Санкт-Петербургский государственный университет

Факультет прикладной математики – процессов управления

Лабораторная работа №3

**Отчет**

по дисциплине «Алгоритмы и структуры данных»

**Решение задачи о коммивояжере с помощью муравьиного алгоритма**

Автор работы: Дацык Р.В.

Группа: 22.Б15-пу

Преподаватель: Дик А. Г.

Санкт-Петербург, 2024

**Оглавление**

[**1.** **Цель работы** 3](#_Toc162084618)

[**2.** **Задачи** 3](#_Toc162084619)

[**3.** **Теоретическая часть** 3](#_Toc162084620)

[**4.** **Алгоритм метода** 4](#_Toc162084621)

[**5.** **Описание программы** 7](#_Toc162084622)

[**6.** **Рекомендации пользователю** 7](#_Toc162084623)

[**7.** **Рекомендации программиста** 8](#_Toc162084624)

[**8.** **Контрольный пример** 8](#_Toc162084625)

[**9.** **Вывод** 13](#_Toc162084626)

[**10.** **Список использованной литературы** 13](#_Toc162084627)

[**11.**  **Приложение А** 14](#_Toc162084628)

# **Цель работы**

Исследование особенностей решения задачи о коммивояжере с помощью муравьиного алгоритма и проведение сравнительного анализа с алгоритмами имитации отжига и ближайшего соседа.

# **Задачи**

Для достижения цели поставлены следующие задачи:

1. Формализовать задачу о коммивояжере.
2. Изучить алгоритм метода имитации отжига.
3. Написать программу, реализующую данный алгоритм.
4. Создать приложение для визуализации.
5. Подготовить контрольный пример.
6. Протестировать программу.
7. Проанализировать полученные результаты.
8. Провести сравнительный анализ с другими алгоритмами.

# **Теоретическая часть**

Муравьиный алгоритм (англ. Ant Colony Optimization, ACO) представляет собой метаэвристический оптимизационный метод, вдохновленный поведением муравьев при поиске пути к источнику пищи. Он был разработан Марко Дориго в начале 1990-х годов и нашел применение в решении множества комбинаторных задач, включая задачу коммивояжера (Travelling Salesman Problem, TSP).

Муравьи оставляют за собой феромоны на пути следования. При выборе пути они предпочитают следовать маршрутам с более высокой концентрацией феромонов. Чем больше муравьев следует по определенному маршруту, тем больше феромонов он содержит. Это приводит к усилению использования путей с высокой степенью феромонов. Однако феромоны испаряются со временем, что позволяет алгоритму избегать локальных оптимумов и искать новые решения.

В контексте задачи коммивояжера города представляются вершинами графа, а ребра между ними — возможными путями между городами. Каждое ребро имеет вес, соответствующий стоимости перемещения между городами. При выборе следующего города муравей оценивает вероятность перемещения в каждый из доступных городов на основе феромонов и стоимости пути. После каждой итерации алгоритма феромоны обновляются в соответствии с найденными маршрутами. Более короткие пути получают больше феромонов.

Муравьиный алгоритм представляет собой эффективный и гибкий метод для решения различных задач оптимизации. Он применим к широкому спектру задач, не требует гладких или выпуклых функций цели. Однако для достижения оптимальных результатов необходимо правильно настроить параметры алгоритма.

# **Алгоритм метода**

1. Создается колония виртуальных муравьев, обычно количество муравьев делают равным количеству вершин в графе. Инициализируется стартовое положение каждого муравья. Все рёбра инициализируются начальным количеством феромона.
2. Муравьи отправляются в свободное путешествие. Муравей последовательно посещает вершины, пока не попадет в тупик или не дойдет до стартовой вершины. Решения муравья зависит от раскиданного феромона и расстояний до вершин.
3. В зависимости от того, выполнил ли муравей поставленную задачу, напрямую зависит, будет ли он откладывать феромоны. Если задача не выполнена, то муравей не оставляет феромоны на своем пути, иначе муравей, в зависимости от длины своего пути, оставляет на своем пути феромоны. Это позволяет усилить путь для будущих муравьев и учитывать локальную информацию о качестве пути.
4. Если муравей выполнил поставленную задачу, его путь и длина этого пути сохраняется (в том случае, если длина этого пути короче уже имеющегося).
5. После того, как все муравьи завершат свою работу, делается глобальное обновление феромонов, которое учитывает локальное обновление феромона и испарение феромона.
6. Процесс построения решений и обновления феромонов повторяется заданное количество раз или до достижения критерия остановки. После каждого прогона муравьев качество решения обычно улучшается.

