**Санкт-Петербургский государственный УНИВЕРСИТЕТ**

**Факультет прикладной математики и процессов управления**

**отчет**

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Муравьиный алгоритм»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22Б16 |  | Шувалов Ф.В. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2023 г.**

**Оглавление**

[1. Цель работы 3](#_Toc163604686)

[3. Введение 3](#_Toc163604687)

[4. Теоретическая часть 3](#_Toc163604688)

[5. Схема выполнения алгоритма 4](#_Toc163604689)

[7. Представление программы на языке программирования 6](#_Toc163604690)

[8. Раскрытие смысла работы алгоритма 6](#_Toc163604691)

[9. Контрольный пример 8](#_Toc163604692)

[10. Результаты тестирования 10](#_Toc163604693)

[11. Модификация 10](#_Toc163604694)

[12. Анализ результатов тестирования 10](#_Toc163604695)

[13. Вывод 11](#_Toc163604696)

[14. Литература 11](#_Toc163604697)

# **Цель работы**

Цель работы – исследование особенностей решения задачи о коммивояжере с помощью муравьиного алгоритма.

1. **Задачи**
2. Необходимо формализовать задачу о коммивояжере с помощью

алгоритма отжига.

1. Подготовить контрольный пример, используя взвешенный орграф
2. Найти кратчайший гамильтонов цикл.

# **Введение**

Название «Задача о коммивояжере» устойчиво закрепилось за одной из самых интересных, практически значимых и одновременно сложных задач теории графов. Задача, берущая свое начало из работ Гамильтона, состоит в определении кратчайшего гамильтонова цикла в графе. Ее решение связано с решением задачи о назначениях и с задачей об остове наименьшего веса.

Исследование алгоритма отжига для задачи нахождения гамильтонова цикла имеет важное значение, так как этот алгоритм представляет собой мощный метод оптимизации, способный находить приближенные решения сложных задач комбинаторной оптимизации. Понимание его работы, свойств и ограничений может быть полезным для разработки более эффективных методов решения задач, таких как маршрутизация в компьютерных сетях или оптимизация планирования маршрутов для транспортных средств, где требуется нахождение оптимальных или приближенных решений..

# **Теоретическая часть**

**Необходимые термины:**

Вот список терминов, связанных с муравьиным алгоритмом для решения задачи о нахождении гамильтонова цикла:

1. Муравьиный алгоритм (Ant Colony Optimization, ACO): Метод оптимизации, инспирированный поведением муравьев при поиске кратчайшего пути к источнику пищи. Муравьиный алгоритм применяется для решения комбинаторных задач, таких как задачи коммивояжера и задачи о рюкзаке.

2. Феромон: В муравьином алгоритме феромон является химическим веществом, которое муравьи оставляют на своем пути. Когда муравей находит источник пищи, он оставляет феромон на своем пути к нему.

3. Вершина (Node): В контексте муравьиного алгоритма вершина представляет собой узел или город в задаче коммивояжера. Каждая вершина соединена с другими вершинами ребрами.

4. Ребро (Edge): Связь между двумя вершинами в графе. Ребра могут иметь вес, представляющий стоимость перемещения между вершинами.

5. Путь (Path): Последовательность вершин, соединенных ребрами в графе. В муравьином алгоритме путь представляет собой маршрут, который муравьи пройдут от начальной вершины к конечной.

6. Вероятность перехода (Transition Probability): Вероятность того, что муравей выберет определенное ребро из текущей вершины, основываясь на феромонном следе и эвристической информации.

7. Эвристическая информация (Heuristic Information): Дополнительная информация, используемая муравьями при принятии решения о выборе следующей вершины. Эвристическая информация может включать в себя расстояние между вершинами или другие характеристики.

