**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. 22.Б15 |  | Агишев А.Б. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2024 г.**

**Содержание**

[1. Цель работы 1](#_Toc151413474)

[2. Задача 1](#_Toc151413475)

[3. Теоретическая часть 1](#_Toc151413476)

[4. Описание алгоритма 2](#_Toc151413477)

[5. Описание программы 4](#_Toc151413478)

[5.1 Описание классов 4](#_Toc151413479)

[5.2. Описание функций 4](#_Toc151413480)

[5.3. Описание переменных 5](#_Toc151413481)

[6. Рекомендации пользователю 7](#_Toc151413482)

[7. Рекомендации программисту 7](#_Toc151413483)

[8. Контрольный пример 8](#_Toc151413484)

[9. Заключение 9](#_Toc151413485)

# **Цель работы**

Разработать алгоритм и программу, которая решает задачу о коммивояжере с помощью алгоритма имитации отжига и сравнить с работой алгоритма ближайшего соседа.

# **Задача**

Приобрести и закрепить знания в области решения задачи о коммивояжере. Написать программу поиска решения. Протестировать программу на контрольном примере. Сделать заключение об алгоритме имитации отжига в решении данной задачи. Сравнить работу с методом ближайшего соседа.

# **Теоретическая часть**

*Гамильтонов цикл в неориентированном графе* — это такой цикл, который проходит через каждую вершину графа ровно один раз и возвращается в начальную вершину. Таким образом, гамильтонов цикл охватывает все вершины графа без повторений.

*Задача коммивояжера* является одной из классических задач комбинаторной оптимизации. Она заключается в поиске кратчайшего маршрута, который проходит через все заданные города (вершины графа) ровно один раз и возвращается в исходный город. Задача коммивояжера является NP-полной, то есть нет известного алгоритма, который бы эффективно решал её для любого входа за полиномиальное время. Однако существуют различные эвристические и точные алгоритмы для её приближенного или оптимального решения.

*Алгоритм имитации отжига* (Simulated Annealing) — это метод оптимизации, вдохновленный процессом отжига металла. Он применяется для решения сложных задач оптимизации, таких как задача коммивояжера или размещение объектов.

Принципы работы алгоритма:

* Начальное состояние: начинается с некоторого начального состояния.
* Случайный поиск: осуществляется случайный поиск по пространству состояний.
* Функция оценки: определяется функция оценки, которая оценивает качество решения.
* Принятие худшего решения: Алгоритм принимает худшее решение с некоторой вероятностью, которая уменьшается по мере "охлаждения".
* Охлаждение: "Температура" постепенно уменьшается, что уменьшает вероятность принятия худшего решения.

Алгоритм имитации отжига не гарантирует нахождение оптимального решения, но обеспечивает приближенное решение за счет случайного поиска и способности принимать временно худшие решения.

# **Описание алгоритма**

1. Получение характеристики графа, параметров для алгоритма через интерфейс программы.
2. Запуск поиска решения задачи о коммивояжере методом имитации отжига.
3. Инициализация начального решения как случайного остовного дерева.
4. Поиск соседнего решения и его веса.
5. Принятие соседнего решения в зависимости от его стоимости и температуры.
6. Обновление текущего решения в случае принятия соседнего.
7. Завершение алгоритма, когда выполнено заданное количество итераций.
8. Вычисление длины маршрута и порядка обхода.
9. Возврат результата - длины маршрута и порядка обхода в качестве ответа.

В рассматриваемом коде была внесена модификация алгоритма имитации отжига, где для вероятностного перехода к случайным изменениям выбирается наилучшее из ранее найденных решений. Эта изменение привело к повышению эффективности алгоритма, однако, следует отметить, что время поиска решения увеличилось.

