**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**Факультет прикладной математики – процессов управления**

**отчет**

**по лабораторной работе**

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

**на тему «Решение задачи о коммивояжере с помощью муравьиного алгоритма»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22.Б15 |  | Агишев А.Б. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2024 г.**

**Содержание**

[1. Цель работы 1](#_Toc151413474)

[2. Задача 1](#_Toc151413475)

[3. Теоретическая часть 1](#_Toc151413476)

[4. Описание алгоритма 2](#_Toc151413477)

[5. Описание программы 7](#_Toc151413478)

[5.1 Описание классов 7](#_Toc151413479)

[5.2. Описание функций 7](#_Toc151413480)

[5.3. Описание переменных 8](#_Toc151413481)

[6. Рекомендации пользователю 12](#_Toc151413482)

[7. Рекомендации программисту 12](#_Toc151413483)

[8. Контрольный пример 13](#_Toc151413484)

[9. Заключение 14](#_Toc151413485)

# **Цель работы**

Разработать алгоритм и программу, которая решает задачу о коммивояжере с помощью муравьиного алгоритма и сравнить с работой алгоритма ближайшего соседа и методом имитации отжига.

# **Задача**

Приобрести и закрепить знания в области решения задачи о коммивояжере. Написать программу поиска решения. Протестировать программу на контрольном примере. Сделать заключение об муравьином алгоритме в решении данной задачи. Сравнить работу с методом ближайшего соседа и имитацией отжига.

# **Теоретическая часть**

*Гамильтонов цикл в неориентированном графе* — это такой цикл, который проходит через каждую вершину графа ровно один раз и возвращается в начальную вершину. Таким образом, гамильтонов цикл охватывает все вершины графа без повторений.

*Задача коммивояжера* является одной из классических задач комбинаторной оптимизации. Она заключается в поиске кратчайшего маршрута, который проходит через все заданные города (вершины графа) ровно один раз и возвращается в исходный город. Задача коммивояжера является NP-полной, то есть нет известного алгоритма, который бы эффективно решал её для любого входа за полиномиальное время. Однако существуют различные эвристические и точные алгоритмы для её приближенного или оптимального решения.

*Муравьиный алгоритм* является одним из эвристических методов решения задачи коммивояжера. Он основан на модели поведения муравьев, которые общаются друг с другом с помощью феромонов, оставляемых на путях, которые они проходят. Основные идеи алгоритма вдохновлены поведением реальных муравьев при поиске кратчайшего пути от места к месту.

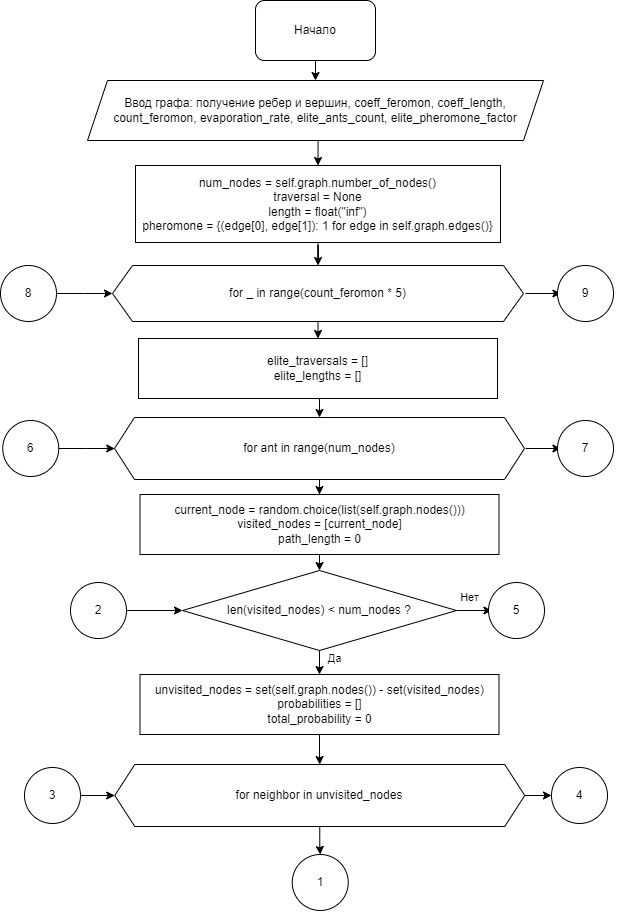
В контексте задачи коммивояжера муравьиный алгоритм работает следующим образом:

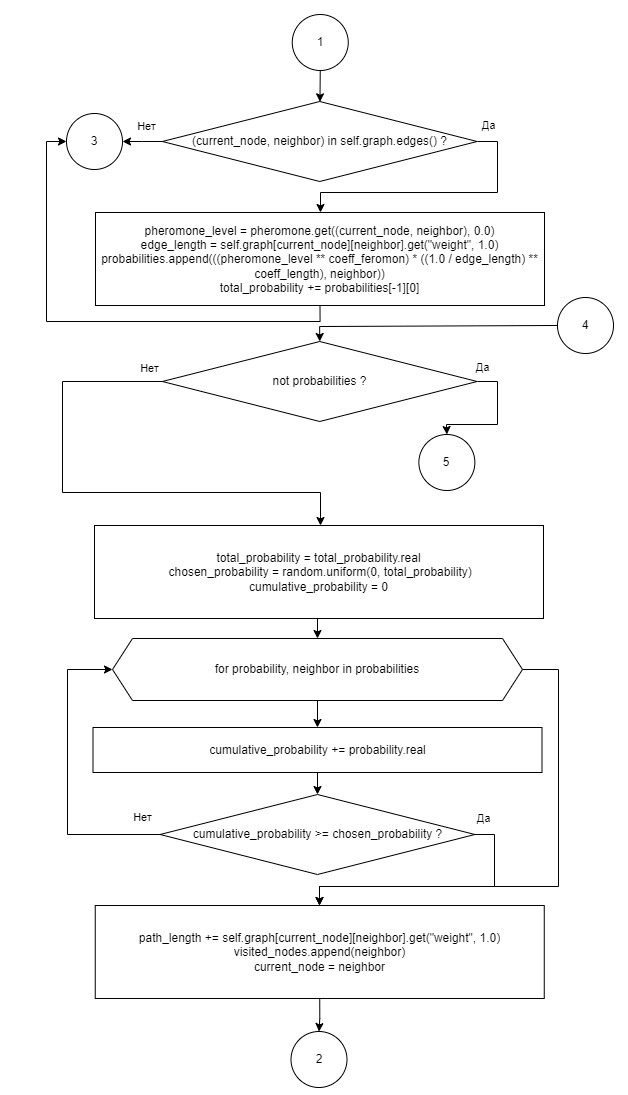
1. Начально каждый муравей располагается в случайной вершине графа (городе) и начинает движение.
2. На каждом шаге муравей выбирает следующий город для посещения с учётом информации о феромонах на рёбрах и эвристической информации (например, расстояния между городами).
3. После того как все муравьи завершили свои путешествия, феромоны обновляются на основе длины пройденных маршрутов: чем короче маршрут, тем больше феромона оставляется на его рёбрах.
4. Феромоны испаряются с течением времени, чтобы избежать застоя в алгоритме.

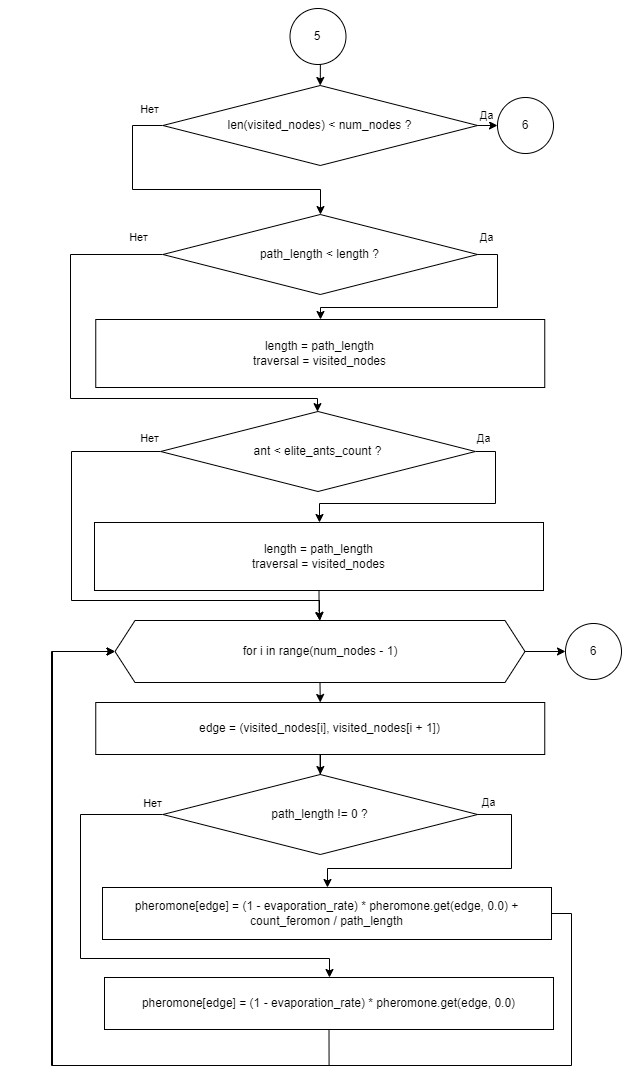
# **Описание алгоритма**

1. Получение характеристики графа, параметров для алгоритма через интерфейс программы.
2. Запуск поиска решения задачи о коммивояжере с помощью муравьиного алгоритма.
3. Для каждого муравья:
   1. Поиск соседнего решения (следующего города для посещения) с учётом феромонов и эвристической информации.
   2. Оценка стоимости (длины маршрута) соседнего решения.
   3. Принятие соседнего решения в зависимости от его стоимости и температуры (как в методе оптимизации отжига). В этом шаге также учитывается информация о лучшем найденном решении (элитные муравьи).
   4. Обновление текущего решения в случае принятия соседнего.
4. Завершение алгоритма, когда выполнено заданное количество итераций.
5. Вычисление длины маршрута и порядка обхода для лучшего найденного решения.
6. Возврат результата - длины маршрута, порядка обхода, количества феромона на каждом ребре в качестве ответа.

В рассмотренном алгоритме муравьиного метода была использована оптимизация с помощью элитных муравьев. Это означает, что при выборе следующего города для посещения каждым муравьем учитывалась информация о лучшем найденном решении, что позволило улучшить качество и эффективность алгоритма.