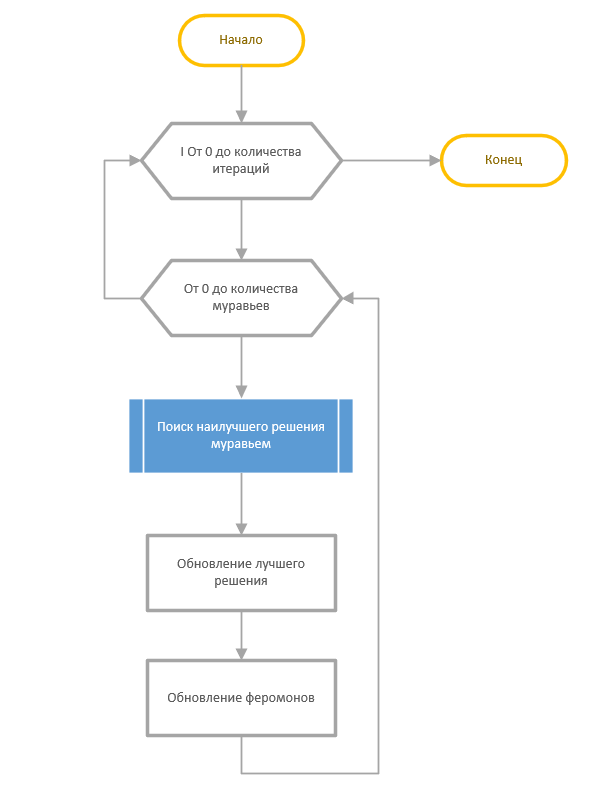
8. Локальное обновление феромона (Local Pheromone Update): Обновление феромонного следа на ребре после прохождения муравьем через него. Локальное обновление обычно происходит после каждого шага муравья.

9. Глобальное обновление феромона (Global Pheromone Update): Обновление феромонного следа на ребрах после завершения прохода всех муравьев по маршрутам. Глобальное обновление обычно происходит путем испарения феромона с последующим добавлением нового.

10. Интенсивность испарения (Evaporation Rate): Параметр, определяющий скорость исчезновения феромонного следа со временем. Высокая интенсивность испарения означает быстрое исчезновение феромона, в то время как низкая означает медленное.

# **Схема выполнения алгоритма**

На рисунках 5.1-5.2 представлена блок-схема алгоритма.

