**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики - процессов управления**

**Программа бакалавриата**

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

**ОТЧЕТ**

**по лабораторной работе №2**

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

**на тему «Алгоритм отжига»**

**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 12](#_Toc178797448)

[12. Источники 12](#_Toc178797448)

[13. Листинг 13](#_Toc178797450)

# Цель работы

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

# Описание задачи (формализация задачи)

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

# Теоретическая часть

Алгоритм имитации отжига — общий алгоритмический метод решения задачи глобальной оптимизации, особенно дискретной и комбинаторной оптимизации. Один из примеров методов Монте-Карло.

Алгоритм основывается на имитации физического процесса, который происходит при кристаллизации вещества, в том числе при отжиге металлов. Предполагается, что атомы вещества уже почти выстроены в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Активность атомов тем больше, чем выше температура, которую постепенно понижают, что приводит к тому, что вероятность переходов в состояния с большей энергией уменьшается. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте.

Параметры алгоритма и их роль:

1. Начальная температура

Чем выше температура, тем выше вероятность исследования различных состояний.

1. Функция охлаждения

Контролирует снижение температуры.

* + Стандартное охлаждение: , где k — номер итерации.
  + Больцмановское охлаждение: , где температура снижается медленнее, что позволяет алгоритму дольше исследовать пространство решений.

1. Функция стоимости (длины пути)

Определяет, насколько хорошо текущее решение. В коде стоимость пути вычисляется как сумма весов рёбер в текущем маршруте.

1. Критерий вероятности перехода

Если новое решение хуже текущего, оно принимается с вероятностью:

где ΔE — разница между длинами маршрутов

# Основные шаги программы

1. Генерируется начальный маршрут случайным образом.
2. На каждой итерации выбираются два случайных узла, и производится перестановка.
3. Вычисляется разница в стоимости пути после перестановки.
4. Если новый путь короче — он принимается.
5. Если новый путь длиннее, он принимается с вероятностью .
6. Температура снижается по заданной схеме.
7. Процесс повторяется, пока температура не достигнет порогового значения или не будет выполнено максимальное число итераций.

# Блок-схема программы

# На рисунке представлена блок-схема алгоритма.

# Рис. 5 Блок-схема алгоритма.

# Описание программы

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

Таблица 6.1 Описание функций

|  |  |
| --- | --- |
| Имя функции | Назначение |
| simulated\_annealing | Применяет алгоритм имитации отжига к заданному графу. |
| calculate\_total\_cost | Вспомогательная функция для вычисления полной цены шага. |

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. |
| get\_sa\_parameters | Принимает параметры алгоритма из интерфейса. |
| run\_simulated\_annealing | Связывает алгоритм вычисления с интерфесом. |
| 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. Проставить параметры алгоритма и выбрать обычный алгоритм или модификация.
4. Нажать кнопку “Найти кратчайший путь”. На графе зелёным цветом отобразится искомый путь, а снизу можно будет увидеть длину пути.

# Рекомендации программиста

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

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

# Исходный код программы

# Исходный код программы и необходимые текстовые файлы доступны по ссылке: <https://github.com/NikiTaku1/spbu_alg_2/tree/main/lab2>

# Контрольный пример

В данном разделе представлен пример, демонстрирующий работу программы.

