**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики - процессов управления**

**Программа бакалавриата**

**“Большие данные и распределенная цифровая платформа”**

**ОТЧЕТ**

**по лабораторной работе №3**

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

**на тему «Муравьиный алгоритм»**

**Студент гр. 23Б15-пу**

**Кубякин Н.А.**

**Преподаватель**

**Дик А.Г.**

**Санкт-Петербург**

**2025 г.**

***Оглавление***

[1. Цель работы 3](#_Toc178797423)

[2. Описание задачи (формализация задачи) 3](#_Toc178797424)

[3. Теоретическая часть 3](#_Toc178797431)

[4. Основные шаги программы 5](#_Toc178797432)

[5. Блок-схема программы 6](#_Toc178797433)

[6. Описание программы 7](#_Toc178797442)

[7. Рекомендации пользователя 8](#_Toc178797443)

[8. Рекомендации программиста 9](#_Toc178797444)

[9. Исходный код программы 9](#_Toc178797445)

[10. Контрольный пример 10](#_Toc178797447)

[11. Вывод 15](#_Toc178797448)

[12. Источники 15](#_Toc178797448)

[13. Листинг 16](#_Toc178797450)

# Цель работы

# Разработать программное обеспечение, которое с помощью муравьиного алгоритма находит кратчайший гамильтонов цикл в направленном взвешенном графе.

# Описание задачи (формализация задачи)

* Необходимо формализовать задачу о коммивояжере с помощью муравьиного алгоритма
* Подготовить контрольный пример, используя взвешенный орграф
* Найти кратчайший гамильтонов цикл
* Сравнить алгоритм с модификацией

# Теоретическая часть

# Муравьиный алгоритм (Ant Colony Optimization, ACO) — эвристический метод оптимизации, основанный на поведении реальных муравьёв при поиске кратчайшего пути от муравейника к пище. Алгоритм основан на том, что муравьи оставляют феромоны на маршруте, усиливая вероятность его выбора другими особями. Со временем наилучшие маршруты формируют оптимальное решение.

Параметры алгоритма и их роль:

1. alpha – влияние феромонов

Если 0, муравьи игнорируют феромоны и выбирают кратчайший путь по расстоянию, если значение высокое то алгоритм быстро сходится к одному решению.

1. beta – влияние расстояния

Коэффициент эвристики, обычно обратное расстояние (1/d)

1. rho – коэффициент испарения феромонов

0 < rho < 1, определяет долю испаряющихся феромонов после каждой итерации.

1. Q – количество феромонов

Коэффициент влияющий на вклад муравья в количество феромонов на каждой итерации.

1. Вероятность того, что муравей в вершине 𝑖 выберет переход в вершину 𝑗:

где:

– — количество феромона на ребре

– — эвристическая привлекательность (обратное расстояние)

– — коэффициенты влияния феромона и расстояния

1. Модификация алгоритма (шаблоны)

В модификации алгоритма запоминаются часто используемые рёбра (шаблоны), которые получают больше феромонов. Это позволяет запоминать рёбра, по которым ходят больше муравьёв.

1. После прохождения всех маршрутов формируется словарь, сохраняющий кол-во вхождений каждого ребра в маршруты.
2. На этапе обновления феромона учитывается это количество:

где:

– p — коэффициент усиления шаблонов

– — количество повторений ребра в маршрутах

– — базовое количество феромона

# Основные шаги программы

1. Инициализация графа с равным числом феромонов на всех рёбрах.
2. На каждой итерации муравьи строят маршруты, выбирая рёбра на основе вероятности перехода.
3. После завершения маршрута муравьи оставляют феромоны на пройденных рёбрах — чем короче путь, тем больше. Если новый путь короче — он обновляется.
4. Феромоны испаряются на неиспользованных вершинах.
5. Температура снижается по заданной схеме.
6. Процесс повторяется, пока алгоритм не сойдётся или не будет выполнено максимальное число итераций.

# Блок-схема программы

# На рисунке представлена блок-схема алгоритма.

# 

# Рис. 5 Блок-схема алгоритма.

# Описание программы

# Алгоритм реализован на языке Python 3.10 с использованием библиотек tkinter [[1]](#tk), networkx [[2]](#nx), math [[3]](#mt), pandas [[4]](#pd), random [[5]](#rn). Логика и интерфейс программы реализованы в следующих классах и функциях:

# Алгоритм реализован через класс AntColonyOptimizer, содержащий следующие функции:

Таблица 6.1 Описание класса расчёта

|  |  |
| --- | --- |
| Имя функции | Назначение |
| update\_templates | Обновляет шаблоны. |
| calculate\_path\_cost | Рассчитывает стоимость пути. |
| select\_next\_node | Выбирает следующую вершину для муравья. |
| update\_pheromone | Обновляет феромоны. |
| apply\_template\_modification | Применяет модификацию шаблонов. |
| run | Запускает алгоритм. |

Интерфейс реализован через класс TSPInteractiveGUI, cодержащий следующие функции:

Таблица 6.2 Описание класса интерфейса

|  |  |
| --- | --- |
| Имя функции | Назначение |
| create\_graph\_canvas | Создает поле для интерактивного рисования графа. |
| create\_distance\_matrix\_frame | Создает фрейм для отображения матрицы расстояний. |
| update\_distance\_matrix | Обновляет отображение матрицы расстояний. |
| create\_control\_frame | Создаёт фрейм с кнопками взаимодействия с графом. |
| create\_result\_frame | Создаёт фрейм с отображением результата программы. |
| on\_canvas\_click | Функция создающая ребро между двумя вершинами. |
| add\_vertex\_event | Отображает новую вершину на графе. |
| get\_midpoint\_coordinates | Находит координаты середины ребра для отображения направления. |
| draw\_arrow | Отображает стрелочку, показывающую направление ребра. |
| draw\_edge | Отображает линию, соответствующую ребру. |
| find\_closest\_vertex | Находит ближайшую вершину. |
| ask\_for\_weight | Принимает вес ребра. |
| set\_start\_vertex | Устанавливает начальную вершину. |
| delete\_last\_vertex | Удаляет последнюю установленную вершину. |
| draw\_graph | Отображает граф в соответствующем поле. |
| clear\_graph | Очищает поле с графом. |
| get\_aco\_parameters | Принимает параметры алгоритма из интерфейса. |
| run\_ant\_colony | Связывает алгоритм вычисления с интерфейсом. |
| draw\_graph\_with\_path | Отображает кратчайший путь на графе. |
| delete\_last\_edge | Удаляет последнее установленное ребро. |
| delete\_last\_path | Обновляет матрицу после удаления ребра. |
| load\_matrix\_from\_table | Загружает матрицу расстояний из таблицы CSV. |
| load\_graph\_from\_matrix | Загружает граф из матрицы расстояний. |
| add\_vertex\_from\_matrix | Добавляет вершину из матрицы. |
| update\_edge\_weight | Обновляет вес выбранного ребра. |

# Рекомендации пользователя

1. Запустить main.py для запуска графического интерфейса программы.
2. Нарисовать граф, используя ЛКМ или загрузить его из файла csv. Для создания новой вершины нажать в любую точку поля слева. Для создания ребра нажать на две вершины в соответствующем порядке и задать длину в окне. При надобности использовать кнопки удаления величин, рёбер, изменения длины пути.
3. Проставить параметры алгоритма и выбрать обычный алгоритм или модификация.
4. Нажать кнопку “Найти кратчайший путь”. На графе зелёным цветом отобразится искомый путь, а снизу можно будет увидеть длину пути.

# Рекомендации программиста

Для запуска программы необходима 64-битная операционная система Windows, Linux или macOS. Для работы с кодом необходима среда разработки, совместимая с python 3.1 и библиотеки tkinter [[1]](#tk), networkx [[2]](#nx), math [[3]](#mt), pandas [[4]](#pd), random [[5]](#rn).

Для запуска программы необходимо установить Python версии не ниже 3.10, а так же библиотеки tkinter [[1]](#tk), networkx [[2]](#nx), math [[3]](#mt), pandas [[4]](#pd), random [[5]](#rn).

# Исходный код программы

# Исходный код программы и необходимые текстовые файлы доступны по ссылке: <https://github.com/NikiTaku1/spbu_alg_2/tree/main/lab3>

# Контрольный пример

В данном разделе представлен пример, демонстрирующий работу программы.

