САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

ОТЧЕТ

по лабораторной работе №8

по дисциплине «Алгоритмы и структуры данных»

на тему «Метод ближайшего соседа»

Выполнила
студентка 2 курса
группы 21-Б15.ПУ
Павлова Ксения Андреевна

Преподаватель
Дик Александр Геннадьевич

Санкт-Петербург
2023

СОДЕРЖАНИЕ

Цель: нахождение кратчайшего гамильтонова цикла во взвешенном графе методом ближайшего соседа.

Задачи:

1. Формализовать задачу о коммивояжере с помощью алгоритма ближайшего соседа.
2. Подготовить контрольный пример, используя взвешенный граф.
3. Найти кратчайший гамильтонов цикл.
4. Написать пользовательский интерфейс для удобства пользования алгоритмом.

Ход работы.

Краткое описание алгоритма метода ближайшего соседа.

Задача о коммивояжёре.

Название «Задача о коммивояжёре» устойчиво закрепилось за одной из самых интересных, практически значимых и одновременно сложных задач теории графов. Задача, берущая свое начало из работ Гамильтона, состоит в определении кратчайшего гамильтонова цикла в графе. Ее решение связано с решением задачи о назначениях и с задачей об остове наименьшего веса.

Алгоритм ближайшего соседа.

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

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

Алгоритм прост в реализации, быстро выполняется, но, как и другие «жадные» алгоритмы, может выдавать неоптимальные решения. Одним из эвристических критериев оценки решения является правило: если путь, пройденный на последних шагах алгоритма, сравним с путём, пройденным на начальных шагах, то можно условно считать найденный маршрут приемлемым, иначе, вероятно, существуют более оптимальные решения. Другой вариант оценки решения заключается в использовании алгоритма нижней граничной оценки.

Для любого количества городов, большего трёх, в задаче коммивояжёра можно подобрать такое расположение городов (значение расстояний между вершинами графа и указание начальной вершины), что алгоритм ближайшего соседа будет выдавать наихудшее решение.

Описание схемы пошагового выполнения алгоритма.

Шаги алгоритма:

1. Поставить все вершины как не посещённые.
2. Выбрать начальную вершину *v* и пометить её, как посещённую.
3. Выбрать ближайшую не посещённую смежную вершину *u* к вершине *v*.
4. Поставить *u* как текущую вершину и пометить как посещённую.
5. Если все вершины посещены, то завершить алгоритм. Иначе, вернуться к шагу 3.

На выходе будем иметь последовательность вершин, предположительно оптимального решения.

Формализация задачи.

Для решения поставленной задачи был выбран язык программирования Python. Визуализация выполнялась с помощью библиотеки TKinter.

1. Объявляется пустой массив посещённых вершин «visited».
2. Из имеющихся вершин случайным образом выбирается начальная текущая вершина «current_vertex», она добавляется в массив посещённых вершин.
3. Путем перебора элементов в массиве рёбер графа «edges» и проверки, является ли текущая вершина началом «edge[0]» или концом «edge[1]» этого ребра, выбирается ближайшая не помещённая смежная вершина. Если текущая вершина является началом ребра и конечная вершина еще не была посещена, то проверяется расстояние до конечной вершины и, если оно меньше, чем текущее минимальное расстояние «min_distance», то оно становится новым минимальным расстоянием, а конечная вершина становится следующей вершиной для посещения «newt_vertex». Если текущая вершина является конечной вершиной ребра и начальная вершина еще не была посещена, то происходит

аналогичная проверка и обновление минимального расстояния и следующей вершины для посещения.

4. Следующая вершина для посещения становится текущей и добавляется в массив посещённых вершин.
5. Если количество элементов в массиве посещённых вершин не меньше количество элементов в массиве всех вершин «vertices», то алгоритм завершает работу, иначе, происходит возвращение вернуться к шагу 3.

Листинг программы.

