**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 алгоритма имитации отжига»**

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

**Санкт-Петербург**

**2025 г.**

**Оглавление**

[**1.** **Цель работы** 3](#_Toc195294793)

[**2.** **Задачи (формализация)** 3](#_Toc195294794)

[**3.** **Теоретическая часть** 3](#_Toc195294795)

[**4.** **Описание алгоритма** 4](#_Toc195294796)

[**5.** **Описание программы** 9](#_Toc195294797)

[**5.1 Описание функций** 9](#_Toc195294798)

[**6.** **Рекомендации пользователю** 10](#_Toc195294799)

[**7.** **Рекомендации программисту** 11](#_Toc195294800)

[**8.** **Контрольный пример** 12](#_Toc195294801)

[**9.** **Анализ результатов** 15](#_Toc195294802)

[**10.** **Вывод** 17](#_Toc195294803)

[**11.** **Листинг программы** 17](#_Toc195294805)

# **Цель работы**

Исследовать эффективность алгоритма имитации отжига для решения задачи коммивояжёра (TSP) и изучить влияние параметров (температуры, скорости охлаждения) на точность решения.

# **Задачи (формализация)**

* *Реализовать алгоритм имитации отжига* (классический и модификация «Сверхбыстрый отжиг»).
* *Разработать графический интерфейс*, позволяющий: вводить и визуализировать граф и настраивать параметры алгоритма (температура, скорость охлаждения, число итераций).
* *Сравнить эффективность классического отжига и VFSA* (по времени выполнения и длине пути). Сопоставить результаты с алгоритмом ближайшего соседа.

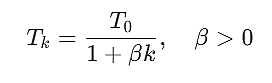
# **Теоретическая часть**

* *Задача коммивояжёра:*Задача коммивояжёра заключается в нахождении кратчайшего замкнутого маршрута, проходящего через все заданные города ровно один раз и возвращающегося в исходную точку. Эта задача относится к классу NP-сложных комбинаторных задач оптимизации.
* *Метод имитации отжига:*  
  Метод имитации отжига — это вероятностный алгоритм оптимизации, вдохновлённый физическим процессом отжига металлов. Алгоритм начинает работу с высокого значения "температуры", что позволяет принимать ухудшающие решения для выхода из локальных минимумов. По мере уменьшения температуры алгоритм становится более "жадным", стремясь к локальному оптимуму.
* *Модификация VFSA:*Модификация VFSA (Very Fast Simulated Annealing) отличается от классического алгоритма имитации отжига более быстрым законом охлаждения. Это позволяет сократить время вычислений при сохранении приемлемого качества решения.
* *Операторы изменения маршрута:*  
  Для генерации новых решений в задаче коммивояжёра используются три основных оператора: перестановка двух вершин (swap), инверсия участка маршрута (reverse) и перемещение одной вершины (insert). Эти операторы позволяют эффективно исследовать пространство возможных решений.
* *Вероятность принятия решения с увеличением длины маршрута на ΔL при температуре T:*

* *Схемы охлаждения:*

а) Экспоненциальное охлаждение (классический SA):



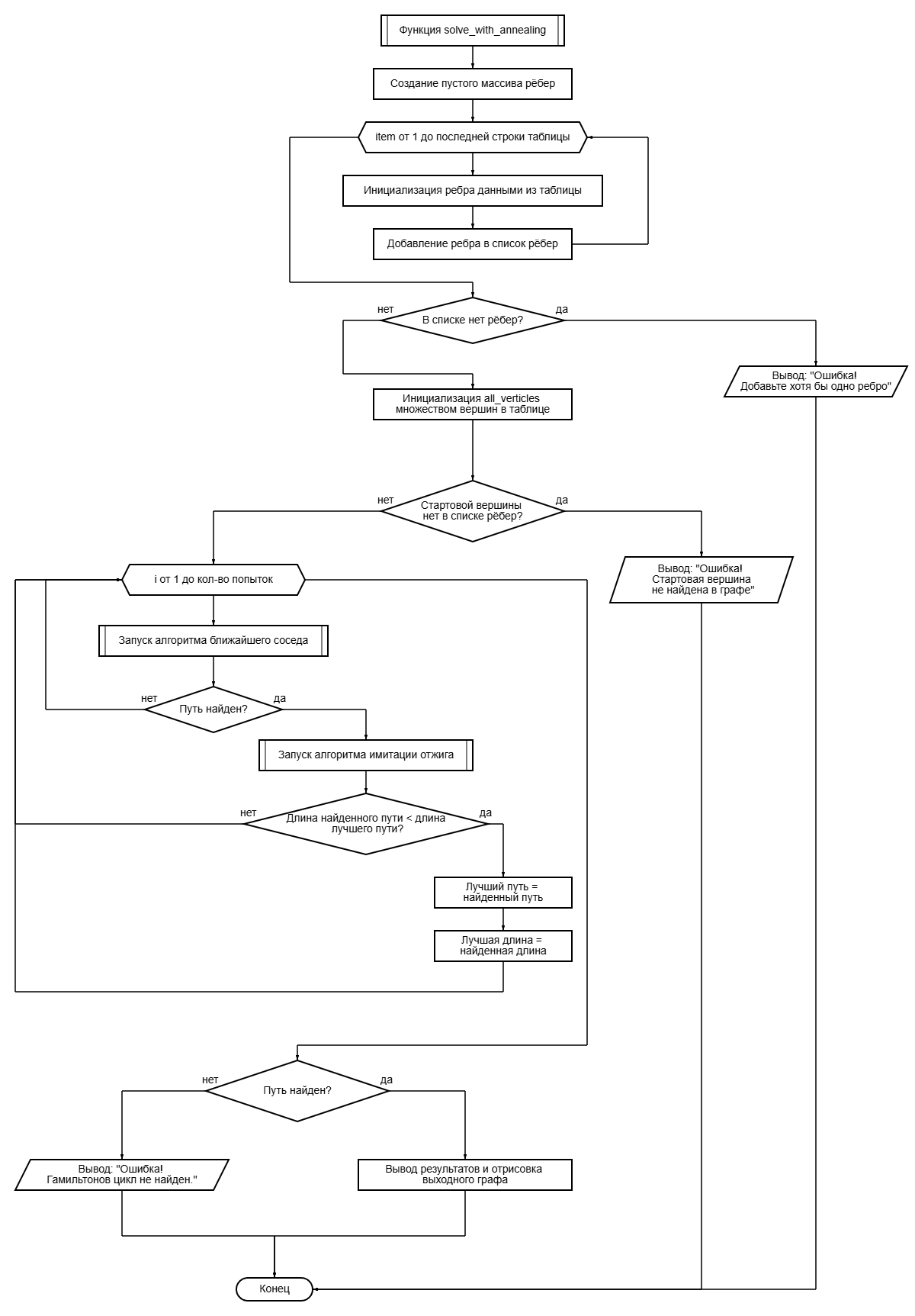
б) Гиперболическое охлаждение (VFSA):

где k – номер итерации.