1) Запуск main.py

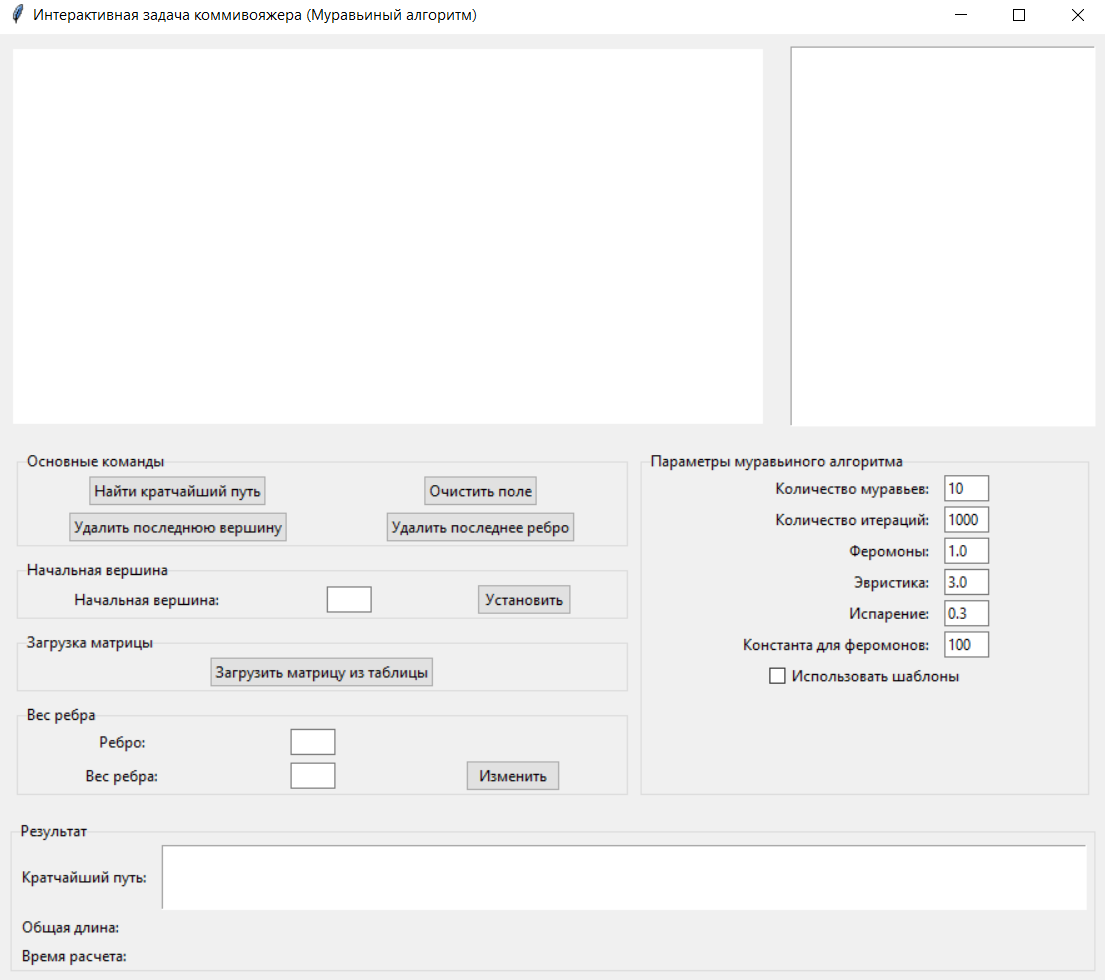


Рис 10.1 Графический интерфейс программы.

2) Задание графа

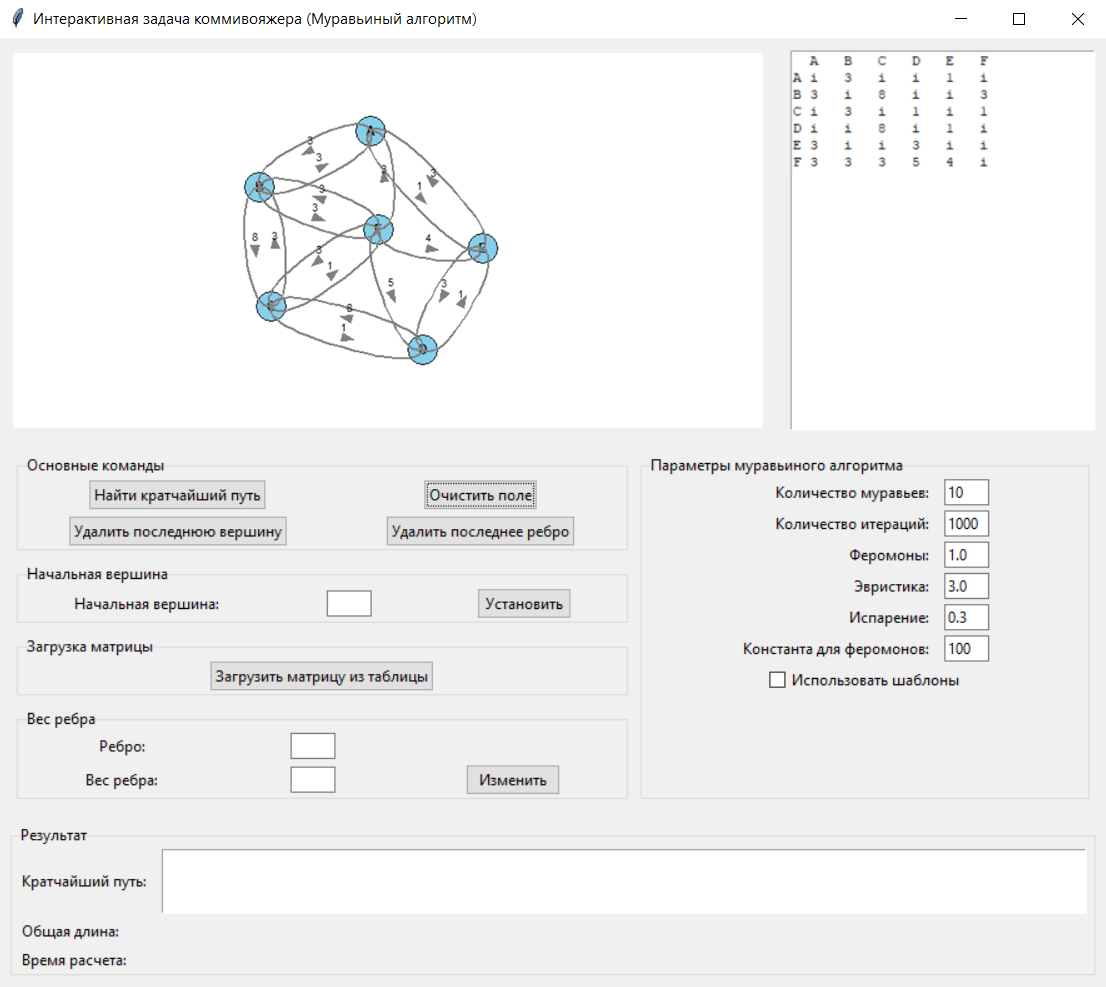


Рис 10.2 Графический интерфейс с заданным графом.

