**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный университет**

**Факультет прикладной математики-процессов управления**

**Кафедра “фундаментальная информатика и информационные технологии”**

**отчет**

**по лабораторной работе №1**

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

**на тему «Решение задачи о коммивояжере с помощью метода ближайшего соседа»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22Б15-пу |  | Добренкова Л.С. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2024 г**

Оглавление

[1 Цель работы 3](#__RefHeading___Toc9662_524586906)

[2 Задачи 3](#__RefHeading___Toc9664_524586906)

[3 Теоретическая часть 3](#__RefHeading___Toc9666_524586906)

[4 Описание алгоритма 4](#__RefHeading___Toc9668_524586906)

[5 Описание программы 4](#__RefHeading___Toc9670_524586906)

[6 Рекомендации для пользователя 6](#__RefHeading___Toc9672_524586906)

[7 Контрольный пример 7](#__RefHeading___Toc9674_524586906)

[8 Рекомендации для программиста 8](#__RefHeading___Toc9676_524586906)

[9 Контрольный пример 9](#__RefHeading___Toc9678_524586906)

[10 Анализ результатов работы алгоритма 11](#__RefHeading___Toc9680_524586906)

[11 Вывод 13](#__RefHeading___Toc9682_524586906)

# ****Цель работы****

Разработать программное обеспечение, которое с помощью метода ближайшего соседа находит гамильтонов цикл в направленном взвешенном графе.

# ****Задачи****

1. Необходимо формализовать задачу о коммивояжере с помощью алгоритма ближайшего соседа
2. Подготовить контрольный пример, используя взвешенный орграф
3. Найти кратчайший гамильтонов цикл
4. Сравнить алгоритм с модификацией

# ****Теоретическая часть****

Гамильтонов цикл — 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.

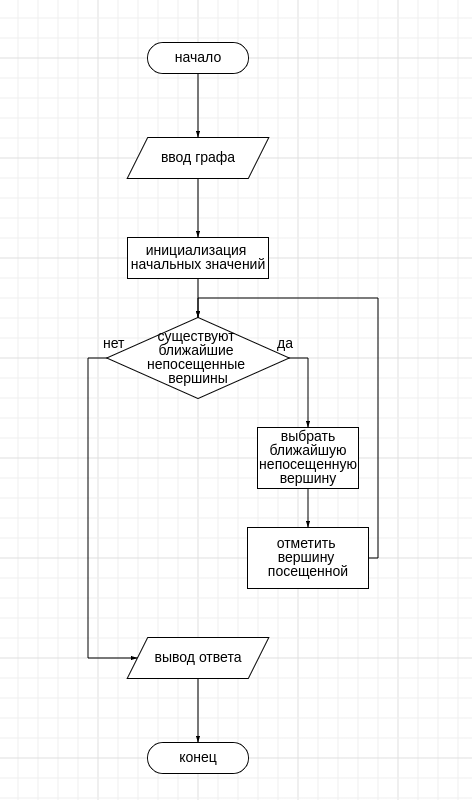
Задача коммивояжера относится к классическим проблемам комбинаторной оптимизации и представляет собой поиск наикратчайшего гамильтонова цикла. Эта задача относится к категории NP-полных задач, что означает отсутствие известных алгоритмов, способных эффективно решать её для всех входных данных в течение полиномиального времени. Тем не менее, для приблизительного или точного решения задачи разработаны разнообразные эвристические и точные методы.

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.

# ****Описание алгоритма****

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

Рисунок 4.1. Основной алгоритм программы

# ****Описание программы****

Программа реализована при помощи языка python 3.8.

Использованные модули: tkinter, math, typing.

В таблицах 5.1, 5.2 представлены описания функций и методов программы.

Таблица 5.1. Описание методов класса App

|  |  |  |
| --- | --- | --- |
| **Метод** | **описание** | **Параметры** |
| add\_vertex | Добавление новой вершины | event |
| draw\_vertex | Добавление вершины на холст | vertex: dict |
| add\_edge | Добавление нового ребра | - |
| draw\_edge | Добавление ребра на холст | edge: dict, isDark: bool |
| draw | Рендеринг | - |
| run | Запуск | - |

Таблица 5.2. Описание методов lib

|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Параметры** |
| build\_distance\_matrix | Инициализация матрицы весов по списку ребер | num\_vertices: int,  edges: List[Tuple[int, int, float]] |
| nearest\_neighbor | Алгоритм ближайшего соседа | dist\_matrix: List[List[float]],  tour: List[Tuple[int, int, float]],  dist: float |
| check\_swap | Проверка перестановок в модификации | dist\_matrix: List[List[float]],  tour: List[Tuple[int, int, float]],  dist: float |

