**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики - процессов управления**

**Программа бакалавриата**

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

**ОТЧЕТ**

**по лабораторной работе №1**

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

**на тему «Метод ближайшего соседа»**

**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 4](#_Toc178797432)

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

[6. Описание программы 6](#_Toc178797442)

[7. Рекомендации пользователя 7](#_Toc178797443)

[8. Рекомендации программиста 8](#_Toc178797444)

[9. Исходный код программы 8](#_Toc178797445)

[10. Контрольный пример 8](#_Toc178797447)

[11. Вывод 11](#_Toc178797448)

[12. Источники 11](#_Toc178797448)

[13. Листинг 12](#_Toc178797450)

# Цель работы

# Разработать программное обеспечение, которое с помощью метода ближайшего соседа находит кратчайший гамильтонов цикл в направленном взвешенном графе.

# Описание задачи (формализация задачи)

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

# Теоретическая часть

Гамильтонов цикл — 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 относится к классическим проблемам комбинаторной оптимизации и представляет собой поиск наикратчайшего гамильтонова цикла. Эта задача относится к категории NP-полных задач, что означает отсутствие известных алгоритмов, способных эффективно решать её для всех входных данных в течение полиномиального времени. Тем не менее, для приблизительного или точного решения задачи разработаны разнообразные эвристические и точные методы.

Алгоритм ближайшего соседа — один из простейших эвристических алгоритмов решения задачи коммивояжёра. Относится к категории «жадных» алгоритмов.

Формулируется следующим образом: пункты обхода плана последовательно включаются в маршрут, причем каждый очередной включаемый пункт должен быть ближайшим к последнему выбранному пункту среди всех остальных, ещё не включенных в состав маршрута.

Модификация алгоритма заключается в запуске алгоритма, начиная из каждой вершины графа и выбор кратчайшего из всех этих путей.

# Основные шаги программы

1. Инициализация графа
2. Поставить все вершины как не посещённые.
3. Выбрать начальную вершину и пометить её, как посещённую.
4. Выбрать наиближайшую не посещённую смежную вершину к начальной вершине.
5. Поставить как текущую вершину и пометить как посещённую.
6. Если все вершины посещены, то завершить алгоритм. Иначе, вернуться к шагу 3.

# 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 функциях:

Таблица 6.1 Описание функций

|  |  |
| --- | --- |
| Имя функции | Назначение |
| nearest\_neighbor | Применяет метод ближайшего соседа к заданному графу. |

Интерфейс реализован через класс TSPInteractiveGUI, cодержащий следующие функции:

Таблица 6.2 Описание функций класса

|  |  |
| --- | --- |
| 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. |
| run\_all\_nearest\_neighbor | Запускает алгоритм, начиная с каждой вершины. (Модификация) |
| 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. 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. Поставить галочку “Модификация” для включения модификации или указать начальную величину обхода.
4. Нажать кнопку “Найти кратчайший путь”. На графе 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/lab1>

# Контрольный пример

В данном разделе представлен пример, демонстрирующий работу программы.

