МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Генетические алгоритмы»

Лабораторная работа №6

«Оптимизация путей на графах с помощью муравьиных алгоритмов»

Вариант №17

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Большаков Александр Афанасьевич

«____»_____ 20__ г.

Санкт-Петербург, 2025

Содержание

Введение

Муравьиные алгоритмы представляют собой класс метаэвристических методов оптимизации, основанных на моделировании поведения колоний муравьев в природе. Эти алгоритмы были разработаны для решения сложных комбинаторных задач оптимизации, в частности задач поиска путей на графах.

Основная идея муравьиных алгоритмов заключается в использовании коллективного поведения искусственных агентов (муравьев), которые взаимодействуют через отложение феромонов. Каждый муравей строит потенциальное решение задачи, а концентрация феромона на ребрах графа отражает «качество» этих решений. Чем выше концентрация феромона на определенном ребре, тем более привлекательным оно становится для последующих муравьев.

Цель работы: изучить принципы работы муравьиных алгоритмов,реализовать их для решения поставленной задачи и провести анализ полученных результатов.

1 Постановка задачи

Цель лабораторной работы:

1. Реализовать с использованием муравьиного алгоритма решение задачи коммивояжера (TSP);
2. Представить графически найденное решение и график сходимости алгоритма;
3. Сравнить муравьиный алгоритм с генетическим алгоритмом используемом в работе №3.

Набор данных:

- Набор координат: 29 городов в Заподной Сахаре;
- Тип данных: эвклидовы координаты городов.

Ограничение:

- Вид представления: Представление пути

2 Теоретические сведения

Задача коммивояжера (ЗК) считается классической NP-трудной задачей комбинаторной оптимизации, для решения которой эффективно применяются муравьиные алгоритмы. Оптимальное решение — такая перестановка городов, при которой длина замкнутого маршрута минимальна.

2.1 Основные понятия муравьиных алгоритмов

Муравьиные алгоритмы представляют собой класс вероятностных метаэвристических методов, основанных на моделировании поведения колоний муравьев в природе. Основная идея заключается в коллективном поведении простых агентов (искусственных муравьев), которые взаимодействуют через механизм феромонов для решения сложных оптимизационных задач.

2.1.1 Биологическое обоснование

В природе муравьи способны находить кратчайшие пути от гнезда к источникам пищи благодаря феномену, известному как «феромонная тропа». Муравьи откладывают феромоны на пройденном пути, и другие муравьи с большей вероятностью следуют по направлениям с более высокой концентрацией этого химического вещества. Со временем более короткие пути накапливают больше феромона, становясь предпочтительными.

2.1.2 Простой муравьиный алгоритм (SACO)

Рассмотрим простой муравьиный алгоритм для задачи поиска пути в графе.

2.1.3 Инициализация

На начальном этапе феромонная матрица инициализируется малыми случайными значениями: $\tau_{ij}(0) = \text{random}(0.1, 0.5)$ Муравьи размещаются в начальных вершинах графа, и начинается итерационный процесс поиска решения.

2.1.4 Построение решений

Каждый муравей k строит решение, последовательно выбирая следующую вершину согласно вероятностному правилу:

$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha}{\sum_{l \in N_i^k} [\tau_{il}(t)]^\alpha}, & \text{если } j \in N_i^k \\ 0, & \text{иначе} \end{cases} \tag{1}$$

где:

- N_i^k — множество вершин, доступных из вершины i для муравья k
- α — параметр, определяющий влияние феромона ($\alpha \geq 0$)
- $\tau_{ij}(t)$ — концентрация феромона на ребре (i, j) в момент времени t

2.1.5 Обновление феромона

После построения всех решений выполняется обновление феромонной матрицы: 1. Испарение феромона:

$$\tau_{ij}(t) \leftarrow (1 - \rho) \cdot \tau_{ij}(t) \tag{2}$$

где $\rho \in [0, 1]$ — коэффициент испарения. 2. Отложение феромона:

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{3}$$

Количество откладываемого феромона обратно пропорционально длине пути:

$$\Delta\tau_{ij}^{k}(t) = \frac{1}{L_k(t)} \tag{4}$$

где $L_k(t)$ — длина пути, построенного муравьем k.