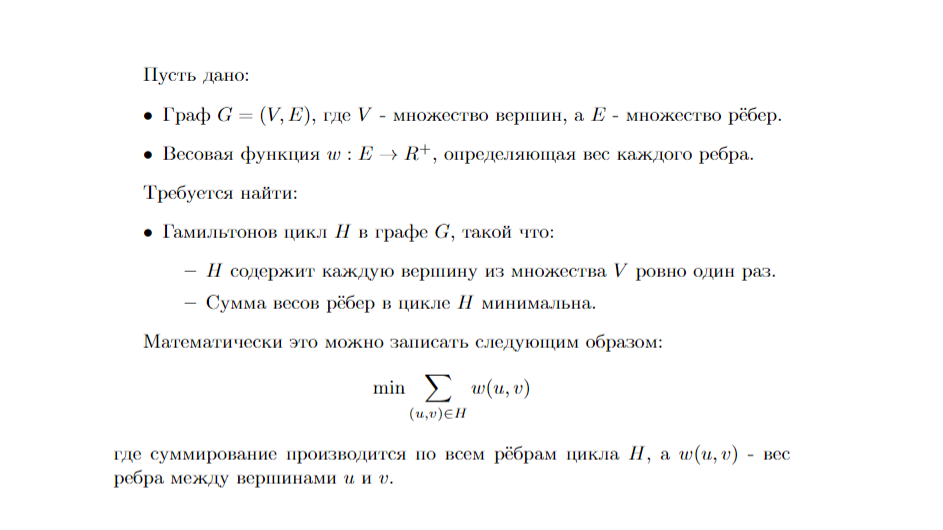


*Рисунок 5.1 Блок-схема основного алгоритма*



*Рисунок 5.2 Блок-схема алгоритма поиска пути*

1. **Формализация задачи**



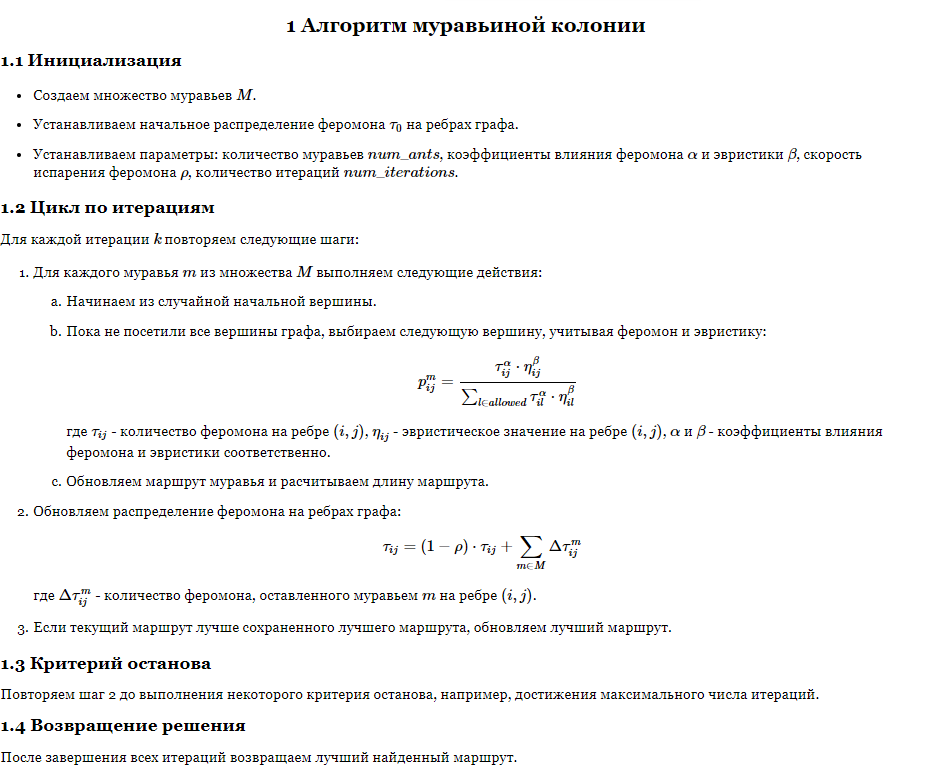
*Рисунок 6.1 Формализация задачи*

# **Представление программы на языке программирования**

Алгоритм реализован на выбранном языке программирования (Python3) с использованием объектно-ориентированного подхода. Программа состоит из двух основных классов: `ACO` и `MACO`. Класс `ACO` представляет реализацию алгоритма муравьиной колонии, а класс ` MACO` представляет его модификацию.