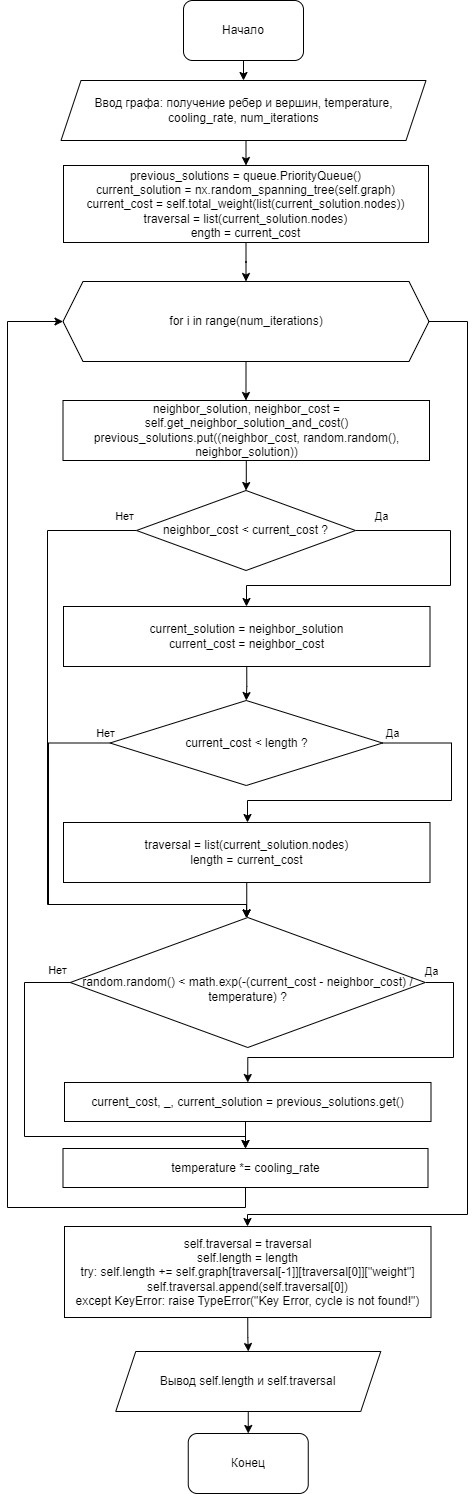


Рисунок 4.1. Блок-схема алгоритма.

# **Описание программы**

## **Описание классов**

В программе используется 3 класса: 2 связанных с интерфейсом программы, 1 связанный с поиском решения задачи. В таблице 5.1 представлено описание классов.

*Таблица 5.1. Описание классов*

|  |  |  |
| --- | --- | --- |
| Имя класса | Наследование | Описание |
| Traveling\_Salesman | — | Поиск решения задачи о коммивояжере |
| GraphEditor | — | Редактирование графа в интерфейсе |
| Interface | — | Графический пользовательский интерфейс |

## **5.2. Описание функций**

Описание функций класса *«Traveling\_Salesman»* представлено в таблице 5.2.

*Таблица 5.2. Описание функций класса «Traveling\_Salesman»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| Экземпляр редактора графа | \_\_init\_\_ | Инициализация класса |
| Порядок обхода | total\_weight | Общий вес ребер порядка обхода |
| — | get\_neighbor\_solution\_and\_cost | Создание случайного порядка обхода |
| Начальная температура, коэффициент охлаждения, количество итераций | simulated\_annealing | Поиск решения задачи о коммивояжере |
| Шаблоны графических элементов | view | Вывод в интерфейс порядка обхода |

Описание функций класса *«GraphEditor»* представлено в таблице 5.3.

*Таблица 5.3. Описание функций класса «GraphEditor»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| Экземпляр методов интерфейса, аргументы | \_\_init\_\_ | Инициализация класса |
| Координаты вершины в интерфейсе | on\_left\_click | Создание вершины в графе |
| Координаты вершины в интерфейсе | on\_right\_click | Создание ребра между вершинами в графе |
| Координаты вершины в интерфейсе | get\_clicked\_vertex | Проверка нажатия на вершину графа |
| — | clear\_graph | Очистка интерфейса графа |

Описание функций класса *«Interface»* представлено в таблице 5.4.

*Таблица 5.4. Описание функций класса «Interface»*

|  |  |  |
| --- | --- | --- |
| Входные данные | Имя | Описание |
| — | \_\_init\_\_ | Инициализация класса |
| — | create\_interface | Создание элементов в интерфейсе |
| Событие | prevent\_typing | Запрет на редактирование  элемента интерфейса |
| — | clear\_output | Очистка интерфейса |
| — | populate\_edge\_table | Вывод таблицы с вершинами и ребрами графа |
| Вершина 1, вершина 2, вес ребра | update\_weight | Обновление веса между ребрами |
| — | threading\_run | Запуск функции *«start\_process»* в отдельном потоке |
| — | start\_process | Запуск решения поиска задачи о коммивояжере |

