**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №2**

**по дисциплине «Построение и анализ алгоритмов»**

**Тема: Кратчайшие пути в графе: коммивояжёр**

**Вариант: 2**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Лобова Е. И. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Целью работы является изучение задачи коммивояжёра, точных и приближенных методов её решения и реализация алгоритма Литтла с модификацией и алгоритма ближайшего соседа.

## Задание

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны заклятием процветания. Для этого ему необходимо посетить каждый город ровно один раз, создавая тропу благополучия, и вернуться обратно в столицу, используя минимум своих чародейских сил. Вашей задачей является помощь в прокладывании маршрута с помощью древнего и могущественного алгоритма ветвей и границ.

Карта дорог Алгоритмии перед Гамильтоном представляет собой полный граф, где каждый город соединён магическими порталами с каждым другим. Стоимость использования портала из города в город занимает определённое количество маны, и Гамильтон стремится минимизировать общее потребление магической энергии для закрепления проклятия.

Входные данные:

Первая строка содержит одно целое число N (N — количество городов). Города нумеруются последовательными числами от 0 до N - 1.

Следующие N строк содержат по N чисел каждая, разделённых пробелами, формируя таким образом матрицу стоимостей M. Каждый элемент M\_i,j этой матрицы представляет собой затраты маны на перемещение из города i в город j.

Выходные данные:

Первая строка: Список из N целых чисел, разделённых пробелами, обозначающих оптимальный порядок городов в магическом маршруте Гамильтона. В начале идёт город 0, с которого начинается маршрут, затем последующие города до тех пор, пока все они не будут посещены.

Вторая строка: Число, указывающее на суммарное количество израсходованной маны для завершения пути.

Sample Input 1:

3

-1 1 3

3 -1 1

1 2 -1

Sample Output 1:

0 1 2

3.0

Sample Input 2:

4

-1 3 4 1

1 -1 3 4

9 2 -1 4

8 9 2 -1

Sample Output 2:

0 3 2 1

6.0

**Вариант 2:**

2 МВиГ: Алгоритм Литтла с модификацией: после приведения матрицы, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. Приближённый алгоритм: АБС. Замечание к варианту 2 Начинать АБС со стартовой вершины.

## Описание алгоритмов

Для решения задания лабораторной работы использовался точный метод: **алгоритм Литтла с модификацией**.

Алгоритм Литтла работает так: он начинает с матрицы стоимостей путей между городами, выполняет её редукцию для получения нижней границы стоимости решения. Затем выбирается ребро (путь между городами) с максимальным штрафом за неиспользование. Создаются две ветви: одна включает это ребро в маршрут, другая - исключает. Для каждой ветви матрица стоимостей корректируется (запрещаются петли, исключаются ребра), редуцируется, и вычисляется нижняя граница стоимости решения. Ветви с наименьшей нижней границей продолжают исследоваться, пока не будет найден полный маршрут, который становится текущим лучшим решением. Другие ветви отсекаются, если их нижняя граница превышает стоимость текущего лучшего решения.

В качестве узла дерева решений используется класс *Node* с полями *matrix* - редуцированная матрица для текущего решения, *bound* - нижняя оценка длины решения, *rout*e - частичные маршруты, то есть ребра, включенные в решение.

Для редукции матрицы используется функция *reduce(matrix)* - функция, принимающая на вход матрицу стоимостей и выполняющая редукцию матрицы по строкам и столбцам, вычитая минимальный элемент из каждой строки и столбца, если минимальный элемент больше нуля. Функция подсчитывает и возвращает общую сумму вычтенных минимумов, а редукция уменьшает значения матрицы, не меняя решения задачи, при этом помогает находить более оптимальное решение.

Для вычисления максимального штрафа за отказ от ребра используется метод *get\_cell\_with\_max\_penalty(self)*, который находит ячейку с нулевым значением в матрице, которая имеет максимальный "штраф". Штраф для ячейки вычисляется как сумма минимального элемента в строке (исключая элемент в столбце) и минимального элемента в столбце (исключая элемент в строке). Функция возвращает кортеж с индексами ячейки и ее штрафом. Эта функция нужна для выбора ветвления в алгоритме ветвей и границ: ячейка с максимальным штрафом указывает на наиболее перспективное направление для дальнейшего поиска решения.