3) Результат алгоритма без модификации

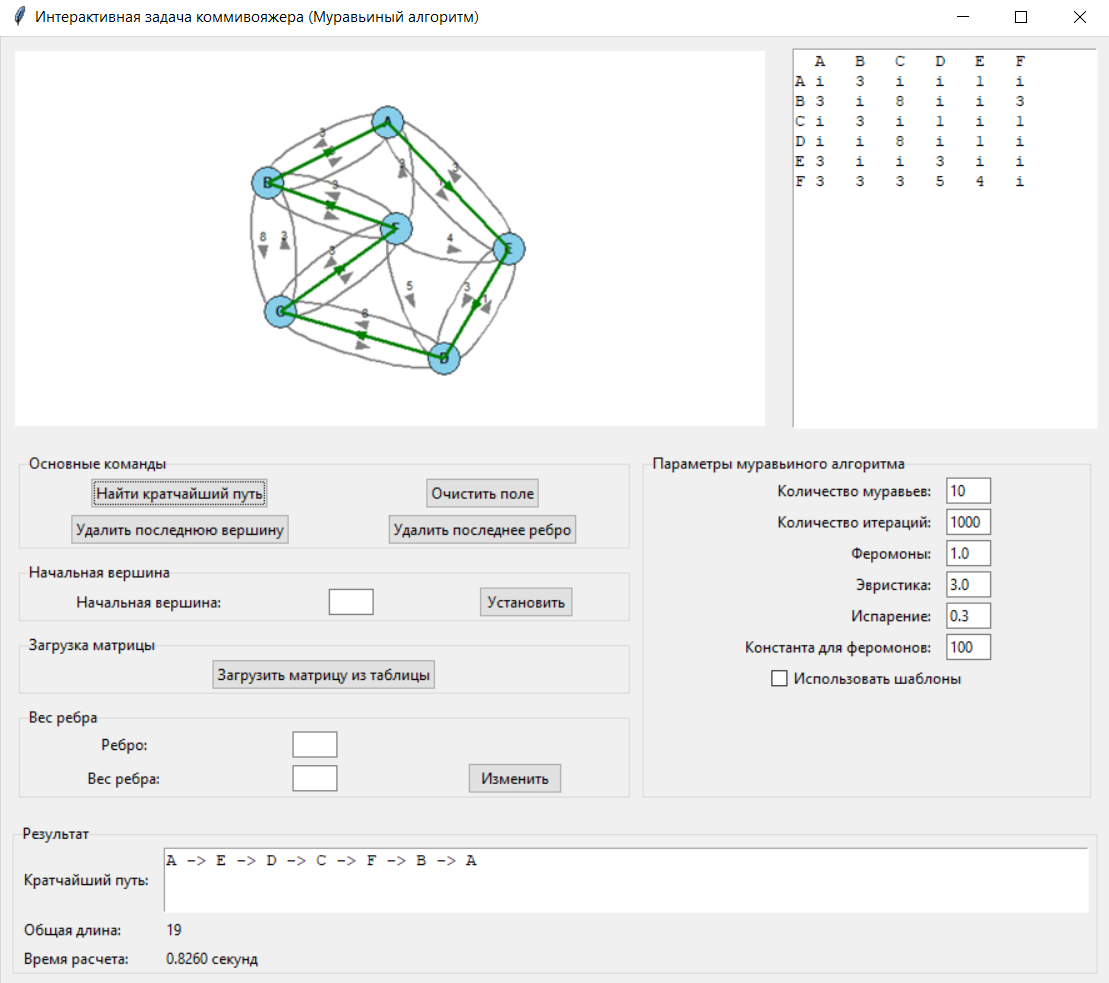


Рис 10.3 Графический интерфейс после выполнения программы без модификации.

4) Результат алгоритма с модификацией

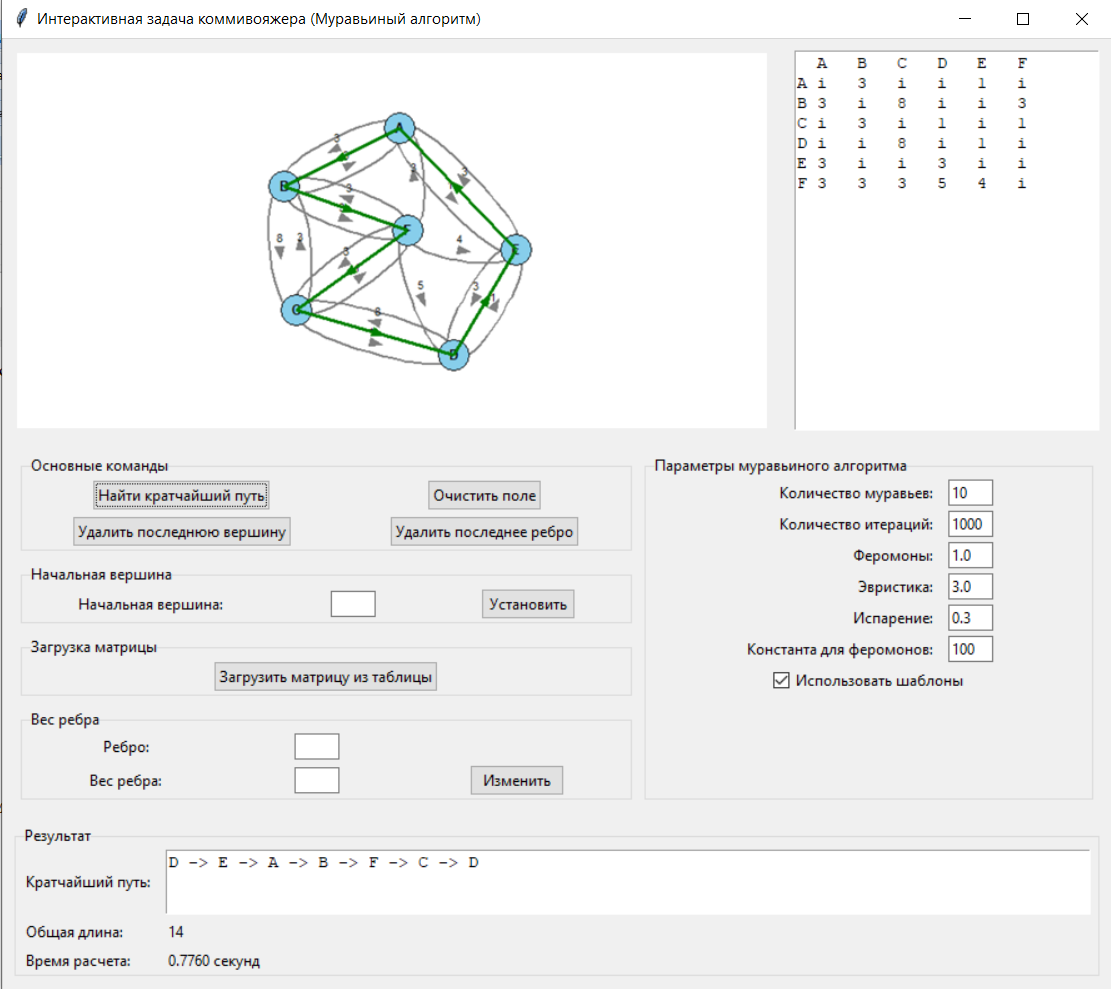


Рис 10.4 Графический интерфейс после выполнения программы c модификацией.

5) Сравнение эффективности алгоритмов метода ближайшего соседа, алгоритма отжига и муравьиного алгоритма с модификациями и без (графы для анализа сгенерированы случайно, параметры модификации оптимизированы)

Таблица 10.5.1 Сравнение алгоритмов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Количество вершин | Время работы без модификации (с) | Время работы с модификацией (с) | Длина пути без модификации | Длина пути с модификацией |
| Метод ближайшего соседа | 6 | 0.0001 | 0.0001 | 97 | 55 |
| 10 | 0.0001 | 0.0013 | 103 | 51 |
| 15 | 0.0011 | 0.0044 | 152 | 120 |
| 30 | 0.0032 | 0.0244 | 234 | 161 |
| 50 | 0.0041 | 0.1042 | 251 | 190 |
| 100 | 0.0223 | 0.9734 | 326 | 233 |
| Алгоритм отжига | 6 | 0.0078 | 0.0088 | 83 | 73 |
| 10 | 0.0118 | 0.0138 | 85 | 74 |
| 15 | 0.0181 | 0.0209 | 165 | 147 |
| 30 | 0.0521 | 0.2363 | 293 | 198 |
| 50 | 0.1212 | 0.3778 | 432 | 299 |
| 100 | 0.4123 | 1.0017 | 912 | 561 |
| Муравьиный алгоритм | 6 | 0.0804 | 0.0840 | 76 | 73 |
| 10 | 0.1660 | 0.1690 | 79 | 74 |
| 15 | 0.3049 | 0.3091 | 107 | 102 |
| 30 | 0.9132 | 0.9141 | 150 | 141 |
| 50 | 3.8592 | 4.2195 | 156 | 151 |
| 100 | 14.5024 | 17.5896 | 210 | 202 |

Таким образом на малом количестве вершин лучший результат показывает метод ближайшего соседа с модификацией обхода всех вершин, тогда как на большом количестве вершин вперёд выходит муравьиный алгоритм с модификацией шаблонов. По скорости метод ближайшего соседа выполняется быстрее. Отжиг немного медленнее, а муравьиный алгоритм выполняется заметно дольше.

# Вывод

В процессе исследования особенностей алгоритма муравьиного алгоритма для решения задачи коммивояжера на ориентированном графе был разработан метод поиска кратчайшего гамильтонова цикла. Написана соответствующая программа, способная находить гамильтонов цикл с использованием муравьиного алгоритма и определять длину маршрута, охватывающего все вершины. Кроме того, была предложена модификация шаблонов. Так же проведён сравнительный анализ алгоритмов метода ближайшего соседа, отжига и муравьиного с целью сравнить эффективность и скорость выполнения.

# Источники

# tkinter — tkinter documentation // Documentation URL: <https://docs.python.org/3/library/tkinter.html> (дата обращения: 25.04.2025).

# networkx — networkx documentation // Documentation URL: [https://networkx.org/documentation/latest/](https://networkx.org/documentation/latest/%20) (дата обращения: 25.04.2025).