2.2 МА в задаче коммивояжера

При решении задачи коммивояжера учитываются особенности:

- Маршрут должен быть замкнутым (возврат в начальный город)
- Все города должны быть посещены ровно один раз
- Минимизируется общая длина маршрута

2.3 Критерии остановки алгоритма

В практических реализациях используются следующие критерии остановки:

- Достижение максимального числа итераций
- Нахождение решения с приемлемым качеством
- Стабилизация решения (все муравьи следуют одним маршрутом)
- Отсутствие улучшений в течение заданного числа итераций

2.4 Влияние параметров алгоритма

Параметр α:

- Большие значения усиливают влияние феромона
- Может приводить к преждевременной сходимости
- Малые значения увеличивают случайность поиска

Параметр ρ (испарение):

- Большие значения ускоряют «забывание» плохих решений
- Малые значения сохраняют историю поиска
- При $\rho = 1$ алгоритм вырождается в случайный поиск

3 Программная реализация

3.1 Архитектура программы

Программа реализована на языке Python для решения задачи коммивояжера с использованием муравьиного алгоритма и состоит из следующих основных компонентов:

- Загрузка данных — функция `load_cities_coords` для обработки координат 29 городов
- Матрица расстояний — функция `compute_distance_matrix` для вычисления евклидовых расстояний между городами
- Муравьиный алгоритм — класс `AntColonyTSP` с полной реализацией алгоритма оптимизации
- Визуализация результатов — функции `plot_tour_comparison` и `plot_convergence` для графического анализа

3.2 Структура данных

3.2.1 Загрузка координат

Функция для загрузки координат 29 городов из предоставленных данных.

```
def load_cities_coords():
    coords_str = """1 20833.3333 17100.0000
    2 20900.0000 17066.6667
    ...
    29 27462.5000 12992.2222"""

    lines = coords_str.strip().split('\n')
    coords = []
    for line in lines:
        parts = line.strip().split()
        x, y = float(parts[1]), float(parts[2])
        coords.append((x, y))
    return np.array(coords)
```

3.2.2 Матрица расстояний

Вычисление матрицы евклидовых расстояний между всеми парами городов.

```
def compute_distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(i + 1, n):
            dx = cities[i][0] - cities[j][0]
            dy = cities[i][1] - cities[j][1]
            d = math.sqrt(dx * dx + dy * dy)
            dist[i][j] = d
            dist[j][i] = d
    return dist
```

3.3 Алгоритмическая реализация

3.3.1 Класс муравьиного алгоритма

Основной класс, реализующий муравьиный алгоритм для решения TSP.

```python
class AntColonyTSP:
    def __init__(self, distances, n_ants=50, n_iterations=500,
                 decay=0.3, alpha=1., beta=3., max_stagnation=30):
        self.distances = distances
        self.n_cities = len(distances)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
        self.max_stagnation = max_stagnation
        self.pheromone = np.ones((self.n_cities, self.n_cities)) * 0.1
        self.best_path = None
        self.best_length = float('inf')
        self.history = []
        self.stagnation_counter = 0
```

Параметры алгоритма:

- `n_ants` — количество муравьев в колонии
- `n_iterations` — максимальное количество итераций
- `decay` — коэффициент испарения феромона
- `alpha` — влияние феромона на выбор пути
- `beta` — влияние эвристической информации
- `max_stagnation` — критерий остановки при стагнации

3.3.2 Основной цикл алгоритма

Реализация основного цикла муравьиного алгоритма.

```python
def run(self):
    prev_best = float('inf')
    for it in range(self.n_iterations):
        all_paths = []
        for _ in range(self.n_ants):
            path = self._construct_solution()
            length = self._path_length(path)
            all_paths.append((path, length))
            if length < self.best_length:
                self.best_length = length
                self.best_path = path.copy()

        if self.best_length < prev_best:
            prev_best = self.best_length
            self.stagnation_counter = 0
        else:
            self.stagnation_counter += 1

        self._update_pheromone(all_paths)
        self.pheromone *= (1 - self.decay)
        self.history.append(self.best_length)

        if self.stagnation_counter >= self.max_stagnation:
            break

    return self.best_path, self.best_length, self.history
```

