**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 метода ближайшего соседа»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 23.Б16 |  | Барафанов А.А. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2025 г.**

**Оглавление**

[**1.** **Цель работы** 3](#_Toc193496828)

[**2.** **Задачи (формализация)** 3](#_Toc193496829)

[**3.** **Теоретическая часть** 4](#_Toc193496830)

[**4.** **Описание алгоритма** 4](#_Toc193496832)

[**5.** **Описание программы** 8](#_Toc193496833)

[**5.1 Описание функций** 8](#_Toc193496834)

[**6.** **Рекомендации пользователю** 9](#_Toc193496835)

[**7.** **Рекомендации программисту** 10](#_Toc193496851)

[**8.** **Контрольный пример** 11](#_Toc193496852)

[**9.** **Анализ результатов** 13](#_Toc193496853)

[**10.** **Вывод** 15](#_Toc193496854)

[**11.** **Листинг программы** 15](#_Toc193496856)

# **Цель работы**

Исследовать эффективность метода ближайшего соседа для решения задачи коммивояжера (TSP) и проверить его с учетом модификации. Разработать программный код, который позволит визуализировать граф и находить кратчайший гамильтонов цикл.

# **Задачи (формализация)**

* Реализовать алгоритм ближайшего соседа и его рандомизированную версию для поиска кратчайшего гамильтонова цикла в графе.
* Разработать графический интерфейс для ввода и визуализации графа, выбора стартовой вершины и отображения результатов работы алгоритма.
* Провести тестирование алгоритма на контрольном примере и сравнить результаты стандартного и рандомизированного методов.

# **Теоретическая часть**

# *Задача коммивояжера (TSP)* — это одна из классических задач теории графов, которая заключается в поиске кратчайшего гамильтонова цикла в графе. Гамильтонов цикл — это замкнутый путь, который проходит через каждую вершину графа ровно один раз и возвращается в исходную вершину.

*Метод ближайшего соседа* — это эвристический алгоритм для решения задачи коммивояжера. Он основан на жадной стратегии: на каждом шаге выбирается ближайшая непосещённая вершина, и так продолжается, пока все вершины не будут посещены. После этого алгоритм возвращается в начальную вершину, замыкая цикл.

Основные этапы работы алгоритма ближайшего соседа:

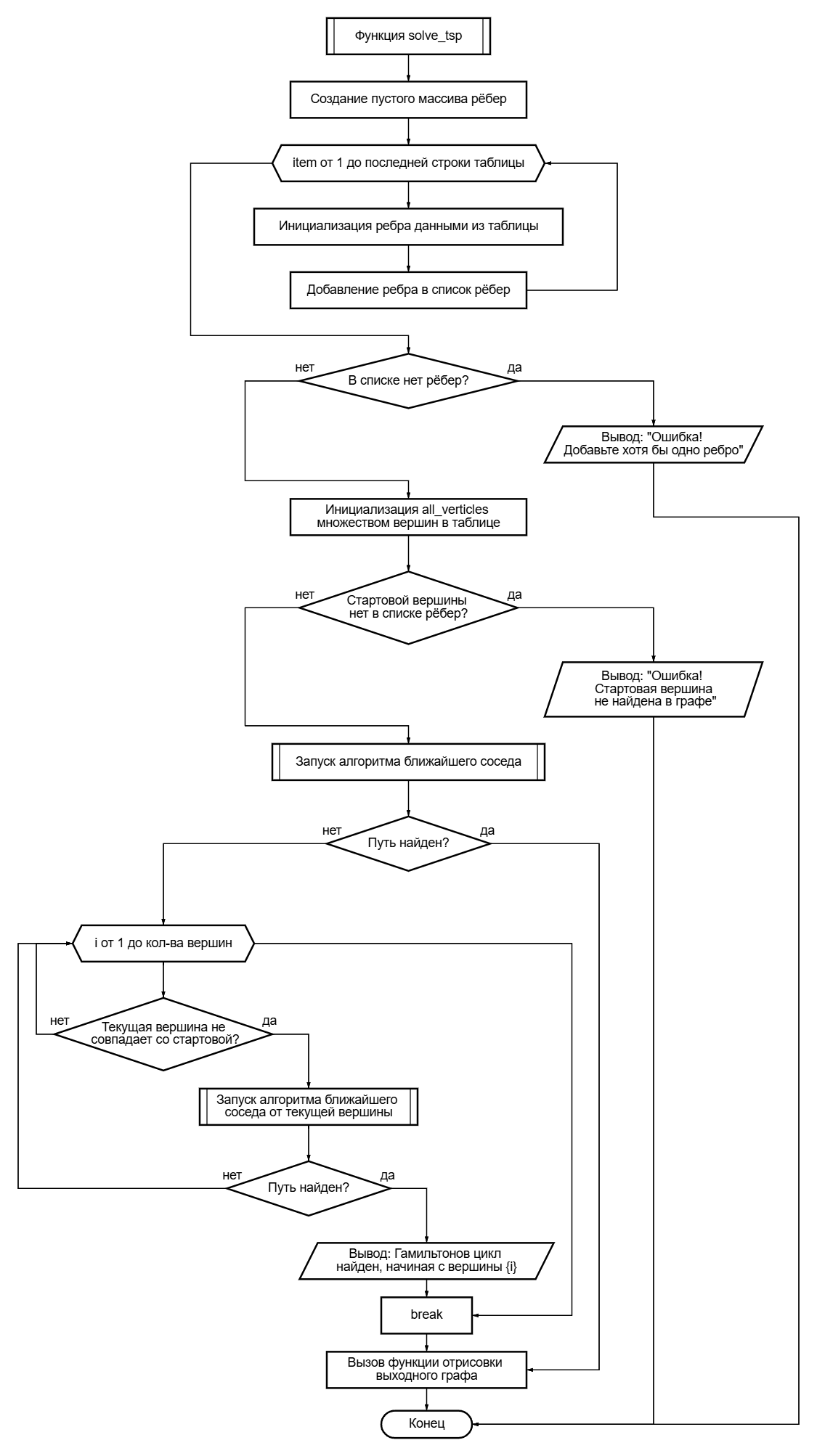
1. Выбор стартовой вершины.
2. На каждом шаге переход к ближайшей непосещённой вершине.
3. Повторение шага 2, пока все вершины не будут посещены.
4. Возврат в начальную вершину для завершения цикла.

*Рандомизированный метод ближайшего соседа* — это модификация стандартного алгоритма, в которой на каждом шаге с определённой вероятностью выбирается случайная вершина вместо ближайшей. Это позволяет увеличить разнообразие решений и избежать зацикливания на локальных минимумах.

