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, *route* - 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: $O(C^n)$, где C = 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: $O(2^n*N^2)$, так как в худшем случае строится полное бинарное дерево, содержащее 2^{n-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: $O(N^2)$, 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
1.	∞ 16 2 16 ∞ 13 7 11 ∞	Длина пути: 29.0 Путь: 0 2 1	Литтл: работает корректно.
2.	0 ∞ 16 2 16 ∞ 13 7 11 ∞	Длина пути: 29 Путь: 0-2-1-0	АБС: работает корректно. Выдал решение совпадающее с точным.
3.	∞ 12 2 12 ∞ 17 2 17 ∞	Длина пути: 31.0 Путь: 0 1 2	Литтл: работает корректно для симметричной матрицы.
4.	1 ∞ 12 2 12 ∞ 17 2 17 ∞	Длина пути: 31 Путь: 1-0-2-1	АБС: работает корректно, начиная обработку с ненулевой вершины.
5.	∞ 1 1 1 ∞ 1 1 1 ∞	Длина пути: 3.0 Путь: 0 1 2	Литтл: работает корректно, когда расстояние между городами одинаковое
6.	∞ 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	Длина пути: 54 Путь: 3-4-0-1-2-3	АБС: работает корректно, выдал не

	4	16	∞	13	6		точное, а
	16	17	18	∞	8		приближенное
	7	19	10	16	∞		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="")
```