## **5.3. Описание переменных**

*Таблица 5.5. Описание переменных функции «simulated\_annealing»*

|  |  |  |
| --- | --- | --- |
| Тип | Имя | Описание |
| PriorityQueue | previous\_solutions | Очередь всех найденных решений |
| networkx.graph | current\_solution | Текущее решение |
| int | current\_cost | Текущий общий вес решения |
| List[int] | traversal | Порядок обхода графа |
| int | length | Длина порядка обхода графа |
| networkx.graph | neighbor\_solution | Соседнее решение |
| int | neighbor\_cost | Общий вес соседнего решения |
| float | temperature | Температура |
| str | result | Результат задачи в виде итоговой длины обхода и порядка обхода |

Полный код программы представлен в Приложении А.

# **Рекомендации пользователю**

Верхнее поле устанавливает редактор для графа с возможностью задания графа путем добавления вершин и ребер. Левой кнопкой мыши создается вершина в графе, а правая кнопка отвечает за создание ребра между двумя вершинам. Поле *«Вес»* изменяет вес ребра на заданное пользователем. Поля *«Начальная температура»*, *«Коэфф. охлаждения»*, *«Количество итераций»* задают значения температуры, коэффициента охлаждения, количества итераций.

Кнопка *«Рассчитать»* отвечает за старт поиска решения задачи о коммивояжере методом ближайшего соседа. Ответ, а также итоговый обход графа выводится в интерфейс.

Кнопка *«Очистить»* отвечает за очистку интерфейса от всех пользовательских данных.

Для завершения работы нажмите на крестик в левом верхнем углу.

