**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: Алгоритмы для решения задачи коммивояжёра

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Гельман П.Е. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

## Цель работы.

Цель данной лабораторной работы состоит в изучении модифицированного алгоритма Литтла и алгоритма ближайшего соседа для нахождения решения задачи коммивояжера.

## 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 Mi,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

**Задание варианта:**

**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.

**Примечания для варианта:**

Независимо от варианта, при сдаче работы должна быть возможность

генерировать матрицу весов (произвольную или симметричную; для варианта 6 —симметричную), сохранять её в файл и использовать в качестве входных данных.

## Выполнение работы.

Реализованный функционал:

1. Класс Node – класс узла бинарного дерева поиска алгоритма МВиГ. Каждый узел содержит поле матрицы, нижней границы, текущего маршрута. def get\_cell\_with\_max\_penalty(self) – метод, который находит в матрице весов клетку с минимальным штрафом за ее удаление (клетку с 0, у которой сумма минимальных элементов в строке и столбце максимальна). def make\_children(self) – метод, которые создает двух потомков текущего узла – левого и правого. Левые узлы отвечают за исключение текущего ребра, правые – за включение ребра в маршрут. Для левого потомка в матрице запрещается текущее ребро (=inf), для правого запрещается обратное ребро, а также столбец и строка соответствующие пункту прибытия и пункту отправления. Также запрещаются все ребра, которые могут создать подциклы в текущем маршруте и досрочно завершить путь.
2. def clone\_matrix(matrix) – клонирование матриц
3. def row\_mins(matrix) – нахождение минимумов в строках матрицы
4. def column\_mins(matrix) – нахождение минимумов в столбцах
5. def sum\_finites(arr) – суммирует минимумы в переданном массиве
6. def reduce\_rows(matrix, mins) – редуцирование матрицы по строкам
7. def reduce\_columns(matrix, mins) – редуцирование матрицы по столбцам
8. def reduce(matrix, route) – полная редукция матрицы с добавлением к нижней оценке веса решения нижней оценки суммарного веса остатка пути на основе МОД.
9. def find\_next\_start\_city(edges, start\_city) – нахождение следующего номера ребра, которое ведет в заданный город.
10. def find\_next\_end\_city(edges, end\_city) – нахождение следующего номера ребра, которое выходит из заданного города.
11. def get\_close\_edges(route) – нахождение ребер, которые могут досрочно завершать поиск пути. Сначала ищутся ребра, продолжающие цепочку в начало, затем которые продолжают в конец.
12. def little(matrix) – функция алгоритма Литтла. Инициализируется корень дерева, ищется минимальная начальная граница стоимости маршрута. Создается очередь с приоритетом для хранения узлов, из нее извлекается узел с минимальной нижней границей, далее идет проверка, если осталось одно ребро до окончания маршрута, то добавлчем его, проверяем текущее решение на минимальность и завершаем программу, иначе находим правого и левого потомков, и добавляем их в очередь.
13. def print\_matrix(matrix) – печать матрицы
14. def generate\_matrix(n, symmetric=False, min\_weight=1, max\_weight=100, infinity\_prob=0.05) – генерация матрицы, n –размер, symmetric – симметричная/произвольная, min/max\_weight – диапазон значений матрицы, infinity\_prob – вероятность, что какой-то элемент не на главной диагонали будет бесконечностью.
15. def save\_matrix\_to\_file(matrix, filename) – сохранение матрицы в файл
16. def load\_matrix\_from\_file(filename) – выгрузка матрицы из файла
17. def prim\_mst(matrix, route) – модифицированный алгоритм Прима с учетом кусков. В функции находим куски текущего маршрута, если все вершины – один кусок, то завершаем, иначе строим новую урезанную матрицу стоимостей допустимых ребер между кусками, к этой матрице применяем обычный алгоритм поиска МОД.
18. def find\_chunks(route, matrix) – поиск кусков в текущем маршруте.
19. def build\_chunk\_matrix(matrix, chunks) – построение матрицы весов из найденных кусков.
20. def standard\_prim(chunk\_matrix) – обычный алгоритм Прима для нахождения МОД.
21. def nearest\_neighbor(matrix, st\_city=0) – приближенный алгоритм ближайшего соседа. Он последовательно строит маршрут, на каждом шаге выбирая ближайший непосещенный город, пока не обойдет все города, после чего возвращается в начальный пункт. Начинается со стартового города.