1) Запуск main.py

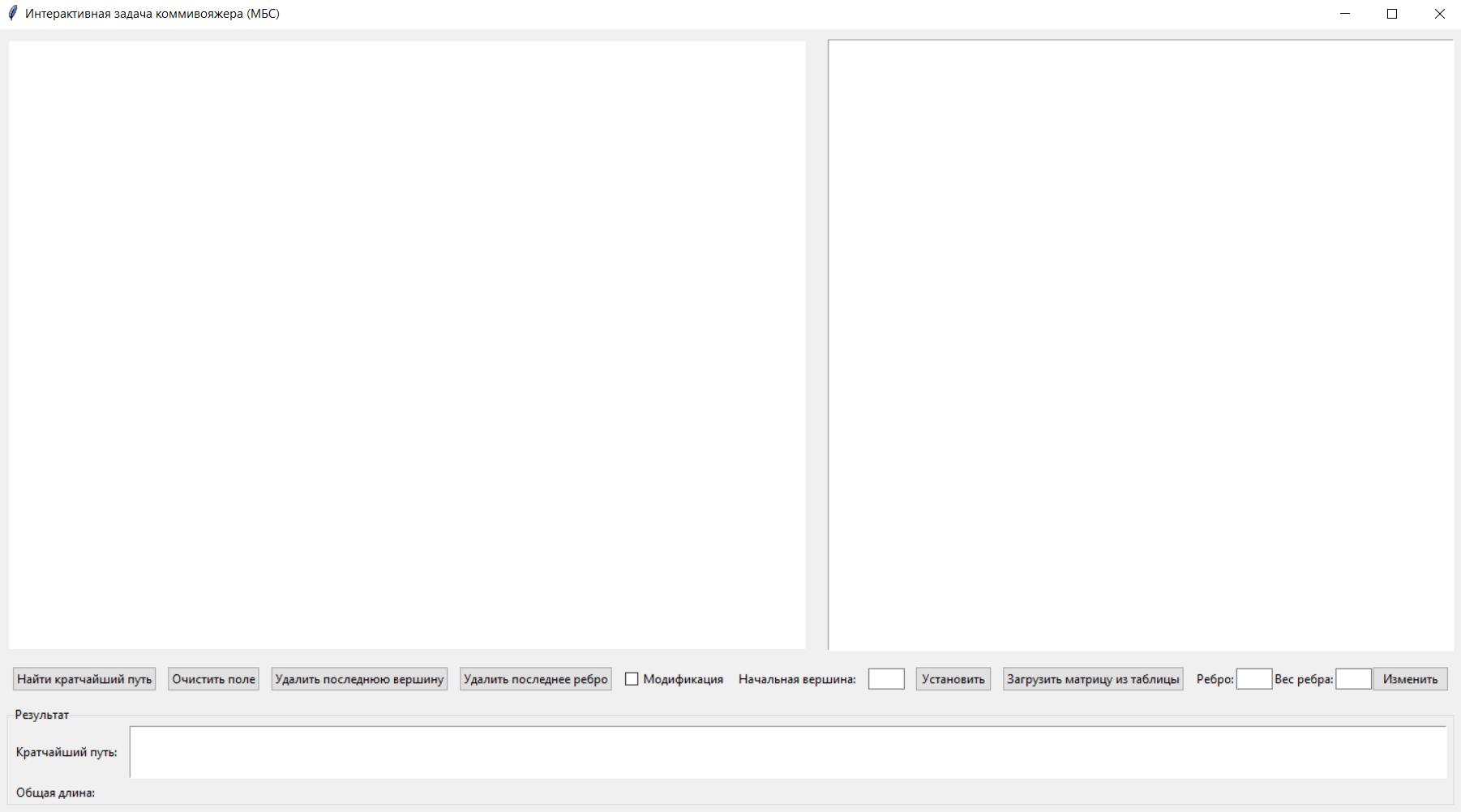


Рис 10.1 Графический интерфейс программы.