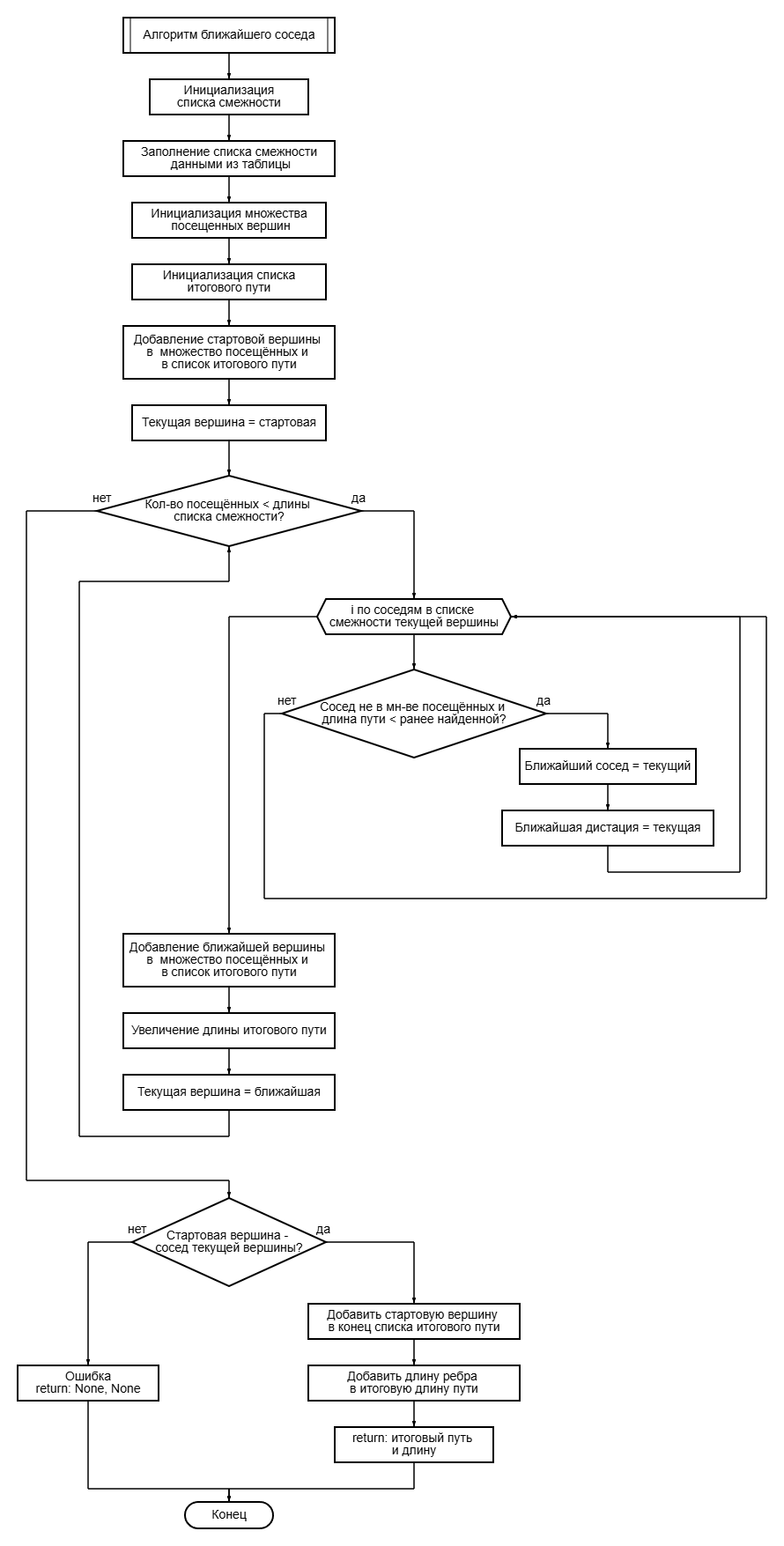
# **Описание алгоритма**

1. *Реализация алгоритма ближайшего соседа:*  
   Алгоритм строит маршрут, на каждом шаге выбирая ближайшую непосещённую вершину, пока не будут посещены все вершины, а затем возвращается в начальную точку.
2. *Реализация рандомизированного метода ближайшего соседа:*  
   Модификация алгоритма, где с заданной вероятностью выбирается случайная вершина вместо ближайшей (тем самым увеличивая разнообразие решений).
3. *Создание графического интерфейса:*  
   Разрабатывается интерфейс с холстами для отображения графов, таблицей для вывода рёбер и кнопками для управления алгоритмом.
4. *Обработка кликов на холсте:*  
   При клике на холст добавляются вершины (или рёбра), а также запрашивается длина ребра, в случае если пользователь вручную задает граф.
5. *Отрисовка входного графа:*  
   Функция draw\_input\_graph отображает вершины и рёбра на холсте, включая подписи длин рёбер.
6. *Отрисовка выходного графа:*  
   Функция draw\_output\_graph визуализирует найденный гамильтонов цикл, (выделен зелёным цветом).
7. *Решение задачи коммивояжера:*  
   Функция solve\_tsp запускает алгоритм ближайшего соседа, проверяет корректность введённых данных и выводит результат (путь и длину цикла).
8. *Очистка графов:*  
   Функция clear\_canvas очищает холсты, таблицу рёбер и результаты, позволяя начать работу с новым графом.

*Блок-схемы, иллюстрирующие шаги алгоритма показаны на рисунках 4.1-4.2*

**

*Рисунок 4.1. Блок-схема основной функции solve\_tsp*

**

*Рисунок 4.2. Блок-схема алгоритма ближайшего соседа*

# **Описание программы**

## **5.1 Описание функций**

В программе используется 10 функций, выполняющие вспомогательные процедуры программы. В таблице 5.1.1 представлено их описание.

*Таблица 5.1.1. Описание функций*

|  |  |  |
| --- | --- | --- |
| Имя функции | Входные данные | Описание функции |
| nearest\_neighbor\_tsp | `graph` (DataFrame), `start\_vertex` (строка) | Реализует алгоритм ближайшего соседа для задачи коммивояжера. Возвращает путь и общее расстояние. |
| randomized\_nearest\_neighbor\_tsp | `graph` (DataFrame), `start\_vertex` (строка), `randomness` (float, опционально) | Рандомизированная версия алгоритма ближайшего соседа. Добавляет случайность в выбор следующей вершины. |
| add\_edge\_to\_table | `v1` (строка), `v2` (строка), `length` (число) | Добавляет ребро в таблицу `tree` для отображения в интерфейсе. |
| on\_click | `event` (событие мыши) | Обрабатывает клик на холсте: добавляет вершину или ребро в граф. |
| draw\_input\_graph | Нет | Отрисовывает входной граф на холсте `input\_canvas`. |
| draw\_output\_graph | `path` (список вершин) | Отрисовывает выходной граф (решение) на холсте `output\_canvas`. |
| solve\_tsp | Нет | Решает задачу коммивояжера с помощью алгоритма ближайшего соседа и отображает результат. |
| solve\_randomized\_tsp | Нет | Решает задачу коммивояжера с помощью рандомизированного алгоритма ближайшего соседа. |
| clear\_canvas | Нет | Очищает холсты, таблицу и сбрасывает все данные графа. |
| set\_graph\_type | `type` (строка: "undirected" или "directed") | Устанавливает тип графа (ориентированный или неориентированный). |
| load\_graph\_from\_file | Нет | Загружает граф из файла (формата csv) |

# **Рекомендации пользователю**

*Запуск программы:*

* Программа запускается с помощью интерпретатора Python. Убедитесь, что у вас установлены все необходимые библиотеки: tkinter, pandas, random.