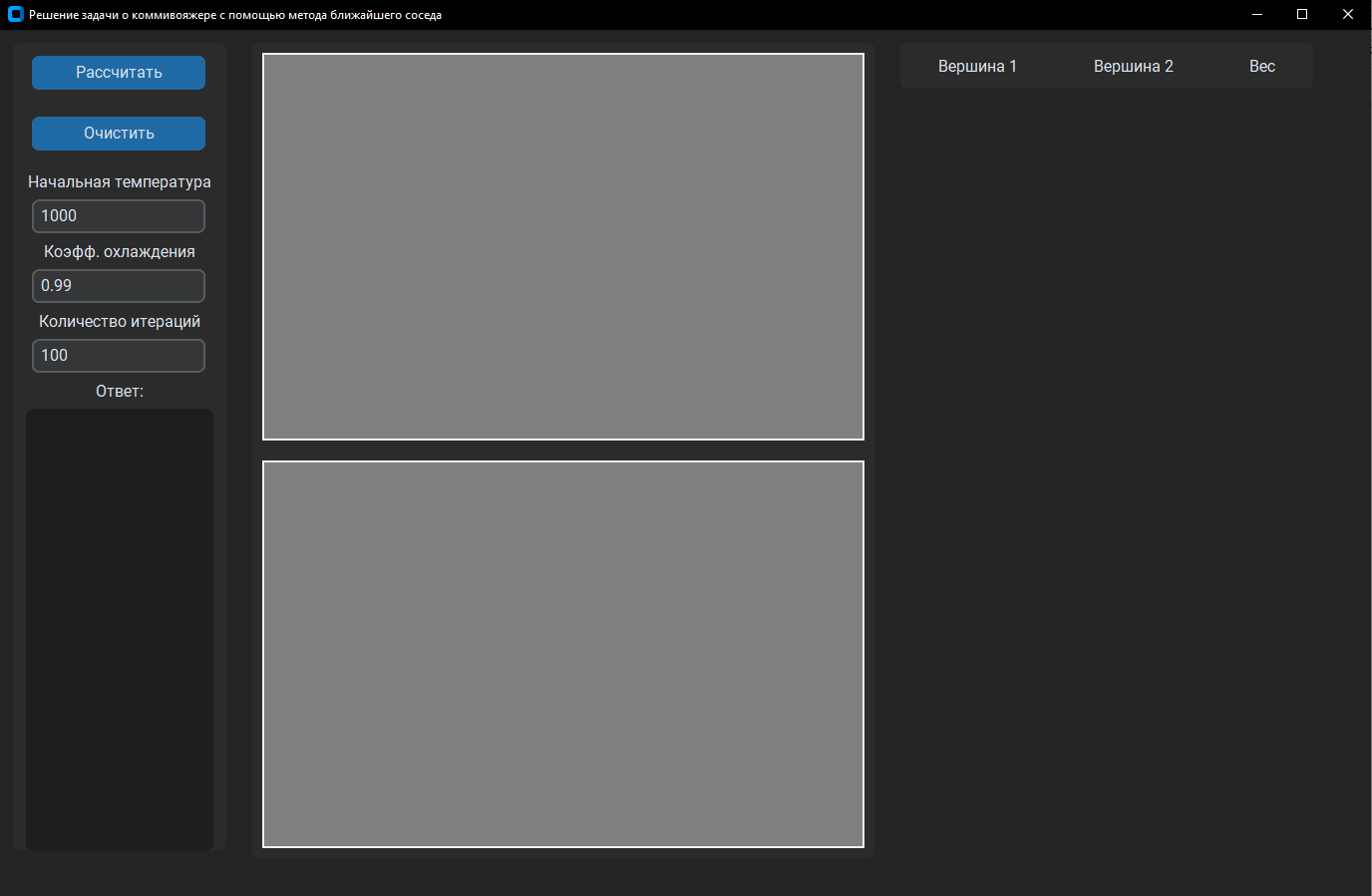
# **Рекомендации программисту**

Для запуска программы необходим Python версии не ниже 3.10.6. Предварительно необходимо установить библиотеки: customtkinter версии не ниже 5.2.0, networkx версии не ниже 3.0.

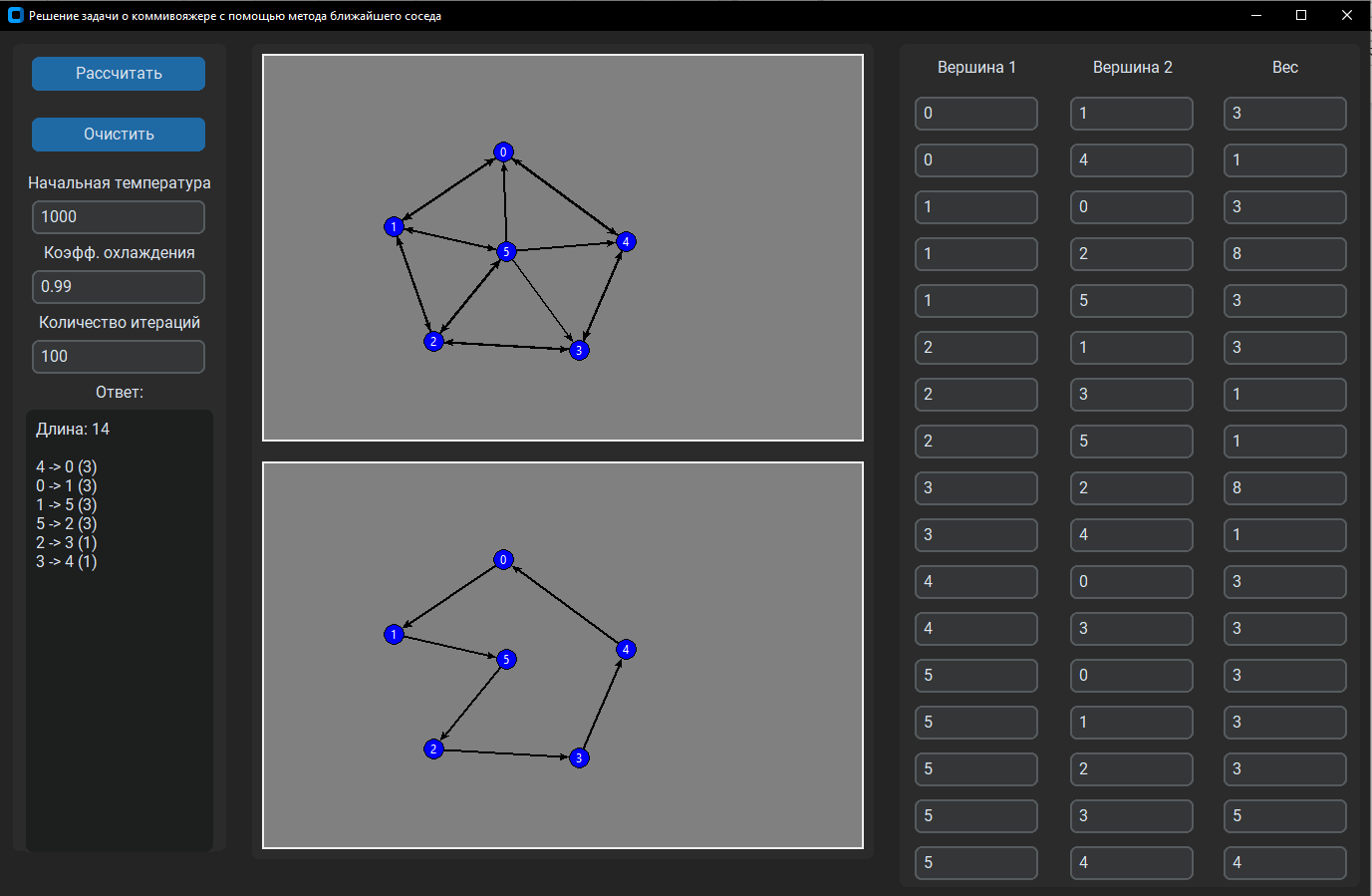
Минимальное необходимое место на диске: 0.5 МБ. Минимальное необходимое количество оперативной памяти: 100 МБ.