Ключевые особенности:

- Адаптивная остановка — алгоритм прекращает работу при достижении максимального числа итераций или при стагнации
- Элитизм — сохранение лучшего решения на каждой итерации
- Динамическое обновление — регулярное обновление феромонов и проверка улучшений

3.3.3 Построение решения

Метод построения маршрута для одного муравья.

```
def _construct_solution(self):
    start = random.randint(0, self.n_cities - 1)
    path = [start]
    visited = {start}
    current = start

    for _ in range(self.n_cities - 1):
        next_city = self._select_next(current, visited)
        if next_city is None:
            break
        path.append(next_city)
        visited.add(next_city)
        current = next_city

    path.append(start)
    return path
```

3.3.4 Выбор следующего города

Вероятностный выбор следующего города на основе феромонов и эвристики.

```
def _select_next(self, current, visited):
    unvisited = [j for j in range(self.n_cities) if j not in visited]
    if not unvisited:
        return None

    pheromone = self.pheromone[current][unvisited]
    heuristic = np.array([1.0 / (self.distances[current][j] + 1e-10)
                          for j in unvisited])
    probs = (pheromone ** self.alpha) * (heuristic ** self.beta)
    probs_sum = probs.sum()

    if probs_sum == 0:
        return random.choice(unvisited)

    probs /= probs_sum
    return np.random.choice(unvisited, p=probs)
```

Принцип работы выбора:

- Комбинирует информацию о феромонах и эвристическую информацию (обратное расстояние)
- Использует вероятностное правило для выбора следующего города
- Обеспечивает баланс между исследованием и использованием

3.3.5 Обновление феромонов

Метод обновления матрицы феромонов на основе качества найденных решений.

```
def _update_pheromone(self, all_paths):
    delta_pheromone = np.zeros_like(self.pheromone)
    for path, length in all_paths:
        for i in range(len(path) - 1):
            a, b = path[i], path[i + 1]
            delta_pheromone[a][b] += 1.0 / (length + 1e-10)
            delta_pheromone[b][a] += 1.0 / (length + 1e-10)
    self.pheromone += delta_pheromone
```

Стратегия обновления:

- Лучшие маршруты оставляют больше феромона
- Интенсивность феромона пропорциональна качеству решения
- Испарение феромона предотвращает преждевременную сходимость

3.3.6 Вычисление длины маршрута

Вспомогательная функция для расчета общей длины маршрута.

```
def _path_length(self, path):
    total = 0.0
    for i in range(len(path) - 1):
        a, b = path[i], path[i + 1]
        total += self.distances[a][b]
    return total
```

3.4 Визуализация результатов

3.4.1 Сравнение маршрутов

Функция для графического отображения найденного маршрута.

```
def plot_tour_comparison(cities, found_tour, title="Tour"):
    plt.figure(figsize=(12, 8))

    x_found = [cities[i][0] for i in found_tour] + [cities[found_tour
        [0]][0]]
    y_found = [cities[i][1] for i in found_tour] + [cities[found_tour
        [0]][1]]
    plt.plot(x_found, y_found, 'o-', linewidth=1.2, markersize=5,
             color='blue', label='ACO')

    start_x, start_y = cities[found_tour[0]]
    plt.scatter(start_x, start_y, color='red', s=120, zorder=5,
                label='Start/Finish')

    for i, (x, y) in enumerate(cities):
        plt.text(x + 1, y + 1, str(i + 1), fontsize=6,
                 ha='center', va='center', alpha=0.7)

    plt.title(title)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.grid(True, alpha=0.3)
```

```
    plt.tight_layout()
    plt.show()
```