Кроме того, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. МОД строится от стянутых вершин, то есть если в текущем решении есть путь соединяющий две и более вершин, они стягиваются в одну. Для этого используются функции *prepare\_matrix\_for\_mst(matrix, route), merge\_vertices(matrix, i, j, old\_transitions), minimum\_spanning\_tree(matrix, transitions\_v)*.

Функция *prepare\_matrix\_for\_mst(matrix, route)* принимает исходную матрицу стоимостей и список включенных в маршрут рёбер, создаёт копию матрицы стоимостей, "стягивает" вершины, соединённые рёбрами из списка, в одну вершину, тем самым подготавливая матрицу для вычисления минимального остовного дерева, и возвращает новую матрицу стоимостей и обновлённый список переходов между вершинами.

Функция *merge\_vertices(matrix, i, j, old\_transitions)* принимает матрицу стоимостей, индексы двух вершин, которые нужно объединить, и список переходов, создаёт новую матрицу меньшего размера, вычисляет стоимости между оставшимися вершинами и объединённой вершиной, обновляет список переходов, чтобы отразить объединение вершин, и возвращает новую матрицу и обновлённый список переходов.

Функция *minimum\_spanning\_tree(matrix, transitions\_v)* вычисляет вес минимального остовного дерева для графа, представленного матрицей. Функция использует алгоритм Прима с приоритетной очередью для построения МОД. Она начинает с произвольной вершины, добавляет её в посещённые, и затем на каждой итерации выбирает ребро наименьшего веса, соединяющее посещённую вершину с непосещённой, добавляя новую вершину в посещённые до тех пор, пока все вершины не будут посещены.

В функции *make\_children(node)* осуществляется создание потомков для текущей вершины, то есть поддерева включающего ребро и не включающего.

В функции *little\_algorithm(matrix)* выполняется последовательное извлечение узлов с наименьшей нижней границей, пока очередь приоритетов не пуста. Если узел представляет почти полный маршрут, вычисляет его полную стоимость и обновляет рекорд, если найден маршрут лучше текущего.

**Оценка сложности точного метода:**

*По времени*: O(Cn), где С = 1.26, полученное экспериментальным путем. Экспоненциальная сложность объясняется тем, что метод ветвей и границ хоть и позволяет отсекать некоторые неперспективные решения, но не избавляет от перебора.

*По памяти*: O(2n\*N2), так как в худшем случае строится полное бинарное дерево, содержащее 2n-2 вершин, так как если не хватает в вершине 2х путей для завершения маршрута, они не строятся, а сразу вычисляются. Для каждой вершины нужно хранить матрицу N\*N.

Для решения задания лабораторной работы использовался приближенный метод: **алгоритм ближайшего соседа**.

Функция *nearest\_neighbor\_algorithm(start, distance\_matrix)* реализует алгоритм ближайшего соседа для задачи коммивояжёра. Она начинает с указанного города (*start*), затем на каждой итерации выбирает ближайший непосещённый город и добавляет его в маршрут. Алгоритм продолжается, пока не будут посещены все города. В конце маршрута происходит возврат в начальный город. Функция возвращает общую стоимость построенного маршрута и сам маршрут.

**Оценка сложности приближенного метода:**

*По времени*: O(N2), так как посещаем все города(N) и для каждого города ищем ближайший к нему, смежных городов N-1.

*По памяти*: O(N), так как храним список не посещенных городов (изначально N-1) и храним путь, который состоит из N+1 вершин.

## Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | ∞ 16 2  16 ∞ 13  7 11 ∞ | Длина пути: 29.0  Путь: 0 2 1 | Литтл: работает корректно. |
|  | 0 ∞ 16 2  16 ∞ 13  7 11 ∞ | Длина пути: 29  Путь: 0-2-1-0 | АБС: работает корректно. Выдал решение совпадающее с точным. |
|  | ∞ 12 2  12 ∞ 17  2 17 ∞ | Длина пути: 31.0  Путь: 0 1 2 | Литтл: работает корректно для симметричной матрицы. |
|  | 1  ∞ 12 2  12 ∞ 17  2 17 ∞ | Длина пути: 31  Путь: 1-0-2-1 | АБС: работает корректно, начиная обработку с ненулевой вершины. |
|  | ∞ 1 1  1 ∞ 1  1 1 ∞ | Длина пути: 3.0  Путь: 0 1 2 | Литтл: работает корректно, когда расстояние между городами одинаковое |
|  | ∞ 17 20 1 13  7 ∞ 9 5 19  4 16 ∞ 13 6  16 17 18 ∞ 8  7 19 10 16 ∞ | Длина пути: 40.0  Путь: 0 3 1 2 4 | Литтл: работает корректно. |
| 7 | ∞ 17 20 1 13  7 ∞ 9 5 19  4 16 ∞ 13 6  16 17 18 ∞ 8  7 19 10 16 ∞ | Длина пути: 54  Путь: 3-4-0-1-2-3 | АБС: работает корректно, выдал не точное, а приближенное решение. |

## 

## Выводы

В ходе работы была успешно изучена задача коммивояжера и алгоритмы для ее точного и приближенного решения. На языке Python были реализованы алгоритмы Литтла с модификацией и ближайшего соседа.

# Приложение А Исходный код программы

Название файла: main.py

from nearest\_neighbor\_algorithm import \*

from little\_algorithm import \*

from matrix import \*

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

filename = "matrix.txt"

print("Загрузить матрицу весов из файла? y/n")

answer = input()

flag\_generate = True

if answer == "y":

try:

dist\_matrix = read\_matrix\_from\_file(filename)

print("Успешно загружена матрица:")

print\_matrix(dist\_matrix)

flag\_generate = False

except (ValueError, FileNotFoundError) as e:

print("Не удалось загрузить, сгенерируем новую")

if flag\_generate:

print("Введите размерность матрицы: ")

n = int(input())

print("Генерировать симметричную матрицу? y/n")

answer = input()

is\_sym = True if answer == "y" else False

print("Сгенерированная матрица:")

dist\_matrix = generate\_weight\_matrix(n, is\_sym, 20)

print\_matrix(dist\_matrix)

print(f"Сохранить матрицу в файл {filename}? y/n")

answer = input()

if answer == "y":

write\_matrix\_to\_file(dist\_matrix, filename)

print("Матрица сохранена!")

start = 0

print("\033[3m\033[36m{}\033[0m".format("\nПРИБЛИЖЕННЫЙ АЛГОРИТМ: АЛГОРИТМ БЛИЖАЙШЕГО СОСЕДА"))

print(f"Какой город принять в качестве стартового? 0 - {len(dist\_matrix) - 1}")

try:

answer = int(input())

if 0 <= answer < len(dist\_matrix):

start = answer

else:

raise ValueError

except ValueError:

print("Число не соответствует номеру города. Начнем поиск с 0го города")

cost, way = nearest\_neighbor\_algorithm(start, dist\_matrix)

print(f"Длина пути: {cost}")

print(f"Путь: {way[0]}", end="")

for i in range(1, len(way)):

print('-', way[i], end='', sep='')

print("\033[3m\033[36m{}\033[0m".format("\n\nТОЧНЫЙ АЛГОРИТМ: АЛГОРИТМА ЛИТТЛА С МОДИФИКАЦИЕЙ"))

result = little\_algorithm(dist\_matrix)

if result:

n = len(dist\_matrix)

ordered\_route = format\_route(result['route'], n)

print(f"Длина пути: {result['length']}")

route = " ".join(map(str, ordered\_route))

print(f"Путь: {route}")