# **Контрольный пример**

В данном разделе представлен контрольный пример, демонстрирующий способность программы находить кратчайший гамильтонов цикл в графе.



*Рисунок 8.1. Интерфейс программы*



*Рисунок 8.2. Вывод решения задачи*

В сравнении поиска решения задачи коммивояжера методом ближайшего соседа алгоритм имитации отжига находит более оптимальные решения, но при этом затрачивает больше времени. Результаты тестирований представлены в таблице 8.3.

*Таблица 8.3. Сравнение двух методов решения задачи коммивояжера.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество вершин в графе | Метод ближайшего соседа, с | Метод ближайшего соседа с оптимизацией, с | Алгоритм имитации отжига, с | Алгоритм имитации отжига с оптимизацией, с |
| 10 | 0.5 | 1 | 2 | 3 |
| 20 | 2 | 4 | 5 | 8 |
| 50 | 20 | 40 | 50 | 90 |

# **Заключение**

В данной работе была исследована задача о коммивояжере, которая представляет собой задачу нахождения кратчайшего маршрута, проходящего через все заданные города ровно один раз и возвращающегося в исходный город. Эта проблема является одной из классических задач комбинаторной оптимизации и относится к классу NP-полных задач.

В работе был рассмотрен алгоритм имитации отжига, который является методом глобальной оптимизации. Для оптимизации алгоритма использовался выбор лучшего решения среди уже найденных, что позволяет найти более оптимальный гамильтонов цикл. Это увеличивает время работы программы, но обеспечивает более точный результат.

В результате сравнения двух алгоритмов было установлено, что алгоритм имитации отжига, с использованием оптимизации запуска метода ближайшего соседа от каждой вершины, демонстрирует более высокую точность нахождения оптимального решения по сравнению с методом ближайшего соседа. Однако, время выполнения алгоритма имитации отжига существенно выше.

1. **Приложение А**

import math  
import queue  
import random  
import threading  
import networkx as nx  
import customtkinter as ctk  
  
  
# Function Traveling Salesman algorithm  
class Traveling\_Salesman:  
 def \_\_init\_\_(self, graph\_editor):  
 self.graph\_editor = graph\_editor  
 self.edges = graph\_editor.edges  
 self.graph = graph\_editor.graph  
 self.length = float("inf")  
 self.traversal = {}  
  
 # Function to calculate total weight of traversal  
 def total\_weight(self, traversal):  
 total = 0  
 for i in range(len(traversal) - 1):  
 edge\_data = self.graph.get\_edge\_data(traversal[i], traversal[i + 1])  
 weight = edge\_data["weight"] if edge\_data else float("inf")  
 total += weight  
 return total  
  
 # Return neigbor solution and cost  
 def get\_neighbor\_solution\_and\_cost(self):  
 for \_ in range(30):  
 neighbor\_solution = nx.random\_spanning\_tree(self.graph)  
 neighbor\_cost = self.total\_weight(list(neighbor\_solution.nodes))  
 if neighbor\_cost < 10 \*\* 9:  
 return neighbor\_solution, neighbor\_cost  
 return None, float("inf")  
  
