**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный университет**

**Факультет прикладной математики-процессов управления**

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

**отчет**

**по лабораторной работе №2**

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

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

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

**Санкт-Петербург**

**2024 г**

Оглавление

[Цель работы 3](#__RefHeading___Toc9662_524586906)

[Задачи 3](#__RefHeading___Toc9664_524586906)

[Теоретическая часть 3](#__RefHeading___Toc9666_524586906)

[Описание алгоритма 4](#__RefHeading___Toc9668_524586906)

[Описание программы 5](#__RefHeading___Toc9670_524586906)

[Рекомендации для пользователя 7](#__RefHeading___Toc9672_524586906)

[Контрольный пример 7](#__RefHeading___Toc8166_812848710)

[Рекомендации для программиста 8](#__RefHeading___Toc9676_524586906)

[Анализ результатов работы алгоритма 9](#__RefHeading___Toc9680_524586906)

[Вывод 11](#__RefHeading___Toc9682_524586906)

[Листинг 11](#__RefHeading___Toc8168_812848710)

# ****Цель работы****

Исследование особенностей решения задачи о коммивояжере с помощью алгоритма имитации отжига.

# ****Задачи****

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

# ****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 имитации отжига – эвристический алгоритм глобальной оптимизации, особенно эффективный при решении дискретных и комбинаторных задач.

Алгоритм основан на реальном процессе – отжиге, способе работы с металлами, при котором задается начальный нагрев и происходит контролируемое остывание. При таком подходе повышается качество металла за счет улучшенной структуры, в которой кристаллическая решетка будет более прочной. При этом нет гарантий, что результат отжига будет исключительно положительным, все зависит от параметров остывания и исходной температуры.

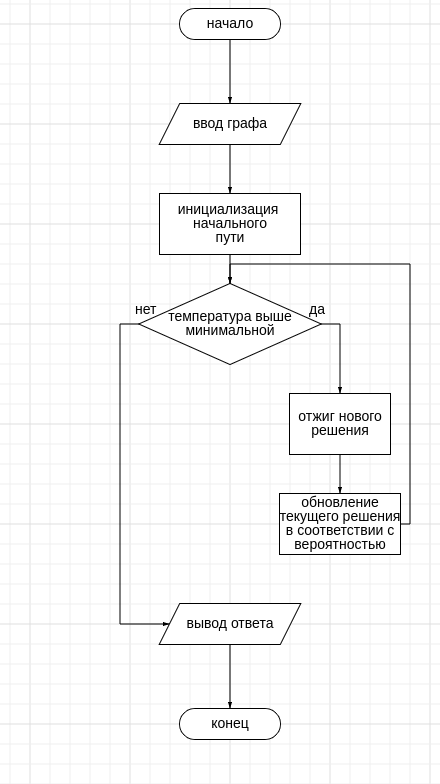
Принципы работы алгоритма:

1. Инициализация: алгоритм начинается с некоторого начального состояния.
2. Случайный поиск: производится случайная модификация пути — перестановка 2 вершин.
3. Функция оценки: определяется функция оценки, которая оценивает качество решения. В нашем случае длина пути.
4. Принятие худшего решения: Алгоритм принимает худшее решение с вероятностью , которая уменьшается по мере "охлаждения".
5. 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. Инициализация графа
2. Поиск любого гамильтонова цикла
3. Случайный «отжиг» нового решения (перестановка)
4. Перейти в новое решение в соответствии с вероятностью
5. Понижение темпертауры
6. Если температура упала до предельного значения, 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, random, 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]] |
| find\_hamiltonian\_cycle | Поиск начального пути | graph: List[List[int]] |
| calculate\_total\_distance | Расчет длины пути | tour,  dist\_matrix |
| simulated\_annealing | Алгоритм имитации отжига | dist\_matrix,  tour: List[int],  initial\_temperature=1000,  cooling\_rate=0.999, |