Название файла: matrix.py

import math

import random

def generate\_weight\_matrix(size: int, symmetric: bool = False, max\_weight: int = 100):

matrix = [[0] \* size for \_ in range(size)]

for i in range(size):

for j in range(size):

if i == j:

matrix[i][j] = math.inf

elif symmetric and j > i:

weight = random.randint(1, max\_weight)

matrix[i][j] = weight

matrix[j][i] = weight

elif not symmetric and j != i:

matrix[i][j] = random.randint(1, max\_weight)

return matrix

def print\_matrix(matrix):

for row in matrix:

print(" ".join(str(x).ljust(5) if x != math.inf else "∞ " for x in row))

def write\_matrix\_to\_file(matrix, filename):

copy\_matrix = [row.copy() for row in matrix]

for i in range(len(copy\_matrix)):

copy\_matrix[i][i] = -1

with open(filename, 'w') as f:

for row in copy\_matrix:

f.write(' '.join(map(str, row)) + '\n')

def read\_matrix\_from\_file(filename):

matrix = []

with open(filename, 'r') as f:

lines = [line.strip() for line in f if line.strip()]

if not lines:

raise ValueError("Файл пустой")

for line in lines:

if line.strip():

matrix.append(list(map(float, line.strip().split())))

for i in range(len(matrix)):

matrix[i][i] = math.inf

return matrix

Название файла:little\_algorithm.py

import math

from heapq import heappush, heappop

from matrix import \*

class Node:

def \_\_init\_\_(self, matrix, bound, route):

self.matrix = matrix # Матрица стоимостей

self.bound = bound # Нижняя граница стоимости

self.route = route # Частичный маршрут (список кортежей (i, j))

def \_\_lt\_\_(self, other):

return self.bound < other.bound

def \_\_str\_\_(self):

return f"Узел с частичным маршрутом {self.route} и нижней границей стоимости {self.bound}"

@staticmethod

def clone\_matrix(matrix):

return [row.copy() for row in matrix]

@staticmethod

def reduce(matrix):

print("РЕДУКЦИЯ:")

print("Матрица до редукции:")

print\_matrix(matrix)

n = len(matrix)

total\_reduction = 0

print("Выполнение редукции по строкам...")

for i in range(n):

try:

min\_val = min(x for x in matrix[i] if not math.isinf(x))

except ValueError:

min\_val = 0

print(f"Минимальное значение в строке {i} = {min\_val}")

if min\_val > 0:

for j in range(n):

if not math.isinf(matrix[i][j]):

matrix[i][j] -= min\_val

total\_reduction += min\_val

print("Выполнение редукции по столбцам...")

for j in range(n):

try:

min\_val = min(matrix[i][j] for i in range(n) if not math.isinf(matrix[i][j]))

except ValueError:

min\_val = 0

print(f"Минимальное значение в столбце {j} = {min\_val}")

if min\_val > 0:

for i in range(n):

if not math.isinf(matrix[i][j]):

matrix[i][j] -= min\_val

total\_reduction += min\_val

print("Матрица после редукции:")

print\_matrix(matrix)

print(f"Сумма минимумов = {total\_reduction}")

return total\_reduction

def get\_cell\_with\_max\_penalty(self):

max\_penalty = -1

best\_cell = None

print("Нахождение ячейки с максимальным штрафом")

for i in range(len(self.matrix)):

for j in range(len(self.matrix)):

if self.matrix[i][j] == 0:

# Вычисляем штраф как сумму минимальных элементов в строке и столбце (исключая текущий 0)

try:

row\_min = min(x for k, x in enumerate(self.matrix[i]) if k != j and not math.isinf(x))

except ValueError:

row\_min = 0

try:

col\_min = min(self.matrix[k][j] for k in range(len(self.matrix)) if

k != i and not math.isinf(self.matrix[k][j]))

except ValueError:

col\_min = 0

penalty = row\_min + col\_min

print(

f"Для ячейки [{i},{j}] минимальный элемент в строке {row\_min}, минимальный элемент в столбце {col\_min}, тогда штраф = {penalty}")

if penalty > max\_penalty:

max\_penalty = penalty

best\_cell = (i, j, penalty)

print(f"Итог: максимальный штраф = {max\_penalty} у ячейки [{best\_cell[0]},{best\_cell[1]}]")

return best\_cell

def find\_next\_start\_city(edges, start\_city):

for i, edge in enumerate(edges):