Сложность алгоритма.

АБС:

* По памяти О(n), так как храним множество непосещенных городов unvisited и посещенных route.
* По времени О(n2). На каждом из n шагов алгоритм перебирает всех непосещённых соседей (в худшем случае n-1, n-2, ..., 1), что даёт сумму арифметической прогрессии.

Алгоритм Литтла с модификацией:

По времени

Сложность точного алгоритма ЗК (методом ВиГ) в среднем (при «случайных» матрицах стоимостей)

* Cn, где C≅1.26

Однако, можно учесть еще сложность алгоритма Прима для кусков

* Алгоритм Прима: O(m log(n)), m – ребра, n - вершины
* Поиск кусков: O(n) анализ ребер маршрута
* Построение матрицы кусков: O(k^2) – k – число кусков (в худ. случае n)

По памяти

* O(n^2) для матрицы весов
* Память на узел O(n^2) на матрицу, O(n) на маршрут, общая память на дерево O(k \* n^2 + k\*n), k – число активных узлов дерева.
* Приоритетная очередь O(k) хранит все активные узлы
* Итого ~ O(k \* n^2)

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 1 2 3  --------------  1 | -1 33 44  2 | 10 -1 91  3 | 78 13 -1 | 0 2 1  67.0 | Верно |
|  | 1 2 3 4  -------------------  1 | -1 6 24 44  2 | 23 -1 46 7  3 | 51 72 -1 83  4 | 78 96 75 -1 | 0 1 3 2  139.0 | Верно |
|  | 1 2 3 4 5  ------------------------  1 | -1 15 95 61 27  2 | 15 -1 64 65 53  3 | 95 64 -1 30 7  4 | 61 65 30 -1 2  5 | 27 53 7 2 -1 | 0 1 2 3 4  138.0 | Верно |
| 4. | 1 2 3 4  -------------------  1 | -1 6 85 87  2 | -1 -1 72 19  3 | 69 86 -1 24  4 | 33 73 53 -1  Стартовый город 1 | РЕЗУЛЬТАТ:  Оптимальный маршрут: 1 → 2 → 4 → 3 → 1  Общая длина пути: 147 | Верно |
| 5. | 1 2 3  --------------  1 | -1 49 47  2 | 49 -1 72  3 | 47 72 -1  2 | РЕЗУЛЬТАТ:  Оптимальный маршрут: 2 → 1 → 3 → 2  Общая длина пути: 168 | Верно |
| 6. | 1 2 3 4 5 6  -----------------------------  1 | -1 -1 71 7 55 10  2 | 43 -1 51 88 86 46  3 | 80 61 -1 74 24 58  4 | 2 8 27 -1 97 70  5 | 47 80 71 67 -1 81  6 | -1 74 83 15 83 -1 | Результат алгоритма Литтла:  0 5 3 1 2 4  155.0  РЕЗУЛЬТАТ:  Оптимальный маршрут: 1 → 4 → 2 → 6 → 3 → 5 → 1  Общая длина пути: 215 | Верно |

## Выводы.

В ходе лабораторной работы был реализован алгоритм Литтла с модификацией и алгоритм ближайшего соседа, проанализирована их временная сложность и сложность по памяти.

# Приложение А Исходный код программы

Название файла: main.py

import heapq

import math

import random

def print\_matrix(matrix):

n = len(matrix)

header = " " \* 5 + " ".join(f"{i + 1:4}" for i in range(n))

print(header)

print(" " \* 5 + "-" \* len(header[5:]))

for i in range(n):

row\_str = f"{i + 1:2} | "

row\_elements = []

for j in range(n):

if matrix[i][j] == math.inf:

row\_elements.append(" -1")

else:

row\_elements.append(f"{int(matrix[i][j]):4}")

row\_str += " ".join(row\_elements)

print(row\_str)

def generate\_matrix(n, symmetric=False, min\_weight=1, max\_weight=100, infinity\_prob=0.05):