1) Запуск main.py

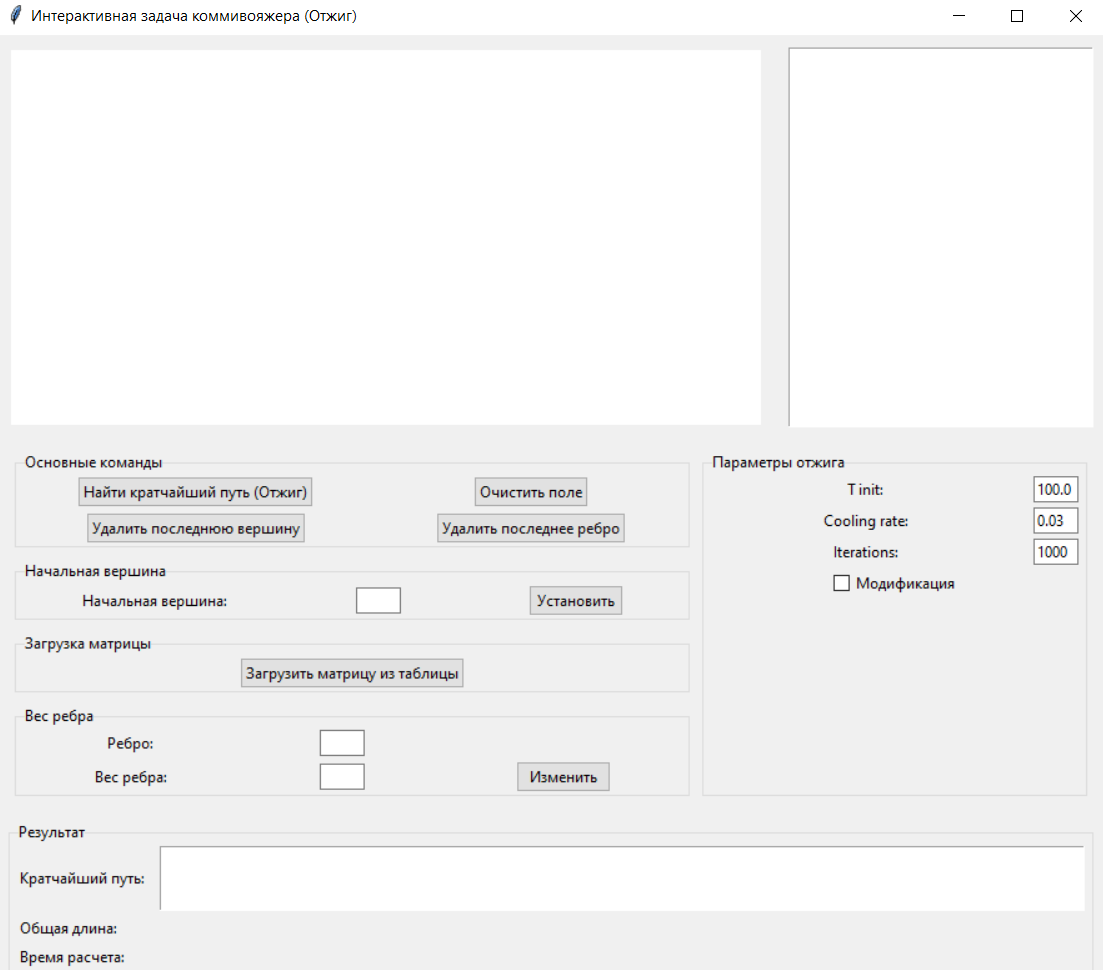


Рис 10.1 Графический интерфейс программы.