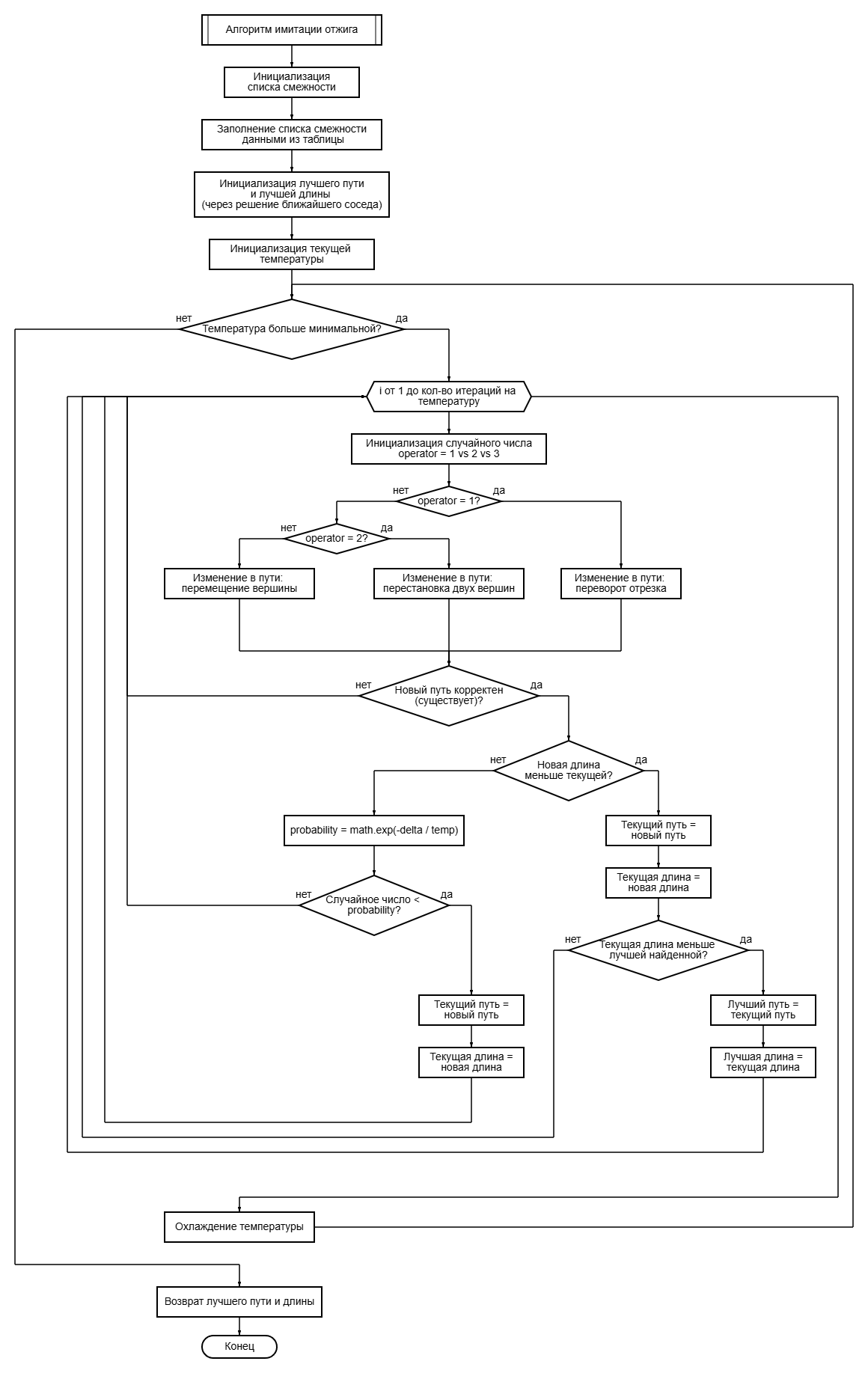
# **Описание алгоритма**

1. *Создание GUI (tkinter)*  
   Инициализация главного окна с тремя основными областями: панель параметров, область визуализации графов и таблица ребер.
2. *Обработка событий мыши (on\_click)*  
   При клике создается вершина (красный круг с меткой), а при последующем клике на другую вершину открывается диалог simpledialog для ввода веса ребра, которое затем добавляется в edges[] и Treeview.
3. *Визуализация графа (draw\_input\_graph)*Отрисовываются овалы вершин, линии ребер (create\_line) с подписями весов (create\_text), обновляя canvas.delete("all") при каждом изменении.
4. *Построение начального пути (randomized\_tsp)*Комбинируем greedy-подход (выбор ближайшего соседа) вероятностью случайного выбора следующей вершины, используя adjacency\_list для хранения структуры графа.
5. *Алгоритм отжига (simulated\_annealing\_tsp)*Применяем три оператора мутации: переворот отрезка (reversed()), перестановку вершин (swap), и перемещение вершины (insert/pop), с проверкой корректности пути.
6. *Запуск ускоренного отжига (very\_fast\_simulated\_annealing\_tsp)*Используется нелинейное охлаждение T = T0/(1+βk) вместо стандартного T = αT, с динамическим расчетом температуры на каждой итерации.
7. *Управление данными (tree/edges)*Treeview отображает ребра в формате (v1, v2, weight), синхронизируясь с глобальным списком edges[], при этом pandas.DataFrame используется для внутренних расчетов.
8. *Запуск расчета (solve\_with\_annealing)*Считывание параметров из Entry (initial\_temp, cooling\_rate и др.), выполняет max\_attempts попыток с замером времени через time.time(), обрабатывая исключения ValueError.
9. *Отображение результата (draw\_output\_graph)*Отображаем финальный путь зелеными линиями (width=2), сохраняя исходную визуализацию вершин, с проверкой path[] на корректность.