if edge[1] == start\_city:

return i

return -1

def find\_next\_end\_city(edges, end\_city):

for i, edge in enumerate(edges):

if edge[0] == end\_city:

return i

return -1

def get\_close\_edges(route):

result = []

edges = route.copy()

while edges:

length = 1

start\_city = edges[0][0]

end\_city = edges[0][1]

edges.pop(0)

index = find\_next\_start\_city(edges, start\_city)

while index != -1:

length += 1

start\_city = edges[index][0]

edges.pop(index)

index = find\_next\_start\_city(edges, start\_city)

index = find\_next\_end\_city(edges, end\_city)

while index != -1:

length += 1

end\_city = edges[index][1]

edges.pop(index)

index = find\_next\_end\_city(edges, end\_city)

if length >= 2:

result.append((end\_city, start\_city))

return result

def prepare\_matrix\_for\_mst(matrix, route):

current\_matrix = [row.copy() for row in matrix]

current\_transitions = [int(x) for x in range(len(matrix))]

for path in route:

i, j = path[0], path[1]

print(f"Стягиваем ребро {i} -> {j} в одну вершину...")

i = current\_transitions[i]

j = current\_transitions[j]

current\_matrix, current\_transitions = merge\_vertices(current\_matrix, i, j, current\_transitions)

return current\_matrix, current\_transitions

def merge\_vertices(matrix, i, j, old\_transitions):

n = len(matrix)

if i == j or i >= n or j >= n:

return matrix

new\_size = n - 1

new\_matrix = [[math.inf] \* new\_size for \_ in range(new\_size)]

merged\_idx = min(i, j)

for new\_row in range(new\_size):

for new\_column in range(new\_size):

if (new\_row < merged\_idx or merged\_idx < new\_row < max(i,j)) and (new\_column < merged\_idx or merged\_idx < new\_column < max(i,j)):

new\_matrix[new\_row][new\_column] = matrix[new\_row][new\_column]

elif (new\_row < merged\_idx or merged\_idx < new\_row < max(i,j)) and new\_column == merged\_idx:

new\_matrix[new\_row][new\_column] = matrix[new\_row][i]

elif (new\_row < merged\_idx or merged\_idx < new\_row < max(i,j)) and new\_column >= max(i,j):

new\_matrix[new\_row][new\_column] = matrix[new\_row][new\_column + 1]

elif new\_row == merged\_idx and (new\_column < merged\_idx or merged\_idx < new\_column < max(i,j)):

new\_matrix[new\_row][new\_column] = matrix[j][new\_column]

elif new\_row == merged\_idx and new\_column == merged\_idx:

new\_matrix[new\_row][new\_column] = math.inf

elif new\_row == merged\_idx and new\_column >= max(i,j):

new\_matrix[new\_row][new\_column] = matrix[j][new\_column + 1]

elif new\_row >= max(i,j) and (new\_column < merged\_idx or merged\_idx < new\_column < max(i,j)):

new\_matrix[new\_row][new\_column] = matrix[new\_row + 1][new\_column]

elif new\_row >= max(i, j) and new\_column == merged\_idx:

new\_matrix[new\_row][new\_column] = matrix[new\_row + 1][i]

elif new\_row >= max(i, j) and new\_column >= max(i,j):

new\_matrix[new\_row][new\_column] = matrix[new\_row + 1][new\_column + 1]

new\_transitions = old\_transitions.copy()

for k in range(len(new\_transitions)):

if new\_transitions[k] == max(i,j):

new\_transitions[k] = merged\_idx

if new\_transitions[k] > max(i,j):

new\_transitions[k] -= 1

return new\_matrix, new\_transitions

def process\_transitions(transitions):

num = set(transitions)

result = [""]\*len(num)

for i in range(len(transitions)):

result[transitions[i]] += str(i)

return result

def minimum\_spanning\_tree(matrix, transitions\_v):

vertices = set(int(x) for x in range(len(matrix)))

if len(vertices) <= 1:

return 0.0

total\_cost = 0.0

visited = set()

start = next(iter(vertices))

print("Построение МОД:")

priority\_queue = [(0.0, start)]

while priority\_queue and len(visited) < len(vertices):

weight, u = heappop(priority\_queue)

print(f"Извлекаем из очереди вершину {transitions\_v[u]} с весом {weight}")

if u in visited:

print("Вершина уже отмечена как посещенная, переходим к следующей...")