В приложении А прикреплен листинг кода с подробными комментариями.

На рисунках 4.1-4.2 представлена блок-схема алгоритма.

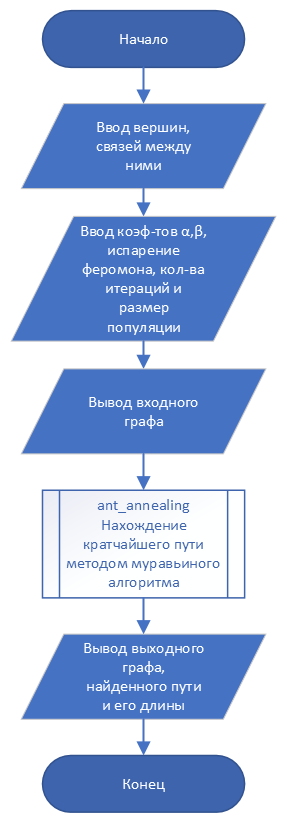


Рисунок 4.1. Блок-схема основной программы

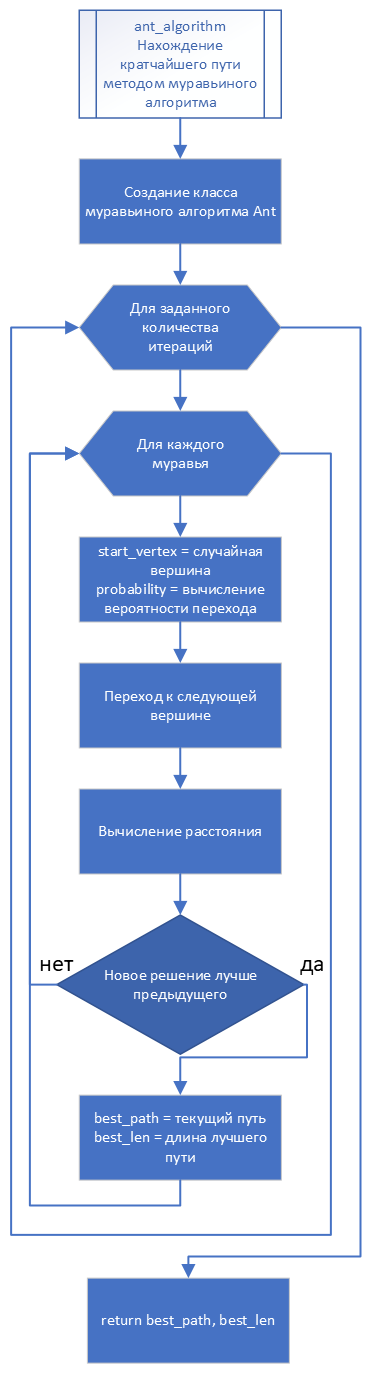


Рисунок 4.2 Блок-схема муравьиного алгоритма

# **Описание программы**

Программа реализована на языке python 3.10 с использованием следующих пакетов: numpy, tkinter, networkx, matplotlib.

В программе используются 8 функций: 5 принадлежащих классу муравьиного алгоритма и 3 для обработки интерфейса. В таблицах 5.1-5.2 представлено описание всех функций.

Таблица 5.1. Описание функций класса GraphGUI.

|  |  |  |
| --- | --- | --- |
| Имя функции | Аргументы функции | Назначение |
| \_\_init\_\_ | self, root | Инициализация окна |
| create\_frames | self | Отображение рамок, границ |
| create\_graph | self | Отображение входного графа |
| set\_calculating\_command | self | Задание метода решения |
| create\_settings | self | Ввод предварительных настроек |

Таблица 5.2. Описание функций класса Ant.

|  |  |  |
| --- | --- | --- |
| Имя функции | Аргументы функции | Назначение |
| \_\_init\_\_ | self, graph | Инициализация класса |
| calculate\_distance | self | Вычисление длины пути |
| ant\_algorithm | self, num\_ants, num\_iterations, pheromon\_coef, alpha, beta | Основная функция, реализующая муравьиный алгоритм. Возвращает лучший путь и значение целевой функции |

# **Рекомендации пользователю**

Для успешного запуска программы необходимо устройство с операционной системой Linux, macOS или Windows, среда разработки, поддерживающая запуск python 3.10.