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

**

*Рисунок 4.1. Блок-схема основной функции solve\_with\_annealing*

**

*Рисунок 4.2. Блок-схема алгоритма имитации отжига*

# **Описание программы**

## **5.1 Описание функций**

В программе используется 12 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 |
| randomized\_tsp | graph (DataFrame), start\_vertex (str), randomness (float, default=0.5) | Генерирует начальное решение TSP, комбинируя жадный подход с рандомным выбором следующей вершины. Возвращает путь и его длину или (None, None) если решение не найдено. |
| simulated\_annealing\_tsp | graph (DataFrame), initial\_path (list), initial\_distance (float), параметры отжига (initial\_temp, cooling\_rate, min\_temp, iterations\_per\_temp) | Реализует классический алгоритм имитации отжига с тремя операторами мутации (swap/reverse/insert). Возвращает лучший найденный путь и его длину. |
| very\_fast\_simulated\_annealing\_tsp | graph (DataFrame), initial\_path (list), initial\_distance (float), параметры VFSA (initial\_temp, min\_temp, iterations\_per\_temp, cooling\_factor) | Модификация алгоритма отжига с гиперболическим охлаждением (VFSA). Возвращает оптимальный путь и расстояние. |
| add\_edge\_to\_table | v1 (str), v2 (str), length (float) | Добавляет ребро в таблицу Treeview для отображения в интерфейсе. |
| on\_click | event (tkinter.Event) | Обрабатывает клики мыши: создает вершины по клику и ребра между ними через диалоговое окно. |
| draw\_input\_graph | - | Отрисовывает текущий граф на входном canvas (вершины как круги, ребра как линии с весами). |
| draw\_output\_graph | path (list) | Визуализирует результирующий путь зеленым цветом на отдельном canvas. |
| solve\_with\_annealing | - | Основная функция решения: читает параметры из GUI, запускает алгоритм отжига и выводит результаты. |
| solve\_with\_very\_fast\_annealing | - | Аналогично solve\_with\_annealing, но для VFSA-версии алгоритма. |
| clear\_canvas | - | Сбрасывает состояние интерфейса: очищает граф, таблицу ребер и результаты. |
| set\_graph\_type | type (str: "directed" или "undirected") | Устанавливает тип графа (ориентированный/неориентированный). |
| load\_graph\_from\_file | - | Загружает граф из CSV-файла (колонки: "вершина 1", "вершина 2", "длина") и отображает его в интерфейсе. |

# **Рекомендации пользователю**

*Запуск программы:*

* Убедитесь, что у вас установлен Python 3.7 или новее и проверьте наличие необходимых библиотек

*Настройка графа:*

* Кликните левой кнопкой мыши на белом холсте, чтобы создать вершину (вершины автоматически нумеруются как V1, V2, V3 и т.д.).
* Или же нажмите кнопку "Загрузить граф из файла" (выберите CSV-файл с колонками: "вершина 1", "вершина 2", "длина").

*Ввод параметров:*

* В поле "Стартовая вершина" введите название вершины (например, V1)
* Установите параметры алгоритма:
* Начальная температура (рекомендуется 1000-5000)
* Скорость охлаждения (рекомендуется 0.95-0.99)
* Число итераций (рекомендуется 50-200)

*Выбор алгоритма:*

* "Рассчитать" - классический алгоритм отжига
* "Рассчитать (сверхбыстрый отжиг)" - ускоренная версия VFSA

*Если решение не найдено,* появится предупреждение - проверьте:

* Все ли вершины связаны;
* Правильно ли указана стартовая вершина;
* И Корректность введенных данных.