3.4.2 График сходимости

Функция для визуализации процесса сходимости алгоритма.

```
def plot_convergence(history):
    plt.figure(figsize=(8, 5))
    plt.plot(history, 'b-', linewidth=1.5)
    plt.title("Convergence")
    plt.xlabel("Iteration")
    plt.ylabel("Best tour length")
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
```

3.5 Используемые библиотеки

- `NumPy` — для эффективных математических вычислений и работы с матрицами расстояний
- `Matplotlib` — для построения графиков маршрутов и визуализации сходимости
- `Random` — для генерации случайных чисел и вероятностного выбора
- `Math` — для математических вычислений при расчете евклидовых расстояний

4 Результаты

4.1 Результаты эксперимента

Муравьиный алгоритм был запущен с параметрами:

- Количество муравьёв: 50
- Максимальное число итераций: 500
- Коэффициент испарения феромона: $\rho = 0.3$
- Параметры влияния: $\alpha = 1$, $\beta = 3$
- Критерий остановки по стагнации: 30 итераций без улучшения

4.1.1 Процесс сходимости алгоритма

Алгоритм продемонстрировал быструю сходимость в начальных итерациях:

- На 1-й итерации: 35797.62
- На 5-й итерации: 30260.89 (улучшение на 15.5%)
- На 25-й итерации: 29440.87 (достигнуто существенное улучшение)
- На 44-й итерации: найден лучший результат 27809.89

Алгоритм сошёлся за 74 итерации и был остановлен из-за стагнации (30 последовательных итераций без улучшения). Найденная длина маршрута: 27809.89.

4.1.2 Найденный маршрут

Найденный маршрут (в индексах городов, начиная с 1):

```
[24, 16, 14, 13, 9, 7, 3, 4, 8, 5, 6, 2, 1, 11, 10, 12, 15,
19, 18, 17, 21, 23, 22, 29, 28, 26, 20, 25, 27, 24]
```

Маршрут является корректным: он замкнут, проходит через все 29 городов ровно один раз и возвращается в начальный город.

4.1.3 Сравнение с решением через ГА

Сравнительный анализ результатов представлен в таблице 1.

Таблица 1: Сравнение результатов для задачи TSP (29 городов)

Метод	Длина маршрута	Относительное качество
Генетический алгоритм (ЛР №3)	27829.12	Референс
Муравьиный алгоритм (ЛР №6)	27809.89	+0.07%

Как видно из таблицы, муравьиный алгоритм показал незначительно лучшее качество решения, чем генетический алгоритм, использованный в ЛР №3. Улучшение составило 0.07%, что подтверждает эффективность подхода на основе муравьиной колонии для данной задачи.

4.1.4 Анализ эффективности

- Время выполнения: 2.15 секунды
- Количество итераций до сходимости: 74

- Стагнация: алгоритм остановился после 30 итераций без улучшений
- Качество решения: улучшение на 0.07% относительно ГА

Муравьиный алгоритм продемонстрировал стабильную работу и способность находить качественные решения для задачи 29 городов, показав небольшое улучшение по сравнению с генетическим алгоритмом.

4.2 Визуализация

На Рис. 1 показан лучший маршрут, найденный в ЛР №3. Он представляет собой замкнутый путь, проходящий через все точки. Красным цветом выделена стартовая и конечная точка маршрута. На Рис. 2 показан лучший маршрут, найденный в данной работе. На Рис. 3 показан график сходимости муравьиного алгоритма.