Интерфейс предлагает пользователю выбрать параметры входного графа: количество вершин и связи между ними. После установки настроек кнопка «Создать граф» визуализирует полученные данные.

Далее необходимо выбрать предварительные настройки: важность расстояния, важность феромона, размер популяции, коэффициент испарения феромона, количество итераций. По умолчанию выбраны некоторые числа.

Алгоритм запустится после нажатия кнопки «Вычислить оптимальный путь». В интерфейсе отобразится длина найденного гамильтонова цикла, порядок вершин, по которому нужно идти, и новый вычисленный граф.

# **Рекомендации программиста**

Для запуска программы необходима 64-битная операционная система Windows, Linux или macOS. Для работы с кодом необходима среда разработки, совместимая с python 3.10, библиотеки numpy, tkinter, networkx, matplotlib.

Исходный код программы доступен по ссылке:

<https://github.com/LisaNota/spbu-ant-colony>

# **Контрольный пример**

В данном разделе представлен контрольный пример, демонстрирующий работу программы.

Для проведения эксперимента установим следующие параметры:

Количество вершин: 6

Ребра: (0, 1, 3); (1, 0, 3); (0, 2, 3); (2, 0, 4); (0, 3, 6); (3, 0, 7); (4, 0, 3); (0, 4, 1); (5, 0, 5); (0, 5, 4); (1, 2, 8); (2, 1, 3); (1, 3, 6); (3, 1, 5); (1, 4, 2); (4, 1, 3);(1, 5, 3); (5, 1, 3); (2, 3, 1); (3, 2, 8); (2, 4, 7); (4, 2, 1);(2, 5, 1); (5, 2, 3);(3, 4, 1); (4, 3, 3);(3, 5, 5); (5, 3, 8);(4, 5, 9); (5, 4, 9)

Важность расстояния: 2.0

Важность феромона: 1.0

Размер популяции: 10

Коэффициент испарения феромона: 0.5

Количество итераций: 100

Настройки изображены на рисунке 8.1.

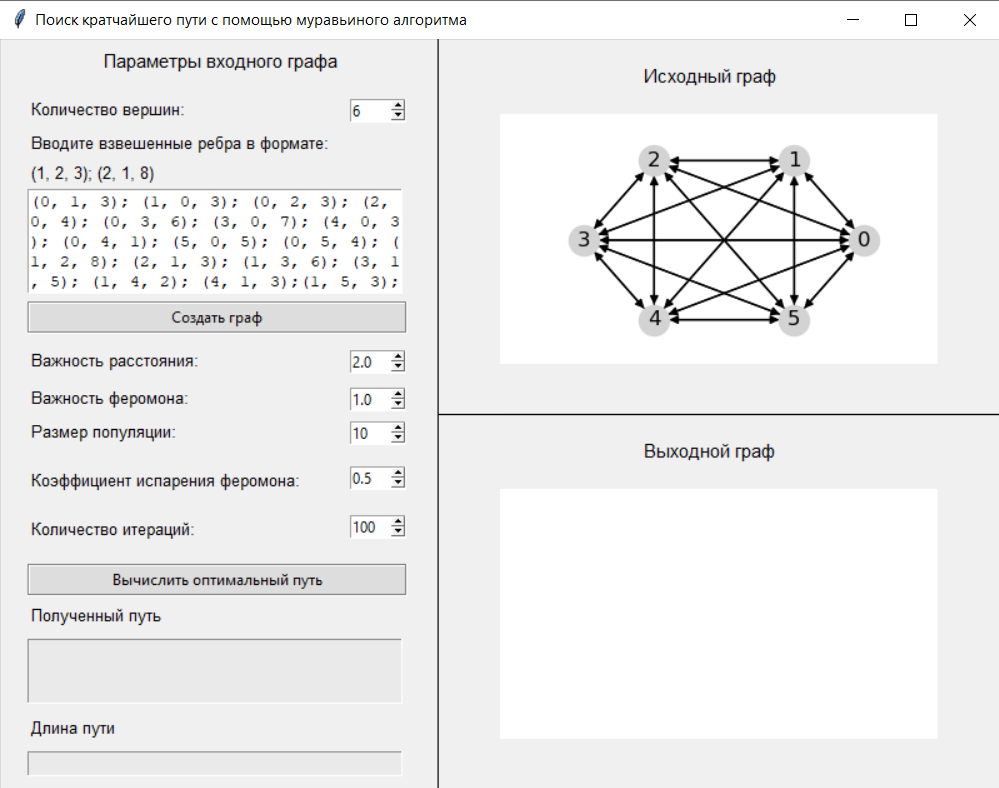


Рисунок 8.1. Выбор предварительных настроек

Найдем минимум целевой функции с помощью метода муравьиного алгоритма. Для этого необходимо нажать на кнопку «Вычислить оптимальный путь». Результат отобразится пользователю в соответствующих графах. Пример вывода показан на рисунке 8.2.