 # Algorithm  
 def simulated\_annealing(self, temperature, cooling\_rate, num\_iterations):  
 previous\_solutions = queue.PriorityQueue()  
 current\_solution = nx.random\_spanning\_tree(self.graph)  
 current\_cost = self.total\_weight(list(current\_solution.nodes))  
 traversal = list(current\_solution.nodes)  
 length = current\_cost  
  
 for i in range(num\_iterations):  
 neighbor\_solution, neighbor\_cost = self.get\_neighbor\_solution\_and\_cost()  
 previous\_solutions.put((neighbor\_cost, random.random(), neighbor\_solution))  
  
 if neighbor\_cost < current\_cost:  
 current\_solution = neighbor\_solution  
 current\_cost = neighbor\_cost  
 if current\_cost < length:  
 traversal = list(current\_solution.nodes)  
 length = current\_cost  
  
 if random.random() < math.exp(-(current\_cost - neighbor\_cost) / temperature):  
 current\_cost, \_, current\_solution = previous\_solutions.get()  
 temperature \*= cooling\_rate  
  
 self.traversal = traversal  
 self.length = length  
  
 try:  
 self.length += self.graph[traversal[-1]][traversal[0]]["weight"]  
 self.traversal.append(self.traversal[0])  
 except KeyError:  
 raise TypeError("Key Error, cycle is not found!")  
  
 result = f"Длина: {self.length}\n\n"  
 for i in range(len(traversal) - 1):  
 result += f'{self.traversal[i]} -> {self.traversal[i + 1]} ({self.graph[self.traversal[i]][self.traversal[i + 1]]["weight"]})\n'  
 return result  
  
 # Draw graph with his traversal  
 def view(self, canvas):  
 canvas.clear\_graph()  
 for i in range(len(self.traversal) - 1):  
 start\_vertex = self.traversal[i]  
 end\_vertex = self.traversal[i + 1]  
 start\_x, start\_y = self.graph\_editor.vertices[start\_vertex]  
 end\_x, end\_y = self.graph\_editor.vertices[end\_vertex]  
 length = ((end\_x - start\_x) \*\* 2 + (end\_y - start\_y) \*\* 2) \*\* 0.5  
 arrow\_offset = 10  
 sx = start\_x + (end\_x - start\_x) \* (arrow\_offset / length)  
 sy = start\_y + (end\_y - start\_y) \* (arrow\_offset / length)  
 ex = end\_x - (end\_x - start\_x) \* (arrow\_offset / length)  
 ey = end\_y - (end\_y - start\_y) \* (arrow\_offset / length)  
 canvas.create\_line(sx, sy, ex, ey, arrow="last", width=2)  
  
 for vertex, (x, y) in enumerate(self.graph\_editor.vertices):  
 canvas.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="blue", tags="vertex")  
 canvas.create\_text(x, y, text=str(vertex), fill="white", tags="vertex\_text")  
  
  
# Class for editing a graph on a canvas  
class GraphEditor(ctk.CTkCanvas):  
 def \_\_init\_\_(self, master, interface, \*\*kwargs):  
 super().\_\_init\_\_(master, \*\*kwargs)  
 self.bind("<Button-1>", self.on\_left\_click)  
 self.bind("<Button-3>", self.on\_right\_click)  
 self.vertices = []  
 self.edges = []  
 self.selected\_vertex = None  
 self.graph = nx.DiGraph()  
 self.interface = interface  
  
 # Function to handle left-click events on the canvas  
 def on\_left\_click(self, event):  
 x, y = event.x, event.y  
 self.create\_oval(x - 10, y - 10, x + 10, y + 10, fill="blue", tags="vertex")  
 self.create\_text(x, y, text=str(len(self.graph)), fill="white", tags="vertex\_text")  
 self.vertices.append((x, y))  
 self.graph.add\_node(len(self.graph))  
  