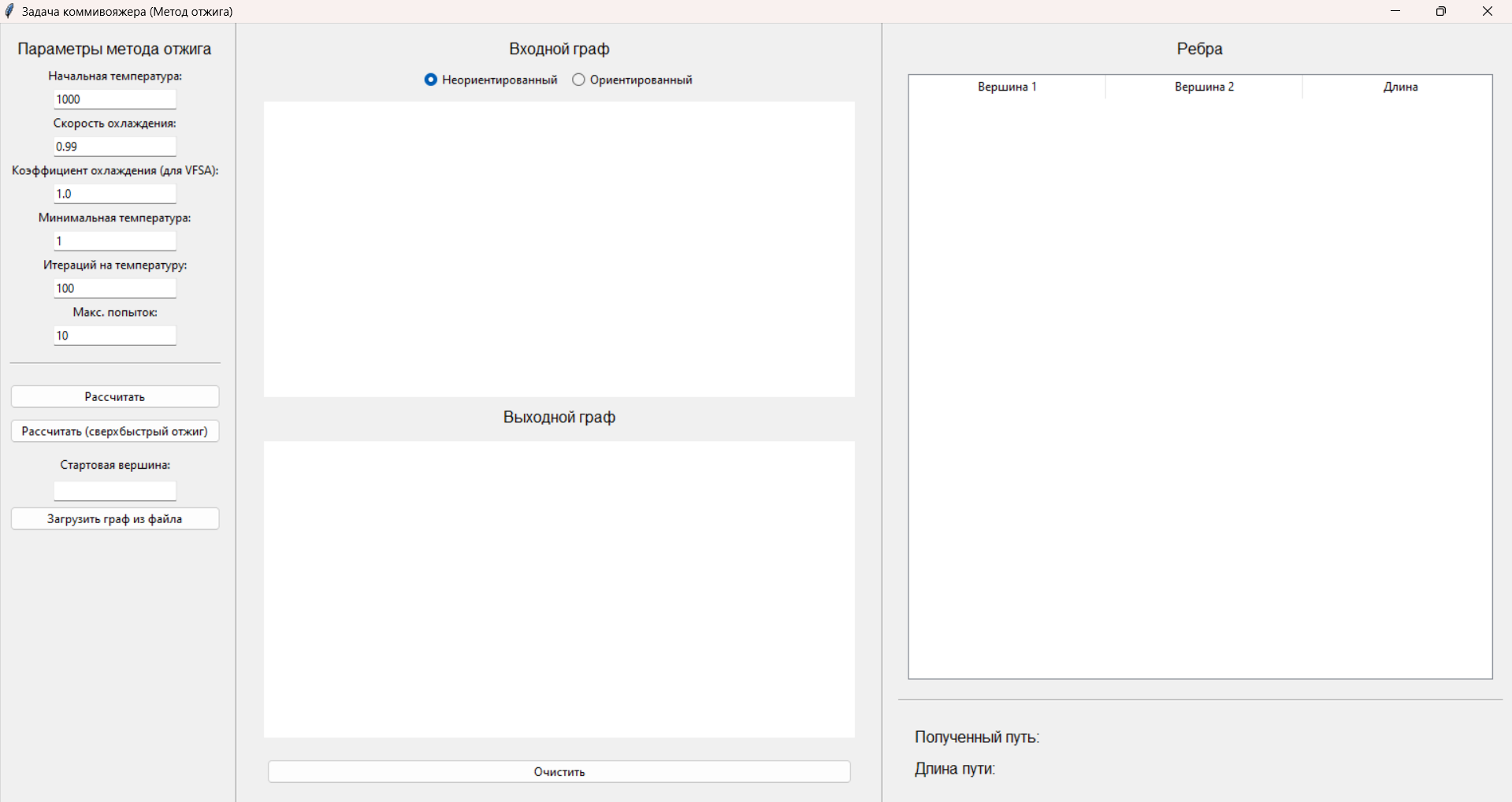
1. **Рекомендации программисту**

* 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 имитации отжига для задачи коммивояжера, а также его сверхбыструю версию. Для иной модификации алгоритм измените формулу охлаждения температуры в функции solve\_with\_very\_fast\_annealing.
* 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 и добавьте соответствующие функции.
* Имейте в виду, что сверхбыстрый отжиг работает быстрее стандартного отжига за счёт нелинейного охлаждения, но увеличивает риск застревания в локальном минимуме. (Cooling\_factor регулирует скорость охлаждения)
* Время выполнения: Для больших графов (>30 вершин) время работы может значительно увеличиться, поэтому рекомендуется начать тестирование с малых графов и постепенно увеличивать размер.

1. **Контрольный пример**

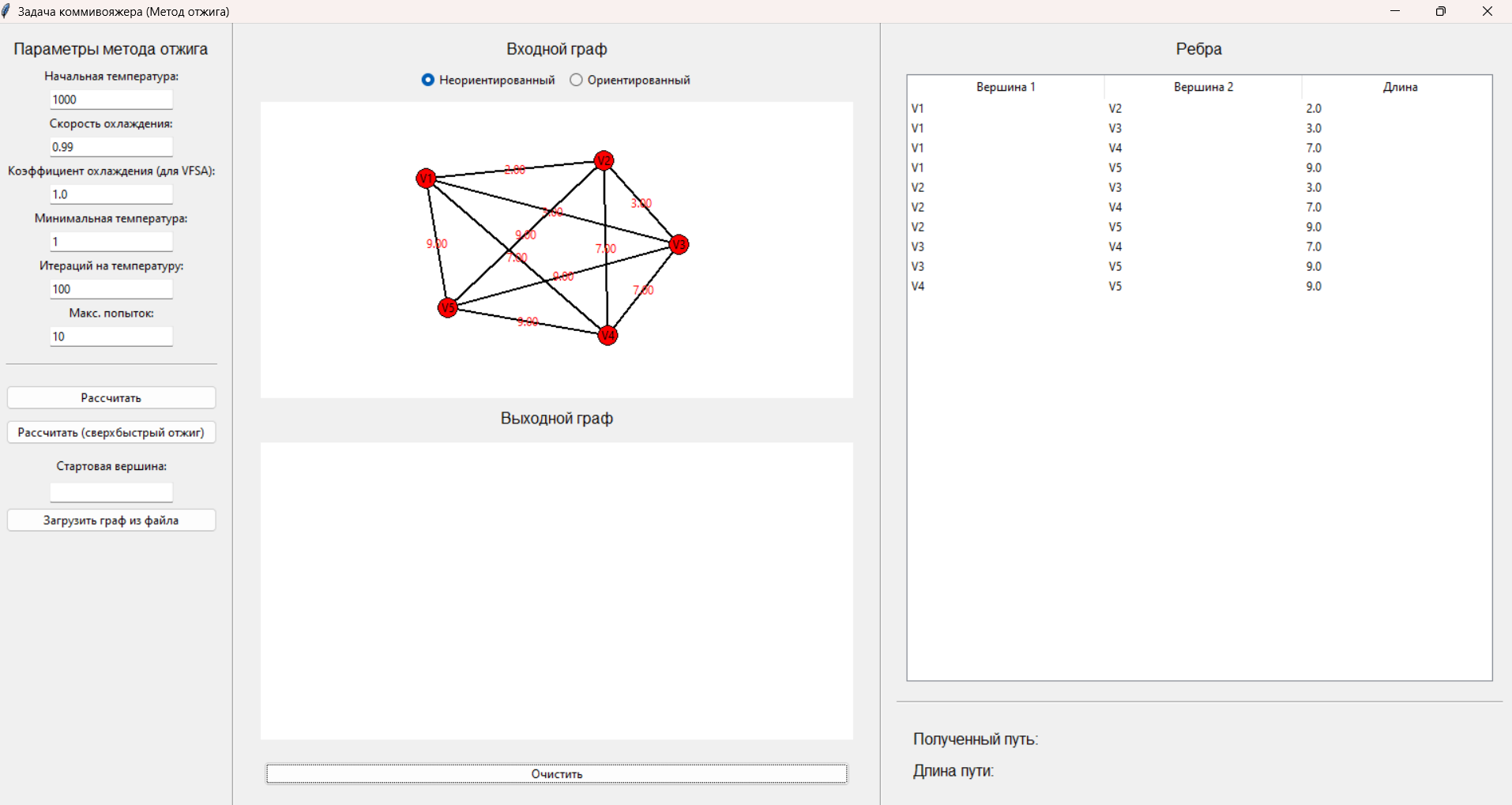
В этом разделе продемонстрирован пример запуска алгоритма имитации отжига на примере заданного неориентированного графа.

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



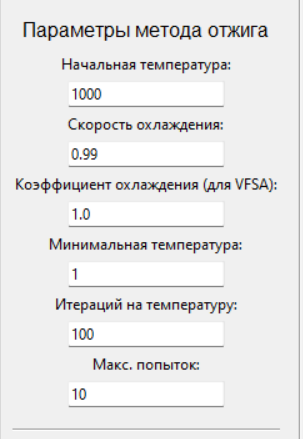
*Рисунок 8.1 Начальный интерфейс программы*

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.