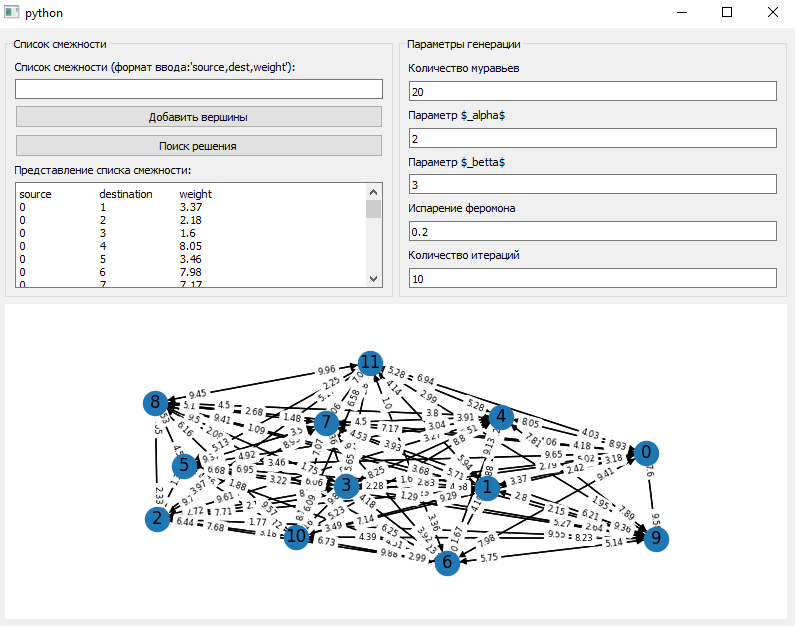
# **Раскрытие смысла работы алгоритма**

Алгоритм муравьиной колонии — это метод оптимизации, вдохновленный поведением муравьев при поиске пути от колонии к источнику пищи. В этом алгоритме муравьи последовательно переходят от начальной точки к конечной, оставляя за собой феромоны, которые влияют на выбор следующего шага. На каждом шаге муравьи оценивают вероятности перехода к каждой из доступных точек, основываясь на количестве феромонов и эвристической информации о расстоянии до каждой точки. Постепенно, через несколько итераций, более оптимальные пути укрепляются за счёт более высокой концентрации феромонов, тем самым приводя к нахождению более кратчайших путей. Алгоритм муравьиной колонии широко применяется в задачах оптимизации, таких как задача коммивояжера, планирование производства и многих других.. Более формальное описание алгоритма представлено на рисунке 8.1.



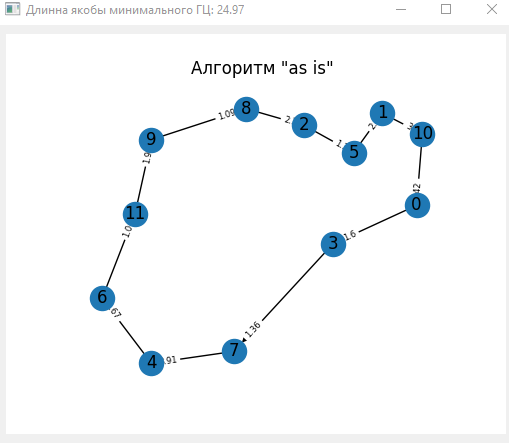
*Рисунок 8.1 Формальное описание алгоритма*

# **Контрольный пример**



*Рисунок 10.1 Визуализация программы*

На рисунке 10.1 изображен пользовательский интерфейс программы. В левой части окна пользователь может вводить параметры, такие как: *граф, коэффициенты alpha и betta, коэффициент испарения феромона и количество итераций.*

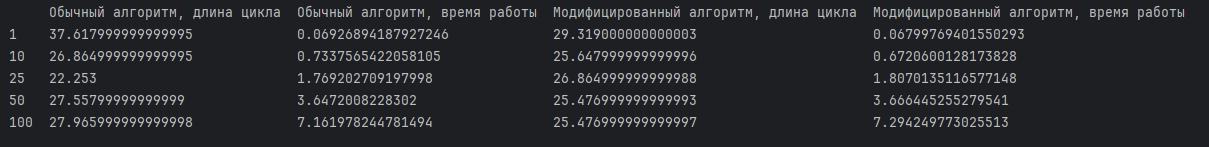
****

*Рисунок 10.2 результат работы алгоритма*

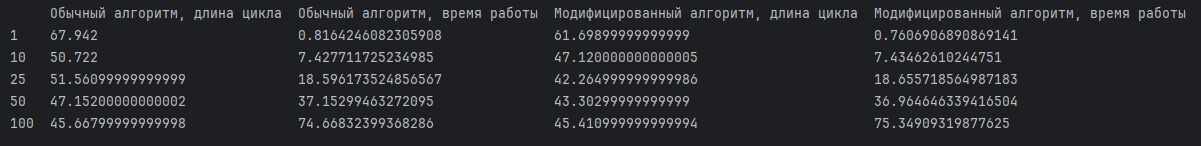
На рисунке 10.2 изображен результат работы алгоритма.