2) Задание графа

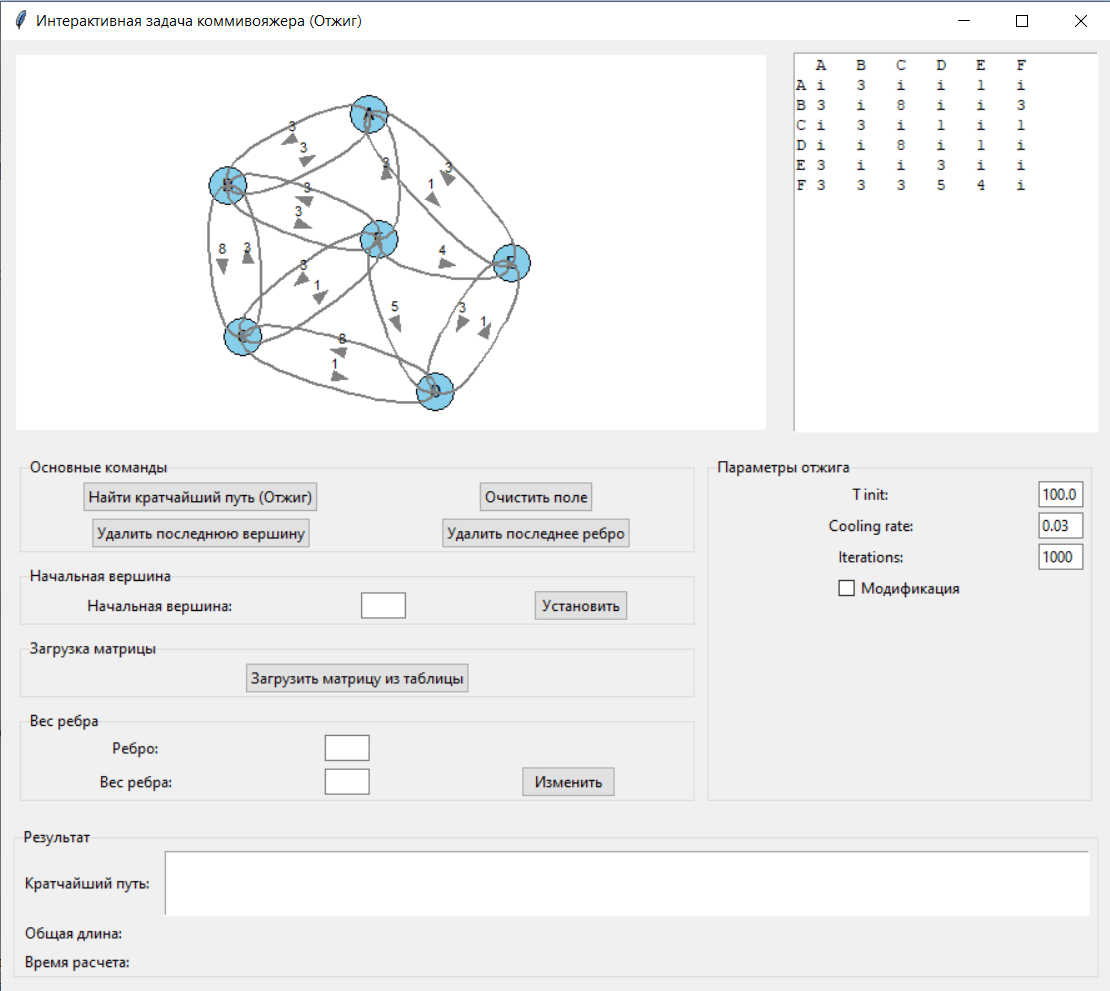


Рис 10.2 Графический интерфейс с заданным графом.