# math — math documentation // Documentation URL: <https://docs.python.org/3/library/math.html> (дата обращения: 25.04.2025).

# pandas — pandas documentation // Documentation URL: [https://pandas.pydata.org/docs/](https://pandas.pydata.org/docs/%20) (дата обращения: 25.04.2025).

# random — random documentation // Documentation URL: <https://docs.python.org/3/library/random.html> (дата обращения: 25.04.2025).

# Листинг

# 1) main.py

import tkinter as tk

from tsp\_gui import TSPInteractiveGUI

if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    gui = TSPInteractiveGUI(root)

    root.mainloop()

# 2) tsp\_solver.py

import random

import math

import numpy as np

from collections import defaultdict

def nearest\_neighbor(graph, start\_node):

    """Finds the nearest neighbor path."""

    graph\_copy = graph.copy()

    if graph\_copy is None or not graph\_copy.nodes() or start\_node is None or start\_node not in graph\_copy.nodes():

        return None, 0

    nodes = list(graph\_copy.nodes())

    path = [start\_node]

    unvisited = set(nodes)

    unvisited.remove(start\_node)

    total\_cost = 0

    while unvisited:

        current\_node = path[-1]

        nearest\_neighbor = None

        min\_distance = float('inf')

        for neighbor in unvisited:

            try:

                distance = graph\_copy[current\_node][neighbor]['weight']

            except KeyError:

                distance = float('inf')

            if distance < min\_distance:

                nearest\_neighbor = neighbor

                min\_distance = distance

        if nearest\_neighbor is None:

            return None, 0

        path.append(nearest\_neighbor)

        unvisited.remove(nearest\_neighbor)

        total\_cost += min\_distance

    try:

        total\_cost += graph\_copy[path[-1]][path[0]]['weight']

        path.append(path[0])

    except KeyError:

        return None, 0

    return path, total\_cost

class AntColonyOptimizer:

    def \_\_init\_\_(self, graph, n\_ants, n\_iterations,

                 alpha, beta, evaporation\_rate,

                 q, use\_templates, template\_update\_frequency=10):  # Добавим частоту обновления шаблонов

        self.graph = graph

        self.n\_ants = n\_ants

        self.n\_iterations = n\_iterations

        self.alpha = alpha  # влияние феромона

        self.beta = beta    # влияние эвристики (1/расстояние)

        self.evaporation\_rate = evaporation\_rate

        self.q = q  # константа для обновления феромона

        self.use\_templates = use\_templates

        self.template\_update\_frequency = template\_update\_frequency # Как часто обновлять шаблоны

        self.nodes = list(graph.nodes())

        self.n\_nodes = len(self.nodes)

        self.node\_indices = {node: i for i, node in enumerate(self.nodes)}

        # Инициализация матрицы феромонов

        self.pheromone = np.ones((self.n\_nodes, self.n\_nodes))

        # Инициализация матрицы расстояний

        self.distances = np.zeros((self.n\_nodes, self.n\_nodes))

        for i, u in enumerate(self.nodes):

            for j, v in enumerate(self.nodes):

                if u == v:

                    self.distances[i,j] = 0

                else:

                    try:

                        self.distances[i,j] = graph[u][v]['weight']

                    except KeyError:

                        self.distances[i,j] = float('inf')

        # Шаблоны для модификации (изначально пустой список)

        self.templates = [] if use\_templates else None

        self.edge\_counts = defaultdict(int) # Словарь для хранения количества вхождений ребер

        self.iteration\_count = 0

    def \_generate\_template\_from\_path(self, path):

        """Создает шаблон из данного пути."""

        # Можно добавить логику для фильтрации или упрощения шаблона,

        # чтобы избежать слишком специфичных шаблонов.

        return path[:]  # Возвращаем копию пути, чтобы избежать изменений исходного пути

    def \_update\_templates(self, ants\_paths):

        """Обновляет шаблоны на основе путей, пройденных муравьями."""

        if not self.use\_templates:

            return

        for path, cost in ants\_paths:

            if cost > 0: # Исключаем невалидные пути

                # Создаем шаблон из пути

                new\_template = self.\_generate\_template\_from\_path(path)

                # Добавляем шаблон, если он еще не существует

                if new\_template not in self.templates:

                    self.templates.append(new\_template)

                # Ограничиваем количество шаблонов (опционально)

                max\_templates = 10 # Максимальное количество шаблонов

                if len(self.templates) > max\_templates:

                    # Удаляем наименее полезный шаблон (здесь - случайный)

                    self.templates.pop(random.randint(0, len(self.templates) - 1))

    def \_calculate\_path\_cost(self, path):

        """Вычисляет стоимость пути."""

        cost = 0

        for i in range(len(path)-1):

            u = path[i]

            v = path[i+1]

            cost += self.distances[self.node\_indices[u], self.node\_indices[v]]

        return cost

    def \_select\_next\_node(self, current\_node, visited):

        """Выбирает следующую вершину для муравья."""

        current\_idx = self.node\_indices[current\_node]

        unvisited = [node for node in self.nodes if node not in visited]

        if not unvisited:

            return None

        # Вычисляем вероятности для всех непосещенных вершин

        probabilities = []

        total = 0.0

        for node in unvisited:

            node\_idx = self.node\_indices[node]

            if self.distances[current\_idx, node\_idx] == float('inf'):

                probabilities.append(0.0)

                continue

            pheromone = self.pheromone[current\_idx, node\_idx] \*\* self.alpha

            heuristic = (1.0 / self.distances[current\_idx, node\_idx]) \*\* self.beta

            prob = pheromone \* heuristic

            probabilities.append(prob)

            total += prob

        if total == 0:

            return random.choice(unvisited)

        # Нормализуем вероятности

        probabilities = [p/total for p in probabilities]

        # Выбираем следующую вершину согласно вероятностям

        return np.random.choice(unvisited, p=probabilities)

    def \_update\_pheromone(self, ants\_paths):

        """Обновляет матрицу феромонов, учитывая частоту ребер."""

        # Испарение феромона

        self.pheromone \*= (1 - self.evaporation\_rate)

        # Сброс edge\_counts перед каждым обновлением феромонов

        self.edge\_counts.clear()

        # Подсчет вхождений ребер в маршруты

        for path, \_ in ants\_paths:

            for i in range(len(path) - 1):

                u, v = path[i], path[i + 1]

                # Учитываем ребро только в одном направлении для ориентированного графа

                self.edge\_counts[(u, v)] += 1

        # Добавление нового феромона

        for path, cost in ants\_paths:

            if cost == 0:

                continue

            delta\_pheromone = self.q / cost

            for i in range(len(path) - 1):

                u, v = path[i], path[i + 1]

                u\_idx, v\_idx = self.node\_indices[u], self.node\_indices[v]

                # Учитываем частоту ребра при обновлении феромона

                edge\_frequency = self.edge\_counts[(u, v)]

                self.pheromone[u\_idx, v\_idx] += delta\_pheromone \* (1 + edge\_frequency) # Умножаем на (1 + частота)

    def \_apply\_template\_modification(self, path):

        """Применяет модификацию шаблонов к пути."""

        if not self.templates:

            return path

        # Выбираем случайный шаблон

        template = random.choice(self.templates)

        # Смешиваем путь с шаблоном

        mixed\_path = []

        template\_set = set(template)

        path\_set = set(path[:-1])  # исключаем последнюю вершину (она совпадает с первой)

        # Добавляем вершины, которые есть и в пути, и в шаблоне

        common = list(path\_set & template\_set)

        random.shuffle(common)

        mixed\_path.extend(common)

        # Добавляем оставшиеся вершины из пути

        remaining = [node for node in path[:-1] if node not in mixed\_path]

        mixed\_path.extend(remaining)

        # Замыкаем цикл

        mixed\_path.append(mixed\_path[0])

        return mixed\_path

    def run(self):  # Убираем параметр start\_node

        """Запускает алгоритм муравьиной колонии."""

        start\_node = random.choice(self.nodes) # Выбираем случайную начальную вершину

        best\_path = None

        best\_cost = float('inf')

        for i in range(self.n\_iterations):

            self.iteration\_count = i

            ants\_paths = []

            for \_ in range(self.n\_ants):

                # Муравей начинает путь со стартовой вершины

                current\_node = start\_node

                visited = {current\_node}

                path = [current\_node]

                # Строим путь

                while len(visited) < self.n\_nodes:

                    next\_node = self.\_select\_next\_node(current\_node, visited)

                    if next\_node is None:

                        break

                    path.append(next\_node)

                    visited.add(next\_node)

                    current\_node = next\_node

                # Замыкаем цикл

                if len(path) == self.n\_nodes and self.distances[self.node\_indices[path[-1]], self.node\_indices[path[0]]] != float('inf'):

                    path.append(path[0])

                    cost = self.\_calculate\_path\_cost(path)