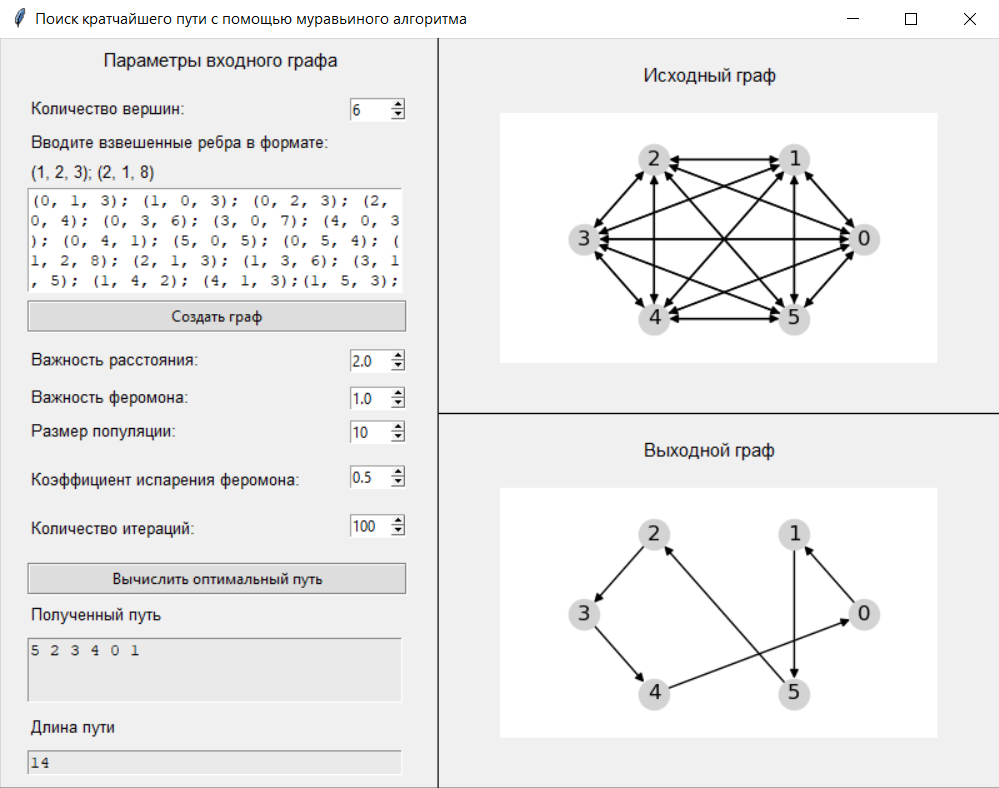


Рисунок 8.2. Результат работы программы

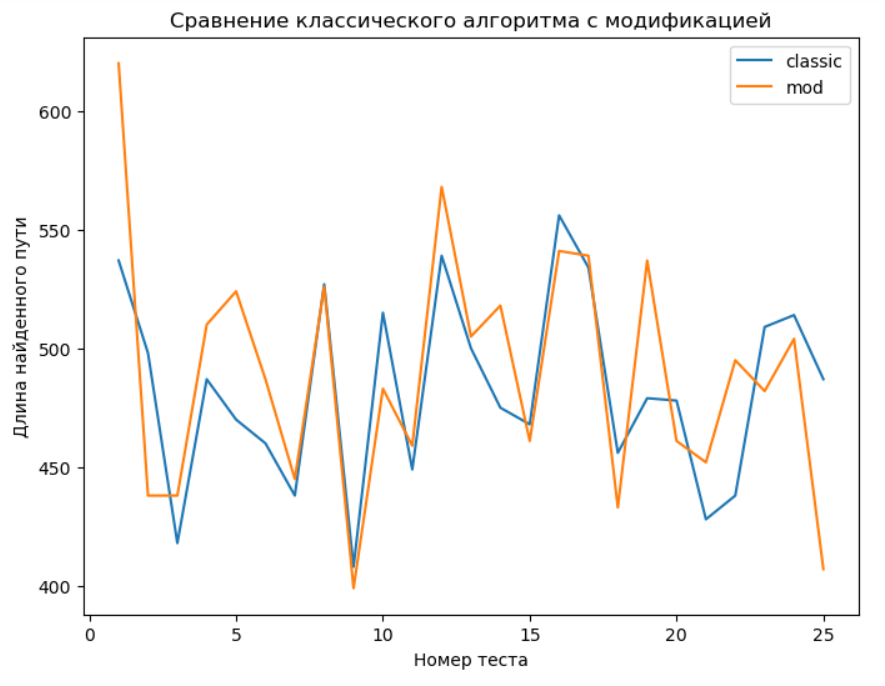
Сравним результаты работы классического алгоритма с его модификацией. Для этого проведем 25 тестов на случайных графах. Исследование изображено на рисунке 8.3.