3) Результат алгоритма без модификации

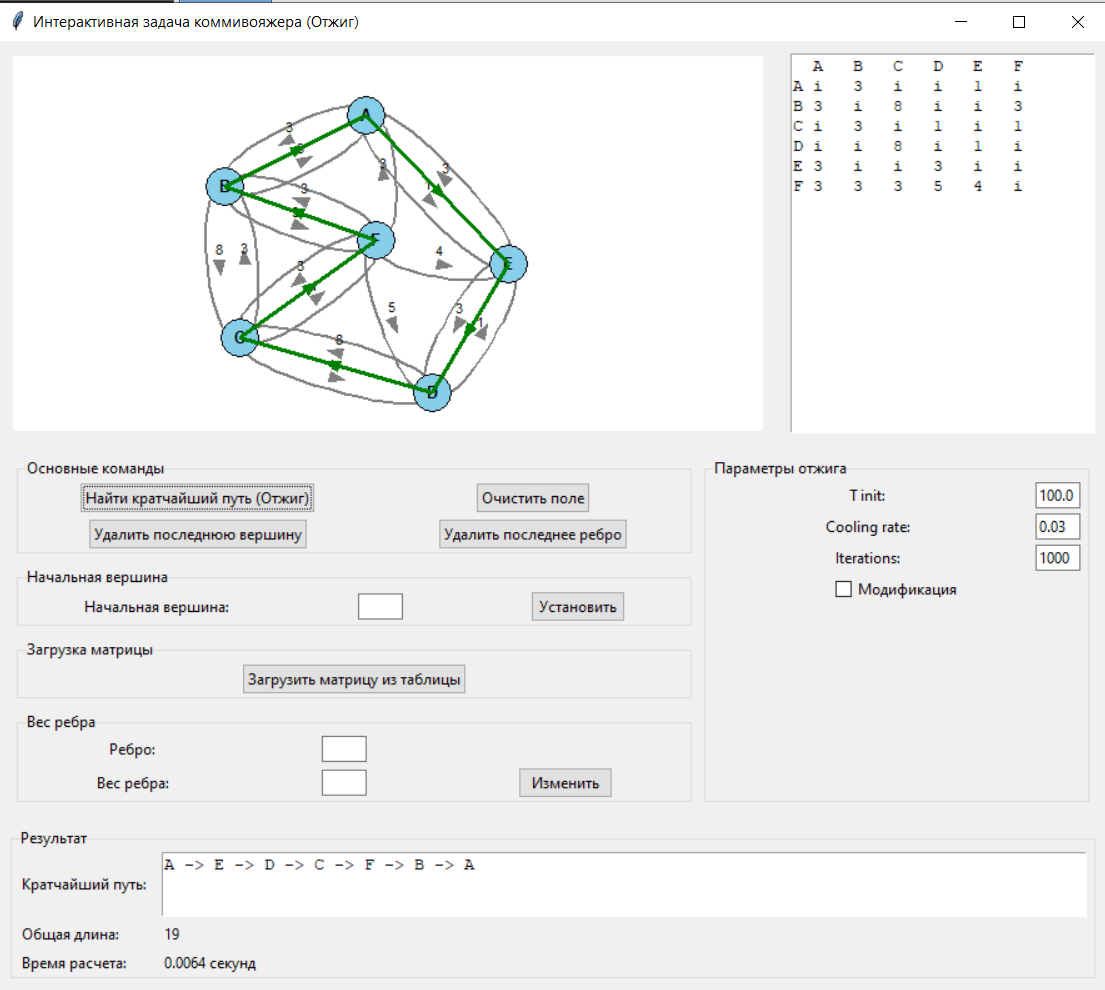


Рис 10.3 Графический интерфейс после выполнения программы без модификации.