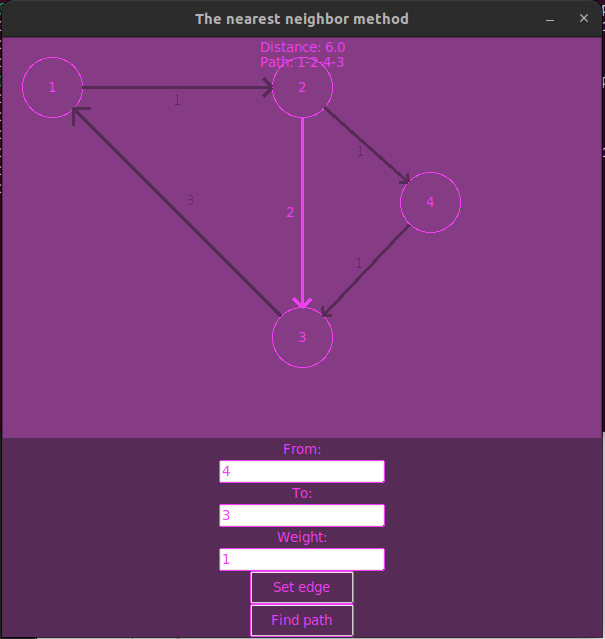
# ****Рекомендации для пользователя****

Для запуска программы необходим интерпретатор python 3.8, пакетный менеджер pip. Необходимо установить библиотеку tkinter.

Системные требования 200мб ОЗУ, 1мб ПЗУ.

Запуск файла main.py выдаст интуитивно понятный графический интерфейс. Для инициализации вершин нужно нажимать ЛКМ на пустое место. Для добавления ребра в граф, необходимо ввести данные о ребре в соответствующие поля, после этого нажать на кнопку «Set edge».

# Контрольный пример

Рисунок 7.1. Пример работы программы

# ****Рекомендации для программиста****

Для внесения изменений необходима ide для python. Код доступен по ссылке: https://github.com/v131v/alg\_labs\_3\_sem/tree/main/nearest\_neighbor

# ****Анализ результатов работы алгоритма****

Метод ближайшего соседа

Преимущества:

1. Простота реализации.
2. Быстрое время выполнения, O(n), где n — количество городов.
3. Подходит для решения больших задач, где требуется быстро получить приемлемое решение.

Недостатки:

1. Решение может быть далеко от оптимального, особенно в случаях, когда ближайший город географически близок, но не является частью кратчайшего пути.
2. Не гарантирует нахождение кратчайшего пути.
3. Чувствителен к выбору начального города.

Метод ближайшего соседа с перестановками

Эта модификация метода ближайшего соседа включает дополнительный этап перестановок, который пытается улучшить найденный маршрут путем перестановки последовательности посещения городов после их первоначального выбора.

Преимущества:

1. Улучшение качества решения по сравнению с обычным методом ближайшего соседа.
2. Может значительно приблизиться к оптимальному решению за счет минимальных изменений маршрута.
3. Сохраняет простоту исходного жадного подхода, добавляя лишь некоторую сложность на этапе оптимизации.

Недостатки:

1. Увеличение времени выполнения из-за необходимости проверки различных перестановок.
2. Не гарантирует нахождение оптимального пути.
3. Эффективность зависит от алгоритма выбора перестановок.

Сравнение

Эффективность: Модификация с перестановками, как правило, дает более короткие маршруты и лучше аппроксимирует оптимальное решение, чем стандартный метод ближайшего соседа, за счет послеоптимизации полученного маршрута.

Скорость выполнения: Стандартный метод ближайшего соседа работает быстрее из-за своей простоты. Метод с перестановками требует дополнительного времени для анализа и выполнения перестановок.