"""Генерация матрицы весов (симметричной или произвольной)"""

matrix = [[math.inf if i == j else 0 for j in range(n)] for i in range(n)]

for i in range(n):

for j in range(n):

if i == j:

continue # диагональ уже заполнена бесконечностями

if symmetric and j < i:

# для симметричной матрицы используем уже заданные значения

matrix[i][j] = matrix[j][i]

else:

# генерируем новое значение

if random.random() < infinity\_prob: # вероятность генерирования инф

weight = math.inf

else:

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

matrix[i][j] = weight

# если матрица симметричная, зеркально отражаем значение

if symmetric and j > i:

matrix[j][i] = weight

return matrix

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

"""Сохранение матрицы в файл"""

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

n = len(matrix)

f.write(f"{n}\n")

for row in matrix:

formatted\_row = [str(int(x) if x != math.inf else -1) for x in row]

f.write(" ".join(formatted\_row) + "\n")

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

"""Чтение матрицы из файла"""

try:

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

n = int(f.readline())

matrix = []

for \_ in range(n):

row = [(x if x != -1 else math.inf) for x in list(map(int, f.readline().split()))]

print(row)

matrix.append(row)

return matrix

except FileNotFoundError:

print(f"Ошибка: файл '{filename}' не найден")

return None

except Exception as e:

print(f"Ошибка при чтении файла: {e}")

return None

def prim\_mst(matrix, route):

"""Модифицированный алгоритм Прима для оценки остатка пути через МОД с учетом кусков"""

n = len(matrix)

if n < 2: # если меньше 2 вершин, мод не существует

return 0

# определяем куски из текущего маршрута

chunks = find\_chunks(route, matrix)

print(f"[PRIM\_MST] Найдены куски: {chunks}")

# если все вершины уже соединены в один кусок

if len(chunks) == 1:

return 0

# строим матрицу допустимых ребер между кусками

chunk\_matrix = build\_chunk\_matrix(matrix, chunks)

print("[PRIM\_MST] Матрица допустимых ребер между кусками:")

print\_matrix(chunk\_matrix)

# применяем алгоритм Прима к этой матрице

return standard\_prim(chunk\_matrix)

def find\_chunks(route, matrix):

"""Находит все куски (связные компоненты) в текущем маршруте"""

if not route:

return []

# собираем все ребра в словарь {from: to}

edges = {src: dst for src, dst in route}

visited = set()

chunks = []

# находим начальные точки кусков (те, которые не являются конечными в других ребрах)

starts = set(edges.keys()) - set(edges.values())

# если все вершины в цикле, берем первую

if not starts:

starts = {route[0][0]}

for start in starts:

if start in visited:

continue

chunk = []

current = start

while current in edges:

chunk.append(current)

visited.add(current)

current = edges[current]

if current not in visited:

chunk.append(current)

visited.add(current)

chunks.append(chunk)

# добавляем изолированные вершины

all\_vertices = set(range(len(matrix)))

isolated = all\_vertices - visited

for v in isolated:

chunks.append([v])

print("chunks", chunks)

return chunks

def build\_chunk\_matrix(matrix, chunks):

"""Строит матрицу допустимых ребер между кусками"""

n\_chunks = len(chunks)

chunk\_matrix = [[math.inf] \* n\_chunks for \_ in range(n\_chunks)]

for i in range(n\_chunks):

for j in range(n\_chunks):

if i == j:

continue

# находим минимальное ребро между последней вершиной i-го куска

# и первой вершиной j-го куска

last\_in\_i = chunks[i][-1]

first\_in\_j = chunks[j][0]

chunk\_matrix[i][j] = matrix[last\_in\_i][first\_in\_j]

return chunk\_matrix

def standard\_prim(chunk\_matrix):

"""Стандартная реализация алгоритма Прима для полученной матрицы кусков"""

n = len(chunk\_matrix)

if n < 2:

return 0

used = [False] \* n

mst\_weight = 0

heap = []

heapq.heappush(heap, (0, 0)) # (weight, chunk\_index)

while heap:

weight, v = heapq.heappop(heap)

if used[v]:

continue

used[v] = True

mst\_weight += weight

for u in range(n):

if not used[u] and chunk\_matrix[v][u] != math.inf:

heapq.heappush(heap, (chunk\_matrix[v][u], u))

return mst\_weight

class Node:

"""Класс для представления узла в дереве поиска алгоритма ветвей и границ"""

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

self.matrix = matrix

self.bound = bound

self.route = route

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

"""

Находит клетку с максимальным штрафом за ее исключение

(клетку с 0, у которой сумма минимальных элементов в строке и столбце максимальна)

"""

print("[CELL WITH MAX PENALTY] Ищем клетку с максимальным штрафом за ее исключение")

max\_penalty = -math.inf # изначально штраф минимальный

cell\_with\_max\_penalty = None

n = len(self.matrix)

for row in range(n):

for col in range(n):

if self.matrix[row][col] == 0: # если клетка содержит 0 (после редукции)

# находим минимальный элемент в строке (исключая текущий столбец)

row\_min = math.inf

for i in range(n):

if i == col or not math.isfinite(self.matrix[row][i]):

continue

if self.matrix[row][i] < row\_min:

row\_min = self.matrix[row][i]

print(f"[CELL WITH MAX PENALTY] Нашли минимальный элемент в строке {row} = {row\_min}")

col\_min = math.inf # находим минимальный элемент в столбце (исключая текущую строку)

for i in range(n):

if i == row or not math.isfinite(self.matrix[i][col]):

continue

if self.matrix[i][col] < col\_min:

col\_min = self.matrix[i][col]

print(f"[CELL WITH MAX PENALTY] Нашли минимальный элемент в столбце {col} = ", col\_min)

penalty = row\_min + col\_min # штраф за исключение этой клетки

print(f"[CELL WITH MAX PENALTY] Штраф за исключение этой клетки [{row},{col}] = ", penalty)

if penalty > max\_penalty:

max\_penalty = penalty

cell\_with\_max\_penalty = (row, col, max\_penalty)

print(f"\n[CELL WITH MAX PENALTY] Текущий штраф [{row},{col}] - максимальный = ", max\_penalty)

return cell\_with\_max\_penalty

def make\_children(self):

"""Создает двух потомков для узла (ветвление)"""

print("[MAKE\_CHILDREN] Создание двух потомков для узла")

cell = self.get\_cell\_with\_max\_penalty() # находим клетку для ветвления

if cell is None: # все клетки обработаны

return None, None

row, col, left\_penalty = cell

print(f"[MAKE\_CHILDREN] Нашли клетку [{row}, {col}] с максимальным штрафом = {left\_penalty}")

# левый потомок - исключаем текущее ребро

left\_matrix = clone\_matrix(self.matrix) # копируем матрицу

print("[MAKE\_CHILDREN] Копируем матрицу в левого потомка")

print("[MAKE CHILDREN] Матрица до запрещения:")

print\_matrix(left\_matrix)

left\_matrix[row][col] = math.inf # запрещаем текущее ребро

print("[MAKE CHILDREN] Матрица после запрещения:")

print\_matrix(left\_matrix)

print(f"[MAKE\_CHILDREN] Запрещаем ребро текущее left\_matrix[{row}][{col}] : {left\_matrix[row][col]}")

left\_bound = self.bound + left\_penalty # обновляем нижнюю границу

print(f"[MAKE\_CHILDREN] Обновляем нижнюю границу для левого потомка: {left\_bound}")

left\_route = self.route.copy() # копируем маршрут

print(f"[MAKE CHILDREN] Копируем маршрут: {left\_route}")

left\_child = Node(left\_matrix, left\_bound, left\_route) # создаем левого потомка

print(f"[MAKE CHILDREN] Создали левого потомка с обновленной матрицей, нижней границей и маршрутом")

# правый потомок - включаем текущее ребро

right\_matrix = clone\_matrix(self.matrix) # копируем матрицу

print("[MAKE CHILDREN] Матрица до запрещения:")

print\_matrix(right\_matrix)

print(f"\n[MAKE CHILDREN] Копируем матрицу для правого потомка")

right\_matrix[col][row] = math.inf # запрещаем обратное ребро

print(f"[MAKE CHILDREN] Запрещаем обратное ребро right\_matrix[{col}][{row}] : {right\_matrix[col][row]}")

print("[MAKE CHILDREN] Матрица после запрещения:")

print\_matrix(right\_matrix)

print(f"[MAKE CHILDREN] Запрещаем все ребра из строки {row} и столбца {col} у правого потомка")