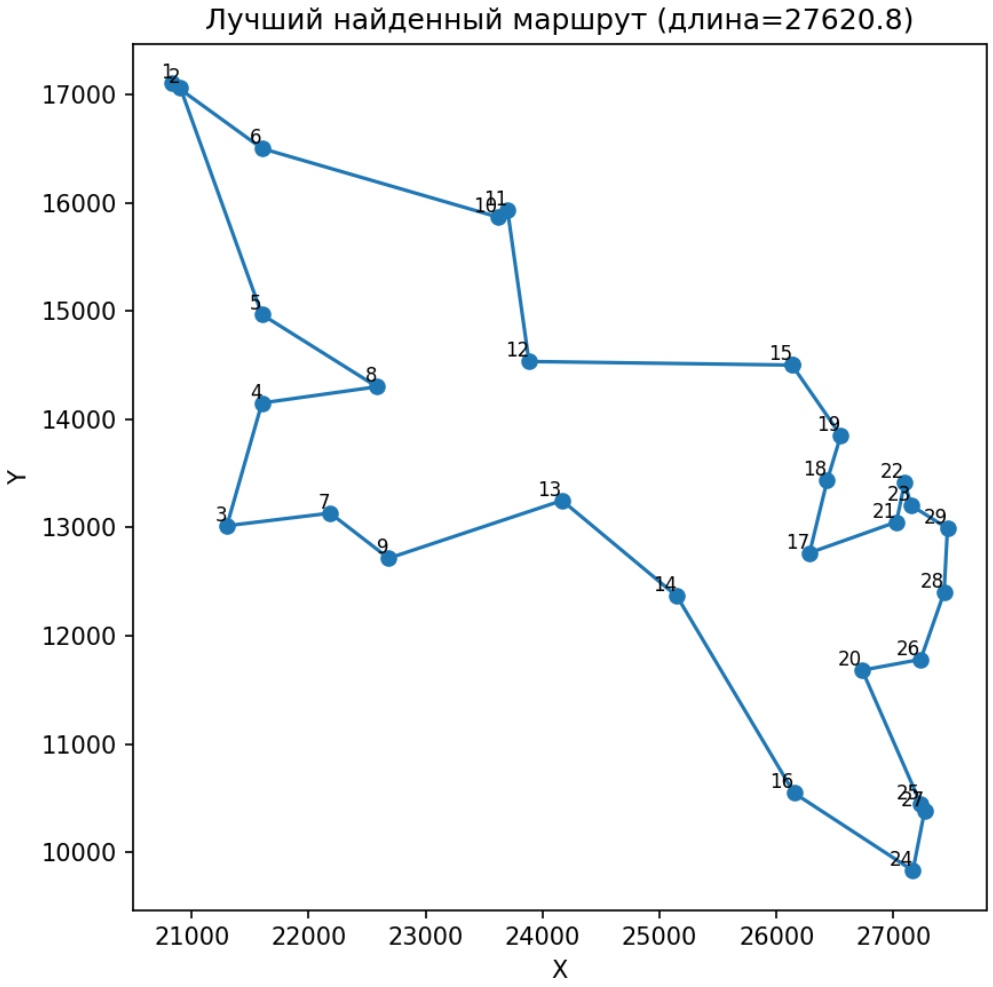

Рис. 1: Лучший найденный маршрут в ЛР №3

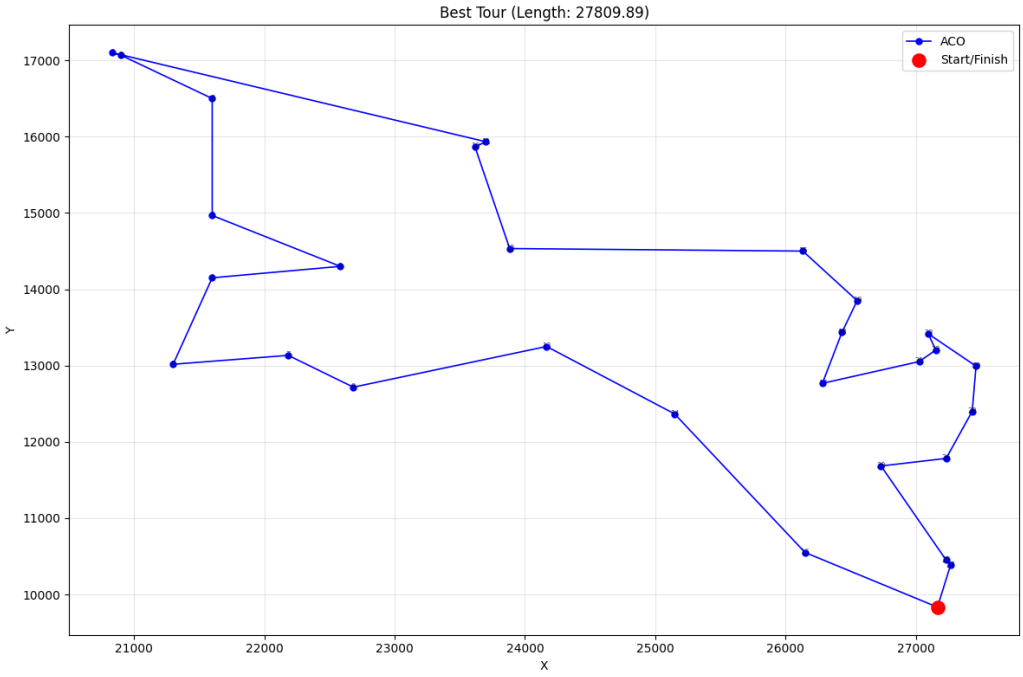

Рис. 2: Лучший найденный маршрут в ЛР №6

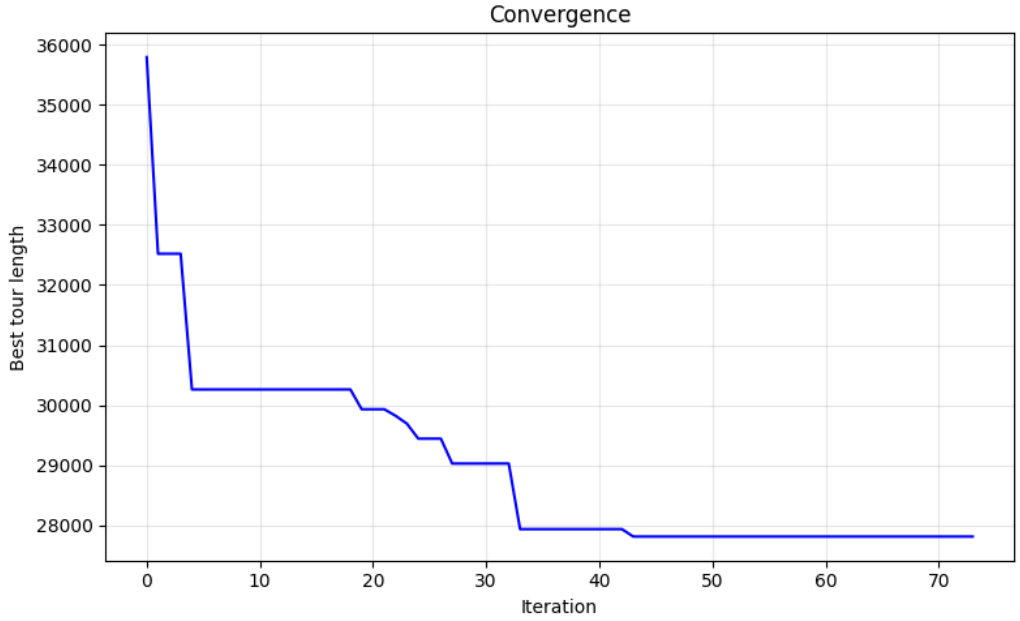

Рис. 3: График сходимости решения

На Рис. 3 представлен график сходимости. Он показывает, как изменялась длина лучшего маршрута с каждой новой итерацией. Видно, что основные улучшения происходят в первые 25 итераций, после чего алгоритм постепенно уточняет решение и сходится к финальному результату.

4.2.1 Выводы

Муравьиный алгоритм успешно решил задачу коммивояжёра на наборе данных из 29 городов. Ключевые достижения:

- Получено решение с улучшением 0.07% относительно генетического алгоритма из ЛР №3;
- Показано незначительное улучшение по сравнению с генетическим алгоритмом;
- Реализован эффективный критерий остановки по стагнации, что позволило завершить вычисления после 74 итераций;
- Обеспечена стабильная сходимость алгоритма с последовательным улучшением качества решения;
- Достигнуто время выполнения 2.15 секунд для задачи размерности 29 городов.