Метод ближайшего соседа.

```
#Блок подключения необходимых библиотек
import random

def nearest_neighbor(canvas):
    #Инициализация
    canvas.best_edges_delete()
    canvas.edges_mass =
canvas.table.check_data(canvas.edges_mass)
    vertices, edges = canvas.circle_mass, canvas.edges_mass
#Массивы вершин и рёбер
    visited = [] #Массив посещённых вершин
    current_vertex = random.choice(vertices)#случайный выбор
начальной вершин
    visited.append(current_vertex[0])
    #Пока количество элементов в массиве посещённых вершин меньше
количества элементов в массиве всех вершин
    while len(visited) < len(vertices):
        next_vertex = None
        min_distance = float("inf")
        #Поиск ближайшей не посещённой смежной вершины и
определение следующей вершины для посещения
        for edge in edges:
            if current_vertex[0] == edge[0] and edge[1] not in
visited:
                if edge[2] < min_distance:
                    min_distance = edge[2]
                    next_vertex = edge[1]
            elif current_vertex[0] == edge[1] and edge[0] not in
visited:
                if edge[2] < min_distance:
                    min_distance = edge[2]
                    next_vertex = edge[0]
        visited.append(next_vertex)
        for vertex in vertices:
            if vertex[0] == next_vertex:
                current_vertex = vertex
```

```
                break
    visited.append(visited[0])
    total_weight = 0
    #Подсчёт общего веса пути
    for i in range(len(visited) - 1):
        for edge in edges:
            if visited[i] == edge[0] and visited[i+1] ==
edge[1]:
                total_weight += edge[2]
            elif visited[i] == edge[1] and visited[i+1] ==
edge[0]:
                total_weight += edge[2]
    #Результаты для блока визуализации
    canvas.best_way = visited
    canvas.best_weight = total_weight
    canvas.all_edges_hidden(True)
    canvas.draw_best_way()
    return canvas
```