# запрещаем все ребра из текущей строки и столбца

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

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

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

print("[MAKE CHILDREN] Матрица после запрещения:")

print\_matrix(right\_matrix)

right\_route = self.route.copy() # копируем маршрут

print(f"[MAKE CHILDREN] Копируем текущий маршрут в правого потомка : {right\_route}")

right\_route.append((row, col)) # включаем текущее ребро в маршрут

print(f"[MAKE CHILDREN] Включаем текущее ребро в маршрут : {right\_route}")

print(f"\n[MAKE CHILDREN] Ищем все подциклы")

close\_edges = get\_close\_edges(right\_route) # ищем все подциклы

for (r, c) in close\_edges:

print(f"[MAKE CHILDREN] Запрещаем ребро ({r},{c}), которое создает подцикл")

right\_matrix[r][c] = math.inf # запрещаем все ребра, которые создают подциклы

print("[MAKE CHILDREN] Матрица после запрещения подциклов:")

print\_matrix(right\_matrix)

print("[MAKE CHILDREN] Редуцируем матрицу")

right\_penalty = reduce(right\_matrix, right\_route) # редуцируем матрицу

print("[MAKE CHILDREN] Штраф у матрицы правого потомка после редукции: ", right\_penalty)

right\_bound = self.bound + right\_penalty # обновляем нижнюю границу

print("[MAKE CHILDREN] Обновленная нижняя граница у правого: ", right\_bound)

right\_child = Node(right\_matrix, right\_bound, right\_route) # создаем правого потомка

print("[MAKE CHILDREN] Создан правый потомок")

return left\_child, right\_child

def clone\_matrix(matrix):

return [row[:] for row in matrix]

def row\_mins(matrix):

print(f"[ROW MIN] Находим минимумы в строках матрицы")

mins = []

for row in matrix:

min\_val = math.inf

for val in row:

if val < min\_val:

min\_val = val

print(f"[ROW MIN] Найден минимум в строке {row}: {min\_val}")

mins.append(min\_val if min\_val != math.inf else 0)

return mins

def column\_mins(matrix):

mins = []

print(f"[СOLUMN MIN] Находим минимумы в столбцах матрицы")

n = len(matrix)

for col in range(n):

min\_val = math.inf

for row in range(n):

if matrix[row][col] < min\_val:

min\_val = matrix[row][col]

print(f"[COLUMN MIN] Найден минимум в столбце {col}: {min\_val}")

mins.append(min\_val if min\_val != math.inf else 0)

return mins

def sum\_finites(arr):

return sum(val for val in arr if val != math.inf)

def reduce\_rows(matrix, mins):

print(f"[REDUCE ROWS] Редуцируем по строкам")

print("[REDUCE ROWS] Матрица до редуцирования")

print\_matrix(matrix)

n = len(matrix)

for row in range(n):

for col in range(n):

if mins[row] != math.inf:

matrix[row][col] -= mins[row]

print("[REDUCE ROWS] Матрица после редуцирования:")

print\_matrix(matrix)

def reduce\_columns(matrix, mins):

print(f"[REDUCE COLUMNS] Редуцируем по столбцам")

print("[REDUCE COLUMNS] Матрица до редуцирования")

print\_matrix(matrix)

n = len(matrix)

for col in range(n):

for row in range(n):

if mins[col] != math.inf:

matrix[row][col] -= mins[col]

print("[REDUCE COLUMNS] Матрица после редуцирования:")

print\_matrix(matrix)

def reduce(matrix, route):