2) Задание графа

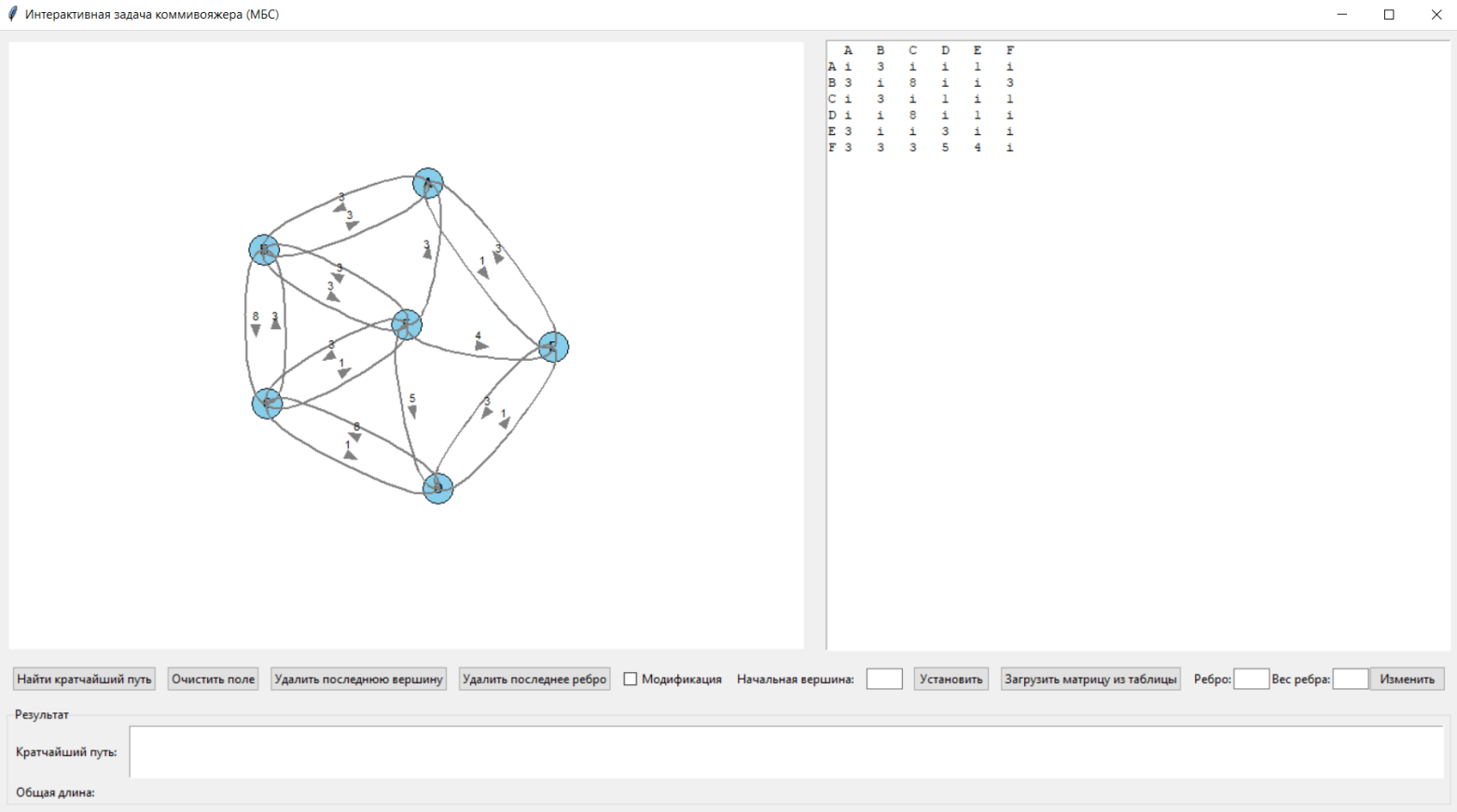


Рис 10.2 Графический интерфейс с заданным графом.

3) Результат алгоритма без модификации (за начальную взята вершина А)