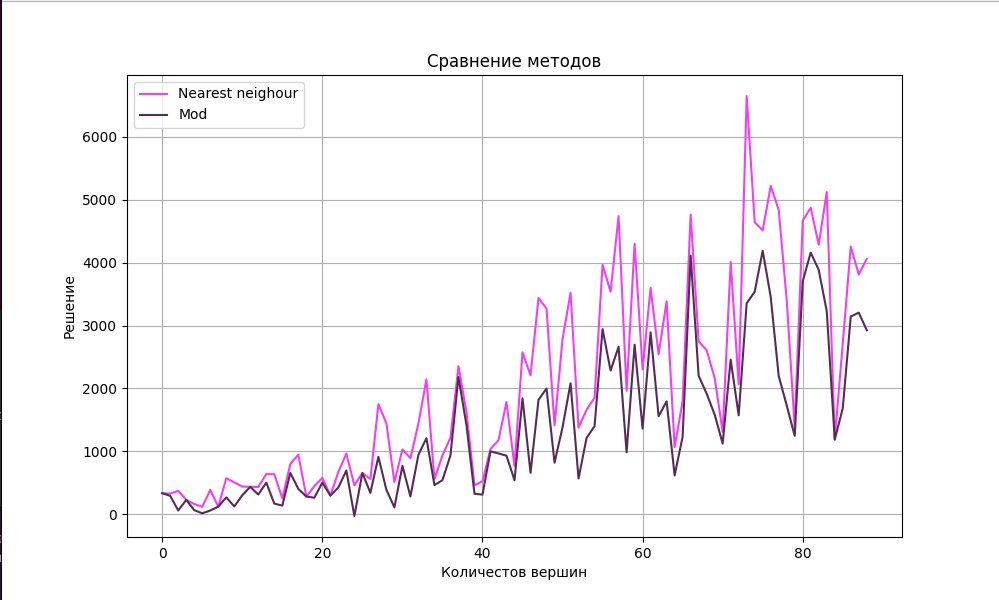
Простота реализации: Метод ближайшего соседа проще в реализации, тогда как метод с перестановками требует дополнительной логики для эффективного выполнения перестановок.

Выбор метода зависит от конкретных требований задачи. Если важна скорость и простота, метод ближайшего соседа может быть предпочтительнее. Однако, если важнее точность решения, стоит рассмотреть метод с перестановками, который может обеспечить более качественное решение за счет улучшения первоначально найденного маршрута.

Ниже представлена сравнительная таблица алгоритмов

Таблица 9.1. Сравнение алгоритмов

|  |  |  |
| --- | --- | --- |
|  | Качество решения, длина пути по медиане | Среднее время работы, секунды |
| Ближайший сосед | 1619 | 0.084 |
| Ближайший сосед с перестановками | 1502 | 0.956 |

Рисунок 9.1. Сравнение решений разных методов

# ****Вывод****

В процессе исследования особенностей алгоритма ближайшего соседа для решения задачи коммивояжера на ориентированном графе был разработан метод поиска гамильтонова цикла. Написана соответствующая программа, способная находить гамильтонов цикл с использованием алгоритма ближайшего соседа и определять длину маршрута, охватывающего все вершины. Кроме того, была предложена модификация, включающая алгоритм перестановок, который может повысить качество решения за счет улучшения изначально найденного маршрута.

# Листинг

Файл main.py

from gui import \*

from lib import \*

from typing import List, Tuple

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

def handler(

n: int, edges: List[Tuple[int, int, float]]

) -> Tuple[list[Tuple[int, int, float], float]]:

matrix = build\_distance\_matrix(n, edges)

tour, dist = nearest\_neighbor(matrix)

print(tour)

best\_tour, best\_dist = check\_swap(matrix, tour, dist)

print(best\_tour)

return best\_tour, best\_dist

App(

"The nearest neighbor method",

handler,

)

Файл gui.py

import tkinter as tk

from tkinter import ttk

import math

from typing import Callable, Tuple

class App:

def \_\_init\_\_(

self,

title: str,

handler: Callable[

[int, list[Tuple[int, int, float]]],

Tuple[list[Tuple[int, int, float], float]],

],

) -> None:

self.\_edge = list(

[

{"from": 1, "to": 2, "weight": 1},

{"from": 2, "to": 3, "weight": 2},

{"from": 3, "to": 1, "weight": 3},

]

) # список ребер

self.\_vertex = list(

[

{"x": 50, "y": 50, "id": 1},

{"x": 300, "y": 50, "id": 2},

{"x": 300, "y": 300, "id": 3},

]