                    # Применяем модификацию шаблонов

                    if self.use\_templates and random.random() < 0.3:  # 30% chance to apply template

                        new\_path = self.\_apply\_template\_modification(path)

                        new\_cost = self.\_calculate\_path\_cost(new\_path)

                        if new\_cost < cost:

                            path = new\_path

                            cost = new\_cost

                    ants\_paths.append((path, cost))

                    if cost < best\_cost:

                        best\_path = path

                        best\_cost = cost

            if ants\_paths:

                self.\_update\_pheromone(ants\_paths)

            # Обновляем шаблоны с заданной частотой

            if self.use\_templates and (i + 1) % self.template\_update\_frequency == 0:

                self.\_update\_templates(ants\_paths)

        return best\_path, best\_cost

def ant\_colony(graph, n\_ants, n\_iterations,

               alpha, beta, evaporation\_rate,

               q, use\_templates, template\_update\_frequency=10):

    """Интерфейс для вызова муравьиного алгоритма."""

    if not graph.nodes():

        return None, 0

    aco = AntColonyOptimizer(graph, n\_ants, n\_iterations, alpha, beta,

                            evaporation\_rate, q, use\_templates, template\_update\_frequency)

    return aco.run()  #  Больше не передаем start\_node

def calculate\_total\_cost(graph, path):

    """Calculates path cost, returns None if any edge is missing."""

    total\_cost = 0

    for i in range(len(path)-1):

        if not graph.has\_edge(path[i], path[i+1]):

            return None

        total\_cost += graph[path[i]][path[i+1]]['weight']

    if not graph.has\_edge(path[-1], path[0]):

        return None

    total\_cost += graph[path[-1]][path[0]]['weight']

    return total\_cost

3) tsp\_gui.py

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

import tkinter.simpledialog

import networkx as nx

import math

from tkinter import filedialog

import pandas as pd

import random

import time

from tsp\_solver import ant\_colony, nearest\_neighbor, calculate\_total\_cost

class TSPInteractiveGUI:

    def \_\_init\_\_(self, master):

        self.master = master

        master.title("Интерактивная задача коммивояжера (Муравьиный алгоритм)")

        # Graph Data Structures

        self.graph = nx.DiGraph()

        self.vertices = {}

        self.edges = {}

        self.edge\_objects = {}

        self.edge\_text\_objects = {}

        self.start\_vertex = None

        self.vertex\_radius = 15

        self.edge\_line\_width = 2

        self.edge\_color = "gray"

        self.vertex\_color = "skyblue"

        self.curve\_offset = 20

        self.arrow\_size = 10

        self.path\_objects = []

        self.best\_path = None

        self.best\_cost = None

        self.last\_edge = None

        self.edge\_history = []

        self.selected\_edge = None

        # Default parameters for Ant Colony

        self.n\_ants = 10

        self.n\_iterations = 1000

        self.alpha = 1.0

        self.beta = 3.0

        self.evaporation\_rate = 0.3

        self.q = 100

        # Main Frame

        self.main\_frame = ttk.Frame(master)

        self.main\_frame.pack(fill=tk.BOTH, expand=True)

        # Create GUI Elements

        self.create\_graph\_canvas(self.main\_frame)

        self.create\_distance\_matrix\_frame(self.main\_frame)

        self.create\_control\_frame(master)

        self.create\_result\_frame(master)

        self.path = None

        self.selected\_vertices = []

    def create\_graph\_canvas(self, master):

        """Создает поле для интерактивного рисования графа."""

        self.canvas\_width = 600

        self.canvas\_height = 300

        self.canvas = tk.Canvas(master, width=self.canvas\_width, height=self.canvas\_height, bg="white")

        self.canvas.grid(row=0, column=0, padx=10, pady=10, sticky=tk.NSEW)

        master.columnconfigure(0, weight=1)

        self.canvas.bind("<Button-1>", self.on\_canvas\_click)

    def create\_distance\_matrix\_frame(self, master):

        """Создает фрейм для отображения матрицы расстояний."""

        self.matrix\_frame = ttk.Frame(master)

        self.matrix\_frame.grid(row=0, column=1, padx=10, pady=10, sticky=tk.NSEW)

        self.matrix\_text = tk.Text(self.matrix\_frame, width=30, height=10, font=('Courier', 10))

        self.matrix\_text.pack(fill=tk.BOTH, expand=True)

        self.matrix\_text.config(state=tk.DISABLED)

        master.columnconfigure(1, weight=2)

    def update\_distance\_matrix(self):

        """Обновляет отображение матрицы расстояний."""

        self.matrix\_text.config(state=tk.NORMAL)

        self.matrix\_text.delete("1.0", tk.END)

        vertices = list(self.vertices.keys())

        if not vertices:

            self.matrix\_text.config(state=tk.DISABLED)

            return

        max\_vertex\_length = max(len(v) for v in vertices)

        header = " " \* (max\_vertex\_length + 1) + "".join([f"{v:<4}" for v in vertices]) + "\n"

        self.matrix\_text.insert("1.0", header)

        reversed\_vertices = list(reversed(vertices))

        for v1 in reversed\_vertices:

            row = f"{v1:<{max\_vertex\_length}} "

            for v2 in vertices:

                try:

                    weight = self.graph[v1][v2]['weight']

                    row += f"{weight:<4}"

                except KeyError:

                    row += "i   "

            row += "\n"

            self.matrix\_text.insert("2.0", row)

        self.matrix\_text.config(state=tk.DISABLED)

    def create\_control\_frame(self, master):

        control\_frame = ttk.Frame(master)

        control\_frame.pack(padx=10, pady=5, fill=tk.X)

        general\_frame = ttk.LabelFrame(control\_frame, text="Основные команды")

        general\_frame.grid(row=0, column=0, padx=5, pady=5, sticky="ew")

        run\_button = ttk.Button(general\_frame, text="Найти кратчайший путь", command=self.run\_ant\_colony)

        run\_button.grid(row=0, column=0, padx=5, pady=2)

        clear\_button = ttk.Button(general\_frame, text="Очистить поле", command=self.clear\_graph)

        clear\_button.grid(row=0, column=1, padx=5, pady=2)

        delete\_vertex\_button = ttk.Button(general\_frame, text="Удалить последнюю вершину", command=self.delete\_last\_vertex)

        delete\_vertex\_button.grid(row=1, column=0, padx=5, pady=2)

        delete\_path\_button = ttk.Button(general\_frame, text="Удалить последнее ребро", command=self.delete\_last\_edge)

        delete\_path\_button.grid(row=1, column=1, padx=5, pady=2)

        start\_vertex\_frame = ttk.LabelFrame(control\_frame, text="Начальная вершина")

        start\_vertex\_frame.grid(row=1, column=0, padx=5, pady=5, sticky="ew")

        self.start\_vertex\_label = ttk.Label(start\_vertex\_frame, text="Начальная вершина:")

        self.start\_vertex\_label.grid(row=0, column=0, padx=5, pady=2)

        self.start\_vertex\_entry = ttk.Entry(start\_vertex\_frame, width=5)

        self.start\_vertex\_entry.grid(row=0, column=1, padx=5, pady=2)

        self.set\_start\_vertex\_button = ttk.Button(start\_vertex\_frame, text="Установить", command=self.set\_start\_vertex)

        self.set\_start\_vertex\_button.grid(row=0, column=2, padx=5, pady=2)

        load\_matrix\_frame = ttk.LabelFrame(control\_frame, text="Загрузка матрицы")

        load\_matrix\_frame.grid(row=2, column=0, padx=5, pady=5, sticky="ew")

        load\_matrix\_button = ttk.Button(load\_matrix\_frame, text="Загрузить матрицу из таблицы", command=self.load\_matrix\_from\_table)

        load\_matrix\_button.grid(row=0, column=0, padx=5, pady=2)

        edge\_weight\_frame = ttk.LabelFrame(control\_frame, text="Вес ребра")

        edge\_weight\_frame.grid(row=3, column=0, padx=5, pady=5, sticky="ew")

        ttk.Label(edge\_weight\_frame, text="Ребро:").grid(row=0, column=0, padx=5, pady=2)

        self.edge\_entry = ttk.Entry(edge\_weight\_frame, width=5)

        self.edge\_entry.grid(row=0, column=1, padx=5, pady=2)

        ttk.Label(edge\_weight\_frame, text="Вес ребра:").grid(row=1, column=0, padx=5, pady=2)

        self.edge\_weight\_entry = ttk.Entry(edge\_weight\_frame, width=5)

        self.edge\_weight\_entry.grid(row=1, column=1, padx=5, pady=2)

        self.update\_edge\_weight\_button = ttk.Button(edge\_weight\_frame, text="Изменить", command=self.update\_edge\_weight)

        self.update\_edge\_weight\_button.grid(row=1, column=2, padx=5, pady=2)