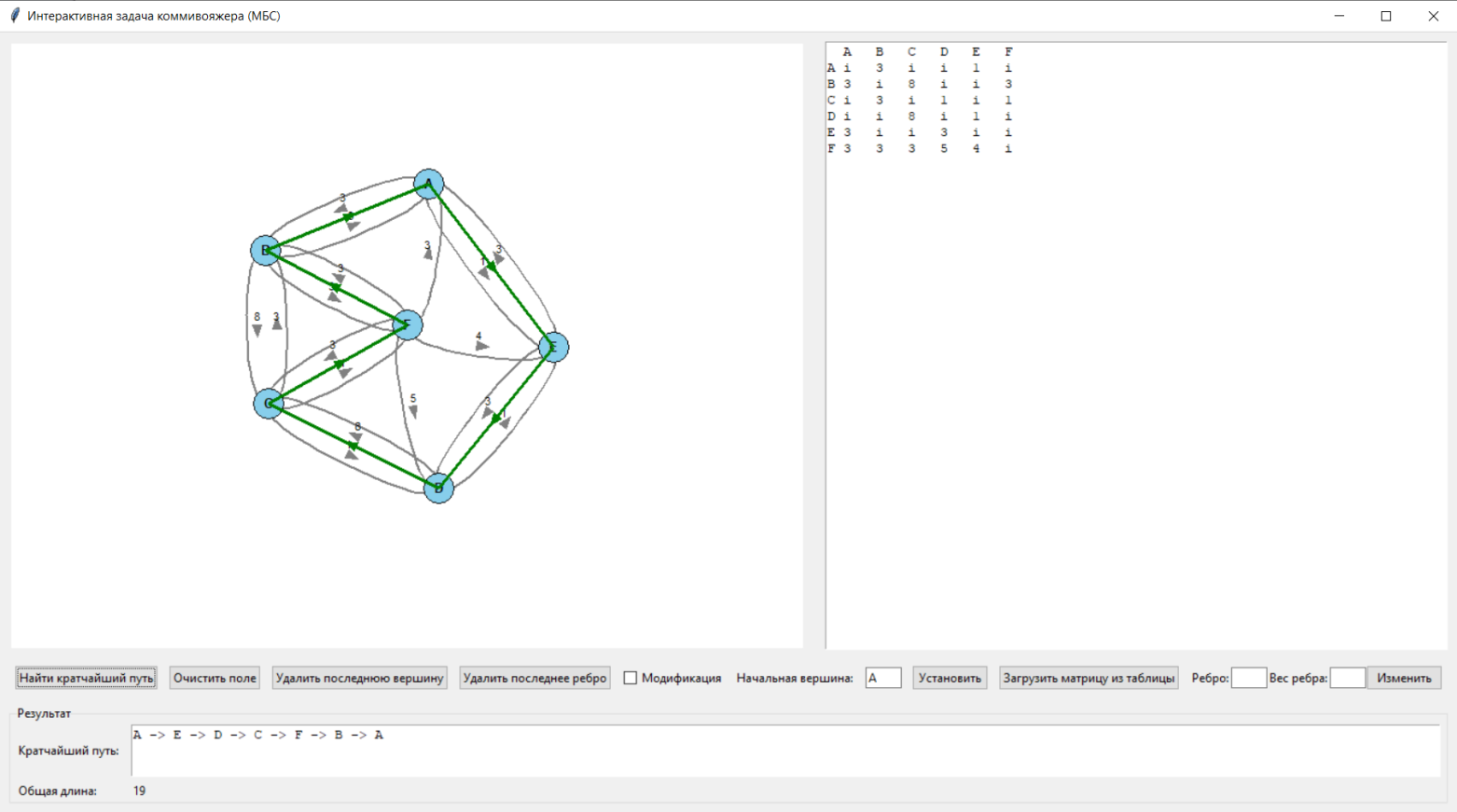


Рис 10.3 Графический интерфейс после выполнения программы без модификации.