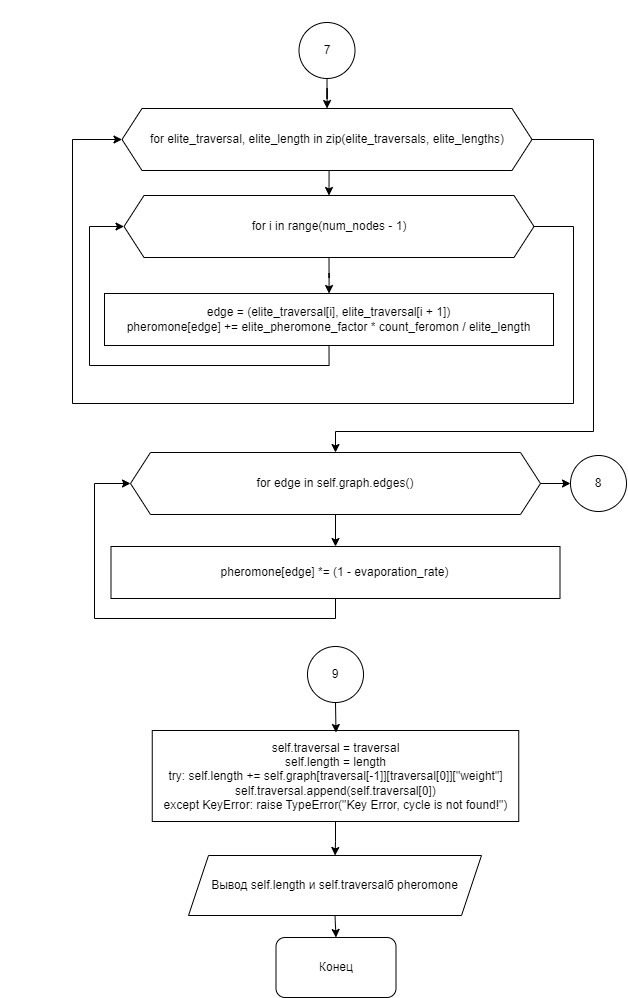


Рисунок 4.1–4.4. Блок-схема алгоритма.

# **Описание программы**

## **Описание классов**

В программе используется 3 класса: 2 связанных с интерфейсом программы, 1 связанный с поиском решения задачи. В таблице 5.1 представлено описание классов.

*Таблица 5.1. Описание классов*

|  |  |  |
| --- | --- | --- |
| Имя класса | Наследование | Описание |
| Traveling\_Salesman | — | Поиск решения задачи о коммивояжере |
| GraphEditor | — | Редактирование графа в интерфейсе |
| Interface | — | Графический пользовательский интерфейс |

## **5.2. Описание функций**

Описание функций класса *«Traveling\_Salesman»* представлено в таблице 5.2.

*Таблица 5.2. Описание функций класса «Traveling\_Salesman»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| Экземпляр редактора графа | \_\_init\_\_ | Инициализация класса |
| Коэффициент значимости феромона, коэффициент значимости длины, количество дополнительного феромона, интенсивность испарения, количество элитных муравьев, фактор феромона для элитных муравьев | ant\_algo | Решение задачи с помощью муравьиного алгоритма |
| Шаблоны графических элементов | view | Вывод в интерфейс порядка обхода |

Описание функций класса *«GraphEditor»* представлено в таблице 5.3.

*Таблица 5.3. Описание функций класса «GraphEditor»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| Экземпляр методов интерфейса, аргументы | \_\_init\_\_ | Инициализация класса |
| Координаты вершины в интерфейсе | on\_left\_click | Создание вершины в графе |
| Координаты вершины в интерфейсе | on\_right\_click | Создание ребра между вершинами в графе |
| Координаты вершины в интерфейсе | get\_clicked\_vertex | Проверка нажатия на вершину графа |
| — | clear\_graph | Очистка интерфейса графа |

Описание функций класса *«Interface»* представлено в таблице 5.4.

*Таблица 5.4. Описание функций класса «Interface»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| — | \_\_init\_\_ | Инициализация класса |
| — | create\_interface | Создание элементов в интерфейсе |
| Событие | prevent\_typing | Запрет на редактирование  элемента интерфейса |
| — | clear\_output | Очистка интерфейса |
| — | populate\_edge\_table | Вывод таблицы с вершинами и ребрами графа |
| Вершина 1, вершина 2, вес ребра | update\_weight | Обновление веса между ребрами |
| — | threading\_run | Запуск функции *«start\_process»* в отдельном потоке |
| — | start\_process | Запуск решения поиска задачи о коммивояжере |

## **5.3. Описание переменных**

*Таблица 5.5. Описание переменных функции «ant\_algo»*

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| int | num\_nodes | Количество вершин в графе |
| Optional[List[int]] | traversal | Путь, найденный алгоритмом |
| float | length | Длина найденного пути |
| Dict[Tuple[int, int], float] | pheromone | Словарь, содержащий уровни феромона на рёбрах графа |
| List[List[int]] | elite\_traversals | Список элитных муравьев, каждый из которых нашел кратчайший путь |
| List[float] | elite\_lengths | Список длин кратчайших путей, найденных элитными муравьями |
| int | ant | Индекс муравья в текущей итерации |
| int | current\_node | Текущая вершина, которую обрабатывает муравей |
| List[int] | visited\_nodes | Список посещенных вершин текущим муравьем |
| float | path\_length | Длина текущего пути муравья |
| Set[int] | unvisited\_nodes | Множество непосещенных вершин |
| List[Tuple[float, int]] | probabilities | Список вероятностей перехода к соседним вершинам |
| float | total\_probability | Суммарная вероятность перехода к соседним вершинам |
| int | neighbor | Соседняя вершина, к которой муравей рассматривает возможный переход |
| float | pheromone\_level | Уровень феромона на ребре между текущей вершиной и соседней вершиной |
| float | edge\_length | Длина ребра между текущей вершиной и соседней вершиной |
| float | chosen\_probability | Случайно выбранная вероятность перехода к соседней вершине |
| float | cumulative\_probability | Кумулятивная вероятность перехода к соседним вершинам |
| float | probability | Вероятность перехода к соседней вершине, рассчитанная на основе феромона и длины ребра |
| Tuple[int, int] | edge | Ребро графа |
| List[int] | self.traversal | Список вершин, представляющий найденный маршрут |
| float | self.length | Длина найденного маршрута |
| str | result | Строка с результатом выполнения алгоритма (длина пути и путь) |

Полный код программы представлен в Приложении А.