**

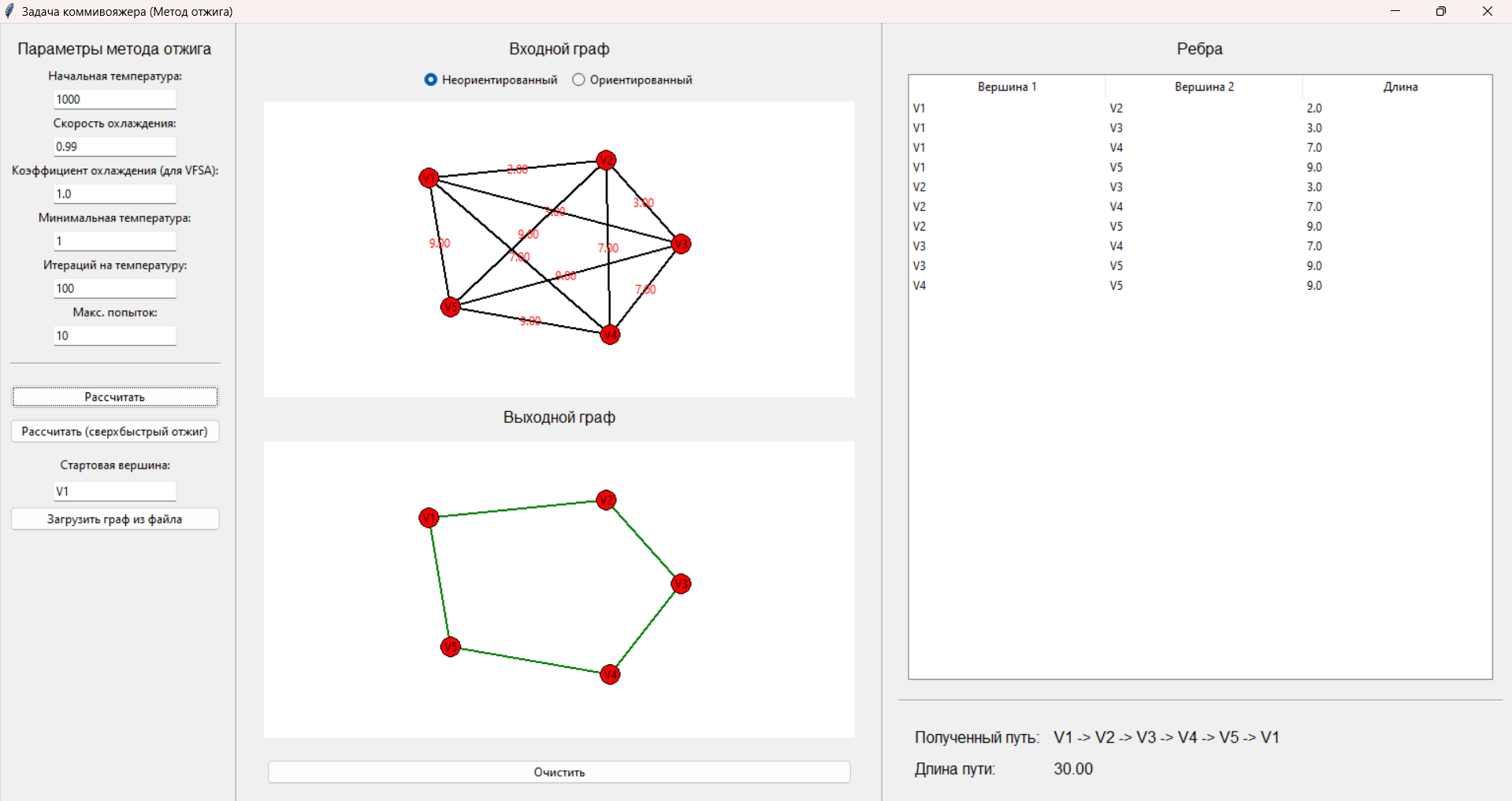
*Рисунок 8.2 Окно программы после ввода графа*

В верхней области левой части окна есть поля для ввода гиперпараметров, вы можете самостоятельно выставить нужные значения (советы по эффективному выбору параметров даны в рекомендациях пользователю и программисту).



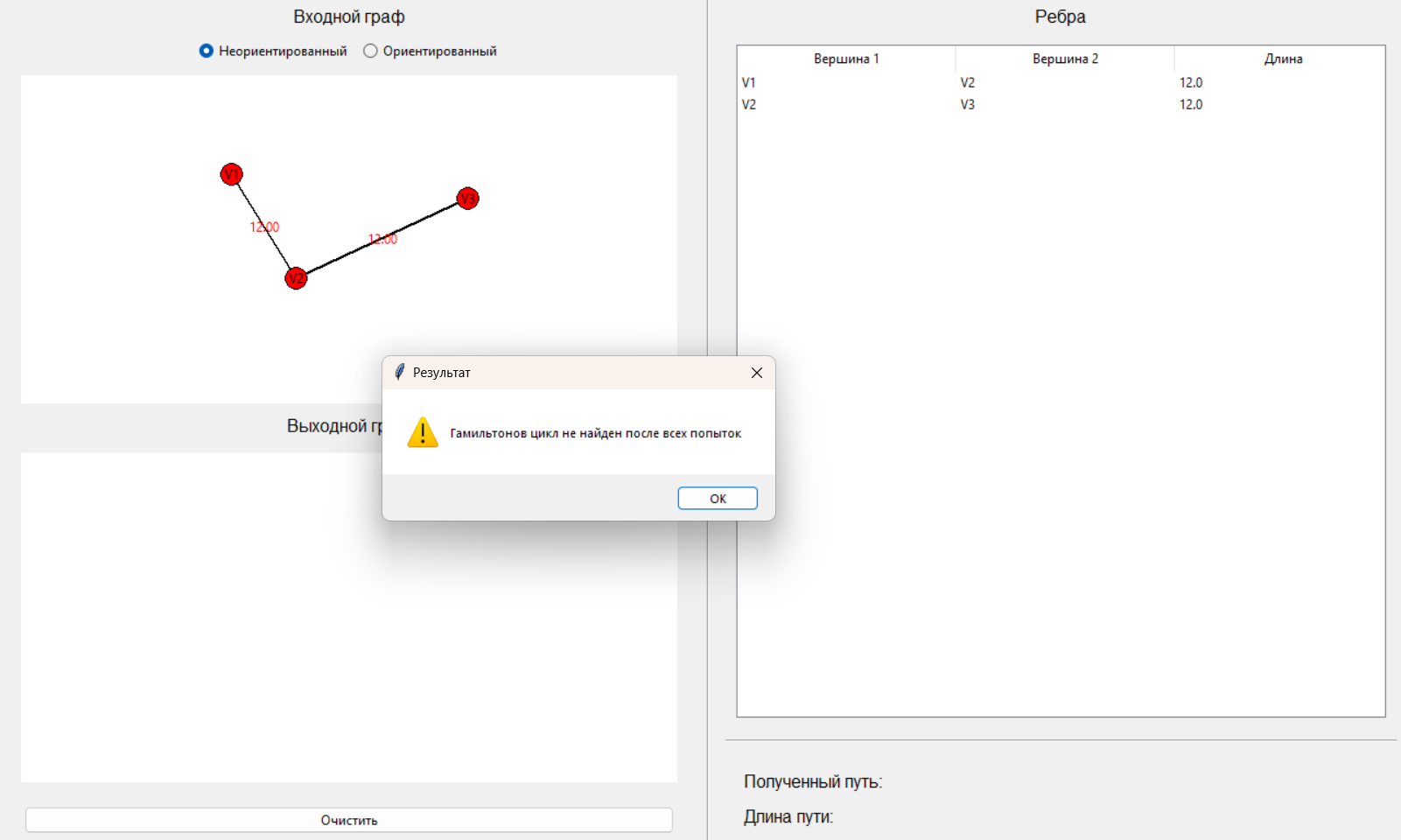
*Рисунок 8.3 Поля для ввода гиперпараметров*