4) Результат алгоритма с модификацией

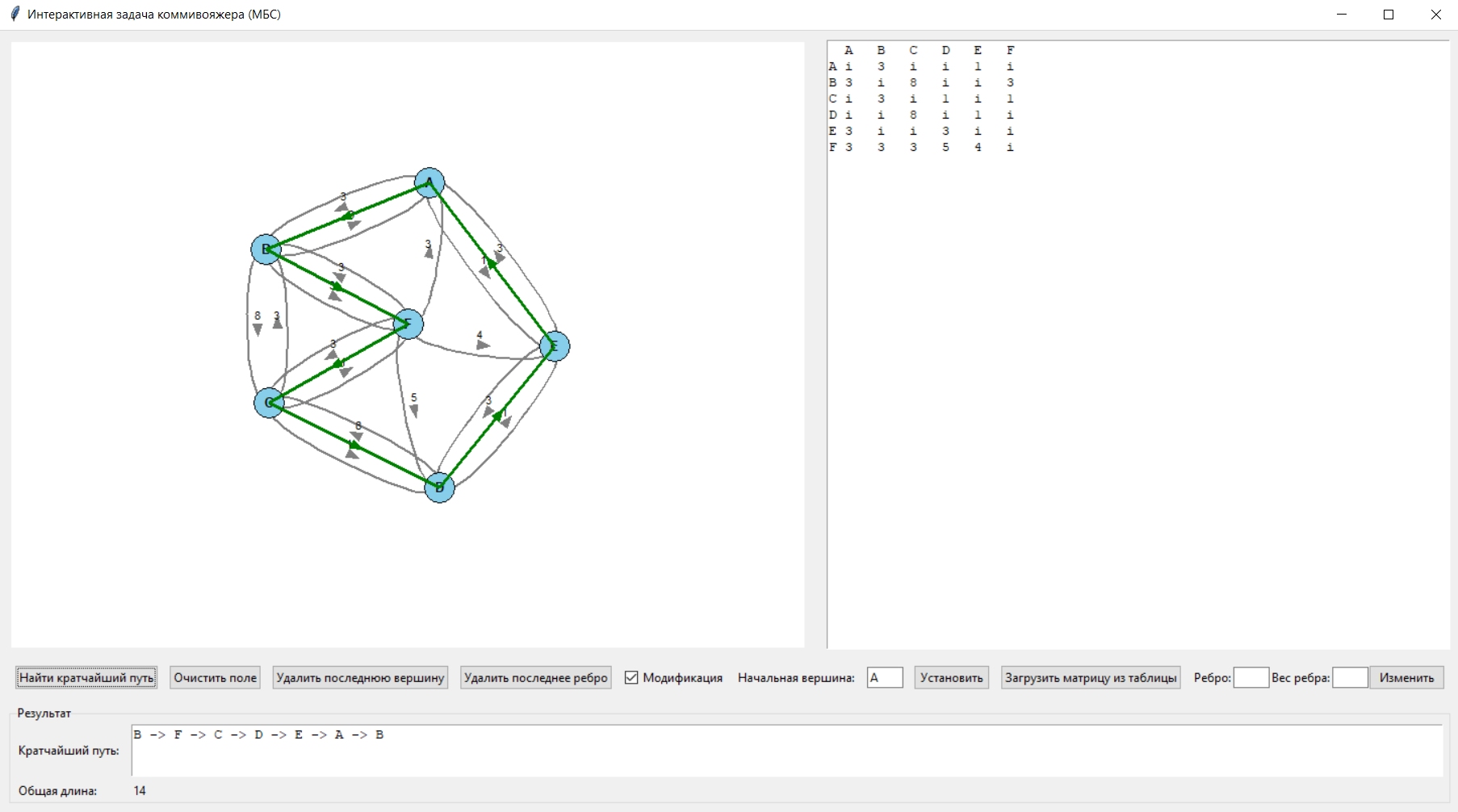


Рис 10.4 Графический интерфейс после выполнения программы c модификацией.

5) Сравнение эффективности алгоритма с модификацией и без (графы для анализа сгенерированы случайно)