4) Результат алгоритма с модификацией

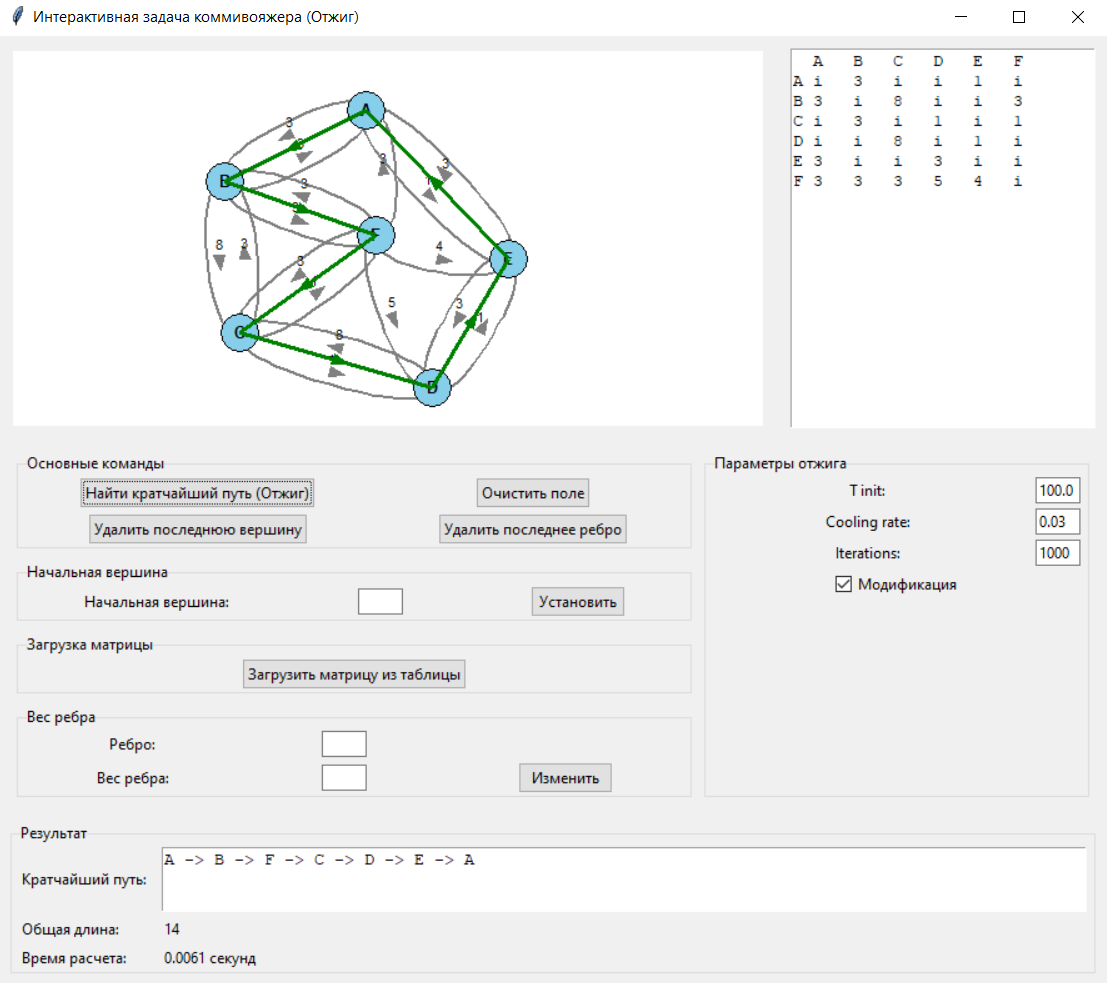


Рис 10.4 Графический интерфейс после выполнения программы c модификацией.

5) Сравнение эффективности алгоритма с модификацией и без (графы для анализа сгенерированы случайно, параметры модификации оптимизированы)

Таблица 10.5 Сравнение эффективности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество вершин | Время работы без модификации (с) | Время работы с модификацией (с) | Длина пути без модификации | Длина пути с модификацией (параметры оптимизированы) |
| 6 | 0.0078 | 0.0088 | 83 | 73 |
| 10 | 0.0118 | 0.0138 | 85 | 74 |
| 15 | 0.0181 | 0.0209 | 165 | 147 |
| 30 | 0.0521 | 0.2363 | 293 | 198 |
| 50 | 0.1212 | 0.3778 | 432 | 299 |
| 100 | 0.4123 | 1.0017 | 912 | 561 |
| 200 | 1.5993 | 2.7025 | 2348 | 1246 |

Сравнение показывает более высокую эффективность модификации при подобранных параметрах (без мод: Т = 100, cooling rate = 0.03, с мод: Т = 10). Значения в таблице взяты при случайных значениях длин рёбер от 1 до 50.

# Вывод

В процессе исследования особенностей алгоритма имитации отжига для решения задачи коммивояжера на ориентированном графе был разработан метод поиска кратчайшего гамильтонова цикла. Написана соответствующая программа, способная находить гамильтонов цикл с использованием алгоритма имитации отжига и определять длину маршрута, охватывающего все вершины. Кроме того, была предложена модификация отжига Больцмана, использующая альтернативную формулу вычисления температуры.

# Источники

# tkinter — tkinter documentation // Documentation URL: <https://docs.python.org/3/library/tkinter.html> (дата обращения: 11.04.2025).

# networkx — networkx documentation // Documentation URL: [https://networkx.org/documentation/latest/](https://networkx.org/documentation/latest/%20) (дата обращения: 11.04.2025).

# math — math documentation // Documentation URL: <https://docs.python.org/3/library/math.html> (дата обращения: 11.04.2025).

# pandas — pandas documentation // Documentation URL: [https://pandas.pydata.org/docs/](https://pandas.pydata.org/docs/%20) (дата обращения: 11.04.2025).

# random — random documentation // Documentation URL: <https://docs.python.org/3/library/random.html> (дата обращения: 11.04.2025).

# Листинг

# 1) main.py

import tkinter as tk

from tsp\_gui import TSPInteractiveGUI

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

    root = tk.Tk()

    gui = TSPInteractiveGUI(root)

    root.mainloop()

# 2) tsp\_solver.py

import random

import math

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

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

    graph\_copy = graph.copy()

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

        return None, 0

    nodes = list(graph\_copy.nodes())

    path = [start\_node]

    unvisited = set(nodes)

    unvisited.remove(start\_node)

    total\_cost = 0

    while unvisited:

        current\_node = path[-1]

        nearest\_neighbor = None

        min\_distance = float('inf')

        for neighbor in unvisited:

            try:

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

            except KeyError:

                distance = float('inf')

            if distance < min\_distance:

                nearest\_neighbor = neighbor

                min\_distance = distance

        if nearest\_neighbor is None:

            return None, 0

        path.append(nearest\_neighbor)

        unvisited.remove(nearest\_neighbor)

        total\_cost += min\_distance

    try:

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

        path.append(path[0])

    except KeyError:

        return None, 0

    return path, total\_cost

def simulated\_annealing(graph, start\_node, initial\_temperature, cooling\_rate, num\_iterations, use\_boltzmann, max\_attempts=20):

    """Implements simulated annealing."""

    nodes = list(graph.nodes())

    if not nodes:

        return None, 0

    if start\_node not in nodes:

        start\_node = random.choice(nodes)

    for attempt in range(max\_attempts):

        current\_solution = nodes.copy()

        random.shuffle(current\_solution)

        if start\_node in current\_solution:

            current\_solution.remove(start\_node)

        current\_solution.insert(0, start\_node)

        current\_cost = calculate\_total\_cost(graph, current\_solution)

        if current\_cost is None:

            continue

        best\_solution = current\_solution.copy()

        best\_cost = current\_cost

        temperature = initial\_temperature

        for i in range(num\_iterations):

            if len(nodes) > 2:

                idx1, idx2 = random.sample(range(1, len(nodes)), 2)

                neighbor\_solution = current\_solution.copy()

                neighbor\_solution[idx1], neighbor\_solution[idx2] = neighbor\_solution[idx2], neighbor\_solution[idx1]

                neighbor\_cost = calculate\_total\_cost(graph, neighbor\_solution)

                if neighbor\_cost is None:

                    continue

                cost\_diff = neighbor\_cost - current\_cost

                if cost\_diff < 0 or (use\_boltzmann and random.random() < math.exp(-cost\_diff / temperature)):

                    current\_solution = neighbor\_solution

                    current\_cost = neighbor\_cost

                    if current\_cost < best\_cost:

                        best\_solution = current\_solution.copy()

                        best\_cost = current\_cost

            if use\_boltzmann:

                temperature = initial\_temperature / math.log(i + 2)

            else:

                temperature \*= (1 - cooling\_rate)

        if best\_cost is not None:

            return best\_solution + [best\_solution[0]], best\_cost

    return None, 0

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

    """Calculates path cost"""

    total\_cost = 0

    for i in range(len(path)-1):

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

            return None

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

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

        return None

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

    return total\_cost

3) tsp\_gui.py

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

import tkinter.simpledialog

import networkx as nx

import math

from tkinter import filedialog

import pandas as pd

import random

import time

from tsp\_solver import simulated\_annealing, nearest\_neighbor

class TSPInteractiveGUI:

    def \_\_init\_\_(self, master):

        self.master = master

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

        # Graph Data Structures

        self.graph = nx.DiGraph()

        self.vertices = {}

        self.edges = {}

        self.edge\_objects = {}

        self.edge\_text\_objects = {}

        self.start\_vertex = None

        self.vertex\_radius = 15

        self.edge\_line\_width = 2

        self.edge\_color = "gray"

        self.vertex\_color = "skyblue"

        self.curve\_offset = 20

        self.arrow\_size = 10

        self.path\_objects = []

        self.best\_path = None

        self.best\_cost = None

        self.last\_edge = None

        self.edge\_history = []

        self.selected\_edge = None

        self.initial\_temperature = 100.0

        self.cooling\_rate = 0.03

        self.num\_iterations = 1000

        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 = 300

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

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

        master.columnconfigure(0, weight=1)

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

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

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

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

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

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

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

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

        master.columnconfigure(1, weight=2)

    def update\_distance\_matrix(self):

        """Обновляет отображение матрицы расстояний."""

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

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

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

        if not vertices:

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

            return

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

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

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

        reversed\_vertices = list(reversed(vertices))

        for v1 in reversed\_vertices:

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

            for v2 in vertices:

                try:

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

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

                except KeyError:

                    row += "i   "

            row += "\n"

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

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

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

        control\_frame = ttk.Frame(master)

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

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

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

        run\_button = ttk.Button(general\_frame, text="Найти кратчайший путь (Отжиг)", command=self.run\_simulated\_annealing)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        sa\_params\_frame = ttk.LabelFrame(control\_frame, text="Параметры отжига")