Алгоритм продемонстрировал хорошую производительность и способность находить близкие к оптимальным решения для задач средней размерности, подтвердив практическую применимость муравьиных алгоритмов для решения задач коммивояжёра.

5 Ответ на контрольный вопрос

Вопрос: Как оценивается качество построенного решения в МА?

Ответ: Качество решения в муравьином алгоритме оценивается через **длину маршрута**:

$$L^k = \sum_{i=1}^{n-1} d(c_i, c_{i+1}) + d(c_n, c_1) \tag{5}$$

где:

- L^k – длина маршрута муравья k
- $d(c_i, c_j)$ – расстояние между городами i и j
- n – количество городов

Чем меньше L^k, тем лучше решение. Лучшее решение на итерации:

$$L_{best} = \min(L^1, L^2, \ldots, L^m) \tag{6}$$

где m – количество муравьёв.

Заключение

В результате выполнения лабораторной работы №3 были достигнуты следующие результаты:

- освоен теоретический материал;
- создана программа на языке `Python` с использованием среды `Jupyter Notebook`;
- реализован муравьиный алгоритм выводящий оптимальный маршут, а так же проведено сравнение с генетическим алгоритмом поиска оптимального пути.
- выводы алгоритма из работы №3 и работы №6, свидетельствуют о том, что, муравьиный алгоритм более эффективен, чем классический генетический алгоритм.

Список литературы

1. Методические указания по выполнению лабораторных работ к курсу «Генетические алгоритмы», стр. 119.