*Настройка графа:*

* Добавьте вершины, кликая на холст. Для добавления рёбер выберите две вершины и введите длину ребра во всплывающем окне.
* Выберите тип графа (ориентированный или неориентированный) с помощью радиокнопки в интерфейсе.

*Запуск алгоритма:*

* Введите стартовую вершину в поле "Стартовая вершина".
* Нажмите кнопку "Рассчитать (обычный)" для запуска стандартного алгоритма ближайшего соседа.
* Нажмите кнопку "Рассчитать (рандомизированный)" для запуска рандомизированной версии алгоритма, где следующий сосед выбирается с некоторой вероятностью случайно.

*Интерпретация результатов:*

* Результат отображается в виде пути (последовательности вершин) и общей длины маршрута.
* На холсте output\_canvas отрисовывается найденный маршрут.

*Советы по использованию:*

* Попробуйте разные стартовые вершины, чтобы найти оптимальный маршрут.
* Если граф не содержит гамильтонова цикла, программа выдаст предупреждение. В этом случае проверьте корректность введённых данных.

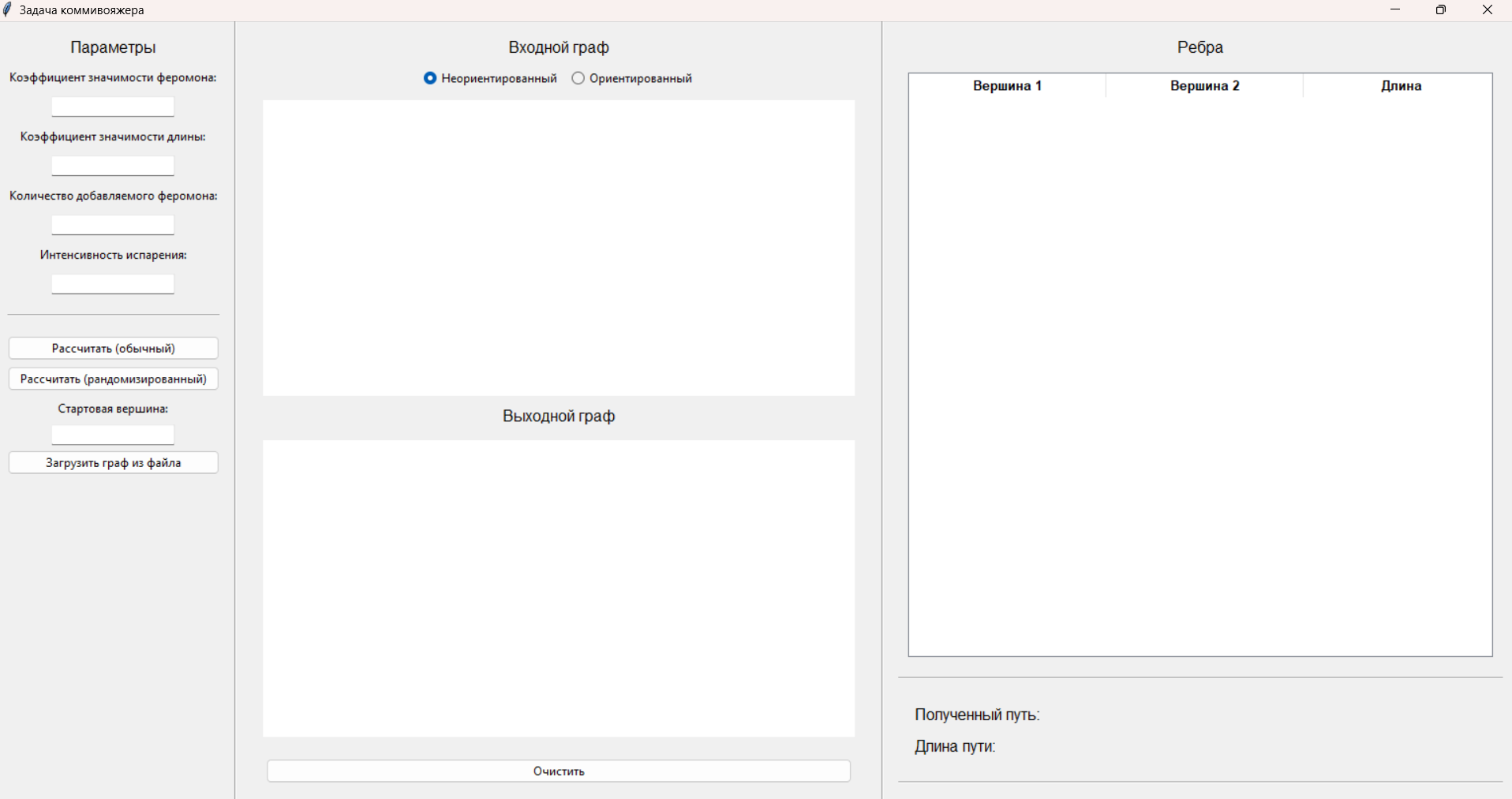
1. **Рекомендации программисту**

* Для работы с программой убедитесь, что установлены все необходимые зависимости: tkinter, pandas и random.
* Запускайте программу через командную строку или IDE (например, PyCharm, VSCode) для удобства отладки и отслеживания ошибок.
* Программа реализует алгоритм ближайшего соседа для задачи коммивояжера, а также его рандомизированную версию. Для модификации алгоритма ближайшего соседа измените логику выбора следующей вершины в функции nearest\_neighbor\_tsp.
* В рандомизированной версии алгоритма (randomized\_nearest\_neighbor\_tsp) параметр randomness определяет вероятность выбора случайного соседа. Экспериментируйте с ним, устанавливая значения от 0.3 до 0.7, чтобы найти баланс между случайностью и жадным выбором. Для задач с большим количеством локальных минимумов увеличьте значение randomness, а для более «жадного» поведения уменьшите его.
* При работе с графом вершины добавляются кликом на холст, а рёбра создаются между двумя вершинами с указанием длины. Если нужно программно добавить вершины и рёбра, модифицируйте глобальные переменные vertices и edges или добавьте соответствующие функции.
* Алгоритм чувствителен к выбору стартовой вершины, поэтому проверьте его работу с разными стартовыми вершинами, чтобы найти оптимальное решение.