 # Function to handle right-click events on the canvas  
 def on\_right\_click(self, event):  
 x, y = event.x, event.y  
 vertex = self.get\_clicked\_vertex(x, y)  
 if vertex is not None:  
 if self.selected\_vertex is None:  
 self.selected\_vertex = vertex  
 else:  
 start\_x, start\_y = self.vertices[self.selected\_vertex]  
 end\_x, end\_y = self.vertices[vertex]  
 length = ((end\_x - start\_x) \*\* 2 + (end\_y - start\_y) \*\* 2) \*\* 0.5  
 if length == 0:  
 return  
 arrow\_offset = 10  
 sx = start\_x + (end\_x - start\_x) \* (arrow\_offset / length)  
 sy = start\_y + (end\_y - start\_y) \* (arrow\_offset / length)  
 ex = end\_x - (end\_x - start\_x) \* (arrow\_offset / length)  
 ey = end\_y - (end\_y - start\_y) \* (arrow\_offset / length)  
 self.create\_line(sx, sy, ex, ey, arrow=ctk.LAST, width=2)  
 self.edges.append((self.selected\_vertex, vertex))  
 self.graph.add\_edge(self.selected\_vertex, vertex, weight=int(length))  
 self.selected\_vertex = None  
 self.interface.populate\_edge\_table()  
  
 # Function to get the index of a clicked vertex on the canvas  
 def get\_clicked\_vertex(self, x, y):  
 for i, (vx, vy) in enumerate(self.vertices):  
 if (x - vx) \*\* 2 + (y - vy) \*\* 2 <= 100:  
 return i  
 return None  
  
 # Function to clear the graph on the canvas  
 def clear\_graph(self):  
 self.vertices = []  
 self.edges = []  
 self.graph.clear()  
 self.delete("all")  
  
  
# Class for the graphical user interface  
class Interface(ctk.CTk):  
 def \_\_init\_\_(self):  
 ctk.CTk.\_\_init\_\_(self)  
 ctk.set\_appearance\_mode("dark")  
 ctk.set\_default\_color\_theme("blue")  
 self.title("Решение задачи о коммивояжере с помощью метода ближайшего соседа")  
 self.geometry("1100x670")  
 self.frame1 = None  
 self.frame2 = None  
 self.frame3 = None  
 self.graph\_view = None  
 self.clear\_button = None  
 self.answer\_label = None  
 self.graph\_editor = None  
 self.output\_text = None  
 self.process\_button = None  
 self.num\_iteration = None  
 self.num\_iteration\_text = None  
 self.coeff\_freeze = None  
 self.coeff\_freeze\_text = None  
 self.temperature = None  
 self.temperature\_text = None  
 self.edge\_table = {}  
 self.create\_interface()  
  
 # Function to create the graphical user interface  
 def create\_interface(self):  
 self.frame1 = ctk.CTkFrame(self)  
 self.frame1.grid(row=0, column=0, padx=10, pady=10, sticky="n")  
  
 self.process\_button = ctk.CTkButton(self.frame1, text="Рассчитать", command=self.threading\_run)  
 self.process\_button.pack(side="top", padx=10, pady=10)  
  
 self.clear\_button = ctk.CTkButton(self.frame1, text="Очистить", command=self.clear\_output)  
 self.clear\_button.pack(side="top", padx=10, pady=10)  
  
 self.temperature\_text = ctk.CTkLabel(self.frame1, text="Начальная температура")  
 self.temperature\_text.pack(side="top", padx=10)  
  
 self.temperature = ctk.CTkEntry(self.frame1, width=140)  
 self.temperature.pack(side="top", padx=10)  
 self.temperature.insert(0, "1000")  
  
 self.coeff\_freeze\_text = ctk.CTkLabel(self.frame1, text="Коэфф. охлаждения")  
 self.coeff\_freeze\_text.pack(side="top", padx=10)  
  
 self.coeff\_freeze = ctk.CTkEntry(self.frame1, width=140)  
 self.coeff\_freeze.pack(side="top", padx=10)  
 self.coeff\_freeze.insert(0, "0.99")  
  
 self.num\_iteration\_text = ctk.CTkLabel(self.frame1, text="Количество итераций")  
 self.num\_iteration\_text.pack(side="top", padx=10)  
  
 self.num\_iteration = ctk.CTkEntry(self.frame1, width=140)  
 self.num\_iteration.pack(side="top", padx=10)  
 self.num\_iteration.insert(0, "100")  
  
 self.answer\_label = ctk.CTkLabel(self.frame1, text="Ответ:")  
 self.answer\_label.pack(side="top", padx=10, fill=ctk.BOTH)  
  