# **Рекомендации пользователю**

Верхнее поле устанавливает редактор для графа с возможностью задания графа путем добавления вершин и ребер. Левой кнопкой мыши создается вершина в графе, а правая кнопка отвечает за создание ребра между двумя вершинам. Поле *«Вес»* изменяет вес ребра на заданное пользователем. Поля *«Коэфф. значимости феромона»*, *«Коэфф. значимости длины»*, *«Кол-во доп. феромона»*, *«Интенсивность испарения»* задают коэффициенты значимости длины, длины, также количества дополнительного феромона, интенсивность испарения.

Кнопка *«Рассчитать»* отвечает за старт поиска решения задачи о коммивояжере муравьином алгоритмом. Ответ, а также итоговый обход графа и количество феромона на ребрах выводится в интерфейс.

Кнопка *«Очистить»* отвечает за очистку интерфейса от всех пользовательских данных.

Для завершения работы нажмите на крестик в левом верхнем углу.

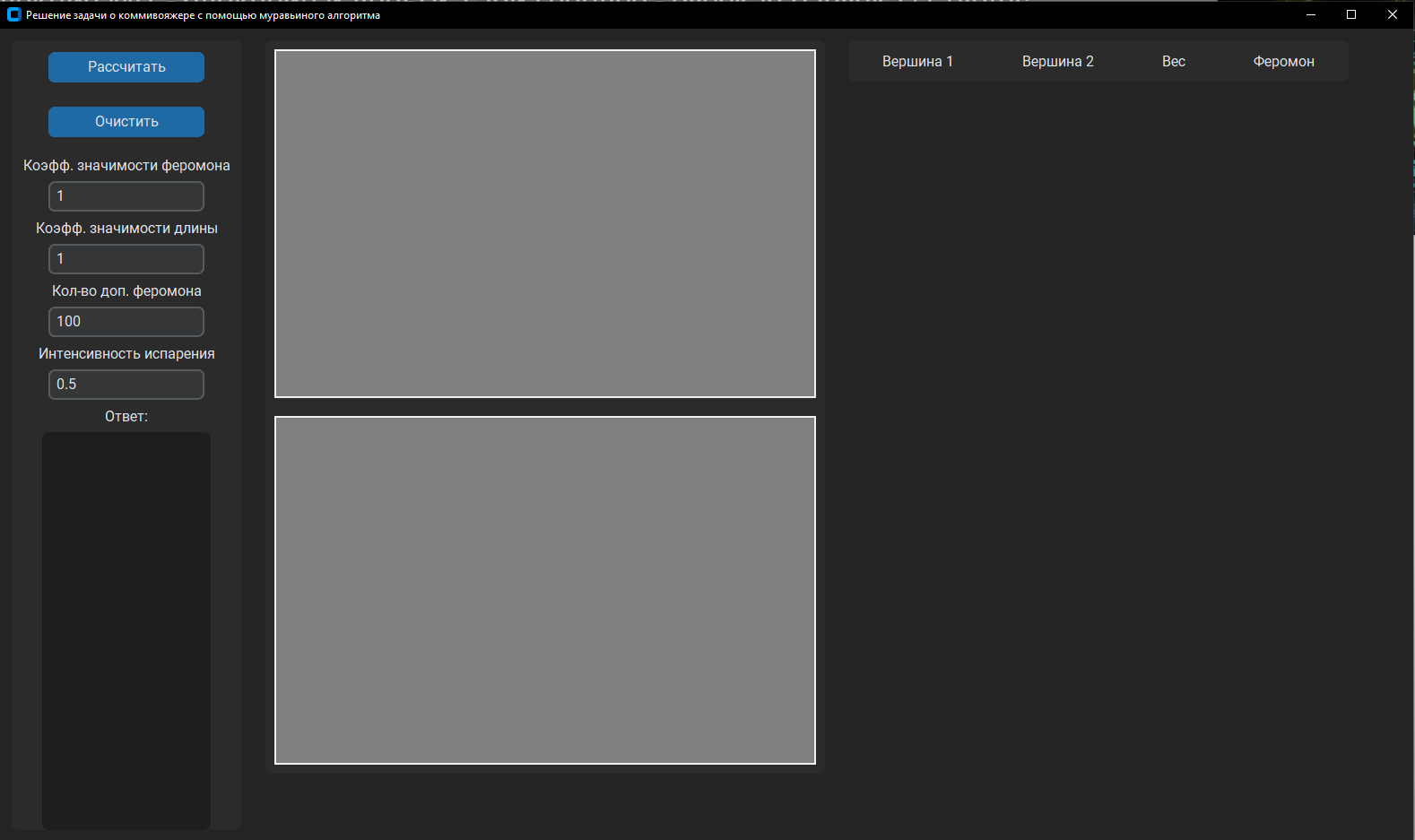
# **Рекомендации программисту**

Для запуска программы необходим Python версии не ниже 3.10.6. Предварительно необходимо установить библиотеки: customtkinter версии не ниже 5.2.0, networkx версии не ниже 3.0.