1. **Контрольный пример**

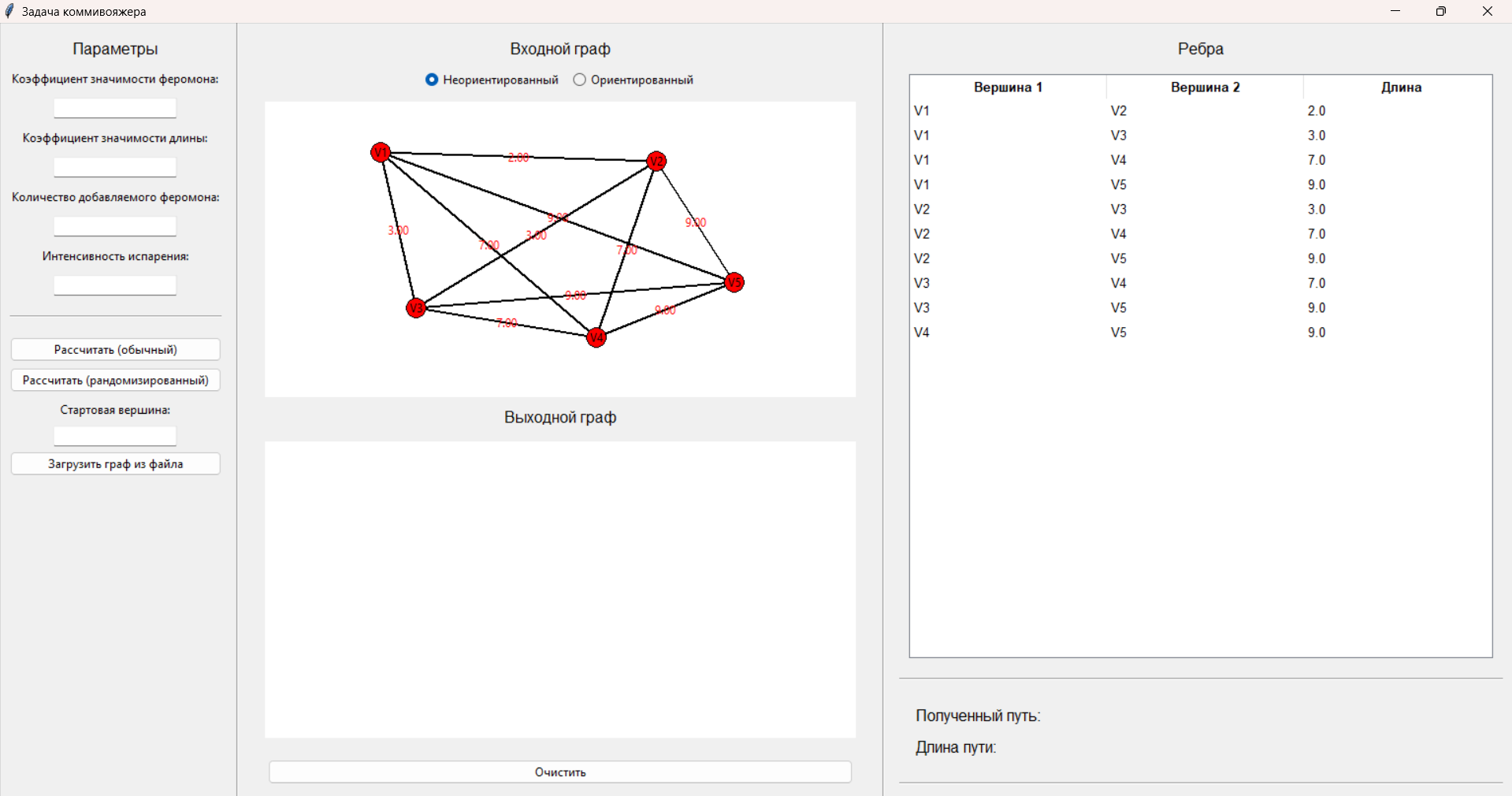
В этом разделе продемонстрирован пример запуска алгоритма ближайшего соседа на примере заданного неориентированного графа.

Тонкости взаимодействия с программой указаны в рекомендациях пользователю, здесь же приведена демонстрация работы с программой.



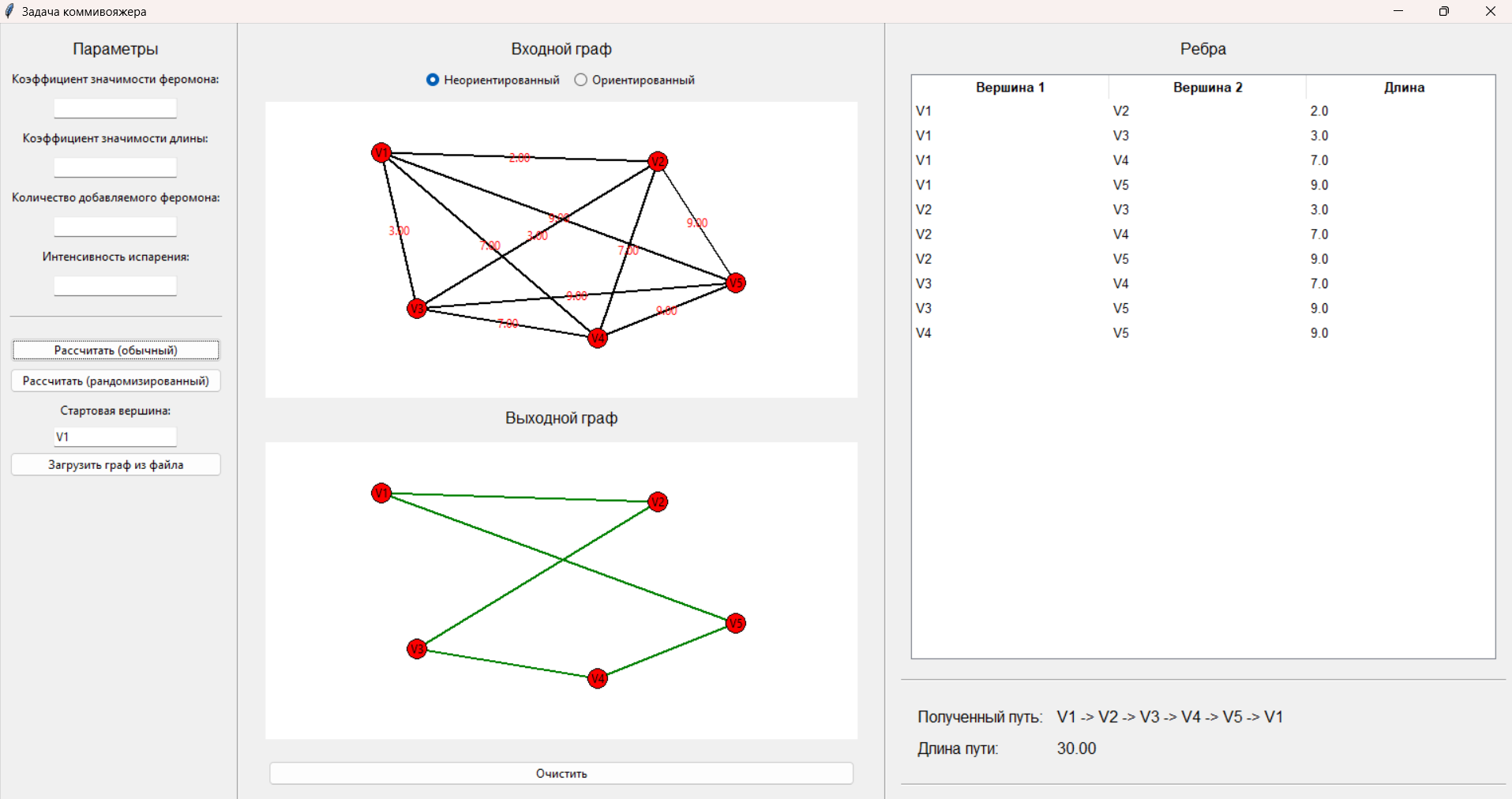
*Рисунок 8.1 Начальный интерфейс программы*