) # список вершин

self.\_selected\_edge = list()

self.\_handler = handler

# интерфейс

light = "#f33ff3"

dark = "#562c56"

self.light = light

self.dark = dark

self.root = tk.Tk()

self.root.title(title)

self.root.geometry("600x600")

self.root.resizable(False, False)

self.root.configure(bg=dark)

styleBtn = ttk.Style()

styleBtn.theme\_use("clam")

styleBtn = styleBtn.configure(

"TButton", foreground=light, background=dark, bordercolor=light

)

styleEntry = ttk.Style()

styleEntry.theme\_use("clam")

styleEntry = styleEntry.configure(

"TEntry", foreground=light, background=dark, bordercolor=light

)

styleLabel = ttk.Style()

styleLabel.theme\_use("clam")

styleLabel = styleLabel.configure("TLabel", foreground=light, background=dark)

self.canvas = tk.Canvas(

self.root, width=600, height=400, bg="#853c85", highlightbackground=dark

)

self.canvas.pack(side=tk.TOP)

self.from\_label = ttk.Label(

self.root, text="From:", foreground=light, background=dark

)

self.from\_label.pack()

self.from\_entry = ttk.Entry(self.root)

self.from\_entry.pack()

self.to\_label = ttk.Label(

self.root, text="To:", foreground=light, background=dark

)

self.to\_label.pack()

self.to\_entry = ttk.Entry(self.root)

self.to\_entry.pack()

self.weight\_label = ttk.Label(

self.root, text="Weight:", foreground=light, background=dark

)

self.weight\_label.pack()

self.weight\_entry = ttk.Entry(self.root)

self.weight\_entry.pack()

self.add\_button = ttk.Button(self.root, text="Set edge", command=self.add\_edge)

self.add\_button.pack()

self.add\_button = ttk.Button(

self.root,

text="Find path",

command=self.run,

)

self.add\_button.pack()

self.canvas.bind("<Button-1>", self.add\_vertex)

self.root.mainloop()

def add\_vertex(

self, event

) -> None: # считывание нажатия и добавления в точку новой вершины

x, y = event.x, event.y # получение координат

self.\_vertex.append({"x": x, "y": y, "id": len(self.\_vertex) + 1})

self.draw()

def add\_edge(self) -> None: # добавление ребра

from\_vertex = int(self.from\_entry.get()) # считывание данных

to\_vertex = int(self.to\_entry.get())

weight = int(self.weight\_entry.get())

if not (

from\_vertex > 0

and from\_vertex <= len(self.\_vertex)

and to\_vertex > 0

and to\_vertex <= len(self.\_vertex)

):

return

for i in range(len(self.\_edge)):

if (

self.\_edge[i]["from"] == from\_vertex

and self.\_edge[i]["to"] == to\_vertex

):

self.\_edge[i], self.\_edge[-1] = self.\_edge[-1], self.\_edge[i]

self.\_edge = self.\_edge[:-1]

break

self.\_edge.append(

{"from": from\_vertex, "to": to\_vertex, "weight": weight}

) # добавление ребра в список

self.draw()

def draw(self):

self.canvas.delete("all")

for v in self.\_vertex:

self.draw\_vertex(v)

for e in self.\_edge:

self.draw\_edge(e)

for e in self.\_selected\_edge:

self.draw\_edge(e, True)

def draw\_vertex(self, vertex: dict) -> None: # рисование вершины в canvas

x, y = vertex["x"], vertex["y"]

self.canvas.create\_oval(x - 30, y - 30, x + 30, y + 30, outline=self.light)

self.canvas.create\_text(x, y, text=str(vertex["id"]), fill=self.light)

def draw\_edge(self, edge: dict, dark: bool = False) -> None: # отображение ребра

color = self.dark if dark else self.light

from\_vertex, to\_vertex, weight = edge["from"], edge["to"], edge["weight"]

from\_x, from\_y = self.get\_vertex\_coordinates(

from\_vertex

) # получение координат исходящей вершины

to\_x, to\_y = self.get\_vertex\_coordinates(

to\_vertex

) # получение координат входящей вершины

# отображение ребра

dx = to\_x - from\_x

dy = to\_y - from\_y

l = (dx\*\*2 + dy\*\*2) \*\* 0.5

ang = math.atan2(dy, dx)

from\_x += math.cos(ang) \* 30

from\_y += math.sin(ang) \* 30

to\_x -= math.cos(ang) \* 30

to\_y -= math.sin(ang) \* 30

l2 = l / 20

x1 = l2 \* math.cos(ang + math.pi + math.pi / 4)

y1 = l2 \* math.sin(ang + math.pi + math.pi / 4)

self.canvas.create\_line(

to\_x, to\_y, to\_x + x1, to\_y + y1, width=3, fill=color, tag="line"

)

x1 = l2 \* math.cos(ang + math.pi - math.pi / 4)

y1 = l2 \* math.sin(ang + math.pi - math.pi / 4)

self.canvas.create\_line(

to\_x, to\_y, to\_x + x1, to\_y + y1, width=3, fill=color, tag="line"

)

self.canvas.create\_line(

from\_x, from\_y, to\_x, to\_y, width=3, fill=color, tag="line"