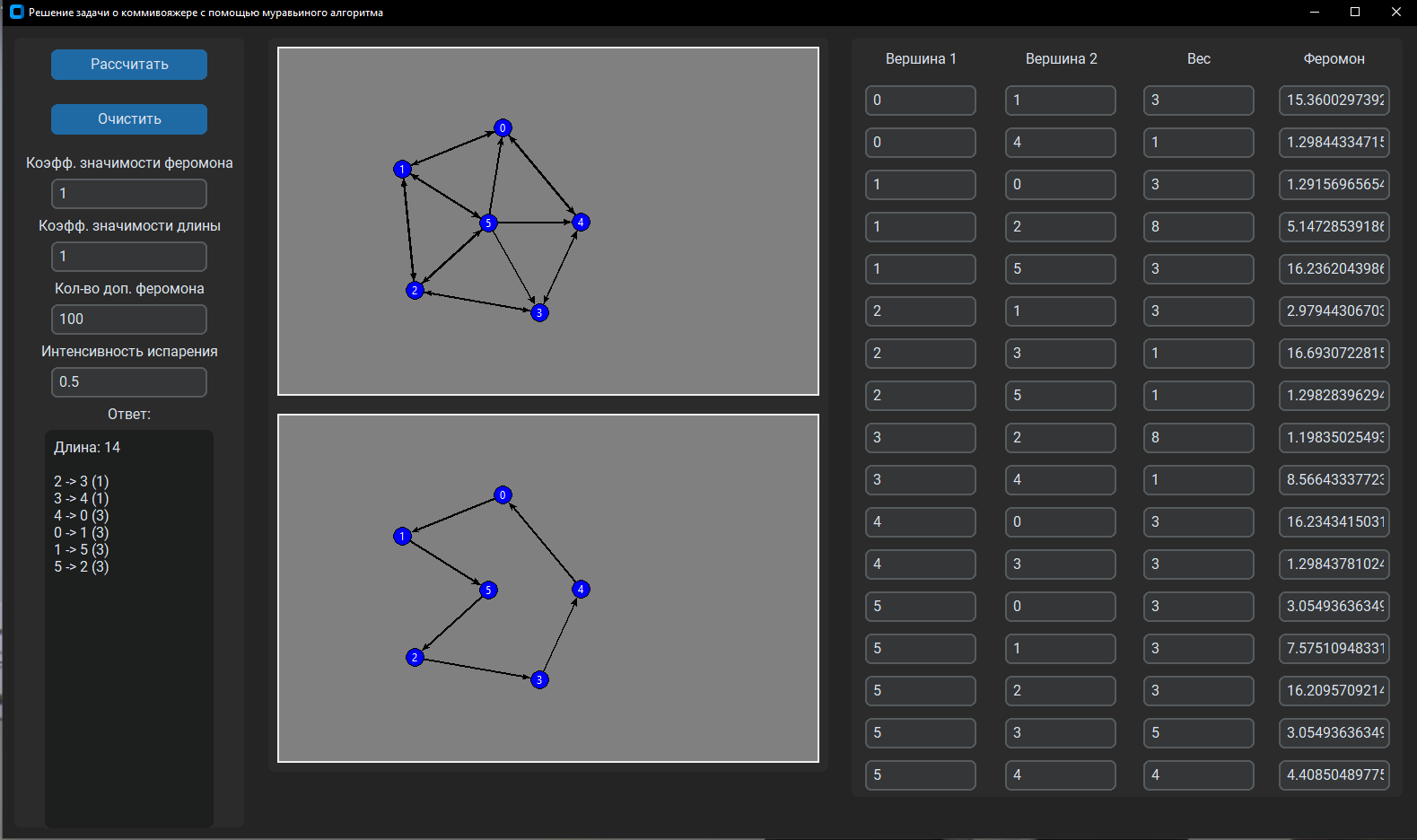
Минимальное необходимое место на диске: 0.5 МБ. Минимальное необходимое количество оперативной памяти: 100 МБ.

# **Контрольный пример**

В данном разделе представлен контрольный пример, демонстрирующий способность программы находить кратчайший гамильтонов цикл в графе.



*Рисунок 8.1. Интерфейс программы*



*Рисунок 8.2. Вывод решения задачи*

Муравьиный алгоритм обеспечивает более высокую точность по сравнению с методом ближайшего соседа, при этом сохраняя приемлемое время выполнения. В то время как метод ближайшего соседа может быть быстрее, он склонен к нахождению локально оптимальных решений. Однако использование муравьиного алгоритма с элитными муравьями может существенно увеличить время выполнения, но при этом обеспечивает дополнительный прирост в точности решения. Таким образом, муравьиный алгоритм является предпочтительным выбором при балансе между точностью результата и временем выполнения. Результаты тестирований представлены в таблице 8.3.

*Таблица 8.3. Сравнение методов решения задачи коммивояжера.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер графа | Метод ближайшего соседа | Метод ближайшего соседа с оптимизацией | Алгоритм имитации отжига | Алгоритм имитации отжига с оптимизацией | Муравьиный алгоритм | Муравьиный алгоритм с оптимизацией |
| 10 | 0.5с | 1с | 2с | 3с | 1.5с | 2с |
| 20 | 2с | 4с | 5с | 8с | 3.5с | 5с |
| 50 | 20с | 40с | 50с | 90с | 25с | 40с |

# **Заключение**

Исследование, направленное на решение задачи коммивояжера с использованием муравьиного алгоритма, а также сравнение его с алгоритмом ближайшего соседа и методом имитации отжига, позволило получить следующие выводы.

Гамильтонов цикл в неориентированном графе является ключевым аспектом задачи о коммивояжере, где требуется определить кратчайший путь, проходящий через все вершины графа ровно один раз и возвращающийся в исходную вершину. Задача коммивояжера представляет собой классическую задачу комбинаторной оптимизации, для которой существует множество методов решения, включая эвристические и точные алгоритмы.

Муравьиный алгоритм основан на модели поведения муравьев, которые общаются между собой с помощью феромонов, оставляемых на путях. Он обеспечивает более высокую точность решения по сравнению с методом ближайшего соседа, сохраняя приемлемое время выполнения. Метод ближайшего соседа, хотя и быстрее, склонен к нахождению локально оптимальных решений.