        # Параметры муравьиного алгоритма

        aco\_params\_frame = ttk.LabelFrame(control\_frame, text="Параметры муравьиного алгоритма")

        aco\_params\_frame.grid(row=0, column=1, rowspan=4, padx=5, pady=5, sticky="nsew")

        ttk.Label(aco\_params\_frame, text="Количество муравьев:").grid(row=0, column=0, padx=5, pady=2, sticky=tk.E)

        self.n\_ants\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.n\_ants\_entry.insert(0, str(self.n\_ants))

        self.n\_ants\_entry.grid(row=0, column=1, padx=5, pady=2, sticky=tk.W)

        ttk.Label(aco\_params\_frame, text="Количество итераций:").grid(row=1, column=0, padx=5, pady=2, sticky=tk.E)

        self.n\_iterations\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.n\_iterations\_entry.insert(0, str(self.n\_iterations))

        self.n\_iterations\_entry.grid(row=1, column=1, padx=5, pady=2, sticky=tk.W)

        ttk.Label(aco\_params\_frame, text="Феромоны:").grid(row=2, column=0, padx=5, pady=2, sticky=tk.E)

        self.alpha\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.alpha\_entry.insert(0, str(self.alpha))

        self.alpha\_entry.grid(row=2, column=1, padx=5, pady=2, sticky=tk.W)

        ttk.Label(aco\_params\_frame, text="Эвристика:").grid(row=3, column=0, padx=5, pady=2, sticky=tk.E)

        self.beta\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.beta\_entry.insert(0, str(self.beta))

        self.beta\_entry.grid(row=3, column=1, padx=5, pady=2, sticky=tk.W)

        ttk.Label(aco\_params\_frame, text="Испарение:").grid(row=4, column=0, padx=5, pady=2, sticky=tk.E)

        self.evaporation\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.evaporation\_entry.insert(0, str(self.evaporation\_rate))

        self.evaporation\_entry.grid(row=4, column=1, padx=5, pady=2, sticky=tk.W)

        ttk.Label(aco\_params\_frame, text="Константа для феромонов:").grid(row=5, column=0, padx=5, pady=2, sticky=tk.E)

        self.q\_entry = ttk.Entry(aco\_params\_frame, width=5)

        self.q\_entry.insert(0, str(self.q))

        self.q\_entry.grid(row=5, column=1, padx=5, pady=2, sticky=tk.W)

        self.modification\_var = tk.BooleanVar()

        modification\_check = ttk.Checkbutton(aco\_params\_frame, text="Использовать шаблоны", variable=self.modification\_var)

        modification\_check.grid(row=6, column=0, columnspan=2, padx=5, pady=2)

        control\_frame.columnconfigure(0, weight=1)

        control\_frame.columnconfigure(1, weight=1)

        aco\_params\_frame.columnconfigure(0, weight=1)

        aco\_params\_frame.columnconfigure(1, weight=1)

        general\_frame.columnconfigure(0, weight=1)

        general\_frame.columnconfigure(1, weight=1)

        start\_vertex\_frame.columnconfigure(0, weight=1)

        start\_vertex\_frame.columnconfigure(1, weight=1)

        start\_vertex\_frame.columnconfigure(2, weight=1)

        load\_matrix\_frame.columnconfigure(0, weight=1)

        edge\_weight\_frame.columnconfigure(0, weight=1)

        edge\_weight\_frame.columnconfigure(1, weight=1)

        edge\_weight\_frame.columnconfigure(2, weight=1)

    def create\_result\_frame(self, master):

        result\_frame = ttk.LabelFrame(master, text="Результат")

        result\_frame.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

        ttk.Label(result\_frame, text="Кратчайший путь:").grid(row=0, column=0, sticky=tk.W, padx=5, pady=2)

        self.path\_text = tk.Text(result\_frame, height=3, width=50)

        self.path\_text.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=2)

        self.path\_text.config(state=tk.DISABLED)

        ttk.Label(result\_frame, text="Общая длина:").grid(row=1, column=0, sticky=tk.W, padx=5, pady=2)

        self.length\_label = ttk.Label(result\_frame, text="")

        self.length\_label.grid(row=1, column=1, sticky=tk.W, padx=5, pady=2)

        ttk.Label(result\_frame, text="Время расчета:").grid(row=2, column=0, sticky=tk.W, padx=5, pady=2)

        self.time\_label = ttk.Label(result\_frame, text="")

        self.time\_label.grid(row=2, column=1, sticky=tk.W, padx=5, pady=2)

        result\_frame.columnconfigure(1, weight=1)

    def on\_canvas\_click(self, event):

        x, y = event.x, event.y

        closest\_vertex = self.find\_closest\_vertex(x, y)

        if closest\_vertex:

            self.selected\_vertices.append(closest\_vertex)

            if len(self.selected\_vertices) == 2:

                v1, v2 = self.selected\_vertices[0], self.selected\_vertices[1]

                self.selected\_vertices = []

                if v1 == v2:

                    messagebox.showinfo("Информация", "Нельзя создать ребро между одной и той же вершиной.")

                else:

                    weight = self.ask\_for\_weight()

                    if weight is not None:

                        if (v1, v2) not in self.edges:

                            self.edges[(v1, v2)] = []

                            self.edge\_objects[(v1,v2)] = []

                            self.edge\_text\_objects[(v1, v2)] = []

                        self.graph.add\_edge(v1, v2, weight=weight)

                        self.edges[(v1, v2)].append(weight)

                        edge\_id, text\_id = self.draw\_edge(v1, v2, weight, len(self.edges[(v1, v2)])-1)

                        self.edge\_objects[(v1,v2)].append(edge\_id)

                        self.edge\_text\_objects[(v1, v2)].append(text\_id)

                        edge = (v1, v2)

                        self.edge\_history.append(edge)

                        self.update\_distance\_matrix()

        else:

            self.add\_vertex\_event(event)

    def add\_vertex\_event(self, event):

        x, y = event.x, event.y

        vertex\_name = chr(65 + len(self.vertices))

        x1, y1 = x - self.vertex\_radius, y - self.vertex\_radius

        x2, y2 = x + self.vertex\_radius, y + self.vertex\_radius

        vertex\_id = self.canvas.create\_oval(x1, y1, x2, y2, fill=self.vertex\_color, outline="black")

        text\_id = self.canvas.create\_text(x, y, text=vertex\_name, fill="black", font=('Arial', 10, 'bold'))

        self.vertices[vertex\_name] = {"x": x, "y": y, "oval\_id": vertex\_id, "text\_id": text\_id}

        self.graph.add\_node(vertex\_name)

        self.update\_distance\_matrix()

    def get\_midpoint\_coordinates(self, x1, y1, ctrl\_x1, ctrl\_y1, ctrl\_x2, ctrl\_y2, x2, y2):

        """Вычисляет координаты середины линии."""

        mid\_x = (x1 + x2 + ctrl\_x1 + ctrl\_x2) / 4

        mid\_y = (y1 + y2 + ctrl\_y1 + ctrl\_y2) / 4

        return mid\_x, mid\_y

    def draw\_arrow(self, x, y, angle, color):

        """Рисует стрелку-треугольник."""

        arrow\_size = self.arrow\_size

        points = [

            x + arrow\_size \* math.cos(angle), y + arrow\_size \* math.sin(angle),

            x + arrow\_size/2 \* math.cos(angle + 2 \* math.pi / 3), y + arrow\_size/2 \* math.sin(angle + 2 \* math.pi / 3),

            x + arrow\_size/2 \* math.cos(angle + 4 \* math.pi / 3), y + arrow\_size/2 \* math.sin(angle + 4 \* math.pi / 3)

        ]

        arrow\_id = self.canvas.create\_polygon(points, fill=color, outline=color)

        return arrow\_id

    def draw\_edge(self, v1, v2, weight, edge\_index):

      """Рисует ребро между двумя вершинами."""

      x1, y1 = self.vertices[v1]["x"], self.vertices[v1]["y"]

      x2, y2 = self.vertices[v2]["x"], self.vertices[v2]["y"]

      dx = x2 - x1

      dy = y2 - y1

      distance = math.sqrt(dx\*\*2 + dy\*\*2)

      if distance == 0:

          return

      dx /= distance

      dy /= distance

      offset\_x = dy \* self.curve\_offset \* (edge\_index - 1)

      offset\_y = -dx \* self.curve\_offset \* (edge\_index - 1)

      ctrl\_x1 = x1 + offset\_x

      ctrl\_y1 = y1 + offset\_y

      ctrl\_x2 = x2 + offset\_x

      ctrl\_y2 = y2 + offset\_y

      edge\_id = self.canvas.create\_line(x1, y1, ctrl\_x1, ctrl\_y1, ctrl\_x2, ctrl\_y2, x2, y2,