)

line = self.canvas.create\_line(

from\_x, from\_y, to\_x, to\_y, width=3, fill=color, tag="line"

)

self.canvas.create\_text(

(to\_x + from\_x) / 2 + math.cos(ang + math.pi / 2) \* l2,

(to\_y + from\_y) / 2 + math.sin(ang + math.pi / 2) \* l2,

text=str(weight),

fill=color,

)

self.canvas.tag\_lower(line)

def get\_vertex\_coordinates(self, vertex: int) -> tuple: # получение координат точки

return (self.\_vertex[vertex - 1]["x"], self.\_vertex[vertex - 1]["y"])

def run(self): # запуск алгоритма

selected\_edge, dist = self.\_handler(

len(self.\_vertex),

[(e["from"] - 1, e["to"] - 1, float(e["weight"])) for e in self.\_edge],

)

if dist == float("inf"):

self.canvas.create\_text(300, 10, text=f"Distance: {dist}", fill=self.light)

return

self.\_selected\_edge = [

{

"from": f + 1,

"to": t + 1,

"weight": int(w),

}

for f, t, w in selected\_edge

]

self.draw()

tour = "-".join([str(x["from"]) for x in self.\_selected\_edge])

self.canvas.create\_text(300, 10, text=f"Distance: {dist}", fill=self.light)

self.canvas.create\_text(300, 25, text=f"Path: {tour}", fill=self.light)

Файл lib.py

import copy

from typing import List, Tuple

def build\_distance\_matrix(

num\_vertices: int, edges: List[Tuple[int, int, float]]

) -> List[List[float]]:

# Инициализируем матрицу большими значениями

dist\_matrix = [[float("inf")] \* num\_vertices for \_ in range(num\_vertices)]

for i in range(num\_vertices):

dist\_matrix[i][i] = 0 # Расстояние от вершины к самой себе равно 0

for start, end, weight in edges:

dist\_matrix[start][end] = weight

print(dist\_matrix)

return dist\_matrix

def nearest\_neighbor(

dist\_matrix: List[List[float]],

) -> Tuple[List[Tuple[int, int, float]], float]:

num\_vertices = len(dist\_matrix)

visited = [False] \* num\_vertices

tour = []

total\_distance = 0.0

current\_city = 0

visited[current\_city] = True

for \_ in range(num\_vertices - 1):

nearest\_distance = float("inf")

nearest\_city = None

for i in range(num\_vertices):

if not visited[i] and dist\_matrix[current\_city][i] < nearest\_distance:

nearest\_distance = dist\_matrix[current\_city][i]

nearest\_city = i

if nearest\_city is None:

total\_distance += nearest\_distance

return tour, total\_distance

visited[nearest\_city] = True

tour.append((current\_city, nearest\_city, nearest\_distance))

total\_distance += nearest\_distance

current\_city = nearest\_city

# Завершаем тур возвращением в начальную точку

total\_distance += dist\_matrix[current\_city][0]

tour.append((current\_city, 0, dist\_matrix[current\_city][0]))

return tour, total\_distance

def check\_swap(

dist\_matrix: List[List[float]], tour: List[Tuple[int, int, float]], dist: float

):

best\_tour = tour

best\_dist = dist

for i in range(len(tour) - 2):

for j in range(i + 1, len(tour) - 1):

ap, \_, \_ = tour[i]

a, an, \_ = tour[i + 1]

bp, \_, \_ = tour[j]

b, bn, \_ = tour[j + 1]

if not (

dist\_matrix[ap][b] < float("inf")

and dist\_matrix[b][an] < float("inf")

and dist\_matrix[bp][a] < float("inf")

and dist\_matrix[a][bn] < float("inf")

):

continue

was\_edges = set([tour[i], tour[i + 1], tour[j], tour[j + 1]])

was\_dist = 0

for \_, \_, w in was\_edges:

was\_dist += w

now\_edges = [

(bp, a, dist\_matrix[bp][a]),

(a, bn, dist\_matrix[a][bn]),

(ap, b, dist\_matrix[ap][b]),

(b, an, dist\_matrix[b][an]),

]

now\_dist = 0

for \_, \_, w in set(now\_edges):

now\_dist += w

if dist + now\_dist - was\_dist < best\_dist:

best\_dist = dist + now\_dist - was\_dist

best\_tour = copy.deepcopy(tour)

best\_tour[i] = now\_edges[0]

best\_tour[i + 1] = now\_edges[1]

best\_tour[j] = now\_edges[2]

best\_tour[j + 1] = now\_edges[3]

return best\_tour, best\_dist