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.

**

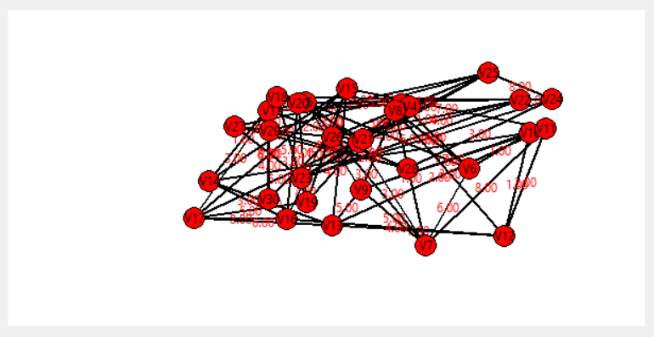
*Рисунок 8.4 Окно программы после нажатия кнопки «Рассчитать»*

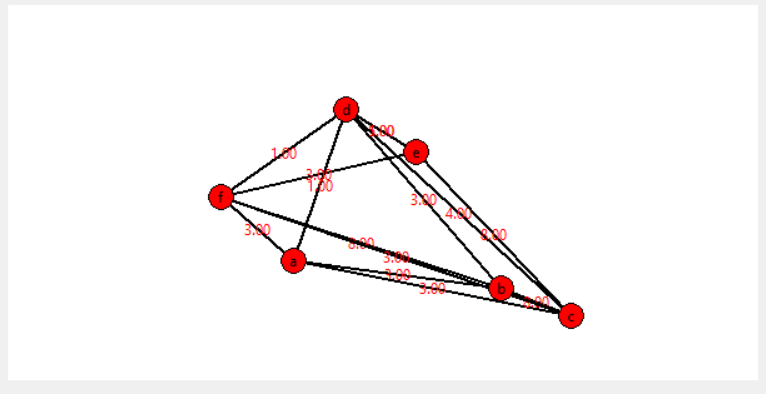
Если алгоритм не найдет гамильтонов цикл, то вы увидите предупреждение во всплывающем окне. В таком случае проверьте корректность введенных данных и выбранный тип графа.

**

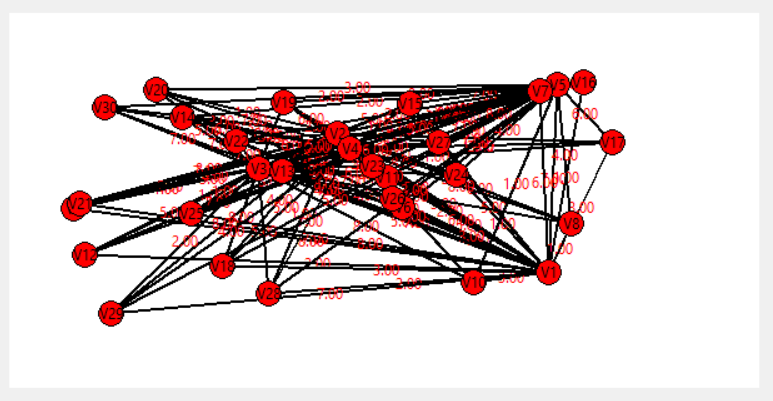
*Рисунок 8.5 Запуск с некорректными данными*

# **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 (в ней также присутствуют предыдущие результаты для алгоритма ближайшего соседа).

*Таблица 9.1. Анализ результатов*

|  |  |  |  |
| --- | --- | --- | --- |
| **Метрика** | **Граф 1** | **Граф 2** | **Граф 3** |
| Кол-во ребер | 14 | 52 | 103 |
| Лучшее решение (ближ. сосед) | 16 | 93 | 96 |
| Время (ближ. сосед) | 0.0010 сек | 0.0032 сек | 0.0065 сек |
| Лучшее решение (ближ. сосед  рандомизированный) | 16 | 93 | 97 |
| Время (ближ. сосед  рандомизированный) | 0.0010 сек | 0.0033 сек | 0.0063 сек |
| Лучшее решение (кл. отжиг) | 16 | 93 | 73 |
| Время (классич. отжиг) | 0.3906 сек | 0.5402 сек | 0.7065 сек |
| Лучшее решение (сверх. отжиг) | 16 | 93 | 87 |
| Время (сверхбыстрый отжиг) | 0.0095 сек | 0.0869 сек | 0.1050 сек |