Визуализация.

```
#Блок подключения необходимых библиотек
import tkinter as tk
from tkinter import ttk
from tkinter import simpledialog

class myWindow:
    def __init__(self, root, color):
        self.root = root
        self.color = color
        #Создание главных фреймов
        self.left_frame = tk.Frame(self.root, height=250,
width=500)
        self.right_frame = tk.Frame(self.root, height=250,
width=500)
        self.left_frame.grid(row=0, column=0, sticky="nsew")
        self.right_frame.grid(row=0, column=1, sticky="nsew")
        #Задание пропорций растяжения для колонок
        self.root.columnconfigure(0, weight=1)
        self.root.columnconfigure(1, weight=1)
        #Задание обработчика изменения размера окна
        self.root.bind("<Configure>", self.on_resize)
        #Создание фреймов в левом фрейме
        self.top_left_frame = tk.LabelFrame(self.left_frame,
text='Input area', bg=self.color, height=125, width=500)
        self.middle_left_frame = tk.LabelFrame(self.left_frame,
text='Table area', bg=self.color, height=125, width=500)
        self.bottom_left_frame = tk.LabelFrame(self.left_frame,
text='Result area', bg=self.color, height=125, width=500)
        self.top_left_frame.pack(side="top", fill="both",
expand=True)
        self.middle_left_frame.pack(side="top", fill="both",
expand=True)
```

```
        self.bottom_left_frame.pack(side="bottom", fill="both",
expand=True)
        #Создание фреймов в правом фрейме
        self.top_right_frame = tk.LabelFrame(self.right_frame,
text='Canvas area', bg=self.color, height=125, width=500)
        self.top_right_frame.pack(side="top", fill="both",
expand=True)

    #Функция запуска
    def on_resize(self, event):
        #Получение текущей ширины окна
        width = self.root.winfo_width()
        #Вычисление новой ширины главных фреймов
        left_width = int(width / 3)
        right_width = width - left_width
        #Изменение ширины колонок
        self.root.columnconfigure(0, minsize=left_width)
        self.root.columnconfigure(1, minsize=right_width)

#Общее описание кнопок
class Button:
    def __init__(self, root,text,command,arg):
        self.arg = arg
        self.command = command
        self.root = root
        self.text = text
        button = tk.Button(self.root, text=self.text,
command=lambda: self.command(self.arg))
        button.pack()

#Описание полей с вводом значений
class labeledSpinbox():
    def __init__(self, root,label_text, spinbox_from,
spinbox_to, spinbox_default,spinbox_step):
        self.root = root
        self.label_text = label_text
        self.spinbox_from = spinbox_from
        self.spinbox_to = spinbox_to
        self.spinbox_default = spinbox_default
        self.spinbox_step = spinbox_step
        #Создание виджета Label
        self.widget_frame = tk.Frame(self.root)
        self.widget_frame.pack(side=tk.TOP)
        self.label = tk.Label(self.widget_frame,
text=self.label_text,font=("Arial", 10))
        self.label.pack(side=tk.LEFT)
        # Создание виджета Spinbox
        self.spinbox = tk.Spinbox(self.widget_frame,
from_=self.spinbox_from, to=self.spinbox_to, width=10,
increment=self.spinbox_step,font=("Arial", 10))
        self.spinbox.delete(0, tk.END)  #Удаление стандартного
значения
```

```
        self.spinbox.insert(0, spinbox_default)  #Установка
стандартного значения
        self.spinbox.pack(side=tk.LEFT)

    def get(self):
        return self.spinbox.get()
#Создание холстов и рисование графов
class canvas:
    def __init__(self, root,table,result):
        self.root = root
        self.table = table
        self.result = result
        self.canvas = tk.Canvas(self.root, width=400,
height=400, bg='white')
        self.canvas.pack(fill="both", expand=True)
        self.circle_count = 0
        self.circle_mass = []
        self.edges_mass = []
        self.best_way = []
        self.best_weight=0
        self.counter = 0
        #Привязка функции рисования к событию нажатия мыши
        self.canvas.bind("<Button-1>", self.draw_circle)
        self.canvas.bind("<Button-3>", self.circle_delete)
        self.canvas.bind("<Button-2>", self.show_data)

    #Функция рисования круга на холсте
    def show_data(self,event):
        print(self.edges_mass)
        print(self.circle_mass)
    def draw_circle(self, event):
        self.best_edges_delete()
        self.all_edges_hidden(False)
        x, y = event.x, event.y
        r = 20
        overlapping_circles = self.canvas.find_enclosed(x - r-
        25, y - r-25, x + r+25, y + r+25)  #поиск
перекрывающихся кругов
        if not overlapping_circles:  #если нет пересечений
            #Добавление новой вершины
            self.circle_mass.append([self.circle_count,
            [int(x),int(y)]])
            self.canvas.create_oval(x - r, y - r, x + r, y + r,
            fill='red',tags='oval')
            self.canvas.create_text(x, y,
            text=str(self.circle_count), font=('Arial', 12,
            'bold'),tags='text')
            self.edges_mass =
            self.table.check_data(self.edges_mass)
            self.draw_edges(self.circle_count)
            self.circle_count += 1
            self.table.update_data(self.edges_mass)
```

```
        self.canvas.tag_raise('oval')
        self.canvas.tag_raise('text')

    #Рисование рёбер
    def draw_edges(self, id):
        l = len(self.circle_mass)
        for i in range(l-1):
            x1, y1 = self.circle_mass[l-1][1]
            x2, y2 = self.circle_mass[i][1]
            id1 = self.circle_mass[l-1][0]
            id2 = self.circle_mass[i][0]
            self.canvas.create_line(x1, y1, x2, y2, width=1,
            fill="black",tags='lines')
            weight = ((x2 - x1)**2 + (y2 - y1)**2)**0.5