"""Полная редукция матрицы с добавлением оценки МОД"""

print(f"[REDUCE] Редукция матрицы с добавлением оценки МОД")

row\_m = row\_mins(matrix)

reduce\_rows(matrix, row\_m)

column\_m = column\_mins(matrix)

reduce\_columns(matrix, column\_m)

print(f"[REDUCE] Сумма минимумов по строкам = {sum\_finites(row\_m)}, по столбцам = {sum\_finites(column\_m)}")

reduction\_cost = sum\_finites(row\_m) + sum\_finites(column\_m)

print(f"[REDUCE] Общая стоимость после редуцирования = {reduction\_cost}")

print(f"[REDUCE] Находим нижнюю оценку суммарного веса остатка пути на основе МОД")

mst\_estimate = prim\_mst(matrix, route)

print(f"[REDUCE] Найденная оценка по МОД: {mst\_estimate}")

print(f"[REDUCE] Итого: {reduction\_cost + mst\_estimate}")

return reduction\_cost + mst\_estimate

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

"""Находит индекс ребра, которое ведет в заданный город"""

print(f"[FIND NEXT START CITY] Находим индекс ребра, которое ведет в город {start\_city}")

for i, (\_, dst) in enumerate(edges): # для каждого ребра в маршруте

if dst == start\_city: # если город назначения совпадает с искомым

print(f"[FIND NEXT START CITY] Найдено ребро {i} {edges[i]}, которое ведет в город {start\_city}")

return i

print(f"[FIND NEXT START CITY] Не найдено ребро, которое ведет в город {start\_city}")

return -1

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

"""Находит индекс ребра, которое начинается в заданном городе"""

print(f"[FIND NEXT END CITY] Находим индекс ребра, которое начинается в городе {end\_city}")

for i, (src, \_) in enumerate(edges): # для каждого ребра в маршруте

if src == end\_city: # если город отправления совпадает с искомым

print(f"[FIND NEXT END CITY] Найдено ребро {i} {edges[i]}, которое начинается в {end\_city}")

return i

print(f"[FIND NEXT END CITY] Не найдено ребро, которое начинается в {end\_city}")

return -1

def get\_close\_edges(route):

"""Находит ребра, которые могут образовывать подциклы"""

print(f"\n[GET CLOSE EDGES] ИЩЕМ ВСЕ РЕБРА, КОТОРЫЕ МОГУТ ОБРАЗОВЫВАТЬ ПОДЦИКЛЫ")

result = []

edges = route.copy()

print(f"[GET CLOSE EDGES] Начальные ребра для анализа: {edges}")

while edges:

print(f"\n[GET CLOSE EDGES] Осталось ребер для обработки: {len(edges)}")

length = 1 # длина текущей цепочки

start\_city, end\_city = edges.pop(0) # берем первое ребро

print(f"[GET CLOSE EDGES] Текущее ребро: ({start\_city}, {end\_city})")

print(f"[GET CLOSE EDGES] Начинаем построение цепочки с этого ребра")

# ищем ребра, которые продолжают цепочку в начало

print("\n[GET CLOSE EDGES] Поиск ребер, продолжающих цепочку В НАЧАЛО:")

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

while index != -1:

old\_start = start\_city

start\_city, \_ = edges.pop(index)

length += 1

print(f"[GET CLOSE EDGES] Найдено продолжающее ребро: ({start\_city}, {old\_start})")

print(f"[GET CLOSE EDGES] Обновленная цепочка: ({start\_city} -> ... -> {end\_city})")

print(f"[GET CLOSE EDGES] Текущая длина цепочки: {length}")

index = find\_next\_start\_city(edges, start\_city) # ищем следующее, которое продолжает

# ищем ребра, которые продолжают цепочку в конец

print("\n[GET CLOSE EDGES] Поиск ребер, продолжающих цепочку В КОНЕЦ:")

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

while index != -1:

old\_end = end\_city

\_, end\_city = edges.pop(index)

length += 1

print(f"[GET CLOSE EDGES] Найдено продолжающее ребро: ({old\_end}, {end\_city})")

print(f"[GET CLOSE EDGES] Обновленная цепочка: ({start\_city} -> ... -> {end\_city})")

print(f"[GET CLOSE EDGES] Текущая длина цепочки: {length}")

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

print(f"\n[GET CLOSE EDGES] Итоговая длина цепочки: {length}")

if length >= 2: # если цепочка может образовать цикл

result.append((end\_city, start\_city)) # добавляем в запрещенные ребра

print(f"[GET CLOSE EDGES] Обнаружена потенциальная цикличность!")