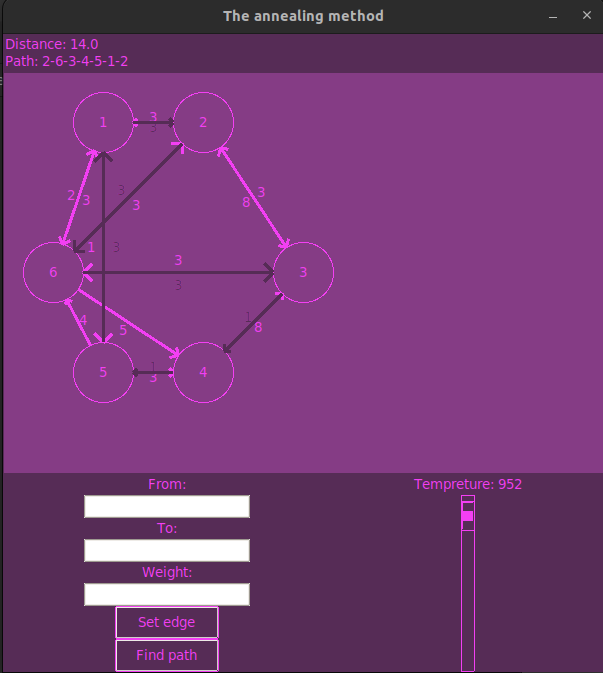
# ****Рекомендации для пользователя****

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/annealing_method>

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

Алгоритм ближайшего соседа

Преимущества:

1. Быстрота: Жадный алгоритм, который быстро находит решение, проходя к ближайшему непосещенному городу.
2. Простота реализации: Легко программировать и понимать.

Недостатки:

1. Качество решения: Не всегда находит кратчайший маршрут и часто зависит от начальной точки.
2. Нет гарантии оптимума: Может застревать в локальных минимумах.

Алгоритм имитации отжига

Преимущества:

1. Избежание локальных минимумов: Вероятностный подход позволяет "выпрыгнуть" из локальных минимумов благодаря принципу имитации процесса отжига.
2. Гибкость: Можно адаптировать к различным типам задач оптимизации.

Недостатки:

1. Сложность настройки: Требует тонкой настройки параметров, таких как температурный режим и скорость охлаждения.
2. Время выполнения: Может требовать значительного времени для нахождения оптимального решения, особенно при больших размерах задач.

Комбинация алгоритма ближайшего соседа и имитации отжига

Комбинирование этих методов может принести лучшие результаты, используя быстроту и простоту ближайшего соседа для получения начального решения и гибкость имитации отжига для его оптимизации.

Преимущества:

1. Улучшение начального решения: Использование ближайшего соседа для получения начального пути, который затем оптимизируется с помощью имитации отжига, может сократить общее время нахождения решения и улучшить его качество.
2. Эффективность: Комбинация жадного подхода и вероятностного метода может эффективно балансировать между исследованием пространства решений и эксплуатацией найденных путей.

Недостатки:

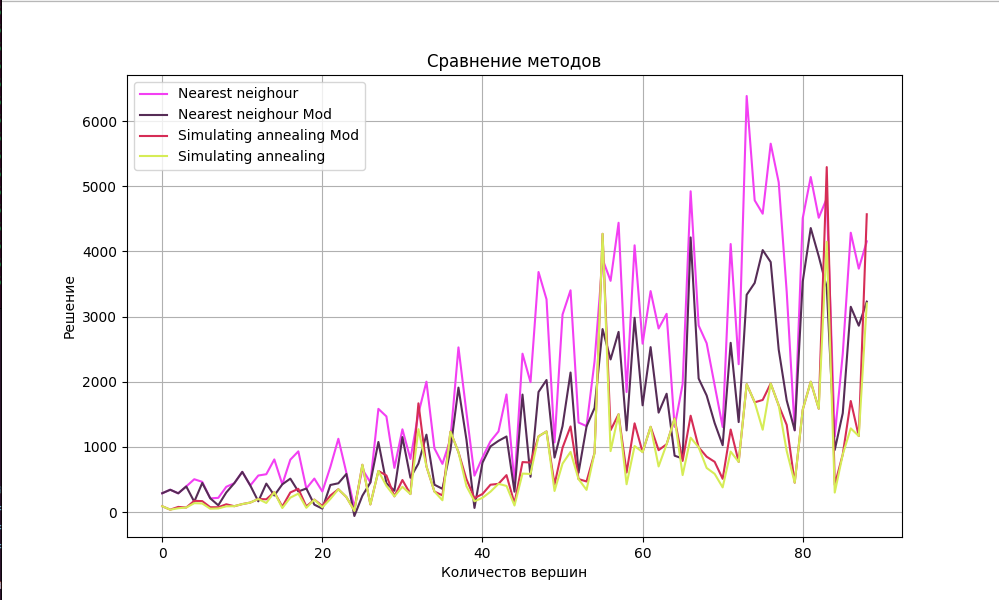
1. Сложность реализации: Требует интеграции двух различных подходов и управления ими.
2. Настройка параметров: Необходимо настроить параметры обоих методов, что может быть непростой задачей.