 self.output\_text = ctk.CTkTextbox(self.frame1, height=355, width=150)  
 self.output\_text.pack(side="top", padx=10)  
 self.output\_text.bind("<KeyPress>", self.prevent\_typing)  
  
 self.frame2 = ctk.CTkFrame(self)  
 self.frame2.grid(row=0, column=1, padx=10, pady=10, sticky="n")  
  
 self.graph\_editor = GraphEditor(self.frame2, width=600, height=385, bg="grey", interface=self)  
 self.graph\_editor.pack(side="top", padx=10, pady=10)  
  
 self.graph\_view = GraphEditor(self.frame2, width=600, height=385, bg="grey", interface=self)  
 self.graph\_view.pack(side="top", padx=10, pady=10)  
 self.graph\_view.bind("<Button-1>", self.prevent\_typing)  
  
 self.frame3 = ctk.CTkFrame(self)  
 self.frame3.grid(row=0, column=2, padx=10, pady=10, sticky="n")  
  
 for col, header in enumerate(["Вершина 1", "Вершина 2", "Вес"]):  
 label = ctk.CTkLabel(self.frame3, text=header)  
 label.grid(row=0, column=col, padx=30, pady=5)  
  
 self.populate\_edge\_table()  
  
 # Function to prevent typing in the output textbox  
 def prevent\_typing(self, event):  
 return "break"  
  
 # Function to clear the output textbox and the graph on the canvas  
 def clear\_output(self):  
 self.graph\_editor.clear\_graph()  
 self.graph\_view.clear\_graph()  
 self.output\_text.delete("1.0", ctk.END)  
 self.populate\_edge\_table()  
  
 # Function to populate the edge table with data from the graph  
 def populate\_edge\_table(self):  
 if self.edge\_table:  
 for widgets in self.edge\_table.values():  
 for widget in widgets:  
 widget.destroy()  
 self.edge\_table = {}  
  
 sorted\_edges = sorted(self.graph\_editor.graph.edges(data="weight"))  
  
 for row, (vertex1, vertex2, weight) in enumerate(sorted\_edges, start=1):  
 entry\_vertex1 = ctk.CTkEntry(self.frame3, width=100)  
 entry\_vertex1.insert(ctk.END, vertex1)  
 entry\_vertex1.grid(row=row, column=0, padx=10, pady=5)  
 entry\_vertex1.bind("<KeyPress>", self.prevent\_typing)  
  
 entry\_vertex2 = ctk.CTkEntry(self.frame3, width=100)  
 entry\_vertex2.insert(ctk.END, vertex2)  
 entry\_vertex2.grid(row=row, column=1, padx=10, pady=5)  
 entry\_vertex2.bind("<KeyPress>", self.prevent\_typing)  
  
 entry\_weight = ctk.CTkEntry(self.frame3, width=100)  
 entry\_weight.insert(ctk.END, weight)  
 entry\_weight.grid(row=row, column=2, padx=10, pady=5)  
 entry\_weight.bind("<FocusOut>",  
 lambda event, vertex1=vertex1, vertex2=vertex2, entry\_weight=entry\_weight: self.update\_weight(vertex1, vertex2,  
 entry\_weight))  
  
 self.edge\_table[row] = [entry\_vertex1, entry\_vertex2, entry\_weight]  
  
 # Function to update weights in graph  
 def update\_weight(self, vertex1, vertex2, entry\_weight):  
 new\_weight = entry\_weight.get()  
 self.graph\_editor.graph[vertex1][vertex2]["weight"] = int(new\_weight)  
  
 # Function to run a process in a separate thread  
 def threading\_run(self):  
 t = threading.Thread(target=self.start\_process)  
 t.start()  
  
 # Placeholder function for starting a process  
 def start\_process(self):  
 self.output\_text.delete("1.0", ctk.END)  
 self.graph\_view.clear\_graph()  
  
 salesman = Traveling\_Salesman(self.graph\_editor)  
 self.output\_text.insert(ctk.END, salesman.simulated\_annealing(int(self.temperature.get()), float(self.coeff\_freeze.get()),  
 int(self.num\_iteration.get())))  
 salesman.view(self.graph\_view)  
  
  
# Main block to run the GUI application  
if \_\_name\_\_ == "\_\_main\_\_":  
 gui = Interface()  
 gui.mainloop()

*Листинг программы.*