                                           smooth=True,

                                           fill=self.edge\_color, width=self.edge\_line\_width)

      mid\_x, mid\_y = self.get\_midpoint\_coordinates(x1, y1, ctrl\_x1, ctrl\_y1, ctrl\_x2, ctrl\_y2, x2, y2)

      angle = math.atan2(y2 - y1, x2 - x1)

      self.draw\_arrow(mid\_x, mid\_y, angle, self.edge\_color)

      text\_id = self.canvas.create\_text(mid\_x , mid\_y - 10, text=str(weight), fill="black", font=('Arial', 8))

      return edge\_id, text\_id

    def find\_closest\_vertex(self, x, y, max\_distance=20):

        closest\_vertex = None

        min\_distance = max\_distance

        for vertex, data in self.vertices.items():

            vx, vy = data["x"], data["y"]

            distance = ((x - vx)\*\*2 + (y - vy)\*\*2)\*\*0.5

            if distance < min\_distance:

                closest\_vertex = vertex

                min\_distance = distance

        return closest\_vertex

    def ask\_for\_weight(self, ):

        weight = None

        while weight is None:

            try:

                weight\_str = tk.simpledialog.askstring("Ввод веса", "Введите вес ребра:")

                if weight\_str is None:

                    return None

                weight = int(weight\_str)

                if weight <= 0:

                    messagebox.showerror("Ошибка ввода", "Вес должен быть положительным числом.")

                    weight = None

            except ValueError:

                messagebox.showerror("Ошибка ввода", "Вес должен быть целым числом.")

                weight = None

        return weight

    def set\_start\_vertex(self):

        start\_vertex = self.start\_vertex\_entry.get()

        if start\_vertex in self.vertices:

            self.start\_vertex = start\_vertex

            messagebox.showinfo("Информация", f"Начальная вершина установлена в {start\_vertex}")

            self.draw\_graph()

        else:

            messagebox.showerror("Ошибка", "Неверная начальная вершина.  Выберите существующую вершину.")

            self.start\_vertex = None

    def delete\_last\_vertex(self):

        if self.vertices:

            vertex\_to\_delete = list(self.vertices.keys())[-1]

            edges\_to\_delete = []

            for (v1, v2) in self.edges:

                if v1 == vertex\_to\_delete or v2 == vertex\_to\_delete:

                    edges\_to\_delete.append((v1, v2))

            for v1, v2 in edges\_to\_delete:

                if (v1, v2) in self.edge\_objects:

                    for edge\_id in self.edge\_objects[(v1, v2)]:

                        self.canvas.delete(edge\_id)

                    del self.edge\_objects[(v1, v2)]

                if (v1, v2) in self.edge\_text\_objects:

                  for text\_id in self.edge\_text\_objects[(v1, v2)]:

                    self.canvas.delete(text\_id)

                  del self.edge\_text\_objects[(v1, v2)]

                if (v1, v2) in self.edges:

                    del self.edges[(v1, v2)]

                if self.graph.has\_edge(v1, v2):

                    self.graph.remove\_edge(v1, v2)

            self.edge\_history = [(v1, v2) for v1, v2 in self.edge\_history if v1 != vertex\_to\_delete and v2 != vertex\_to\_delete]

            self.canvas.delete(self.vertices[vertex\_to\_delete]["oval\_id"])

            self.canvas.delete(self.vertices[vertex\_to\_delete]["text\_id"])

            del self.vertices[vertex\_to\_delete]

            self.graph.remove\_node(vertex\_to\_delete)

            self.update\_distance\_matrix()

            self.draw\_graph()

            if self.start\_vertex == vertex\_to\_delete:

                self.start\_vertex = None

    def draw\_graph(self):

        self.canvas.delete("all")

        for (v1, v2), weights in self.edges.items():

            for i, weight in enumerate(weights):

                edge\_id, text\_id = self.draw\_edge(v1,v2, weight, i)

                self.edge\_objects[(v1,v2)].append(edge\_id)

                self.edge\_text\_objects[(v1, v2)].append(text\_id)

        for vertex, data in self.vertices.items():

            x, y = data["x"], data["y"]

            x1, y1 = x - self.vertex\_radius, y - self.vertex\_radius

            x2, y2 = x + self.vertex\_radius, y + self.vertex\_radius

            oval\_id = self.canvas.create\_oval(x1, y1, x2, y2, fill=self.vertex\_color, outline="black")

            text\_id = self.canvas.create\_text(x, y, text=vertex, fill="black", font=('Arial', 10, 'bold'))

            self.vertices[vertex]["oval\_id"] = oval\_id

            self.vertices[vertex]["text\_id"] = text\_id

    def clear\_graph(self):

        self.graph.clear()

        self.vertices.clear()

        self.edges.clear()

        self.edge\_objects.clear()

        self.edge\_text\_objects.clear()

        self.start\_vertex = None

        self.path = None

        self.path\_text.config(state=tk.NORMAL)

        self.path\_text.delete("1.0", tk.END)

        self.path\_text.config(state=tk.DISABLED)

        self.length\_label.config(text="")

        self.time\_label.config(text="")

        self.canvas.delete("all")

        self.update\_distance\_matrix()

    def get\_aco\_parameters(self):

        """Получает параметры муравьиного алгоритма из полей ввода."""

        try:

            self.n\_ants = int(self.n\_ants\_entry.get())

            self.n\_iterations = int(self.n\_iterations\_entry.get())

            self.alpha = float(self.alpha\_entry.get())

            self.beta = float(self.beta\_entry.get())

            self.evaporation\_rate = float(self.evaporation\_entry.get())

            self.q = float(self.q\_entry.get())

            if (self.n\_ants <= 0 or self.n\_iterations <= 0 or self.alpha <= 0 or

                self.beta <= 0 or self.evaporation\_rate <= 0 or self.evaporation\_rate >= 1 or

                self.q <= 0):

                messagebox.showerror("Ошибка", "Все параметры должны быть положительными числами. Испарение должно быть между 0 и 1.")

                return False

            return True

        except ValueError:

            messagebox.showerror("Ошибка", "Пожалуйста, введите корректные числовые значения для параметров.")

            return False

    def run\_ant\_colony(self):

        """Запускает муравьиный алгоритм для решения задачи коммивояжера."""

        if not self.get\_aco\_parameters():

            return

        if not self.graph.nodes():

            messagebox.showerror("Ошибка", "Пожалуйста, создайте граф.")

            return

        nodes = list(self.graph.nodes())

        if self.start\_vertex is None or self.start\_vertex not in nodes:

            self.start\_vertex = random.choice(nodes)

        self.clear\_temporary\_edge()

        self.delete\_last\_path()

        start\_time = time.time()

        best\_path, best\_cost = ant\_colony(

            self.graph,

            n\_ants=self.n\_ants,

            n\_iterations=self.n\_iterations,

            alpha=self.alpha,

            beta=self.beta,

            evaporation\_rate=self.evaporation\_rate,

            q=self.q,

            use\_templates=self.modification\_var.get()

        )

        end\_time = time.time()

        calculation\_time = end\_time - start\_time

        if best\_path:

            self.best\_path = best\_path

            self.best\_cost = best\_cost

            self.path\_text.config(state=tk.NORMAL)

            self.path\_text.delete("1.0", tk.END)

            self.path\_text.insert(tk.END, " -> ".join(best\_path))

            self.path\_text.config(state=tk.DISABLED)

            # Изменяем форматирование вывода длины пути

            if best\_cost.is\_integer():

                self.length\_label.config(text=str(int(best\_cost)))

            else:

                self.length\_label.config(text=str(best\_cost))

            self.time\_label.config(text=f"{calculation\_time:.4f} секунд")

            self.draw\_graph\_with\_path()

        else:

            messagebox.showerror("Ошибка", "Решение не найдено.")

    def calculate\_total\_cost(self, graph, path):

        """Calculates the total cost of a given path."""

        total\_cost = 0

        for i in range(len(path) - 1):

            try:

                total\_cost += graph[path[i]][path[i + 1]]['weight']

            except KeyError:

                return float('inf')

        return total\_cost

    def run\_all\_nearest\_neighbor(self):

        """(Мод) Перебирает все вершины как стартовые и находит лучший путь."""