Для ввода вершин графа достаточно кликнуть мышью в нужное место верхнего холста, а для ввода ребра нужно последовательно кликнуть на две уже существующие вершины и ввести длину в появившемся диалоговом окне. На рисунке 8.2 показан пример введенного графа и соответствующая ему таблица ребер.



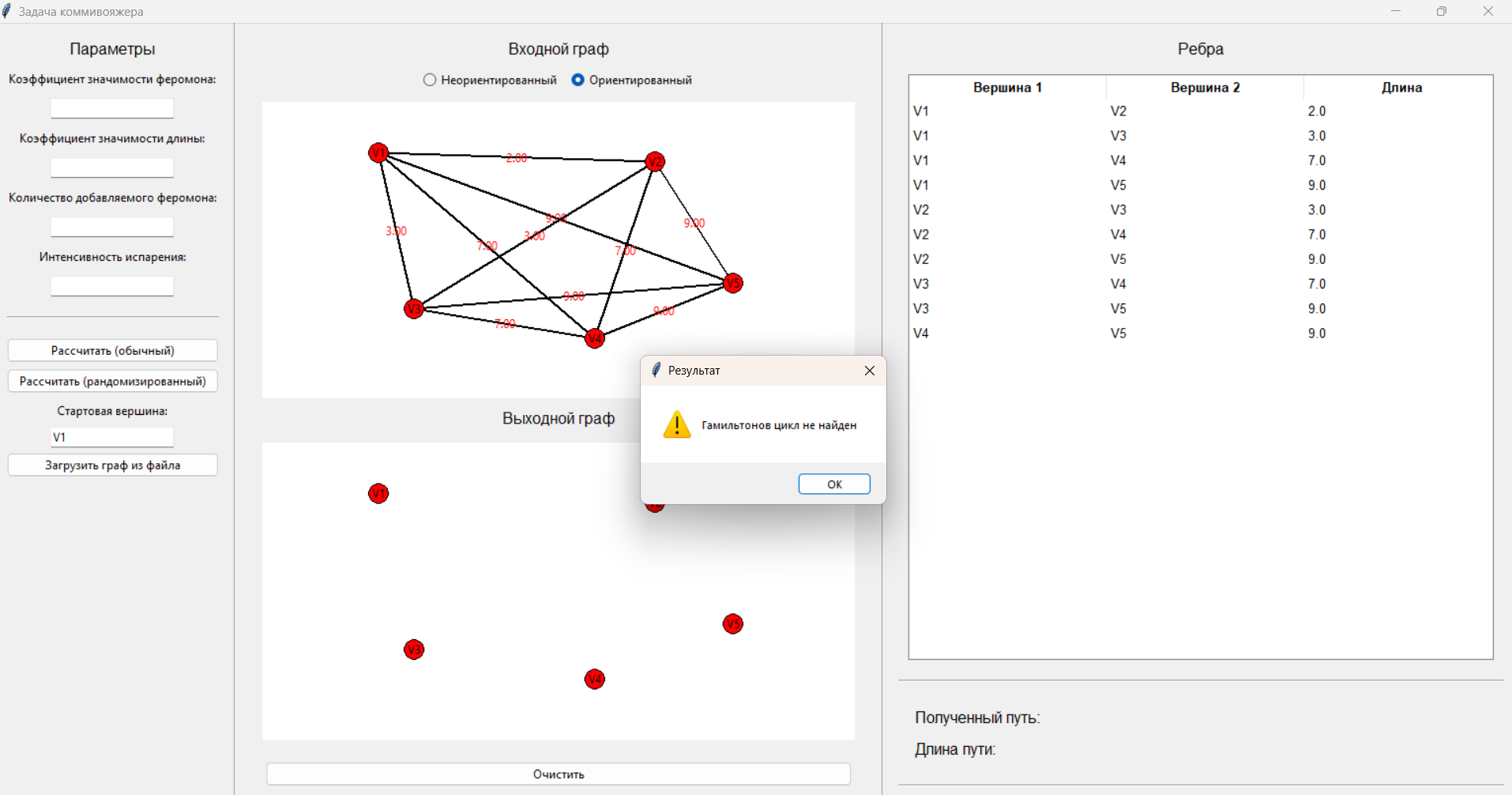
*Рисунок 8.2 Окно программы после ввода графа*

Для получения результата необходимо после ввода графа выбрать стартовую вершину и нажать на кнопку «Рассчитать» (в зависимости от ваших целей можете выбрать либо расчет по классическому алгоритму, либо по модифицированной версии). Результат появится в нижнем правом углу окна.

**

*Рисунок 8.3 Окно программы после нажатия кнопки «Рассчитать»*

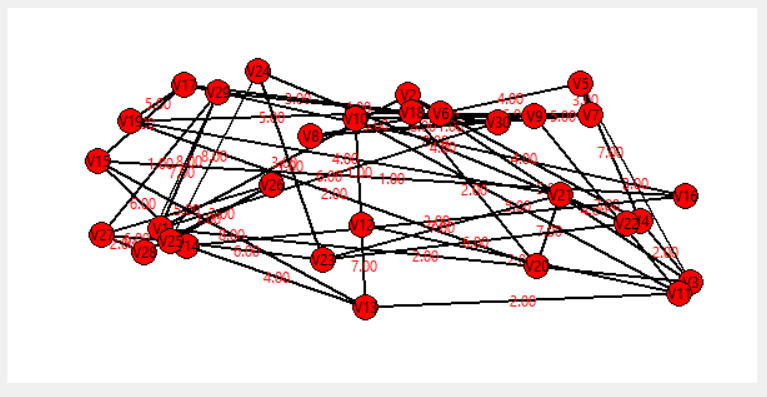
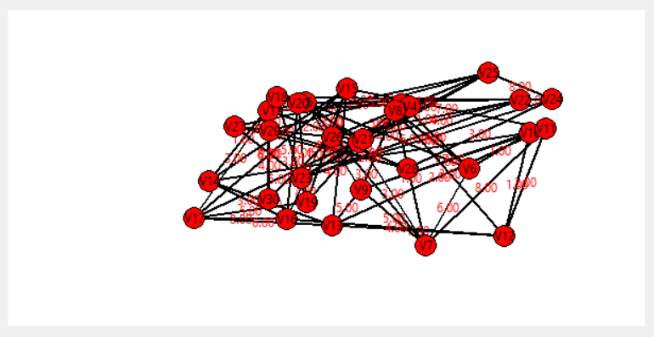
Вы также можете изменить тип графа, нажав на соответствующую радиокнопку. Если алгоритм не найдет гамильтонов цикл, то вы увидите предупреждение во всплывающем окне. В таком случае проверьте корректность введенных данных и выбранный тип графа.