print(f"[GET CLOSE EDGES] Добавляем запретное ребро для предотвращения цикла: ({end\_city}, {start\_city})")

else:

print("[GET CLOSE EDGES] Цепочка слишком коротка для образования цикла - пропускаем")

print(f"\n[GET CLOSE EDGES] Итоговый список запретных ребер: {result}")

return result

def little(matrix):

"""Алгоритм Литтла"""

node = Node(matrix, 0, [])

root\_matrix = clone\_matrix(matrix)

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

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

print(f"\nСоздаем корень дерева с матрицей {root\_matrix} и мин.границей {min\_bound}")

priority\_queue = [] # Используем кучу для хранения узлов по приоритету (нижней границе)

heapq.heappush(priority\_queue, (root.bound, id(root), root)) # (bound, id, node)

record = None # лучший найденный маршрут

print(f"Добавляем корень в очередь с приоритетом (граница, номер корня, объект корня): {priority\_queue}")

while priority\_queue: # пока есть узлы

print("\n\nПытаемся извлечь из очереди узел с минимальной нижней границей")

try:

mn, \_, min\_node = heapq.heappop(priority\_queue) # Извлекаем узел с минимальной границей

print(f"Извлекли узел с минимальной границей {mn}")

except IndexError:

print("Очередь пуста, заверщаем работу")

break # если очередь пуста, завершаем работу

print(f"\nТекущий путь: {min\_node.route}, текущая нижняя граница: {min\_node.bound}")

if record and record['length'] <= min\_node.bound: # если есть запись о меньшем маршруте, завершаем

print("Найдена запись о меньшем маршруте, завершаем просмотр очереди")

break

if len(min\_node.route) == len(matrix) - 2: # если маршрут почти полный

print("\nМаршрут почти полный (не хватает двух городов)")

print("Добавляем последние ребра для завершения просмотра")

for row in range(len(matrix)): # добавляем последние ребра для завершения цикла

for col in range(len(matrix)):

if math.isfinite(min\_node.matrix[row][col]):

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

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

print("Обновленная нижняя граница: ", min\_node.bound)

print("Обновленный путь для вершины с мин.границей: ", min\_node.route)

if record is None or record['length'] > min\_node.bound: # обновляем запись, если нашли лучшее решение

print("Было найдено лучшее решение, обновляем запись: ")

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

print(record)

else:

print("\n Находим правого (включает ребро) и левого потомка (исключает ребро) для текущей вершины")

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

# добавляем потомков в очередь, если они существуют

if left\_child is not None:

print(f"Левый потомок найден, добавляем его в очередь")

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

if right\_child is not None:

print(f"Правый потомок найден, добавляем его в очередь")

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

print(record)

return record

def nearest\_neighbor(matrix, st\_city=0):

"""Алгоритм ближайшего соседа для ЗК"""

n = len(matrix)

if n < 2:

return []

print("Входная матрица:")

print\_matrix(matrix)

route = [st\_city]

print(f"Текущий маршрут: {[x + 1 for x in route]}")

unvisited = set(range(n)) - {st\_city} # множество непосещенных городов

print(f"Непосещенные города: {[x + 1 for x in unvisited]}")

total\_distance = 0

current\_city = st\_city

while unvisited:

print(f"\nТекущий город {current\_city + 1}")

# находим ближайшего непосещенного соседа

nearest\_city = None

min\_distance = math.inf

for city in unvisited:

distance = matrix[current\_city][city]

print(f"Дистанция до города {city + 1} от текущего города {current\_city + 1} = {distance}")

if distance != math.inf and distance < min\_distance:

print(f"Найденная дистанция на данный момент минимальная, текущий ближайший город - {city + 1}")

min\_distance = distance

nearest\_city = city

else:

print("Найденная дистанция на данный момент не минимальная, не добавляем ее")

# если нет доступных городов

if nearest\_city is None:

print(f"Доступных ближайших городов не нашлось. Маршрута не существует.")