При использовании муравьиного алгоритма с элитными муравьями можно получить дополнительный прирост в точности решения, однако это приводит к значительному увеличению времени выполнения. Тем не менее, муравьиный алгоритм остается предпочтительным выбором при балансе между точностью и временем выполнения.

Исследование позволило провести сравнение времени работы различных алгоритмов на конкретном примере задачи коммивояжера. Результаты тестирования представлены в таблице, где видно, что муравьиный алгоритм превосходит другие методы по точности решения при разумном времени выполнения.

1. **Приложение А**

import random  
import threading  
import networkx as nx  
import customtkinter as ctk  
  
  
# Function Traveling Salesman algorithm  
class Traveling\_Salesman:  
 def \_\_init\_\_(self, graph\_editor):  
 self.graph\_editor = graph\_editor  
 self.edges = graph\_editor.edges  
 self.graph = graph\_editor.graph  
 self.length = float("inf")  
 self.traversal = {}  
  
 # Algorithm  
 def ant\_algo(self, coeff\_feromon, coeff\_length, count\_feromon, evaporation\_rate, elite\_ants\_count=1, elite\_pheromone\_factor=2):  
 num\_nodes = self.graph.number\_of\_nodes()  
 traversal = None  
 length = float("inf")  
 pheromone = {(edge[0], edge[1]): 1 for edge in self.graph.edges()}  
  
 for \_ in range(count\_feromon \* 5):  
 elite\_traversals = []  
 elite\_lengths = []  
  
 for ant in range(num\_nodes):  
 current\_node = random.choice(list(self.graph.nodes()))  
 visited\_nodes = [current\_node]  
 path\_length = 0  
  
 while len(visited\_nodes) < num\_nodes:  
 unvisited\_nodes = set(self.graph.nodes()) - set(visited\_nodes)  
 probabilities = []  
 total\_probability = 0  
  
 for neighbor in unvisited\_nodes:  
 if (current\_node, neighbor) in self.graph.edges():  
 pheromone\_level = pheromone.get((current\_node, neighbor), 0.0)  
 edge\_length = self.graph[current\_node][neighbor].get("weight", 1.0)  
 probabilities.append(((pheromone\_level \*\* coeff\_feromon) \* ((1.0 / edge\_length) \*\* coeff\_length), neighbor))  
 total\_probability += probabilities[-1][0]  
  
 if not probabilities:  
 break  
  
 total\_probability = total\_probability.real  
 chosen\_probability = random.uniform(0, total\_probability)  
 cumulative\_probability = 0  
  
 for probability, neighbor in probabilities:  
 cumulative\_probability += probability.real  
 if cumulative\_probability >= chosen\_probability:  
 break  
  
 path\_length += self.graph[current\_node][neighbor].get("weight", 1.0)  
 visited\_nodes.append(neighbor)  
 current\_node = neighbor  
  
 if len(visited\_nodes) < num\_nodes:  
 # If the path is incomplete, skip this ant  
 continue  
  
 if path\_length < length:  
 length = path\_length  
 traversal = visited\_nodes  
  
 if ant < elite\_ants\_count:  
 elite\_traversals.append(visited\_nodes)  
 elite\_lengths.append(path\_length)  
  
 for i in range(num\_nodes - 1):  
 edge = (visited\_nodes[i], visited\_nodes[i + 1])  
 if path\_length != 0:  
 pheromone[edge] = (1 - evaporation\_rate) \* pheromone.get(edge, 0.0) + count\_feromon / path\_length  
 else:  
 pheromone[edge] = (1 - evaporation\_rate) \* pheromone.get(edge, 0.0)  
  
 for elite\_traversal, elite\_length in zip(elite\_traversals, elite\_lengths):  
 for i in range(num\_nodes - 1):  
 edge = (elite\_traversal[i], elite\_traversal[i + 1])  
 pheromone[edge] += elite\_pheromone\_factor \* count\_feromon / elite\_length  
  
 for edge in self.graph.edges():  
 pheromone[edge] \*= (1 - evaporation\_rate)  
  
 self.traversal = traversal  
 self.length = length  
  
 try:  
 self.length += self.graph[traversal[-1]][traversal[0]]["weight"]  
 self.traversal.append(self.traversal[0])  
 except KeyError:  
 raise TypeError("Key Error, cycle is not found!")  
  
 result = f"Длина: {self.length}\n\n"  
 for i in range(len(self.traversal) - 1):  
 result += f'{self.traversal[i]} -> {self.traversal[i + 1]} ({self.graph[self.traversal[i]][self.traversal[i + 1]]["weight"]})\n'  
 return result, pheromone  
  
 # Draw graph with his traversal  
 def view(self, canvas):  
 canvas.clear\_graph()  
 for i in range(len(self.traversal) - 1):  
 start\_vertex = self.traversal[i]  
 end\_vertex = self.traversal[i + 1]  
 start\_x, start\_y = self.graph\_editor.vertices[start\_vertex]  
 end\_x, end\_y = self.graph\_editor.vertices[end\_vertex]  
 length = ((end\_x - start\_x) \*\* 2 + (end\_y - start\_y) \*\* 2) \*\* 0.5  
 arrow\_offset = 10  
 sx = start\_x + (end\_x - start\_x) \* (arrow\_offset / length)  
 sy = start\_y + (end\_y - start\_y) \* (arrow\_offset / length)  
 ex = end\_x - (end\_x - start\_x) \* (arrow\_offset / length)  
 ey = end\_y - (end\_y - start\_y) \* (arrow\_offset / length)  
 canvas.create\_line(sx, sy, ex, ey, arrow="last", width=2)  
  
 for vertex, (x, y) in enumerate(self.graph\_editor.vertices):  
 canvas.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="blue", tags="vertex")  
 canvas.create\_text(x, y, text=str(vertex), fill="white", tags="vertex\_text")  
  