**

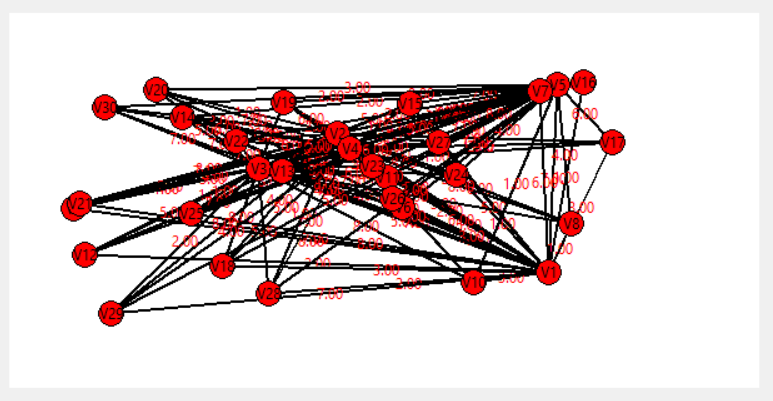
*Рисунок 8.4 Результат запуска при смене типа графа на «Ориентированный»*

# **Анализ результатов**

Запустим алгоритм (классический и с модификацией) на трех тестовых графах (их картинки можно увидеть ниже). Каждый из них состоит из 30 вершин, но имеет разную конфигурацию ребер.



*Рисунок 9.1. Входной граф №1 Рисунок 9.2. Входной граф №2*



*Рисунок 9.3. Входной граф №3*

Результаты работы после пяти запусков можно представить в виде следующей таблицы:

*Таблица 9.1. Анализ результатов*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер графа | Кол-во ребер | Лучшее решение классического алгоритма | Время выполнения (классич-кий алгоритм) | Лучшее решение рандом-нной модификации | Время выполнения (модификация) |
| 1 | 52 | 93 | 0.0035 сек | 106 | 0.0043 сек |
| 2 | 103 | Нет | - | 96 | 0.0065 сек |
| 3 | 135 | Нет | - | Нет | - |

1. Граф 1:  
   Классический алгоритм нашёл решение с длиной 93, что лучше, чем результат модификации (106). То есть для данного графа (с относительно небольшим количеством ребер) жадный подход оказался эффективнее (из-за отсутствия сложных циклов и локальных минимумов).
2. Граф 2:  
   Классический алгоритм не смог найти решение, что может быть связано с высокой сложностью графа (103 ребра). Рандомизированный алгоритм же нашёл решение с длиной 96, что подтверждает его способность находить пути в более сложных графах благодаря элементу случайности в выборе пути.
3. Граф 3:  
   Ни один из алгоритмов не смог найти решение, что указывает на очень высокую сложность графа (135 рёбер). Обе модификации алгоритма ближайшего соседа не могут найти гамильтонов цикл, а «уходят в дебри» реберной структуры этого графа.
4. Итого, можно сделать вывод:  
   Классический алгоритм демонстрирует хорошие результаты на простых графах, но может застревать в локальных минимумах (или вовсе не находить решение) на более сложных. Рандомизированный алгоритм, напротив, показывает свою эффективность на более сложных графах, но его результаты могут быть менее точными. Для графов с очень высокой сложностью (граф 3) требуется использование более мощных алгоритмов.
5. **Вывод**

В ходе выполнения лабораторной работы был реализован алгоритм ближайшего соседа для решения задачи коммивояжера, а также его модификация с использованием рандомизированного выбора соседа. А также был проведён анализ результатов работы обоих алгоритмов. Для удобства работы с алгоритмами была разработана программа с графическим интерфейсом пользователя, которая позволяет визуализировать граф, задавать вершины и рёбра, а также отображать найденные решения TSP.

# **Листинг программы**

import tkinter as tk

from tkinter import ttk, messagebox, simpledialog, filedialog

import pandas as pd

import random

​

# Глобальные переменные для хранения вершин и рёбер

vertices = {}

edges = []

current\_edge = None

graph\_type = "undirected"

​

def nearest\_neighbor\_tsp(graph, start\_vertex):

   adjacency\_list = {}

   for \_, row in graph.iterrows():

       v1, v2, length = row['вершина 1'], row['вершина 2'], float(row['длина'])

       if v1 not in adjacency\_list:

           adjacency\_list[v1] = {}

       if v2 not in adjacency\_list:

           adjacency\_list[v2] = {}

       adjacency\_list[v1][v2] = length

       if graph\_type == "undirected":

           adjacency\_list[v2][v1] = length

​

   visited = set()

   path = [start\_vertex]

   visited.add(start\_vertex)

   total\_distance = 0

​

   current\_vertex = start\_vertex

​

   while len(visited) < len(adjacency\_list):

       nearest\_vertex = None

       nearest\_distance = float('inf')

​

       for neighbor, distance in adjacency\_list[current\_vertex].items():

           if neighbor not in visited and distance < nearest\_distance:

               nearest\_vertex = neighbor

               nearest\_distance = distance

​

       if nearest\_vertex is None:

           break

​

       path.append(nearest\_vertex)

       visited.add(nearest\_vertex)

       total\_distance += nearest\_distance

       current\_vertex = nearest\_vertex

​

   if len(path) == len(adjacency\_list) and start\_vertex in adjacency\_list[current\_vertex]:

       path.append(start\_vertex)

       total\_distance += adjacency\_list[current\_vertex][start\_vertex]

       return path, total\_distance

   else:

       return None, None

​

​

def randomized\_nearest\_neighbor\_tsp(graph, start\_vertex, randomness=0.5):

   adjacency\_list = {}

   for \_, row in graph.iterrows():

       v1, v2, length = row['вершина 1'], row['вершина 2'], float(row['длина'])

       if v1 not in adjacency\_list:

           adjacency\_list[v1] = {}

       if v2 not in adjacency\_list:

           adjacency\_list[v2] = {}

       adjacency\_list[v1][v2] = length

       if graph\_type == "undirected":

           adjacency\_list[v2][v1] = length

​

   visited = set()

   path = [start\_vertex]

   visited.add(start\_vertex)

   total\_distance = 0

​

   current\_vertex = start\_vertex

​

   while len(visited) < len(adjacency\_list):

       neighbors = [neighbor for neighbor in adjacency\_list[current\_vertex].items() if neighbor[0] not in visited]

       if not neighbors:

           break

​

       # С вероятностью randomness выбираем случайного соседа

       if random.random() < randomness:

           nearest\_vertex, nearest\_distance = random.choice(neighbors)

       else:

           nearest\_vertex, nearest\_distance = min(neighbors, key=lambda x: x[1])