return None

# добавляем город в маршрут

route.append(nearest\_city)

print(f"Добавляем найденный город в маршрут : {[x + 1 for x in route]}")

unvisited.remove(nearest\_city) # удаляем из непосещенных

print(f"Непосещенные города: {[x + 1 for x in unvisited]}")

total\_distance += min\_distance # добавляем стоимость

print(f"Текущая стоимость {total\_distance}")

current\_city = nearest\_city # текущий - теперь ближайший

# после прохода по каждому городу, возвращаемся в начальный город

return\_distance = matrix[current\_city][st\_city]

print(f"Стоимость возврата в город, из которого отправились: {return\_distance}")

if return\_distance == math.inf: # нет обратного пути

print("Обратного пути нет")

return None

total\_distance += return\_distance

route.append(st\_city)

print(f"Добавляем стартовый город в путь: {[x + 1 for x in route]}")

return {

'route': route,

'distance': total\_distance

}

def main():

cur\_matrix = None

n = 2

while True:

print('\n')

print("=" \* 150)

print("Команды:")

print("1. Сгенерировать случайную матрицу")

print("2. Загрузить матрицу из файла")

print("3. Сохранить матрицу в файл")

print("4. Решить задачу методом ближайшего соседа")

print("5. Решить задачу методом ветвей и границ (Литтла)")

print("6. Показать текущую матрицу")

print("7. Ввести матрицу вручную")

print("8. Выход")

c = input("Выберите действие (1-8): ")

if c == "1":

n = int(input("Введите количество городов N: "))

sym = input("Симметричная матрица? (y/n): ").lower() == 'y'

cur\_matrix = generate\_matrix(n, sym)

print("\nСгенерированная матрица:")

print\_matrix(cur\_matrix)

elif c == "2":

filename = input("Введите имя файла: ")

loaded\_matrix = load\_matrix\_from\_file(filename)

n = len(loaded\_matrix)

if loaded\_matrix is not None:

cur\_matrix = loaded\_matrix

print("\nЗагруженная матрица:")

print\_matrix([[math.inf if x == -1 else x for x in row] for row in cur\_matrix])

elif c == "3":

if cur\_matrix is None:

print("Нет матрицы для сохранения")

continue

filename = input("Введите имя файла для сохранения: ")

save\_matrix\_to\_file(cur\_matrix, filename)

print(f"Матрица сохранена в файл '{filename}'")

elif c == "7":

n = int(input("Введите количество городов N: "))

cur\_matrix = []

print(f"\nВведите матрицу {n}x{n}:")

cost\_matrix = []

for \_ in range(n):

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

cost\_matrix.append(row)

for i in range(n):

matrix\_row = []

for j in range(n):

if cost\_matrix[i][j] == -1:

matrix\_row.append(math.inf)

else:

matrix\_row.append(cost\_matrix[i][j])

cur\_matrix.append(matrix\_row)

print("\nВведенная матрица:")

print\_matrix(cur\_matrix)

elif c == "4":

if cur\_matrix is None:

print("Сначала загрузите или создайте матрицу")

continue

n = len(cur\_matrix)

while True:

try:

start\_city = int(input(f"Введите стартовый город (1-{len(cur\_matrix)}): "))

if 1 <= start\_city <= len(cur\_matrix):

break

print("Номер города вне допустимого диапазона")

except ValueError:

print("Введите целое число")

result = nearest\_neighbor(cur\_matrix, start\_city - 1)

print("\n" + "=" \* 50)

print("РЕЗУЛЬТАТ:")

if result:

print(f"Оптимальный маршрут: {' → '.join(map(str, [x + 1 for x in result['route']]))}")

print(f"Общая длина пути: {result['distance']}")

else:

print("Невозможно построить маршрут для данной матрицы")

elif c == "5":

if cur\_matrix is not None:

result = little(cur\_matrix)

n = len(cur\_matrix)

print("\nРезультат алгоритма Литтла:")

if result:

route = [0] \* n

cur\_city = 0

route[0] = 0

next\_city = {}

for (src, dst) in result['route']:

next\_city[src] = dst

for i in range(1, n):

route[i] = next\_city.get(cur\_city, 0)

cur\_city = route[i]

print(' '.join(map(str, route)))

print(f"{result['length']:.1f}")

else:

print("Матрица не введена")

elif c == "6":

if cur\_matrix is not None:

print\_matrix(cur\_matrix)

else:

print("Матрица не существует")

elif c == "8":

print("Выход")

exit(0)

else:

print("Некорректная команда")

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

main()