  
# Class for editing a graph on a canvas  
class GraphEditor(ctk.CTkCanvas):  
 def \_\_init\_\_(self, master, interface, \*\*kwargs):  
 super().\_\_init\_\_(master, \*\*kwargs)  
 self.bind("<Button-1>", self.on\_left\_click)  
 self.bind("<Button-3>", self.on\_right\_click)  
 self.vertices = []  
 self.edges = []  
 self.selected\_vertex = None  
 self.graph = nx.DiGraph()  
 self.interface = interface  
  
 # Function to handle left-click events on the canvas  
 def on\_left\_click(self, event):  
 x, y = event.x, event.y  
 self.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="blue", tags="vertex")  
 self.create\_text(x, y, text=str(len(self.graph)), fill="white", tags="vertex\_text")  
 self.vertices.append((x, y))  
 self.graph.add\_node(len(self.graph))  
  
 # Function to handle right-click events on the canvas  
 def on\_right\_click(self, event):  
 x, y = event.x, event.y  
 vertex = self.get\_clicked\_vertex(x, y)  
 if vertex is not None:  
 if self.selected\_vertex is None:  
 self.selected\_vertex = vertex  
 else:  
 start\_x, start\_y = self.vertices[self.selected\_vertex]  
 end\_x, end\_y = self.vertices[vertex]  
 length = ((end\_x - start\_x) \*\* 2 + (end\_y - start\_y) \*\* 2) \*\* 0.5  
 if length == 0:  
 return  
 arrow\_offset = 10  
 sx = start\_x + (end\_x - start\_x) \* (arrow\_offset / length)  
 sy = start\_y + (end\_y - start\_y) \* (arrow\_offset / length)  
 ex = end\_x - (end\_x - start\_x) \* (arrow\_offset / length)  
 ey = end\_y - (end\_y - start\_y) \* (arrow\_offset / length)  
 self.create\_line(sx, sy, ex, ey, arrow=ctk.LAST, width=2)  
 self.edges.append((self.selected\_vertex, vertex))  
 self.graph.add\_edge(self.selected\_vertex, vertex, weight=int(length))  
 self.selected\_vertex = None  
 self.interface.populate\_edge\_table()  
  
 # Function to get the index of a clicked vertex on the canvas  
 def get\_clicked\_vertex(self, x, y):  
 for i, (vx, vy) in enumerate(self.vertices):  
 if (x - vx) \*\* 2 + (y - vy) \*\* 2 <= 100:  
 return i  
 return None  
  
 # Function to clear the graph on the canvas  
 def clear\_graph(self):  
 self.vertices = []  
 self.edges = []  
 self.graph.clear()  
 self.delete("all")  
  
  
# Class for the graphical user interface  
class Interface(ctk.CTk):  
 def \_\_init\_\_(self):  
 ctk.CTk.\_\_init\_\_(self)  
 ctk.set\_appearance\_mode("dark")  
 ctk.set\_default\_color\_theme("blue")  
 self.title("Решение задачи о коммивояжере с помощью муравьиного алгоритма")  
 self.geometry("1250x670")  
 self.frame1 = None  
 self.frame2 = None  
 self.frame3 = None  
 self.graph\_view = None  
 self.clear\_button = None  
 self.answer\_label = None  
 self.graph\_editor = None  
 self.output\_text = None  
 self.process\_button = None  
 self.evaporation\_rate = None  
 self.evaporation\_rate\_text = None  
 self.count\_feromon = None  
 self.count\_feromon\_text = None  
 self.coeff\_length = None  
 self.coeff\_length\_text = None  
 self.coeff\_feromon = None  
 self.coeff\_feromon\_text = None  
 self.edge\_table = {}  
 self.create\_interface()  
  
 # Function to create the graphical user interface  
 def create\_interface(self):  
 self.frame1 = ctk.CTkFrame(self)  
 self.frame1.grid(row=0, column=0, padx=10, pady=10, sticky="n")  
  
 self.process\_button = ctk.CTkButton(self.frame1, text="Рассчитать", command=self.threading\_run)  
 self.process\_button.pack(side="top", padx=10, pady=10)  
  
 self.clear\_button = ctk.CTkButton(self.frame1, text="Очистить", command=self.clear\_output)  
 self.clear\_button.pack(side="top", padx=10, pady=10)  
  
 self.coeff\_feromon\_text = ctk.CTkLabel(self.frame1, text="Коэфф. значимости феромона")  
 self.coeff\_feromon\_text.pack(side="top", padx=10)  
  
 self.coeff\_feromon = ctk.CTkEntry(self.frame1, width=140)  
 self.coeff\_feromon.pack(side="top", padx=10)  
 self.coeff\_feromon.insert(0, "1")  
  
 self.coeff\_length\_text = ctk.CTkLabel(self.frame1, text="Коэфф. значимости длины")  
 self.coeff\_length\_text.pack(side="top", padx=10)  
  
 self.coeff\_length = ctk.CTkEntry(self.frame1, width=140)  
 self.coeff\_length.pack(side="top", padx=10)  
 self.coeff\_length.insert(0, "1")  
  
 self.count\_feromon\_text = ctk.CTkLabel(self.frame1, text="Кол-во доп. феромона")  
 self.count\_feromon\_text.pack(side="top", padx=10)  
  
 self.count\_feromon = ctk.CTkEntry(self.frame1, width=140)  
 self.count\_feromon.pack(side="top", padx=10)  
 self.count\_feromon.insert(0, "100")  
  