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

        ttk.Label(sa\_params\_frame, text="T init:").grid(row=0, column=0, padx=5, pady=2) #Use grid

        self.initial\_temperature\_entry = ttk.Entry(sa\_params\_frame, width=5)

        self.initial\_temperature\_entry.insert(0, str(self.initial\_temperature))

        self.initial\_temperature\_entry.grid(row=0, column=1, padx=5, pady=2) #Use grid

        ttk.Label(sa\_params\_frame, text="Cooling rate:").grid(row=1, column=0, padx=5, pady=2) #Use grid

        self.cooling\_rate\_entry = ttk.Entry(sa\_params\_frame, width=5)

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

        self.cooling\_rate\_entry.grid(row=1, column=1, padx=5, pady=2) #Use grid

        ttk.Label(sa\_params\_frame, text="Iterations:").grid(row=2, column=0, padx=5, pady=2) #Use grid

        self.num\_iterations\_entry = ttk.Entry(sa\_params\_frame, width=5)

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

        self.num\_iterations\_entry.grid(row=2, column=1, padx=5, pady=2) #Use grid

        self.modification\_var = tk.BooleanVar()

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

        modification\_check.grid(row=3, column=0, columnspan=2, padx=5, pady=2) #Use grid

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        x, y = event.x, event.y

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

        if closest\_vertex:

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

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

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

                self.selected\_vertices = []

                if v1 == v2:

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

                else:

                    weight = self.ask\_for\_weight()

                    if weight is not None:

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

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

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

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

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

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

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

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

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

                        edge = (v1, v2)

                        self.edge\_history.append(edge)

                        self.update\_distance\_matrix()

        else:

            self.add\_vertex\_event(event)

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

        x, y = event.x, event.y

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

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

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

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

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

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

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

        self.update\_distance\_matrix()

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

        """Вычисляет координаты середины линии."""

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

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

        return mid\_x, mid\_y

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

        """Рисует стрелку-треугольник."""

        arrow\_size = self.arrow\_size

        points = [

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

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

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

        ]

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

        return arrow\_id

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

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

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

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

      dx = x2 - x1

      dy = y2 - y1

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

      if distance == 0:

          return

      dx /= distance

      dy /= distance

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

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

      ctrl\_x1 = x1 + offset\_x

      ctrl\_y1 = y1 + offset\_y

      ctrl\_x2 = x2 + offset\_x

      ctrl\_y2 = y2 + offset\_y

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

                                           smooth=True,

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

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

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

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

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

      return edge\_id, text\_id

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

        closest\_vertex = None

        min\_distance = max\_distance

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

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

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

            if distance < min\_distance:

                closest\_vertex = vertex

                min\_distance = distance

        return closest\_vertex

    def ask\_for\_weight(self, ):

        weight = None

        while weight is None:

            try:

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

                if weight\_str is None:

                    return None

                weight = int(weight\_str)

                if weight <= 0:

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

                    weight = None

            except ValueError:

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

                weight = None

        return weight

    def set\_start\_vertex(self):

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

        if start\_vertex in self.vertices:

            self.start\_vertex = start\_vertex

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

            self.draw\_graph()

        else:

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

            self.start\_vertex = None

    def delete\_last\_vertex(self):

        if self.vertices:

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

            edges\_to\_delete = []

            for (v1, v2) in self.edges:

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

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

            for v1, v2 in edges\_to\_delete:

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

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

                        self.canvas.delete(edge\_id)

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

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

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

                    self.canvas.delete(text\_id)

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

                if (v1, v2) in self.edges:

                    del self.edges[(v1, v2)]

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

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

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

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

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

            del self.vertices[vertex\_to\_delete]

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

            self.update\_distance\_matrix()

            self.draw\_graph()

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

                self.start\_vertex = None

    def draw\_graph(self):

        self.canvas.delete("all")

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

            for i, weight in enumerate(weights):

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

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

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

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

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

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

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

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

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

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

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

    def clear\_graph(self):

        self.graph.clear()

        self.vertices.clear()

        self.edges.clear()

        self.edge\_objects.clear()

        self.edge\_text\_objects.clear()

        self.start\_vertex = None

        self.path = None

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

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

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

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

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

        self.canvas.delete("all")

        self.update\_distance\_matrix()

    def get\_sa\_parameters(self):

        """Gets simulated annealing parameters from entry fields."""

        try:

            self.initial\_temperature = float(self.initial\_temperature\_entry.get())

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

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

            if self.initial\_temperature <= 0 or self.cooling\_rate <= 0 or self.cooling\_rate >= 1 or self.num\_iterations <= 0:

                messagebox.showerror("Ошибка", "Параметры отжига должны быть положительными. Охлаждение должно быть от 0 до 1.")

                return False

            return True

        except ValueError:

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

            return False

    def run\_simulated\_annealing(self):

        """Runs simulated annealing to find the TSP solution."""