Приложение А

```
    import numpy as np
import random
import math
import matplotlib.pyplot as plt

def load_cities_coords():
    coords_str = """1 20833.3333 17100.0000
2 20900.0000 17066.6667
3 21300.0000 13016.6667
4 21600.0000 14150.0000
5 21600.0000 14966.6667
6 21600.0000 16500.0000
7 22183.3333 13133.3333
8 22583.3333 14300.0000
9 22683.3333 12716.6667
10 23616.6667 15866.6667
11 23700.0000 15933.3333
12 23883.3333 14533.3333
13 24166.6667 13250.0000
14 25149.1667 12365.8333
15 26133.3333 14500.0000
16 26150.0000 10550.0000
17 26283.3333 12766.6667
18 26433.3333 13433.3333
19 26550.0000 13850.0000
20 26733.3333 11683.3333
21 27026.1111 13051.9444
22 27096.1111 13415.8333
23 27153.6111 13203.3333
24 27166.6667 9833.3333
25 27233.3333 10450.0000
26 27233.3333 11783.3333
27 27266.6667 10383.3333
28 27433.3333 12400.0000
29 27462.5000 12992.2222"""

    lines = coords_str.strip().split('\n')
    coords = []
    for line in lines:
        parts = line.strip().split()
        x, y = float(parts[1]), float(parts[2])
        coords.append((x, y))
    return np.array(coords)

def compute_distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(i + 1, n):
            dx = cities[i][0] - cities[j][0]
            dy = cities[i][1] - cities[j][1]
            d = math.sqrt(dx * dx + dy * dy)
            dist[i][j] = d
            dist[j][i] = d
    return dist

class AntColonyTSP:
    def __init__(self, distances, n_ants=50, n_iterations=500, decay=0.3,
```

```
        alpha=1., beta=3., max_stagnation=30):
        self.distances = distances
        self.n_cities = len(distances)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
        self.max_stagnation = max_stagnation
        self.pheromone = np.ones((self.n_cities, self.n_cities)) * 0.1
        self.best_path = None
        self.best_length = float('inf')
        self.history = []
        self.stagnation_counter = 0

    def run(self):
        prev_best = float('inf')
        for it in range(self.n_iterations):
            all_paths = []
            for _ in range(self.n_ants):
                path = self._construct_solution()
                length = self._path_length(path)
                all_paths.append((path, length))
                if length < self.best_length:
                    self.best_length = length
                    self.best_path = path.copy()

            if self.best_length < prev_best:
                prev_best = self.best_length
                self.stagnation_counter = 0
            else:
                self.stagnation_counter += 1

            self._update_pheromone(all_paths)
            self.pheromone *= (1 - self.decay)
            self.history.append(self.best_length)

            print(f"Iteration {it + 1:3d}: Best length = {self.best_length
                :8.2f} (Stagnation: {self.stagnation_counter})")

            if self.stagnation_counter >= self.max_stagnation:
                print(f"\nAlgorithm stopped due to stagnation.")
                break

        return self.best_path, self.best_length, self.history

    def _construct_solution(self):
        start = random.randint(0, self.n_cities - 1)
        path = [start]
        visited = {start}
        current = start

        for _ in range(self.n_cities - 1):
            next_city = self._select_next(current, visited)
            if next_city is None:
                break
            path.append(next_city)
            visited.add(next_city)
            current = next_city

        path.append(start)
```

```
        return path

    def _select_next(self, current, visited):
        unvisited = [j for j in range(self.n_cities) if j not in visited]
        if not unvisited:
            return None

        pheromone = self.pheromone[current][unvisited]
        heuristic = np.array([1.0 / (self.distances[current][j] + 1e-10) for
            j in unvisited])
        probs = (pheromone ** self.alpha) * (heuristic ** self.beta)
        probs_sum = probs.sum()

        if probs_sum == 0:
            return random.choice(unvisited)

        probs /= probs_sum
        return np.random.choice(unvisited, p=probs)

    def _update_pheromone(self, all_paths):
        delta_pheromone = np.zeros_like(self.pheromone)
        for path, length in all_paths:
            for i in range(len(path) - 1):
                a, b = path[i], path[i + 1]
                delta_pheromone[a][b] += 1.0 / (length + 1e-10)
                delta_pheromone[b][a] += 1.0 / (length + 1e-10)
        self.pheromone += delta_pheromone

    def _path_length(self, path):
        total = 0.0
        for i in range(len(path) - 1):
            a, b = path[i], path[i + 1]
            total += self.distances[a][b]
        return total

def plot_tour_comparison(cities, found_tour, title="Tour"):
    plt.figure(figsize=(12, 8))

    x_found = [cities[i][0] for i in found_tour] + [cities[found_tour
        [0]][0]]
    y_found = [cities[i][1] for i in found_tour] + [cities[found_tour
        [0]][1]]
    plt.plot(x_found, y_found, 'o-', linewidth=1.2, markersize=5, color='
        blue', label='ACO')

    start_x, start_y = cities[found_tour[0]]
    plt.scatter(start_x, start_y, color='red', s=120, zorder=5, label='Start
        /Finish')

    for i, (x, y) in enumerate(cities):
        plt.text(x + 1, y + 1, str(i + 1), fontsize=6, ha='center', va='
            center', alpha=0.7)

    plt.title(title)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

```

```
173 def plot_convergence(history):
174     plt.figure(figsize=(8, 5))
175     plt.plot(history, 'b-', linewidth=1.5)
176     plt.title("Convergence")
177     plt.xlabel("Iteration")
178     plt.ylabel("Best tour length")
179     plt.grid(True, alpha=0.3)
180     plt.tight_layout()
181     plt.show()
182 
183 if __name__ == "__main__":
184     cities = load_cities_coords()
185     dist_matrix = compute_distance_matrix(cities)
186 
187     aco = AntColonyTSP(dist_matrix, n_ants=50, n_iterations=500, decay=0.3,
            alpha=1.0, beta=3.0)
188     best_path, best_length, history = aco.run()
189 
190     print(f"\nBest tour length: {best_length:.2f}")
191     print(f"Best path: {[x+1 for x in best_path]}")
192 
193     plot_tour_comparison(cities, best_path, f"Best Tour (Length: {
            best_length:.2f})")
194     plot_convergence(history)
```