 self.evaporation\_rate\_text = ctk.CTkLabel(self.frame1, text="Интенсивность испарения")  
 self.evaporation\_rate\_text.pack(side="top", padx=10)  
  
 self.evaporation\_rate = ctk.CTkEntry(self.frame1, width=140)  
 self.evaporation\_rate.pack(side="top", padx=10)  
 self.evaporation\_rate.insert(0, "0.5")  
  
 self.answer\_label = ctk.CTkLabel(self.frame1, text="Ответ:")  
 self.answer\_label.pack(side="top", padx=10, fill=ctk.BOTH)  
  
 self.output\_text = ctk.CTkTextbox(self.frame1, height=355, width=150)  
 self.output\_text.pack(side="top", padx=10)  
 self.output\_text.bind("<KeyPress>", self.prevent\_typing)  
  
 self.frame2 = ctk.CTkFrame(self)  
 self.frame2.grid(row=0, column=1, padx=10, pady=10, sticky="n")  
  
 self.graph\_editor = GraphEditor(self.frame2, width=600, height=385, bg="grey", interface=self)  
 self.graph\_editor.pack(side="top", padx=10, pady=10)  
  
 self.graph\_view = GraphEditor(self.frame2, width=600, height=385, bg="grey", interface=self)  
 self.graph\_view.pack(side="top", padx=10, pady=10)  
 self.graph\_view.bind("<Button-1>", self.prevent\_typing)  
  
 self.frame3 = ctk.CTkFrame(self)  
 self.frame3.grid(row=0, column=2, padx=10, pady=10, sticky="n")  
  
 for col, header in enumerate(["Вершина 1", "Вершина 2", "Вес", "Феромон"]):  
 label = ctk.CTkLabel(self.frame3, text=header)  
 label.grid(row=0, column=col, padx=30, pady=5)  
  
 self.populate\_edge\_table()  
  
 # Function to prevent typing in the output textbox  
 def prevent\_typing(self, event):  
 return "break"  
  
 # Function to clear the output textbox and the graph on the canvas  
 def clear\_output(self):  
 self.graph\_editor.clear\_graph()  
 self.graph\_view.clear\_graph()  
 self.output\_text.delete("1.0", ctk.END)  
 self.populate\_edge\_table()  
  
 # Function to update weights in graph  
 def update\_weight(self, vertex1, vertex2, entry\_weight):  
 new\_weight = entry\_weight.get()  
 self.graph\_editor.graph[vertex1][vertex2]["weight"] = int(new\_weight)  
  
 # Function to run a process in a separate thread  
 def threading\_run(self):  
 t = threading.Thread(target=self.run\_process)  
 t.start()  
  
 # Function to run a process in the main thread  
 def run\_process(self):  
 self.output\_text.delete("1.0", ctk.END)  
 self.graph\_view.clear\_graph()  
 self.after(10, self.start\_process)  
  
 # Function to start a process  
 def start\_process(self):  
 salesman = Traveling\_Salesman(self.graph\_editor)  
 result, pheromone = salesman.ant\_algo(float(self.coeff\_feromon.get()), float(self.coeff\_length.get()),  
 int(self.count\_feromon.get()), float(self.evaporation\_rate.get()))  
 self.output\_text.insert(ctk.END, result)  
 self.populate\_edge\_table(pheromone)  
 salesman.view(self.graph\_view)  
  
 # Function to populate the edge table with data from the graph  
 def populate\_edge\_table(self, pheromone=None):  
 if self.edge\_table:  
 for widgets in self.edge\_table.values():  
 for widget in widgets:  
 widget.destroy()  
 self.edge\_table = {}  
  
 sorted\_edges = sorted(self.graph\_editor.graph.edges(data="weight"))  
  
 for row, (vertex1, vertex2, weight) in enumerate(sorted\_edges, start=1):  
 entry\_vertex1 = ctk.CTkEntry(self.frame3, width=100)  
 entry\_vertex1.insert(ctk.END, vertex1)  
 entry\_vertex1.grid(row=row, column=0, padx=10, pady=5)  
 entry\_vertex1.bind("<KeyPress>", self.prevent\_typing)  
  
 entry\_vertex2 = ctk.CTkEntry(self.frame3, width=100)  
 entry\_vertex2.insert(ctk.END, vertex2)  
 entry\_vertex2.grid(row=row, column=1, padx=10, pady=5)  
 entry\_vertex2.bind("<KeyPress>", self.prevent\_typing)  
  
 entry\_weight = ctk.CTkEntry(self.frame3, width=100)  
 entry\_weight.insert(ctk.END, weight)  
 entry\_weight.grid(row=row, column=2, padx=10, pady=5)  
 entry\_weight.bind("<FocusOut>",  
 lambda event, vertex1=vertex1, vertex2=vertex2, entry\_weight=entry\_weight: self.update\_weight(vertex1, vertex2,  
 entry\_weight))  
  
 self.edge\_table[row] = [entry\_vertex1, entry\_vertex2, entry\_weight]  
  
 if pheromone:  
 for row, (vertex1, vertex2, weight) in enumerate(sorted\_edges, start=1):  
 entry\_pheromone = ctk.CTkEntry(self.frame3, width=100)  
 entry\_pheromone.insert(ctk.END, pheromone.get((vertex1, vertex2), 1))  
 entry\_pheromone.grid(row=row, column=3, padx=10, pady=5)  
 entry\_pheromone.bind("<KeyPress>", self.prevent\_typing)  
  
 self.edge\_table[row].append(entry\_pheromone)  
  
  
# Main block to run the GUI application  
if \_\_name\_\_ == "\_\_main\_\_":  
 gui = Interface()  
 gui.mainloop()

*Листинг программы.*