            self.edges_mass.append([id1,id2,int(weight)])

    #Удаление вершины
    def circle_delete(self, event):
        self.edges_mass = self.table.check_data(self.edges_mass)
        x = event.x
        y = event.y
        r = 20
        objects = self.canvas.find_overlapping(x-r,y-r,x+r,y+r)
        for obj in objects:
            if self.canvas.type(obj) == 'oval':
                coords = self.canvas.coords(obj)
                circle_x = (coords[0] + coords[2]) // 2
                circle_y = (coords[1] + coords[3]) // 2
                for oval in self.circle_mass:
                    if oval[1][0] == circle_x and oval[1][1] ==
                    circle_y:
                        self.delId = oval[0]
                        self.circle_mass.remove(oval)
                        self.edges_delete()
            self.canvas.delete(obj)
        self.table.update_data(self.edges_mass)

    #удаление ребра
    def edges_delete(self):
        l = len(self.edges_mass)
        self.edges_mass = [inner_lst for inner_lst in
        self.edges_mass if self.delId not in inner_lst]

    #скрытие и показ рёбер
    def all_edges_hidden(self,Bool):
        items = self.canvas.find_withtag("lines")
        for item in items:
            if Bool:
                self.canvas.itemconfig(item, state='hidden')
            else:
                self.canvas.itemconfig(item, state='normal')
```

```
    #Удаление рёбер найденного пути
    def best_edges_delete(self):
        items = self.canvas.find_withtag("bestlines")
        for item in items:
          self.canvas.delete(item)

    #Отрисовка найденного пути
    def draw_best_way(self):
        self.result.update_data_res(self.best_way,
        self.best_weight,self.counter)
        coordinates = []
        for i in self.best_way:
            for j in self.circle_mass:
                if j[0] == i:
                    coordinates.append(j[1])
        coordinates.append(coordinates[0])
        for i in range(len(coordinates)-1):
            self.canvas.create_line(coordinates[i][0],
coordinates[i][1], coordinates[i+1][0], coordinates[i+1][1],
width=2, fill="red",tags='bestlines')
        self.canvas.tag_raise('text')

    #Очистка холста
    def clear_canvas(self,bool):
        self.canvas.delete("all")
        self.edges_mass = []
        self.circle_mass = []
        self.best_way = []
        self.best_weight = 0
        self.circle_count=0
        self.counter = 0
        self.table.update_data(self.edges_mass)
        self.result.update_data_res(self.best_way,
        self.best_weight,self.counter)

#Задание таблицы смежности
class myTable:
    def __init__(self,root,headers):
        self.root = root
        self.headers = headers
        self.changed_mass = []
        self.changed_val = []
        self.table = ttk.Treeview(self.root,
        columns=self.headers, show='headings')
        for header in self.headers:
            self.table.heading(header, text=header.title())
        for col in self.table["columns"]:
            self.table.column(col, width=100)
        self.table.pack(side=tk.TOP, fill=tk.X, expand=1)
        self.table.bind("<Double-1>", self.update_cell)
```

```
    #Обновление данных
    def update_data(self,mass):
        self.table.delete(*self.table.get_children())
        for row in mass:
            self.table.insert("", "end", values=row)
    #Проверка данных
    def check_data(self,list1):
        if len(self.changed_mass):
            for i in range(len(list1)):
                if list1[i] in self.changed_mass:
                    list1[i][2] = self.changed_val.pop(0)
            self.changed_mass = []
            self.changed_val = []
        return list1
    #Обновление значение ячеек
    def update_cell(self,event):
        #Получение ссылки на таблицу и на выбранную строку
        self.table = event.widget
        self.item = self.table.selection()[0]
        #Получение индекса выбранной колонки и имени выбранного
столбца
        self.column = self.table.identify_column(event.x)
        self.column_name =
self.table.heading(self.column)['text']
        #Получение старого значения ячейки и запрос у
пользователя нового значения
        self.old_value = self.table.set(self.item, self.column)
        self.id1 = self.table.set(self.item, '#1')
        self.id2 = self.table.set(self.item, '#2')
        self.new_value = simpledialog.askstring('Изменение
значения', f'Введите новое значение для
{self.column_name}:',initialvalue=self.old_value)
        #Обновление значения ячейки, если пользователь ввёл
новое значение
        if self.new_value:
            self.changed_mass.append([int(self.id1),
            int(self.id2),int(self.old_value)])
            self.changed_val.append(int(self.new_value))
            self.table.set(self.item, self.column,
            self.new_value)

#Задание блока вывода результатов
class myResult:
    def __init__(self,root):
        self.root = root
        self.visited = []
        self.best_weight = 0
        self.counter = 0
        self.label = tk.Label(self.root, text=f"Путь:
        {self.visited}")
        self.label2 = tk.Label(self.root, text=f"Общий вес пути:
```

```
        {self.best_weight}")
        self.label3 = tk.Label(self.root, text=f"Количество
        поколений: {self.counter}")
        self.label.pack()
        self.label2.pack()
    #Обновление результатов
    def update_data_res(self,vis,wg,counter):
        self.visited = vis
        self.best_weight = wg
        self.counter = counter
        self.label.config(text=f"Путь: {self.visited}")
        self.label2.config(text=f"Общий вес пути:
        {self.best_weight}")
        if self.counter:
            self.label3.config(text=f"Количество поколений:
            {self.counter}")
            self.label3.pack()
```