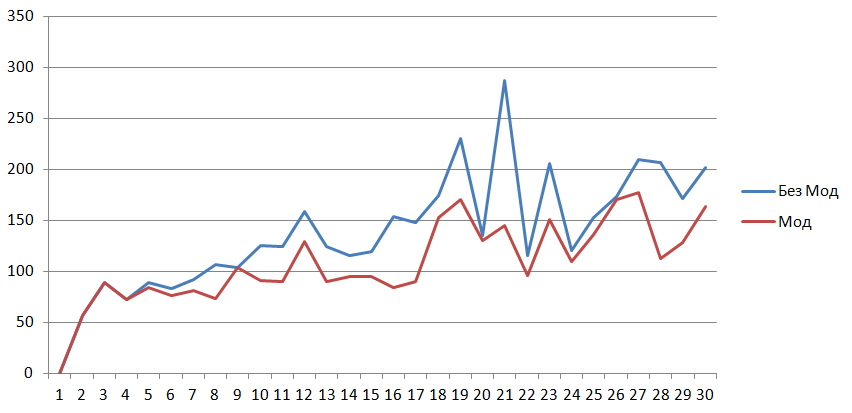


Рис 10.5 График зависимости длины пути от количества вершин

Сравнение показывает более высокую эффективность модификации ввиду нахождения более короткого пути почти во всех случаях.

# Вывод

В процессе исследования особенностей алгоритма ближайшего соседа для решения задачи коммивояжера на ориентированном графе был разработан метод поиска кратчайшего гамильтонова цикла. Написана соответствующая программа, способная находить гамильтонов цикл с использованием алгоритма ближайшего соседа и определять длину маршрута, охватывающего все вершины. Кроме того, была предложена модификация, включающая запуск алгоритма из всех вершин и выбор кратчайшего.

# Источники

# tkinter — tkinter documentation // Documentation URL: <https://docs.python.org/3/library/tkinter.html> (дата обращения: 21.03.2025).

# networkx — networkx documentation // Documentation URL: [https://networkx.org/documentation/latest/](https://networkx.org/documentation/latest/%20) (дата обращения: 21.03.2025).

# math — math documentation // Documentation URL: <https://docs.python.org/3/library/math.html> (дата обращения: 21.03.2025).

# pandas — pandas documentation // Documentation URL: [https://pandas.pydata.org/docs/](https://pandas.pydata.org/docs/%20) (дата обращения: 21.03.2025).

# random — random documentation // Documentation URL: <https://docs.python.org/3/library/random.html> (дата обращения: 21.03.2025).

# 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

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

    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

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 tsp\_solver import nearest\_neighbor

from tkinter import filedialog

import pandas as pd

import random

class TSPInteractiveGUI:

    def \_\_init\_\_(self, master):

        self.master = master

        master.title("Интерактивная задача коммивояжера (МБС)")

        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

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

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

        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 = 600

        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)

        run\_button = ttk.Button(control\_frame, text="Найти кратчайший путь", command=self.run\_all\_nearest\_neighbor)

        run\_button.pack(side=tk.LEFT, padx=5)

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

        clear\_button.pack(side=tk.LEFT, padx=5)

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

        delete\_vertex\_button.pack(side=tk.LEFT, padx=5)

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

        delete\_path\_button.pack(side=tk.LEFT, padx=5)

        self.modification\_var = tk.BooleanVar()

        modification\_check = ttk.Checkbutton(control\_frame, text="Модификация", variable=self.modification\_var)

        modification\_check.pack(side=tk.LEFT, padx=5)

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

        self.start\_vertex\_label.pack(side=tk.LEFT, padx=5)

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

        self.start\_vertex\_entry.pack(side=tk.LEFT, padx=5)

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

        self.set\_start\_vertex\_button.pack(side=tk.LEFT, padx=5)

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

        load\_matrix\_button.pack(side=tk.LEFT, padx=5)

        self.edge\_weight\_frame = ttk.Frame(control\_frame)

        self.edge\_weight\_frame.pack(side=tk.LEFT, padx=5)

        ttk.Label(self.edge\_weight\_frame, text="Ребро:").pack(side=tk.LEFT)

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

        self.edge\_entry.pack(side=tk.LEFT)

        ttk.Label(self.edge\_weight\_frame, text="Вес ребра:").pack(side=tk.LEFT)

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

        self.edge\_weight\_entry.pack(side=tk.LEFT)

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

        self.update\_edge\_weight\_button.pack(side=tk.LEFT)

    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)

        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.canvas.delete("all")

        self.update\_distance\_matrix()

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

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

        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

        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.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.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."""

        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]

                    if not isinstance(weight, str) and not 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))