​

       path.append(nearest\_vertex)

       visited.add(nearest\_vertex)

       total\_distance += nearest\_distance

       current\_vertex = nearest\_vertex

​

   if len(path) == len(adjacency\_list) and start\_vertex in adjacency\_list[current\_vertex]:

       path.append(start\_vertex)

       total\_distance += adjacency\_list[current\_vertex][start\_vertex]

       return path, total\_distance

   else:

       return None, None

​

​

def add\_edge\_to\_table(v1, v2, length):

   tree.insert("", "end", values=(v1, v2, length, ""))

​

​

def on\_click(event):

   global current\_edge

​

   for vertex, (x, y) in vertices.items():

       if abs(event.x - x) < 10 and abs(event.y - y) < 10:  # 10 пикселей

           if current\_edge is None:

               current\_edge = vertex

           else:

               length = simpledialog.askfloat("Длина ребра", f"Введите длину ребра между {current\_edge} и {vertex}:")

               if length is not None:

                   edges.append((current\_edge, vertex, length))

                   add\_edge\_to\_table(current\_edge, vertex, length)

                   current\_edge = None

                   draw\_input\_graph()

           break

   else:

       vertex = f"V{len(vertices) + 1}"

       vertices[vertex] = (event.x, event.y)

       draw\_input\_graph()

​

​

def draw\_input\_graph():

   input\_canvas.delete("all")

​

   for v1, v2, length in edges:

       x1, y1 = vertices[v1]

       x2, y2 = vertices[v2]

       input\_canvas.create\_line(x1, y1, x2, y2, width=2, fill="black")

       input\_canvas.create\_text((x1 + x2) / 2, (y1 + y2) / 2, text=f"{length:.2f}", fill="red")

​

   for vertex, (x, y) in vertices.items():

       input\_canvas.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="red")

       input\_canvas.create\_text(x, y, text=vertex, fill="black")

​

​

def draw\_output\_graph(path):

   output\_canvas.delete("all")

​

   for i in range(len(path) - 1):

       v1, v2 = path[i], path[i + 1]

       x1, y1 = vertices[v1]

       x2, y2 = vertices[v2]

       output\_canvas.create\_line(x1, y1, x2, y2, width=2, fill="green")

​

   for vertex, (x, y) in vertices.items():

       output\_canvas.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="red")

       output\_canvas.create\_text(x, y, text=vertex, fill="black")

​

​

def solve\_tsp():

   edges\_table = []

   for item in tree.get\_children():

       edge = tree.item(item, "values")

       edges\_table.append(edge)

​

   if not edges\_table:

       messagebox.showwarning("Ошибка", "Добавьте хотя бы одно ребро!")

       return

​

   graph = pd.DataFrame(edges\_table, columns=["вершина 1", "вершина 2", "длина", "феромон"])

​

   start\_vertex = entry\_start.get().strip()

   if not start\_vertex:

       messagebox.showwarning("Ошибка", "Введите стартовую вершину!")

       return

​

   all\_vertices = set(graph['вершина 1']).union(set(graph['вершина 2']))

   if start\_vertex not in all\_vertices:

       messagebox.showwarning("Ошибка", f"Стартовая вершина '{start\_vertex}' не найдена в графе!")

       return

​

   path, total\_distance = nearest\_neighbor\_tsp(graph, start\_vertex)

​

   if path is None:

       for vertex in all\_vertices:

           if vertex != start\_vertex:

               path, total\_distance = nearest\_neighbor\_tsp(graph, vertex)

               if path is not None:

                   messagebox.showinfo("Результат", f"Гамильтонов цикл найден, начиная с вершины {vertex}")

                   break

​

   if path is None:

       messagebox.showwarning("Результат", "Гамильтонов цикл не найден")

       result\_path.config(text="")

       result\_distance.config(text="")

       draw\_output\_graph([])

   else:

       result\_path.config(text=f"{' -> '.join(path)}")

       result\_distance.config(text=f"{total\_distance:.2f}")

       draw\_output\_graph(path)

​

​

def solve\_randomized\_tsp():

   edges\_table = []

   for item in tree.get\_children():

       edge = tree.item(item, "values")

       edges\_table.append(edge)

​

   if not edges\_table:

       messagebox.showwarning("Ошибка", "Добавьте хотя бы одно ребро!")

       return

​

   graph = pd.DataFrame(edges\_table, columns=["вершина 1", "вершина 2", "длина", "феромон"])

​

   start\_vertex = entry\_start.get().strip()

   if not start\_vertex:

       messagebox.showwarning("Ошибка", "Введите стартовую вершину!")

       return

​

   all\_vertices = set(graph['вершина 1']).union(set(graph['вершина 2']))

   if start\_vertex not in all\_vertices:

       messagebox.showwarning("Ошибка", f"Стартовая вершина '{start\_vertex}' не найдена в графе!")

       return

​

   path, total\_distance = randomized\_nearest\_neighbor\_tsp(graph, start\_vertex, randomness=0.5)

​

   # Если гамильтонов цикл не найден, запускаем от других вершин

   if path is None:

       for vertex in all\_vertices:

           if vertex != start\_vertex:

               path, total\_distance = randomized\_nearest\_neighbor\_tsp(graph, vertex, randomness=0.5)

               if path is not None:

                   messagebox.showinfo("Результат", f"Гамильтонов цикл найден, начиная с вершины {vertex}")

                   break

​

   if path is None:

       messagebox.showwarning("Результат", "Гамильтонов цикл не найден")

       result\_path.config(text="")

       result\_distance.config(text="")

       draw\_output\_graph([])

   else:

       result\_path.config(text=f"{' -> '.join(path)}")

       result\_distance.config(text=f"{total\_distance:.2f}")

       draw\_output\_graph(path)

​

​

def clear\_canvas():

   global vertices, edges, current\_edge

   vertices = {}

   edges = []

   current\_edge = None

   input\_canvas.delete("all")

   output\_canvas.delete("all")

   tree.delete(\*tree.get\_children())

   result\_path.config(text="")

   result\_distance.config(text="")

   draw\_input\_graph()

   draw\_output\_graph([])

​

​

def set\_graph\_type(type):

   global graph\_type

   graph\_type = type

   messagebox.showinfo("Тип графа", f"Тип графа изменён на {'неориентированный' if type == 'undirected' else 'ориентированный'}")

​

def load\_graph\_from\_file():

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

   if not file\_path:

       return

​

   try:

       df = pd.read\_csv(file\_path)

       if not all(col in df.columns for col in ["вершина 1", "вершина 2", "длина"]):

           messagebox.showerror("Ошибка", "Файл должен содержать колонки 'вершина 1', 'вершина 2', 'длина'")

           return

​

       clear\_canvas()

       for \_, row in df.iterrows():

           v1, v2, length = row['вершина 1'], row['вершина 2'], float(row['длина'])

           if v1 not in vertices:

               vertices[v1] = (random.randint(50, 550), random.randint(50, 250))

           if v2 not in vertices:

               vertices[v2] = (random.randint(50, 550), random.randint(50, 250))

           edges.append((v1, v2, length))

           add\_edge\_to\_table(v1, v2, length)