1. Граф 1:  
   Все алгоритмы нашли наилучшее решение, что говорит о том, что граф 1 имеет простую реберную структуру. Для графов такой структуры (с малым количеством вершин и ребер) нет смысла использовать сложные метаэвристики – достаточно быстрого и предсказуемого алгоритма ближайшего соседа.
2. Граф 2:  
   Классический отжиг показал такой же результат, как и алгоритм ближайшего соседа (93), но работал значительно медленнее. Можно сделать вывод, что для простых графов с малым количеством рёбер стохастические методы избыточны — жадный алгоритм справляется быстрее без серьезной потери точности. Сверхбыстрый отжиг сильно сократил время работы классического варианта, сохранив результат, что делает его более практичным в определенных задачах.
3. Граф 3:  
   Классический отжиг оказался эффективнее остальных методов. Это подтверждает его способность вырываться из локальных минимумов за счёт вероятностного принятия худших решений на ранних этапах. Сверхбыстрый отжиг, хотя и улучшил результат ближайшего соседа, не дотянул до классического варианта, вероятно, из-за слишком быстрого снижения температуры, которое ограничило исследование пространства решений.
4. Общий итог:

Алгоритм имитации отжига подтвердил свою эффективность для графов сложной и средней сложности, где жадные алгоритмы дают заметно худшие результаты, но его применение должно быть осмысленным: на простых графах он проигрывает в скорости, а на сверхсложных требует модификаций.

1. **Вывод**

Реализован алгоритм имитации отжига для решения задачи коммивояжера (классическая версия алгоритма и модификация «сверхбыстрый отжиг»). Создана программа с удобным и понятным интерфейсом пользователя с несколькими функциональностями: создание графа, загрузка графа из файла, установка гиперпараметров и др. Проведен сравнительный анализ по времени выполнения и точности найденного решения с алгоритмом ближайшего соседа (на графах сложной структуры, заданных на 30+ вершинах).

# **Листинг программы**

import math

import random

import tkinter as tk

from tkinter import ttk, messagebox, simpledialog, filedialog

import pandas as pd

import time

import numpy as np

​

# Глобальные переменные для хранения вершин и рёбер

vertices = {}

edges = []

current\_edge = None

graph\_type = "undirected"

​

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

   """Рандомизированный алгоритм для нахождения начального решения"""

   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 simulated\_annealing\_tsp(graph, initial\_path, initial\_distance,

                          initial\_temp=1000, cooling\_rate=0.99,

                          min\_temp=1, iterations\_per\_temp=100):

   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

​

   if initial\_path[0] != initial\_path[-1]:

       raise ValueError("Начальный путь должен быть циклом (первая и последняя вершины совпадают)")

​

   current\_path = initial\_path.copy()

   current\_distance = initial\_distance

   best\_path = current\_path.copy()

   best\_distance = current\_distance

   temp = initial\_temp

   while temp > min\_temp:

       for \_ in range(iterations\_per\_temp):

           new\_path = current\_path.copy()

           intermediate\_vertices = new\_path[1:-1]  # Вершины между началом и концом

           if len(intermediate\_vertices) >= 2:

               operator = random.choice([1, 2, 3])

               if operator == 1:  # переворот отрезка

                   i, j = sorted(random.sample(range(1, len(new\_path) - 1), 2))

                   new\_path[i:j+1] = reversed(new\_path[i:j+1])

               elif operator == 2:  # Перестановка двух вершин

                   i, j = random.sample(range(1, len(new\_path) - 1), 2)

                   new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

               else:  # Перемещение вершины

                   i = random.randint(1, len(new\_path) - 2)

                   j = random.randint(1, len(new\_path) - 2)

                   if i != j:

                       vertex = new\_path.pop(i)

                       new\_path.insert(j, vertex)

           if new\_path[0] != new\_path[-1]:

               new\_path[-1] = new\_path[0]

           new\_distance = 0

           valid = True

           for k in range(len(new\_path) - 1):

               v1, v2 = new\_path[k], new\_path[k+1]

               if v2 in adjacency\_list[v1]:

                   new\_distance += adjacency\_list[v1][v2]

               else:

                   valid = False

                   break

           if not valid:

               continue

           if new\_distance < current\_distance:

               current\_path = new\_path

               current\_distance = new\_distance

               if new\_distance < best\_distance:

                   best\_path = new\_path.copy()

                   best\_distance = new\_distance

           else:

               delta = new\_distance - current\_distance

               probability = math.exp(-delta / temp)

               if random.random() < probability:

                   current\_path = new\_path

                   current\_distance = new\_distance

       temp \*= cooling\_rate

   return best\_path, best\_distance

​

def very\_fast\_simulated\_annealing\_tsp(graph, initial\_path, initial\_distance,

                                    initial\_temp=1000, min\_temp=1,

                                    iterations\_per\_temp=100, cooling\_factor=1.0):

   """Модификация метода отжига со сверхбыстрым охлаждением (VFSA)"""

   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

​

   if initial\_path[0] != initial\_path[-1]:

       raise ValueError("Начальный путь должен быть циклом (первая и последняя вершины совпадают)")

​

   current\_path = initial\_path.copy()

   current\_distance = initial\_distance

   best\_path = current\_path.copy()

   best\_distance = current\_distance

   temp = initial\_temp

   iteration = 0

   while temp > min\_temp:

       for \_ in range(iterations\_per\_temp):

           iteration += 1

           new\_path = current\_path.copy()

           intermediate\_vertices = new\_path[1:-1]

           if len(intermediate\_vertices) >= 2:

               operator = random.choice([1, 2, 3])

               if operator == 1:

                   i, j = sorted(random.sample(range(1, len(new\_path) - 1), 2))

                   new\_path[i:j+1] = reversed(new\_path[i:j+1])

               elif operator == 2:

                   i, j = random.sample(range(1, len(new\_path) - 1), 2)

                   new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

               else:

                   i = random.randint(1, len(new\_path) - 2)

                   j = random.randint(1, len(new\_path) - 2)

                   if i != j:

                       vertex = new\_path.pop(i)

                       new\_path.insert(j, vertex)

           if new\_path[0] != new\_path[-1]:

               new\_path[-1] = new\_path[0]

           new\_distance = 0

           valid = True

           for k in range(len(new\_path) - 1):

               v1, v2 = new\_path[k], new\_path[k+1]

               if v2 in adjacency\_list.get(v1, {}):

                   new\_distance += adjacency\_list[v1][v2]

               else:

                   valid = False

                   break

           if not valid:

               continue

           if new\_distance < current\_distance:

               current\_path = new\_path

               current\_distance = new\_distance

               if new\_distance < best\_distance:

                   best\_path = new\_path.copy()

                   best\_distance = new\_distance

           else:

               delta = new\_distance - current\_distance

               probability = math.exp(-delta / temp)

               if random.random() < probability:

                   current\_path = new\_path

                   current\_distance = new\_distance

       # Сверхбыстрое охлаждение (VFSA)

       temp = initial\_temp / (1 + cooling\_factor \* iteration)

   return best\_path, best\_distance

​

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:

           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\_with\_annealing():

   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

​

   try:

       initial\_temp = float(entry\_initial\_temp.get())

       cooling\_rate = float(entry\_cooling\_rate.get())

       min\_temp = float(entry\_min\_temp.get())

       iterations = int(entry\_iterations.get())

       max\_attempts = int(entry\_max\_attempts.get())

   except ValueError:

       messagebox.showwarning("Ошибка", "Пожалуйста, введите корректные числовые параметры!")

       return

​

   # max\_attempts попыток

   best\_path = None

   best\_distance = float('inf')

   execution\_time = 0

   for attempt in range(1, max\_attempts + 1):

       # Находим начальное решение рандомизированным алгоритмом

       start\_time = time.time()

       initial\_path, initial\_distance = randomized\_tsp(graph, start\_vertex, randomness=0.5)

       end\_time = time.time()

       execution\_time += end\_time - start\_time

       if initial\_path is None:

           continue

       start\_time = time.time()

       path, distance = simulated\_annealing\_tsp(

           graph, initial\_path, initial\_distance,

           initial\_temp=initial\_temp,

           cooling\_rate=cooling\_rate,

           min\_temp=min\_temp,

           iterations\_per\_temp=iterations

      )

       end\_time = time.time()

       execution\_time += end\_time - start\_time

       if distance < best\_distance:

           best\_path = path

           best\_distance = distance

       if best\_path is not None:

           break

​

   print(f"Время выполнения: {execution\_time:.4f} секунд")

   print(f"Попыток: {attempt}")

​

   if best\_path is None:

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

       result\_path.config(text="")

       result\_distance.config(text="")

       draw\_output\_graph([])

   else:

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

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

       draw\_output\_graph(best\_path)

​

def solve\_with\_very\_fast\_annealing():

   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

​

   try:

       initial\_temp = float(entry\_initial\_temp.get())

       min\_temp = float(entry\_min\_temp.get())

       iterations = int(entry\_iterations.get())

       max\_attempts = int(entry\_max\_attempts.get())

       cooling\_factor = float(entry\_cooling\_factor.get())

   except ValueError:

       messagebox.showwarning("Ошибка", "Пожалуйста, введите корректные числовые параметры!")

       return

​

   # max\_attempts попыток

   best\_path = None

   best\_distance = float('inf')

   execution\_time = 0

   for attempt in range(1, max\_attempts + 1):

       start\_time = time.time()

       initial\_path, initial\_distance = randomized\_tsp(graph, start\_vertex, randomness=0.5)

       end\_time = time.time()

       execution\_time += end\_time - start\_time

       if initial\_path is None:

           continue

       start\_time = time.time()

       path, distance = very\_fast\_simulated\_annealing\_tsp(

           graph, initial\_path, initial\_distance,

           initial\_temp=initial\_temp,

           min\_temp=min\_temp,

           iterations\_per\_temp=iterations,

           cooling\_factor=cooling\_factor

      )

       end\_time = time.time()

       execution\_time += end\_time - start\_time

       if distance < best\_distance:

           best\_path = path

           best\_distance = distance

       if best\_path is not None:

           break

​

   print(f"Время выполнения (сверхбыстрый отжиг): {execution\_time:.4f} секунд")

   print(f"Попыток: {attempt}")

​

   if best\_path is None:

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

       result\_path.config(text="")

       result\_distance.config(text="")

       draw\_output\_graph([])

   else:

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

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

       draw\_output\_graph(best\_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}")

​

# Создание интерфейса

root = tk.Tk()

root.title("Задача коммивояжера (Метод отжига)")

​

# Левый фрейм для параметров

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 метода отжига", font=("Arial", 12))

label\_parameters.pack(pady=5)

​

label\_initial\_temp = ttk.Label(left\_frame, text="Начальная температура:")

label\_initial\_temp.pack(pady=2)

entry\_initial\_temp = ttk.Entry(left\_frame)

entry\_initial\_temp.insert(0, "1000")

entry\_initial\_temp.pack(pady=2)

​

label\_cooling\_rate = ttk.Label(left\_frame, text="Скорость охлаждения (для эксп.):")

label\_cooling\_rate.pack(pady=2)

entry\_cooling\_rate = ttk.Entry(left\_frame)

entry\_cooling\_rate.insert(0, "0.99")

entry\_cooling\_rate.pack(pady=2)

​

label\_cooling\_factor = ttk.Label(left\_frame, text="Коэффициент охлаждения (для VFSA):")

label\_cooling\_factor.pack(pady=2)

entry\_cooling\_factor = ttk.Entry(left\_frame)

entry\_cooling\_factor.insert(0, "1.0")

entry\_cooling\_factor.pack(pady=2)

​

label\_min\_temp = ttk.Label(left\_frame, text="Минимальная температура:")

label\_min\_temp.pack(pady=2)

entry\_min\_temp = ttk.Entry(left\_frame)

entry\_min\_temp.insert(0, "1")

entry\_min\_temp.pack(pady=2)

​

label\_iterations = ttk.Label(left\_frame, text="Итераций на температуру:")

label\_iterations.pack(pady=2)

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

entry\_iterations.insert(0, "100")

entry\_iterations.pack(pady=2)

​

label\_max\_attempts = ttk.Label(left\_frame, text="Макс. попыток:")

label\_max\_attempts.pack(pady=2)

entry\_max\_attempts = ttk.Entry(left\_frame)

entry\_max\_attempts.insert(0, "10")

entry\_max\_attempts.pack(pady=2)

​

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 (эксп. охлаждение)", command=solve\_with\_annealing)

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

​

button\_solve\_vfsa = ttk.Button(left\_frame, text="Рассчитать (сверхбыстрый отжиг)", command=solve\_with\_very\_fast\_annealing)

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

​

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 графиков

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)

​

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

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")

​root.mainloop()