continue

total\_cost += weight

print(f"Итовую строимость увеличиваем на {weight} и получаем стоимость = {total\_cost}")

visited.add(u)

for v in vertices - visited:

edge\_weight = matrix[u][v]

if edge\_weight != math.inf:

print(f"Добавляем смежную вершину {transitions\_v[v]} с весом перехода {edge\_weight} в очередь")

heappush(priority\_queue, (edge\_weight, v))

print(f"Итоговый вес МОД: {total\_cost}")

return total\_cost if len(visited) == len(vertices) else math.inf

def make\_children(min\_node):

row, column, left\_penalty = min\_node.get\_cell\_with\_max\_penalty()

print("\nСОЗДАНИЕ ЛЕВОГО ПОТОМКА...")

print(f"В левом потомке исключаем дугу {row} -> {column}")

left\_matrix = [row.copy() for row in min\_node.matrix]

left\_matrix[row][column] = math.inf

Node.reduce(left\_matrix)

left\_bound = min\_node.bound + left\_penalty

left\_route = min\_node.route.copy()

left\_child = Node(left\_matrix, left\_bound, left\_route)

print(f"ИТОГ: Для маршрута {left\_route}, не проходящего через дугу {row} -> {column} нижняя оценка длины маршрута = {left\_bound}")

print("\nСОЗДАНИЕ ПРАВОГО ПОТОМКА...")

print(f"В маршрут правого потомка включаем дугу {row} -> {column}")

right\_matrix = [row.copy() for row in min\_node.matrix]

print(f"Запрещаем обратную дугу {column} -> {row}")

right\_matrix[column][row] = math.inf

print(f"Запрещаем выезжать из города {row} и въезжать в город {column}, то есть дуги ", end="")

for i in range(len(right\_matrix)):

right\_matrix[row][i] = math.inf

right\_matrix[i][column] = math.inf

print(f"{row} -> {i}, {i} -> {column}", end="")

if i != len(right\_matrix):

print(", ", end="")

right\_route = min\_node.route.copy()

right\_route.append((row, column))

close\_edges = get\_close\_edges(right\_route)

print(f"\nЗапрещаем дуги, которые могут создать подциклы, то есть дуги ", end="")

for curr\_row, curr\_edge in close\_edges:

right\_matrix[curr\_row][curr\_edge] = math.inf

print(f"{curr\_row} -> {curr\_edge}; ", end="")

print()

right\_penalty = Node.reduce(right\_matrix)

print("ДОБАВЛЕНИЕ ОЦЕНКИ НА ОСНОВЕ МОД")

matrix\_for\_mst, transitions = prepare\_matrix\_for\_mst(right\_matrix, right\_route)

vertices = process\_transitions(transitions)

print("Матрица для построения МОД:")

print\_matrix(matrix\_for\_mst)

mst\_bound = minimum\_spanning\_tree(matrix\_for\_mst, vertices)

right\_bound = min\_node.bound + right\_penalty + mst\_bound

right\_child = Node(right\_matrix, right\_bound, right\_route)

print(f"ИТОГ: Для маршрута {left\_route}, проходящего через дугу {row} -> {column} нижняя оценка длины маршрута = {right\_bound}")

return left\_child, right\_child

def little\_algorithm(matrix):

root\_matrix = Node.clone\_matrix(matrix)

min\_bound = Node.reduce(root\_matrix)

root = Node(root\_matrix, min\_bound, [])

print(f"Создаем начальный узел, соответствующий пустому маршруту с нижней оценкой длины маршрута {min\_bound}")

priority\_queue = []

heappush(priority\_queue, (root.bound, root))

record = None

while priority\_queue:

print(f"\nИзвлекаем из очереди с приоритетом узел с наименьшей нижней оценкой длины маршрута:")