Подключение визуализации и основного алгоритма.

```
#Блок подключения необходимых библиотек
import tkinter as tk
from tkinter import ttk, messagebox
import gui
from alg import nearest_neighbor

#Функция закрытия программы
def on_closing():
    if messagebox.askokcancel("Выход", "Вы действительно хотите
выйти?"): root.destroy()

#Создание окна с вкладками и фреймов
root = tk.Tk()
root.geometry("1000x500")
root.protocol("WM_DELETE_WINDOW", on_closing)

notebook = ttk.Notebook(root)
notebook.pack(fill='both', expand=True)

tab1 = tk.Frame(notebook)
tab2 = tk.Frame(notebook)
tab3 = tk.Frame(notebook)
notebook.add(tab1, text='Метод ближайшего соседа')

#Запуск алгоритма ближайшего соседа
neighbor_window = gui.myWindow(tab1,'#cad7e8')
neighbor_table =
gui.myTable(neighbor_window.middle_left_frame,['Вершина 1',
'Вершина 2', 'Вес'])
neighbor_result =
gui.myResult(neighbor_window.bottom_left_frame)
neighbor_canvas =
gui.canvas(neighbor_window.top_right_frame,neighbor_table,neighb
```

```
or_result)
neighbor_startButton =
gui.Button(neighbor_window.top_left_frame,'Начать алгоритм',
nearest_neighbor.nearest_neighbor, neighbor_canvas)
neighbor_clearButton =
gui.Button(neighbor_window.top_left_frame,'Стереть
данные',neighbor_canvas.clear_canvas,True)

root.mainloop()
```