Выбор метода или их комбинации зависит от специфики задачи, доступных ресурсов и требуемой точности решения. Ближайший сосед отлично подходит для быстрого получения рабочего решения, но оно может быть далеко от оптимального. Имитация отжига предлагает более мощный, но более ресурсоемкий подход к поиску глобального оптимума. Их комбинация может обеспечить баланс между скоростью и качеством решения, делая его особенно полезным для больших и сложных задач.

Ниже представления сравнительная таблица.

Таблица 9.1. Сравнение алгоритмов

|  |  |  |
| --- | --- | --- |
|  | Качество решения, длина пути по медиане | Среднее время работы, секунды |
| Алгоритм ближайшего соседа | 1619 | 0.084 |
| Алгоритм ближайшего соседа с перестановками | 1502 | 0.96 |
| Метод отжига | 840 | 2.31 |
| Метод отжига с ближайшим соседом | 983 | 2.104 |

Рисунок 9.1. Сравнение решений разных методов

# ****Вывод****

В рамках данной работы были исследованы характеристики алгоритма отжига для решения задачи коммивояжера на ориентированном графе. Создан метод поиска гамильтонова цикла, используя алгоритм отжига. Реализована программа, которая с помощью DFS обнаруживает гамильтонов цикл и оптимизирует его длину, уменьшая общее расстояние. Также была разработана модификация с использованием алгоритма ближайшего соседа, которая значительно улучшает результаты на полносвязных графах.

# 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]], tempr: float

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

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

tour = find\_hamiltonian\_cycle(matrix)

print(tour)

if tour is not None:

tour, dist = simulated\_annealing(

matrix, tour, initial\_temperature=tempr + 0.1

)

print(tour)

return tour, dist

App(

"The annealing 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]], 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},

{"from": 2, "to": 4, "weight": 1},

{"from": 4, "to": 3, "weight": 1},

]

) # список ребер

self.\_vertex = list(

[

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

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

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

{"x": 400, "y": 200, "id": 4},

]

) # список вершин

self.\_selected\_edge = list()

self.\_edge = list(

[

{"from": 1, "to": 2, "weight": 3},

{"from": 1, "to": 5, "weight": 1},

{"from": 1, "to": 6, "weight": 2},

{"from": 2, "to": 1, "weight": 3},

{"from": 2, "to": 6, "weight": 3},

{"from": 2, "to": 3, "weight": 8},

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

{"from": 3, "to": 6, "weight": 3},

{"from": 3, "to": 4, "weight": 1},

{"from": 4, "to": 3, "weight": 8},

{"from": 4, "to": 5, "weight": 1},

{"from": 5, "to": 1, "weight": 3},

{"from": 5, "to": 4, "weight": 3},

{"from": 5, "to": 6, "weight": 4},

{"from": 6, "to": 1, "weight": 3},

{"from": 6, "to": 2, "weight": 3},

{"from": 6, "to": 3, "weight": 3},

{"from": 6, "to": 4, "weight": 5},

]

)

self.\_vertex = list(

[

{"x": 100, "y": 50, "id": 1},

{"x": 200, "y": 50, "id": 2},

{"x": 300, "y": 200, "id": 3},

{"x": 200, "y": 300, "id": 4},

{"x": 100, "y": 300, "id": 5},

{"x": 50, "y": 200, "id": 6},

]

)

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("default")

styleEntry = styleEntry.configure(

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

)

styleScale = ttk.Style()

styleScale.theme\_use("clam")

styleScale = styleScale.configure(

"Vertical.TScale",

foreground=light,

background=dark,

bordercolor=light,

troughcolor=dark,

darkcolor=dark,

lightcolor=light,

highlightbackground=light,

)

self.res\_label = ttk.Label(

self.root, text="There will be result...", foreground=light, background=dark

)

self.res\_label.pack(fill="x", expand=True)

self.canvas = tk.Canvas(

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

)

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

self.frame = tk.Frame(self.root, bg=dark)

self.frame.pack(side="left", fill="both", expand=True)

self.from\_label = ttk.Label(

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

)

self.from\_label.pack()

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

self.from\_entry.pack()

self.to\_label = ttk.Label(

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

)

self.to\_label.pack()