1. Результаты тестирования

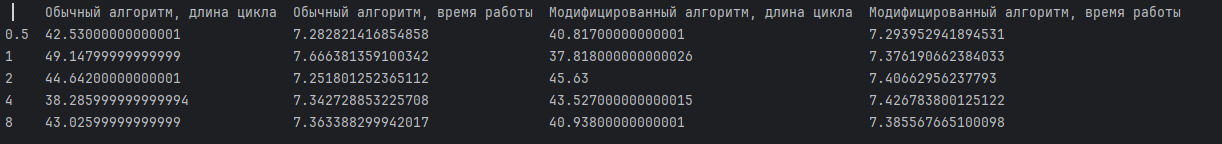
На рисунках 10.1-10.8 приведены результаты тестирования алгоритма.



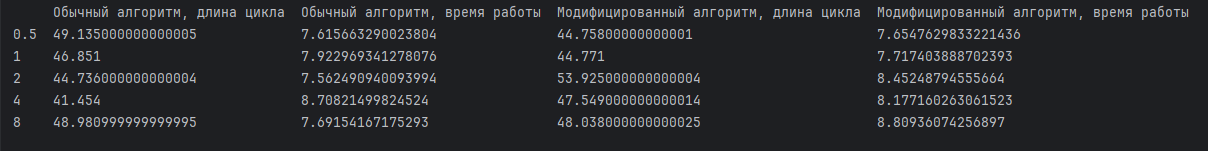
*Рисунок 10.1 Результаты тестирования (количество итераций как переменная)*



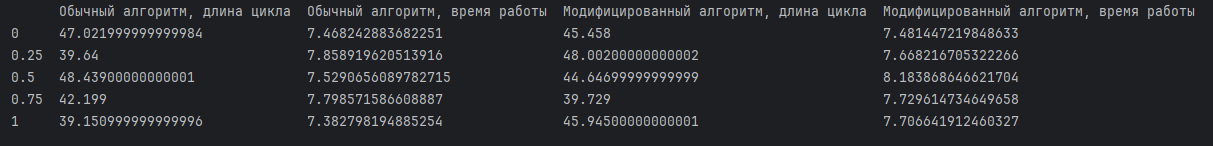
*Рисунок 10.2 Результаты тестирования (количество муравьев как переменная)*

**

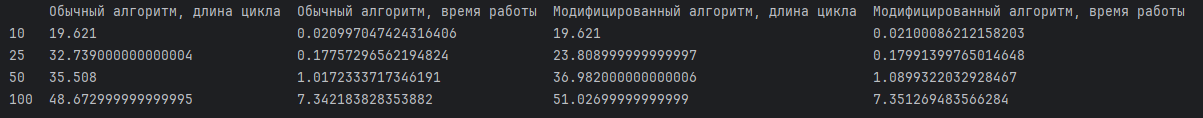
*Рисунок 10.3 Результаты тестирования (коэффициент альфа как переменная)*

**

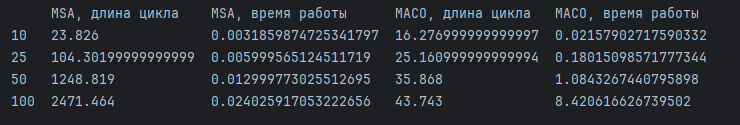
*Рисунок 10.4 Результаты тестирования (коэффициент бета как переменная)*

**

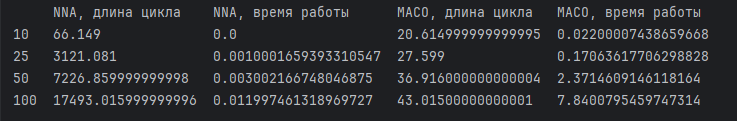
*Рисунок 10.5 Результаты тестирования (коэффициент испарения как переменная)*

**

*Рисунок 10.6 Результаты тестирования (количество вершин в графе как переменная)*