​

       draw\_input\_graph()

   except Exception as e:

       messagebox.showerror("Ошибка", f"Не удалось загрузить файл: {e}")

​

​

root = tk.Tk()

root.title("Задача коммивояжера")

​

# Левый фрейм для параметров и кнопок

left\_frame = ttk.Frame(root)

left\_frame.pack(side=tk.LEFT, fill=tk.BOTH, padx=10, pady=10)

​

label\_parameters = ttk.Label(left\_frame, text="Параметры", font=("Arial", 12))

label\_parameters.pack(pady=5)

​

label\_pheromone = ttk.Label(left\_frame, text="Коэффициент значимости феромона:")

label\_pheromone.pack(pady=5)

entry\_pheromone = ttk.Entry(left\_frame)

entry\_pheromone.pack(pady=5)

​

label\_length = ttk.Label(left\_frame, text="Коэффициент значимости длины:")

label\_length.pack(pady=5)

entry\_length = ttk.Entry(left\_frame)

entry\_length.pack(pady=5)

​

label\_add\_pheromone = ttk.Label(left\_frame, text="Количество добавляемого феромона:")

label\_add\_pheromone.pack(pady=5)

entry\_add\_pheromone = ttk.Entry(left\_frame)

entry\_add\_pheromone.pack(pady=5)

​

label\_evaporation = ttk.Label(left\_frame, text="Интенсивность испарения:")

label\_evaporation.pack(pady=5)

entry\_evaporation = ttk.Entry(left\_frame)

entry\_evaporation.pack(pady=5)

​

separator\_count = ttk.Separator(left\_frame, orient=tk.HORIZONTAL)

separator\_count.pack(fill=tk.X, pady=15)

​

button\_solve = ttk.Button(left\_frame, text="Рассчитать (обычный)", command=solve\_tsp)

button\_solve.pack(fill=tk.BOTH, pady=5)

​

button\_solve\_randomized = ttk.Button(left\_frame, text="Рассчитать (рандомизированный)", command=solve\_randomized\_tsp)

button\_solve\_randomized.pack(fill=tk.BOTH, pady=1)

​

label\_start = ttk.Label(left\_frame, text="Стартовая вершина:")

label\_start.pack(pady=7)

entry\_start = ttk.Entry(left\_frame)

entry\_start.pack()

​

button\_load = ttk.Button(left\_frame, text="Загрузить граф из файла", command=load\_graph\_from\_file)

button\_load.pack(fill=tk.BOTH, pady=5)

​

separator\_left = ttk.Separator(root, orient=tk.VERTICAL)

separator\_left.pack(side=tk.LEFT, fill=tk.Y, padx=5)

​

# Центральный фрейм для холстов

center\_frame = ttk.Frame(root)

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

​

label\_input\_graph = ttk.Label(center\_frame, text="Входной граф", font=("Arial", 12))

label\_input\_graph.pack(pady=5)

​

graph\_type\_frame = ttk.Frame(center\_frame)

graph\_type\_frame.pack(pady=5)

​

graph\_type\_var = tk.StringVar(value="undirected")

radio\_undirected = ttk.Radiobutton(graph\_type\_frame, text="Неориентированный", variable=graph\_type\_var, value="undirected", command=lambda: set\_graph\_type("undirected"))

radio\_undirected.pack(side=tk.LEFT, padx=5)

radio\_directed = ttk.Radiobutton(graph\_type\_frame, text="Ориентированный", variable=graph\_type\_var, value="directed", command=lambda: set\_graph\_type("directed"))

radio\_directed.pack(side=tk.LEFT, padx=5)

​

input\_canvas = tk.Canvas(center\_frame, width=600, height=300, bg="white")

input\_canvas.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)

​

label\_output\_graph = ttk.Label(center\_frame, text="Выходной граф", font=("Arial", 12))

label\_output\_graph.pack(pady=2)

​

output\_canvas = tk.Canvas(center\_frame, width=600, height=300, bg="white")

output\_canvas.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

​

control\_frame = ttk.Frame(center\_frame)

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

​

button\_clear = ttk.Button(control\_frame, text="Очистить", command=clear\_canvas)

button\_clear.pack(fill=tk.BOTH, padx=5)

​

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

​

separator\_right = ttk.Separator(root, orient=tk.VERTICAL)

separator\_right.pack(side=tk.LEFT, fill=tk.Y, padx=5)

​

# Правый фрейм для таблицы и результатов

right\_frame = ttk.Frame(root, width=300)

right\_frame.pack(side=tk.RIGHT, fill=tk.BOTH, padx=10, pady=10)

​

label\_edges = ttk.Label(right\_frame, text="Ребра", font=("Arial", 12))

label\_edges.pack(pady=5)

​

style = ttk.Style()

style.layout("my.Treeview", [

  ("Treeview.field", {"sticky": "nswe", "border": "1", "children": [

      ("Treeview.padding", {"sticky": "nswe", "children": [

          ("Treeview.treearea", {"sticky": "nswe"})

      ]})

  ]})

])

style.configure("my.Treeview", font=("Arial", 10), rowheight=25, borderwidth=1, relief="solid")

style.configure("my.Treeview.Heading", font=("Arial", 10, "bold"), background="gray")

style.map("my.Treeview", background=[("selected", "#347083")])

​

tree = ttk.Treeview(right\_frame, columns=("вершина 1", "вершина 2", "длина", "феромон"), show="headings", style="my.Treeview")

tree.heading("вершина 1", text="Вершина 1")

tree.heading("вершина 2", text="Вершина 2")

tree.heading("длина", text="Длина")

tree.heading("феромон", text="Феромон")

tree.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

​

separator\_results\_top = ttk.Separator(right\_frame, orient=tk.HORIZONTAL)

separator\_results\_top.pack(fill=tk.X, pady=10)

​

result\_frame = ttk.Frame(right\_frame)

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

​

label\_path = ttk.Label(result\_frame, text="Полученный путь:", font=("Arial", 12))

label\_path.grid(row=0, column=0, padx=5, pady=5, sticky="w")

result\_path = ttk.Label(result\_frame, text="", font=("Arial", 12), wraplength=400)

result\_path.grid(row=0, column=1, padx=5, pady=5, sticky="w")

​

label\_distance = ttk.Label(result\_frame, text="Длина пути:", font=("Arial", 12))

label\_distance.grid(row=1, column=0, padx=5, pady=5, sticky="w")

result\_distance = ttk.Label(result\_frame, text="", font=("Arial", 12))

result\_distance.grid(row=1, column=1, padx=5, pady=5, sticky="w")

​

separator\_results\_bottom = ttk.Separator(right\_frame, orient=tk.HORIZONTAL)

separator\_results\_bottom.pack(fill=tk.X, pady=10)

​root.mainloop()