Спецификация программы.

Пользовательский интерфейс (рис. 1)

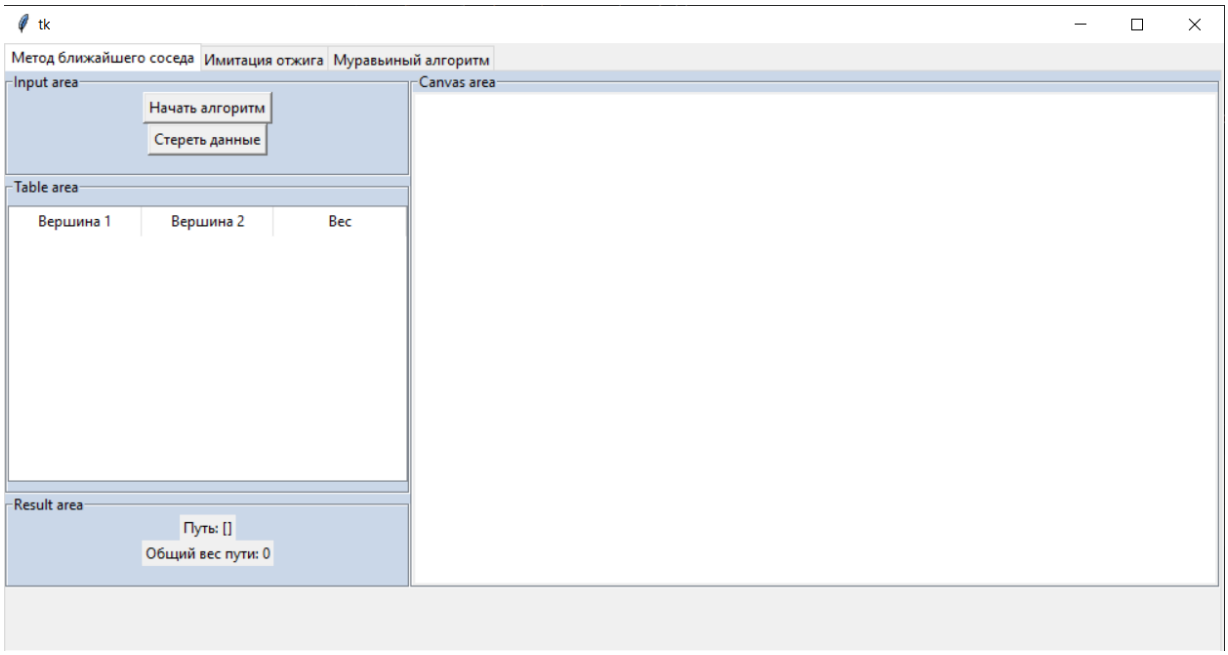

Рис. 1. Пользовательский интерфейс

Для создания графа используется правый фрейм. При нажатии ЛКМ создаётся вершина, которая соединяется со всеми остальными вершинами. Каждому ребру задаётся вес в соответствии с расстоянием до других вершин. Чтобы удалить вершину или ребро нужно нажать на соответствующий элемент ПКМ. Чтобы запустить алгоритм надо нажать кнопку «Начать алгоритм» в левой верхней части окна. Под ней располагается кнопка «Стереть данные», позволяющая очистить область рисования. Также в середине левого фрейма находится таблица со списком рёбер и весами. У каждого ребра можно вручную изменить вес, нажав на него в таблице дважды ЛКМ. В нижней части левого фрейма находится область вывода результата. При нажатии на знак крестика в правом верхнем углу осуществляется выход из программы,

который нужно подтвердить. Если не ввести граф, то алгоритм выдаст ошибку «IndexError: list index out of range».

Контрольный пример.

Тестирование программы производилось на графе, представленном на рис. 2 с случайно заданными весами рёбер.