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

            return

        start\_time = time.time()

        if not self.graph.nodes():

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

            return

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

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

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

        self.clear\_temporary\_edge()

        self.delete\_last\_path()

        best\_path, best\_cost = simulated\_annealing(

            self.graph,

            self.start\_vertex,

            self.initial\_temperature,

            self.cooling\_rate,

            self.num\_iterations,

            self.modification\_var.get(),

            max\_attempts=200

        )

        end\_time = time.time()

        calculation\_time = end\_time - start\_time

        if best\_path:

            self.best\_path = best\_path

            self.best\_cost = best\_cost

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

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

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

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

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

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

            self.draw\_graph\_with\_path()

        else:

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

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

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

        total\_cost = 0

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

            try:

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

            except KeyError:

                return float('inf')

        return total\_cost

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

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

        start\_time = time.time()

        graph\_copy = self.graph.copy()

        edges\_copy = self.edges.copy()

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

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

        best\_path = None

        min\_cost = float('inf')

        best\_start\_node = None

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

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

            return

        if self.modification\_var.get():

            start\_nodes = self.vertices.keys()

        else:

            start\_nodes = [self.start\_vertex]

        for start\_node in start\_nodes:

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

            if path and total\_cost < min\_cost:

                min\_cost = total\_cost

                best\_path = path

                best\_start\_node = start\_node

        end\_time = time.time()

        calculation\_time = end\_time - start\_time

        self.graph = graph\_copy

        self.edges = edges\_copy

        self.edge\_objects = edge\_objects\_copy

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

        self.draw\_graph()

        self.update\_distance\_matrix()

        if best\_path:

            self.best\_path = best\_path

            self.best\_cost = min\_cost

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

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

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

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

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

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

            self.draw\_graph\_with\_path()

        else:

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

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

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

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

            return

        self.path\_objects = []

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

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

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

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

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

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

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

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

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

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

    def delete\_last\_edge(self):

        """Удаляет последнее добавленное ребро."""

        if self.edge\_history:

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

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

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

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

                  self.canvas.delete(edge\_id)

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

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

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

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

                    self.canvas.delete(text\_id)

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

            if (v1, v2) in self.edges:

                del self.edges[(v1, v2)]

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

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

            self.update\_distance\_matrix()

            self.draw\_graph()

            if not self.edge\_history:

                self.last\_edge = None

    def delete\_last\_path(self):

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

        for obj\_id in self.path\_objects:

            self.canvas.delete(obj\_id)

        self.path\_objects = []

        self.best\_path = None

        self.best\_cost = None

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

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

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

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

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

        self.draw\_graph()

    def clear\_temporary\_edge(self):

        self.selected\_vertices = []

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

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

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

        if filename:

            try:

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

                vertices = list(df.index)

                distances = df.to\_dict()

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

            except FileNotFoundError:

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

            except Exception as e:

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

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

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

        self.clear\_graph()

        for vertex in vertices:

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

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

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

        for v1 in vertices:

            for v2 in vertices:

                if v1 != v2:

                    weight = distances[v2][v1]

                    if weight <= 0:

                        continue

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

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

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

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

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

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

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

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

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

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

                        edge = (v1, v2)

                        self.edge\_history.append(edge)

        self.update\_distance\_matrix()

        self.draw\_graph()

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

        """Добавляет вершину из матрицы."""

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

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

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

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

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

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

    def update\_edge\_weight(self):

        """Обновляет вес выбранного ребра."""

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

        try:

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

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

                raise ValueError("Указанная вершина не существует.")

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

                raise ValueError("Указанное ребро не существует.")

            try:

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

                if new\_weight <= 0:

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

                    return

            except ValueError:

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

                return

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

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

            self.update\_distance\_matrix()

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

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

                    self.canvas.delete(text\_id)

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

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

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

            self.draw\_graph()

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

        except ValueError as e:

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