\_, min\_node = heappop(priority\_queue)

print(f"ТЕКУЩИЙ УЗЕЛ: {min\_node}")

if record is not None and record['length'] <= min\_node.bound:

print(f"Нижняя оценка текущего маршрута больше, чем существующее решение-рекорд, поэтому переходим дальше...")

continue

# Если маршрут почти завершен (n-2 ребра)

if len(min\_node.route) == len(matrix) - 2:

print("Маршрут почти завершен, добавляем последние два ребра ", end="")

for row in range(len(matrix)):

for column in range(len(matrix)):

if not math.isinf(min\_node.matrix[row][column]):

print(f"{row}->{column}, ", end="")

min\_node.bound += min\_node.matrix[row][column]

min\_node.route.append((row, column))

print()

if record is None or record['length'] > min\_node.bound:

if record:

print(f"Длина текущего маршрута, равная {min\_node.bound} оказалась меньше длины решения-рекорда, равной {record['length']}. Обновляем рекорд...")

else:

print("Получили первый маршрут, проходящий через все города, и приравниваем его к рекорду...")

print(f"Новый рекорд - маршрут {min\_node.route} с длиной {min\_node.bound}")

record = {'length': min\_node.bound, 'route': min\_node.route}

else:

print(f"Создание потомков узла...")

left\_child, right\_child = make\_children(min\_node)

# Добавляем потомков в очередь

heappush(priority\_queue, (left\_child.bound, left\_child))

heappush(priority\_queue, (right\_child.bound, right\_child))

return record

def format\_route(route, n):

if not route:

return []

route\_map = {}

for u, v in route:

route\_map[u] = v

# Начинаем с города 0

current = 0

ordered\_route = [current]

# Строим маршрут по цепочке

while len(ordered\_route) < n:

current = route\_map.get(current)

if current is None:

break

ordered\_route.append(current)

return ordered\_route

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

n = int(input())

cost\_matrix = []

for i in range(n):

row = list(map(int,input().split()))

row[i] = math.inf

cost\_matrix.append(row)

result = little\_algorithm(cost\_matrix)

if result:

n = len(cost\_matrix)

ordered\_route = format\_route(result['route'], n)

print(" ".join(map(str, ordered\_route)))

print(float(result['length']))

Название файла:nearest\_neighbor\_algorithm.py

import math

def nearest\_neighbor\_algorithm(start: int, distance\_matrix: list):

num\_cities = len(distance\_matrix)

unvisited = set(range(num\_cities))

print(f'Начинаем путь с вершины №{start}')

unvisited.remove(start)

total\_cost = 0

path = [start]

current\_city = start

iteration = 1

while unvisited:

print(f'Итерация №{iteration}')

# Находим ближайший непосещенный город

nearest\_city = None

min\_distance = math.inf

for city in unvisited:

distance = distance\_matrix[current\_city][city]

print(f'Расстояние от города №{current\_city} до города №{city} = {distance}')

if 0 < distance < min\_distance:

nearest\_city = city

min\_distance = distance

if nearest\_city is None:

break

unvisited.remove(nearest\_city)

path.append(nearest\_city)

total\_cost += min\_distance

current\_city = nearest\_city

print(f'Выбран путь {path[-2]} → {nearest\_city}, длина: {min\_distance}')

print(f'Текущий путь: {path}, общая стоимость: {total\_cost}\n')

iteration += 1

# Возвращаемся в начальный город

return\_cost = distance\_matrix[current\_city][start]

if not math.isinf(return\_cost) and return\_cost > 0:

total\_cost += return\_cost

path.append(start)

print(f'Возвращаемся в начальный город: {current\_city} → {start}, длина: {return\_cost}')

print('Решение найдено!')

return total\_cost, path

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

n = int(input())

cost\_matrix = []

for i in range(n):

row = list(map(int, input().split()))

row[i] = math.inf

cost\_matrix.append(row)

start = 0

cost, way = nearest\_neighbor\_algorithm(start, cost\_matrix)

print(f"Длина пути: {cost}")

print(f"Путь: {way[0]}", end="")