**

*Рисунок 10.7 Результаты тестирования (количество вершин в графе как переменная), сравнительная таблица*

**

*Рисунок 10.8 Результаты тестирования (количество вершин в графе как переменная), сравнительная таблица*

1. Модификация

В данной лабораторной работе была реализована также модификация алгоритма муравьиной колонии, включающая в себя два ключевых аспекта:

1. Наличие “рациональных муравьев”: такие муравьи имеют повышенное влияние эвристики и пониженное влияние феромонов на выбор решения. Реализовано это в виде изменения коэффициентов альфа и бета.
2. Динамический выбор муравьев: на каждой итерации с определенной вероятностью будет выбран один из типов муравьев.

Вместе эти два аспекта обеспечивают более эффективный поиск гамильтонова цикла в графе. Наличие разных типов муравьев позволяет более эффективно исследовать пространство поиска.

1. Анализ результатов тестирования

Оптимальные параметры для алгоритма, полученные по результатам тестирования отображены на рисунке 12.1.



*Рисунок 12.1 Оптимальные параметры для алгоритма*

Анализ результатов тестирования выявил следующие ключевые особенности алгоритма ACO:

1. Значительное ухудшение скорости сходимости при увеличении размера (количества ребер и вершин) графа; для полного графа зависимость времени работы от числа вершин приблизительно O(n^3).
2. Увеличение количества итераций оказывает значительного эффекта на улучшение скорости сходимости.
3. Алгоритм ACO значительно превосходит по точности алгоритм SA, и на многие порядки - NNA
4. Алгоритм ACO значительно проигрывает по скорости алгоритмам NNA и SA по скорости работы в полном графе.

# **Вывод**

При выполнении данной работы были получены все необходимые навыки и знания по реализации алгоритма ACO . В ходе реализации задачи была создана программа для визуализации его работы.

# **Литература**

* + 1. Документация библиотеки PyQT [Электронный ресурс] - URL: <https://doc.qt.io/qtforpython-6/> (дата обращения: 25.09.2023)

1. **Листинг**

import networkx as nx  
import random  
from networkx import DiGraph  
  
  
class ACO:  
 def \_\_init\_\_(self, graph: nx.DiGraph, num\_ants: int=1, alpha: float=2, betta: float=3, evaporation\_rate: float=0.2):  
 self.graph = graph  
 self.num\_ants = num\_ants  
 self.alpha = alpha  
 self.betta = betta  
 self.evaporation\_rate = evaporation\_rate  
 self.best\_path = None  
 self.best\_path\_length = float('inf')  
 self.ant = Ant(self.graph, alpha=self.alpha, betta=self.betta, evaporation\_rate=self.evaporation\_rate)  
  
 def to\_DiGraph(self):  
 return self.convert\_path\_to\_graph(self.best\_path)  
  
 def run(self, iterations: int):  
 for \_ in range(iterations):  
 self.iteration()  
 return self.best\_path  
  
 def solve(self, args: dict = {}):  
 self.num\_ants = args['num\_ants']  
 self.alpha = args['alpha']  
 self.betta = args['betta']  
 self.evaporation\_rate = args['evaporation\_rate']  
 iterations = args['iterations']  
 path = self.run(iterations)  
 self.length = self.calculate\_path\_length(path)  
 return path  
  
 def iteration(self):  
 for ant\_index in range(self.num\_ants):  
 path, path\_length = self.ant.find\_hamiltonian\_cycle()  
 if path\_length < self.best\_path\_length:  
 self.best\_path = path  
 self.best\_path\_length = path\_length  
 self.ant.update\_pheromones(path\_length)  
  
 def get\_best\_path(self):  
 return self.best\_path  
  
 def convert\_path\_to\_graph(self, path):  
 path\_graph = nx.DiGraph()  
 for i in range(len(path) - 1):  
 source = path[i]  
 target = path[i + 1]  
 weight = self.graph[source][target]['weight']   
 path\_graph.add\_edge(source, target, weight=weight)  
 return path\_graph  
  