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

self.to\_entry.pack()

self.weight\_label = ttk.Label(

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

)

self.weight\_label.pack()

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

self.weight\_entry.pack()

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

self.add\_button.pack()

self.add\_button = ttk.Button(

self.frame,

text="Find path",

command=self.start\_algoritm,

)

self.add\_button.pack()

self.frame2 = tk.Frame(self.root, bg=dark)

self.frame2.pack(side="left", fill="both", expand=True)

self.tempr\_label = ttk.Label(

self.frame2, text="Tempreture: 0", foreground=light, background=dark

)

self.tempr\_label.pack()

self.scale1 = ttk.Scale(

self.frame2,

from\_=1000,

to=0,

orient="vertical",

command=self.upd\_tempreture,

)

self.scale1.pack(fill="y", expand=True)

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

self.root.mainloop()

def upd\_tempreture(self, val):

self.tempr\_label.configure(text=f"Tempreture: {int(float(val))}")

self.draw()

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 start\_algoritm(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],

float(self.scale1.get()),

)

if dist == float("inf"):

self.res\_label.configure(text=f"Distance: {dist}")

return

self.\_selected\_edge = [

{

"from": f + 1,

"to": t + 1,

"weight": int(w),

}

for f, t, w in selected\_edge

]

# self.canvas.delete("all")

self.draw()

tour = (

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

+ "-"

+ str(self.\_selected\_edge[-1]["to"])

)

self.res\_label.configure(text=f"Distance: {dist}\nPath: {tour}")

Файл lib.py

import copy

from typing import List, Tuple, Optional

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 find\_hamiltonian\_cycle(graph: List[List[int]]) -> Optional[List[int]]:

n = len(graph)

path: List[int] = []

def dfs(vertex: int, path: List[int]) -> Optional[List[int]]:

path.append(vertex)

if len(path) == n:

if (

graph[path[-1]][path[0]] != 0

): # Проверяем, можем ли вернуться в начальную вершину

path.append(path[0]) # Замыкаем цикл

return path

else:

path.pop()

return None

for next\_vertex in range(n):

if graph[vertex][next\_vertex] != 0 and next\_vertex not in path:

result = dfs(next\_vertex, path)

if result:

return result

path.pop() # Backtracking

return None

for start\_vertex in range(n): # Можем начать с любой вершины

result = dfs(start\_vertex, [])

if result:

return result

return None

import random

import math

def calculate\_total\_distance(tour, dist\_matrix):

total\_distance = 0

num\_cities = len(tour)

for i in range(num\_cities):

total\_distance += dist\_matrix[tour[i]][tour[(i + 1) % num\_cities]]

return total\_distance

def simulated\_annealing(

dist\_matrix,

tour: List[int],

initial\_temperature=1000,

cooling\_rate=0.999,

):

num\_cities = len(dist\_matrix)

current\_distance = calculate\_total\_distance(tour, dist\_matrix)

current\_tour = tour[:-1]

best\_tour = current\_tour[:]

best\_distance = current\_distance

temperature = initial\_temperature

while temperature > 0.1:

new\_tour = current\_tour[:]

# Производим мутацию текущего маршрута (поменяем местами два города)

city1, city2 = random.sample(range(num\_cities), 2)

new\_tour[city1], new\_tour[city2] = new\_tour[city2], new\_tour[city1]

new\_distance = calculate\_total\_distance(new\_tour, dist\_matrix)

delta\_distance = new\_distance - current\_distance

if new\_distance < float("inf"):

print(temperature, " dist: ", new\_distance, " tour: ", new\_tour)

if delta\_distance < 0 or random.random() <= math.exp(

-delta\_distance / temperature

):

current\_tour = new\_tour

current\_distance = new\_distance

if current\_distance < best\_distance:

best\_tour = current\_tour[:]

best\_distance = current\_distance

# Уменьшаем температуру

temperature \*= cooling\_rate

ans\_tour = []

for i in range(len(best\_tour) - 1):

ans\_tour.append(

(

best\_tour[i],

best\_tour[i + 1],

dist\_matrix[best\_tour[i]][best\_tour[i + 1]],

)

)

ans\_tour.append(

(best\_tour[-1], best\_tour[0], dist\_matrix[best\_tour[-1]][best\_tour[0]])

)

return ans\_tour, calculate\_total\_distance(best\_tour, dist\_matrix)