Рисунок 8.3. Сравнение классического алгоритма с модификацией

Графики показывают неоднозначные результаты: классический алгоритм на разных графах работает как лучше, так и хуже модификации, а иногда их ответы совпадают. Это может быть связано с тем, что, модифицируя алгоритм выбором лучших муравьев, метод может застревать в локальных минимумах.

Проведем другое исследование – сравним алгоритм имитации отжига с алгоритмом ближайшего соседа, а также рассмотрим их модификации.

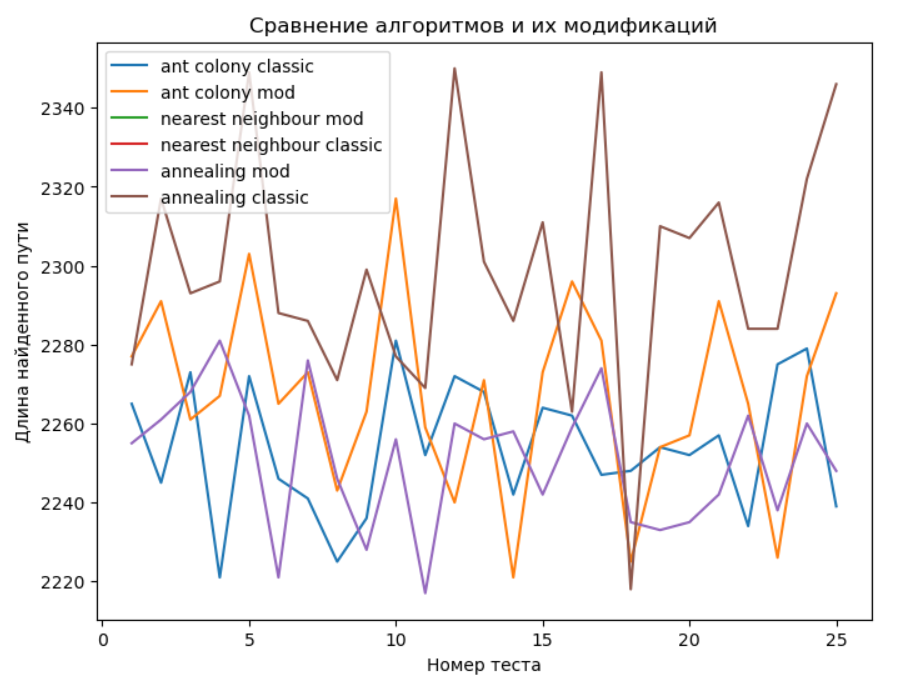


Рисунок 8.4. Сравнение алгоритмов и их модификаций

Графики показывают, что лучший результат показывают методы муравьиного алгоритма. Методы ближайшего соседа работают менее точно, однако являются более стабильными, чем методы имитации отжига. Классический метод имитации отжига показывает наименее устойчивый результат.

Теперь измерим время работы алгоритмов. В таблице 8.1 записаны полученные значения.

Таблица 8.1. Время работы алгоритмов на полном графе на 50 вершинах.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод ближайшего соседа (классический) | Метод ближайшего соседа (модификация) | Метод имитации отжига (классический) | Метод имитации отжига (модификация) | Метод муравьиного алгоритма (классический) | Метод муравьиного алгоритма (модификация) |
| 0.052 | 0.065 | 0.027 | 0.041 | 0.035 | 0.037 |

Из таблицы видно, что наиболее быстрым является классический метод модификации отжига. Более медленной показала себя модификация этого алгоритма, поскольку в качестве начального решения предлагает метод ближайшего соседа. Классический алгоритм ближайшего соседа работает примерно за такое же время, однако его модификация выполняется дольше, поскольку перебирает все вершины в качестве стартовых. Примерно одинаковый результат показали классический и модификация алгоритмы муравьиного отжига.