 def calculate\_path\_length(self, path):  
 length = 0  
 for i in range(len(path) - 1):  
 length += self.graph[path[i]][path[i + 1]]['weight']  
 return length  
  
  
class Ant:  
 def \_\_init\_\_(self, graph: nx.DiGraph, alpha: float, betta: float, evaporation\_rate: float):  
 self.alpha = alpha  
 self.betta = betta  
 self.graph = graph  
 self.path = []  
 self.modified = False  
 self.evaporation\_rate = evaporation\_rate  
 self.pheromones = {(u, v): 1.0 for u, v in graph.edges()}  
  
 def find\_hamiltonian\_cycle(self):  
 start\_node = random.choice(list(self.graph.nodes()))  
 visited = set([start\_node])  
 current\_node = start\_node  
 path = [current\_node]  
 path\_length = 0  
  
 while len(visited) < len(self.graph.nodes()):  
 next\_node = self.choose\_next\_node(current\_node, visited)  
 path\_length += self.graph[current\_node][next\_node]['weight']  
 visited.add(next\_node)  
 path.append(next\_node)  
 current\_node = next\_node  
  
 path\_length += self.graph[path[-1]][start\_node]['weight']  
 path.append(str(start\_node))  
 self.path = path  
 return path, path\_length  
  
 def choose\_next\_node(self, current\_node, visited):  
 unvisited\_neighbors = [n for n in self.graph.neighbors(current\_node) if n not in visited]  
 probabilities = [self.calculate\_probability(current\_node, neighbor) for neighbor in unvisited\_neighbors]  
 return random.choices(unvisited\_neighbors, weights=probabilities)[0]  
  
 def calculate\_probability(self, current\_node, next\_node):  
 pheromone = self.pheromones[(current\_node, next\_node)]  
 distance = self.graph[current\_node][next\_node]['weight']  
 total\_pheromone = sum([self.pheromones[(current\_node, n)] for n in self.graph.neighbors(current\_node)])  
 return (pheromone \*\* self.alpha) \* ((1.0 / distance) \*\* self.betta) / total\_pheromone  
  
 def update\_pheromones(self, path\_length):  
 evaporation = 1 - self.evaporation\_rate  
 for edge in self.graph.edges():  
 self.pheromones[edge] \*= evaporation  
 for i in range(len(self.path) - 1):  
 edge = (self.path[i], self.path[i + 1])  
 self.pheromones[edge] += 1.0 / path\_length  
  
 def convert(self):  
 out = RationalAnt(self.graph, self.alpha, self.betta, self.evaporation\_rate)  
 out.pheromones = self.pheromones  
 out.path = self.path  
 return out  
  
  
class RationalAnt(Ant):  
 def \_\_init\_\_(self, graph, alpha, betta, evaporation\_rate):  
 super().\_\_init\_\_(graph, alpha, betta, evaporation\_rate)  
 self.alpha /= 4  
 self.betta \*= 2  
 self.modified = True  
  
 def convert(self):  
 out = Ant(self.graph, self.alpha \* 4, self.betta / 2, self.evaporation\_rate)  
 out.pheromones = self.pheromones  
 out.path = self.path  
 return out  
  
  
  
class MACO(ACO):  
 def \_\_init\_\_(self, graph: nx.DiGraph, num\_ants: int=1, alpha: float=2, betta: float=3, evaporation\_rate: float=0.2):  
 super().\_\_init\_\_(graph, num\_ants, alpha, betta, evaporation\_rate)  
  
 def iteration(self):  
 for ant\_index in range(self.num\_ants):  
 if random.random() > (0.75 - int(self.ant.modified) / 2):  
 self.ant = self.ant.convert()  
 path, path\_length = self.ant.find\_hamiltonian\_cycle()  
 if path\_length < self.best\_path\_length:  
 self.best\_path = path  
 self.best\_path\_length = path\_length  
 self.ant.update\_pheromones(path\_length)