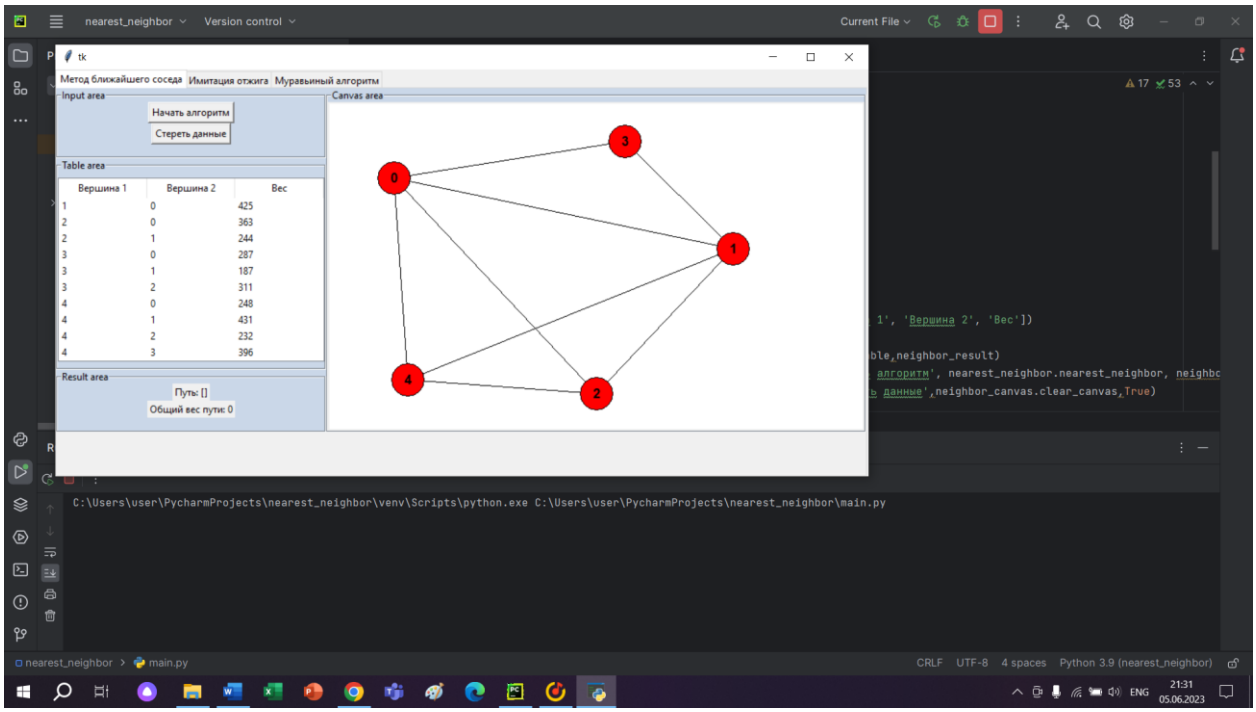

Рис. 2. Начало работы алгоритма

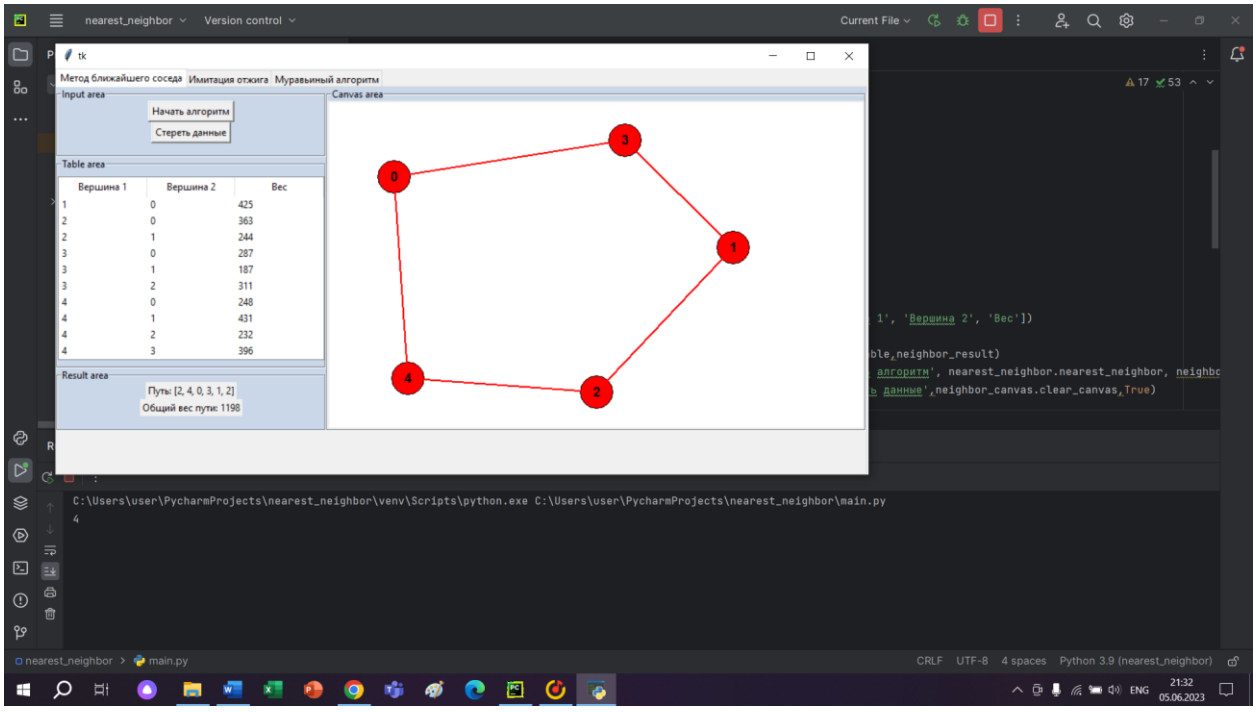

Рис. 3. Окончание работы алгоритма

Таким образом, найден кратчайший гамильтонов цикл, представленный на рис. 3, путь цикла по вершинам: 2, 4, 0, 3, 1, 2.

Заключение.

В ходе работы изучен и реализован метод ближайшего соседа, а также проведён анализ результатов работы алгоритма. Из полученных данных сделаны выводы:

- Метод ближайшего соседа прост в реализации, эффективен для простых графов или графов с близким расположением объектов, показывает хороший результат и быстрое время работы.
- Однако, так как метод ближайшего соседа относится к классу жадных алгоритмов и сильно зависит от выбора начальной вершины, он может выдавать разные решения, не являющиеся оптимальными.

Список литературы.

1. Introducing Python. Modern computing in simple packages / Bill Lobanovic.
2. http://www.machinelearning.ru
3. http://twt.mpei.ac.ru/MCS/Worksheets/nva.xmcd
4. Ссылка на код https://github.com/ksenkap/Nearest_neighbor