Отладка алгоритма имитации отжига: в таблицах 8.1-8.3 продемонстрированы зависимости точности от различных предварительных настроек на полном графе на 50 вершинах.

Таблица 8.2. Зависимость точности алгоритма от размера популяции.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | 20 | 50 | 75 | 100 | 250 |
| 2262 | 2242 | 2105 | 2223 | 2244 | 2252 |

Таблица 8.3. Зависимость точности алгоритма от коэффициента испарения феромона.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0.1 | 0.2 | 0.5 | 0.7 | 0.8 | 0.9 |
| 2308 | 2262 | 2111 | 2271 | 2145 | 2287 |

Таблица 8.4. Зависимость точности алгоритма от количества итераций.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | 20 | 50 | 75 | 100 | 250 |
| 2409 | 2346 | 2267 | 2244 | 2235 | 2140 |

Данные исследования показывают, что точность растет при увеличении одного из параметров до определенного момента (для размера популяции – 50, для коэффициента испарения феромона – 0.5, для количества итераций – 250), затем точность алгоритма растет совсем незначительно, а в некоторых случаях даже падает.

# **Вывод**

В результате выполнения работы реализован муравьиный алгоритм с модификацией, а также проведен сравнительный анализ с методами ближайшего соседа и имитации отжига. Реализован интерфейс, подробно демонстрирующий работу написанной программы.

# **Список использованной литературы**

[1] Описание работы генетического алгоритма: <https://ru.wikipedia.org/wiki/Муравьиный_алгоритм>

[2] Документация библиотеки Tkinter: <https://docs.python.org/3/library/tkinter.html>

[3] Полезная статья о муравьином алгоритме в контексте задачи коммивояжера: https://habr.com/ru/companies/timeweb/articles/754462/

[4] Статья о работе с tkinter: <https://pythonru.com/uroki/obuchenie-python-gui-tkinter>

# **Приложение А**

**import** numpy **as** np

**class** **Ant**:

**def** **\_\_init\_\_**(self, graph):

self.graph = graph

self.num\_cities = len(graph)

self.visited = [**False**] \* self.num\_cities

self.tour = []

self.distance = 0.0

**def** **calculate\_distance**(self):

distance = 0

num\_cities = len(self.tour)

**for** i **in** range(num\_cities):

distance += self.graph[self.tour[i % num\_cities]][self.tour[(i + 1) % num\_cities]]

**return** distance

**def** **ant\_algorithm**(self, num\_ants, num\_iterations, pheromon\_coef=0.5, alpha=1, beta=2):

num\_cities = self.num\_cities

pheromones = np.ones((num\_cities, num\_cities))

best\_distance = float('inf')

best\_tour = []

**for** \_ **in** range(num\_iterations):

ants = [Ant(self.graph) **for** \_ **in** range(num\_ants)]

**for** ant **in** ants:

start\_city = np.random.choice(list(self.graph.keys()))

ant.tour.append(start\_city)

ant.visited[start\_city] = **True**

**for** \_ **in** range(num\_cities - 1):

current\_city = ant.tour[-1]

probabilities = []

**for** city, distance **in** self.graph[current\_city].items():

**if** **not** ant.visited[city]:

pheromone = pheromones[current\_city][city]

probabilities.append((city, (pheromone \*\* alpha) \* ((1 / distance) \*\* beta)))

probabilities = np.array(probabilities)

probabilities[:, 1] /= np.sum(probabilities[:, 1])

next\_city = int(np.random.choice(probabilities[:, 0], p=probabilities[:, 1]))

ant.tour.append(next\_city)

ant.visited[next\_city] = **True**

ant.distance = ant.calculate\_distance()

**if** ant.distance < best\_distance:

best\_distance = ant.distance

best\_tour = ant.tour

pheromones \*= (1 - pheromon\_coef)

**for** ant **in** ants:

**for** i **in** range(num\_cities):

pheromones[ant.tour[i % num\_cities]][ant.tour[(i + 1) % num\_cities]] += 1 / ant.distance

**return** best\_tour, best\_distance

Листинг 4.1. Класс Ant