        start\_time = time.time()

        graph\_copy = self.graph.copy()

        edges\_copy = self.edges.copy()

        edge\_objects\_copy = self.edge\_objects.copy()

        edge\_text\_objects\_copy = self.edge\_text\_objects.copy()

        best\_path = None

        min\_cost = float('inf')

        best\_start\_node = None

        if self.start\_vertex is None and not self.modification\_var.get():

            messagebox.showerror("Ошибка", "Пожалуйста, установите начальную вершину.")

            return

        if self.modification\_var.get():

            start\_nodes = self.vertices.keys()

        else:

            start\_nodes = [self.start\_vertex]

        for start\_node in start\_nodes:

            path, total\_cost = nearest\_neighbor(graph\_copy, start\_node)

            if path and total\_cost < min\_cost:

                min\_cost = total\_cost

                best\_path = path

                best\_start\_node = start\_node

        end\_time = time.time()

        calculation\_time = end\_time - start\_time

        self.graph = graph\_copy

        self.edges = edges\_copy

        self.edge\_objects = edge\_objects\_copy

        self.edge\_text\_objects = edge\_text\_objects\_copy

        self.draw\_graph()

        self.update\_distance\_matrix()

        if best\_path:

            self.best\_path = best\_path

            self.best\_cost = min\_cost

            self.path\_text.config(state=tk.NORMAL)

            self.path\_text.delete("1.0", tk.END)

            self.path\_text.insert(tk.END, " -> ".join(best\_path))

            self.path\_text.config(state=tk.DISABLED)

            self.length\_label.config(text=str(min\_cost))

            self.time\_label.config(text=f"{calculation\_time:.4f} секунд") # Форматированный вывод

            self.draw\_graph\_with\_path()

        else:

            messagebox.showerror("Ошибка", "Некорректный граф.")

    def draw\_graph\_with\_path(self):

        """Отображает граф с найденным путем."""

        if self.graph is None or self.best\_path is None:

            return

        self.path\_objects = []

        for i in range(len(self.best\_path) - 1):

            v1, v2 = self.best\_path[i], self.best\_path[i + 1]

            x1, y1 = self.vertices[v1]["x"], self.vertices[v1]["y"]

            x2, y2 = self.vertices[v2]["x"], self.vertices[v2]["y"]

            line\_id = self.canvas.create\_line(x1, y1, x2, y2, fill="green", width=3)

            self.path\_objects.append(line\_id)

            mid\_x, mid\_y = (x1 + x2) / 2, (y1 + y2) / 2

            angle = math.atan2(y2 - y1, x2 - x1)

            arrow\_id = self.draw\_arrow(mid\_x, mid\_y, angle, "green")

            self.path\_objects.append(arrow\_id)

    def delete\_last\_edge(self):

        """Удаляет последнее добавленное ребро."""

        if self.edge\_history:

            v1, v2 = self.edge\_history.pop()

            if (v1, v2) in self.edge\_objects:

                if self.edge\_objects[(v1, v2)]:

                  edge\_id = self.edge\_objects[(v1, v2)].pop()

                  self.canvas.delete(edge\_id)

                if not self.edge\_objects[(v1, v2)]:

                  del self.edge\_objects[(v1, v2)]

            if (v1, v2) in self.edge\_text\_objects:

                for text\_id in self.edge\_text\_objects[(v1,v2)]:

                    self.canvas.delete(text\_id)

                del self.edge\_text\_objects[(v1,v2)]

            if (v1, v2) in self.edges:

                del self.edges[(v1, v2)]

            if self.graph.has\_edge(v1, v2):

               self.graph.remove\_edge(v1,v2)

            self.update\_distance\_matrix()

            self.draw\_graph()

            if not self.edge\_history:

                self.last\_edge = None

    def delete\_last\_path(self):

        """Удаляет с холста последний нарисованный путь и восстанавливает матрицу."""

        for obj\_id in self.path\_objects:

            self.canvas.delete(obj\_id)

        self.path\_objects = []

        self.best\_path = None

        self.best\_cost = None

        self.path\_text.config(state=tk.NORMAL)

        self.path\_text.delete("1.0", tk.END)

        self.path\_text.config(state=tk.DISABLED)

        self.length\_label.config(text="")

        self.time\_label.config(text="")

        self.draw\_graph()

    def clear\_temporary\_edge(self):

        self.selected\_vertices = []

    def load\_matrix\_from\_table(self):

        """Загружает матрицу расстояний из таблицы CSV."""

        filename = filedialog.askopenfilename(filetypes=[("CSV files", "\*.csv")])

        if filename:

            try:

                df = pd.read\_csv(filename, index\_col=0)

                vertices = list(df.index)

                distances = df.to\_dict()

                self.load\_graph\_from\_matrix(vertices, distances)

            except FileNotFoundError:

                messagebox.showerror("Ошибка", "Файл не найден.")

            except Exception as e:

                messagebox.showerror("Ошибка", f"Не удалось загрузить матрицу из таблицы. Проверьте формат. Ошибка: {e}")

    def load\_graph\_from\_matrix(self, vertices, distances):

        """Загружает граф из матрицы расстояний, игнорируя значения <= 0 как отсутствие пути."""

        self.clear\_graph()

        for vertex in vertices:

            x = random.randint(50, self.canvas\_width - 50)

            y = random.randint(50, self.canvas\_height - 50)

            self.add\_vertex\_from\_matrix(vertex, x, y)

        for v1 in vertices:

            for v2 in vertices:

                if v1 != v2:

                    weight = distances[v2][v1]  # Access the distance correctly

                    # Check if the weight is zero or negative, indicating no path

                    if weight <= 0:

                        continue

                    # Also check if it is not NaN if you are getting the value from a pandas dataframe

                    if not isinstance(weight, str) and not (isinstance(weight, float) and math.isnan(weight)):

                        if (v1, v2) not in self.edges:

                            self.edges[(v1, v2)] = []

                            self.edge\_objects[(v1,v2)] = []

                            self.edge\_text\_objects[(v1, v2)] = []

                        self.graph.add\_edge(v1, v2, weight=weight)

                        self.edges[(v1, v2)].append(weight)

                        edge\_id, text\_id = self.draw\_edge(v1, v2, weight, len(self.edges[(v1, v2)])-1)

                        self.edge\_objects[(v1,v2)].append(edge\_id)

                        self.edge\_text\_objects[(v1, v2)].append(text\_id)

                        edge = (v1, v2)

                        self.edge\_history.append(edge)

        self.update\_distance\_matrix()

        self.draw\_graph()

    def add\_vertex\_from\_matrix(self, vertex\_name, x, y):

        """Добавляет вершину из матрицы."""

        x1, y1 = x - self.vertex\_radius, y - self.vertex\_radius

        x2, y2 = x + self.vertex\_radius, y + self.vertex\_radius

        vertex\_id = self.canvas.create\_oval(x1, y1, x2, y2, fill=self.vertex\_color, outline="black")

        text\_id = self.canvas.create\_text(x, y, text=vertex\_name, fill="black", font=('Arial', 10, 'bold'))

        self.vertices[vertex\_name] = {"x": x, "y": y, "oval\_id": vertex\_id, "text\_id": text\_id}

        self.graph.add\_node(vertex\_name)

    def update\_edge\_weight(self):

        """Обновляет вес выбранного ребра."""

        edge\_str = self.edge\_entry.get().upper()

        try:

            v1, v2 = edge\_str[0], edge\_str[1]

            if v1 not in self.vertices or v2 not in self.vertices:

                raise ValueError("Указанная вершина не существует.")

            if (v1, v2) not in self.edges:

                raise ValueError("Указанное ребро не существует.")

            try:

                new\_weight = int(self.edge\_weight\_entry.get())

                if new\_weight <= 0:

                    messagebox.showerror("Ошибка", "Вес должен быть положительным числом.")

                    return

            except ValueError:

                messagebox.showerror("Ошибка", "Пожалуйста, введите целое число для веса.")

                return

            self.graph[v1][v2]['weight'] = new\_weight

            self.edges[(v1, v2)][0] = new\_weight

            self.update\_distance\_matrix()

            if (v1, v2) in self.edge\_text\_objects:

                for text\_id in self.edge\_text\_objects[(v1, v2)]:

                    self.canvas.delete(text\_id)

            edge\_id, text\_id = self.draw\_edge(v1, v2, new\_weight, 0)

            self.edge\_objects[(v1, v2)] = [edge\_id]

            self.edge\_text\_objects[(v1, v2)] = [text\_id]

            self.draw\_graph()

            messagebox.showinfo("Информация", "Вес ребра успешно изменен.")

        except ValueError as e:

            messagebox.showerror("Ошибка", str(e))