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 графах: коммивояжёр
Вариант: 2

Студентка гр. 3388 _____ Титкова С.Д.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы:

Изучить принципы работы алгоритмов на графах. Решить с помощью них задачу Коммивояжёра.

Задание:

Решить задачу Коммивояжёра 2 различными способами. 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. Начинать АБС со стартовой вершины.

Реализация

Описание алгоритма Литтла с модификацией:

Задача коммивояжера заключается в нахождении кратчайшего замкнутого маршрута (гамильтонова цикла), проходящего через все вершины полного взвешенного графа ровно один раз с возвращением в начальную точку. Алгоритм Литтла применяет метод ветвей и границ для эффективного поиска такого маршрута, избегая полного перебора всех возможных путей.

Алгоритм начинает с исходной матрицы расстояний между вершинами (городами) и использует редукцию для получения начальной нижней границы стоимости маршрута. Затем он итеративно разбивает задачу на подзадачи (ветвление), оценивает их перспективность через нижние границы и отсекает неперспективные варианты, пока не найдет оптимальный цикл.

Исходные данные представляют собой квадратную матрицу n×n где n — число вершин. Элемент matrix[i][j] — это стоимость пути из вершины i в вершину j, а диагональные элементы (i=j) равны бесконечности (∞), так как петли запрещены.

Выполняется начальная редукция матрицы. Для каждой строки находится минимальное значение (игнорируя ∞) и вычитается из всех элементов строки. Это приводит к появлению хотя бы одного нуля в каждой строке. Аналогично, для каждого столбца находится минимальное значение и вычитается из всех элементов столбца. Сумма всех вычтенных минимальных значений становится начальной нижней границей — минимально возможной стоимостью маршрута.

Создается корневая подзадача с редуцированной матрицей, пустым маршрутом и начальной границей.

Для каждого узла определяется нижняя граница — минимальная стоимость, которую можно достичь, продолжая маршрут из текущего состояния. Используется два подхода для вычисления нижней границы:

1. **Сумма редукций**: исходная граница увеличивается на основе изменений при ветвлении (штрафы или стоимость включенных дуг).

2. **Граф минимального остовного дерева (МОД)**: строится граф на основе текущих кусков маршрута, находящихся в процессе формирования. Вес минимального остова этого графа добавляется к границе, чтобы оценить стоимость соединения оставшихся вершин. Если эта оценка выше текущей границы, она используется как более точная.

На каждом шаге выбирается дуга (ребро) для включения или исключения из маршрута на основе анализа матрицы:

Выбор дуги: среди всех ячеек с нулевым значением в матрице определяется та, исключение которой приведет к наибольшему увеличению нижней границы (штраф). Штраф вычисляется как сумма минимальных значений в строке и столбце этой ячейки (исключая саму ячейку). Создаются два потомка:

Исключение дуги (левый потомок): Выбранная дуга запрещается (ее стоимость устанавливается в ∞), а также запрещается дополнительная дуга для предотвращения подциклов. Матрица редуцируется заново, и нижняя граница увеличивается на штраф.

Включение дуги (правый потомок): Выбранная дуга добавляется в маршрут, вся строка и столбец этой дуги запрещаются (заполняются ∞), исключая возможность повторного посещения вершин. Матрица редуцируется, и нижняя граница корректируется на основе новой редукции.

Запрет подциклов: на каждом этапе проверяется, чтобы добавление дуги не замкнуло цикл меньше n вершин. Это достигается выбором дополнительной дуги для запрета в левом потомке, например, обратной дуги на первом шаге или дуги, замыкающей треугольник на последующих.

Все узлы хранятся в очереди с приоритетом, где приоритет определяется нижней границей. На каждой итерации выбирается узел с минимальной нижней границей для дальнейшего ветвления. Если граница узла превышает текущую лучшую стоимость полного маршрута, узел отсекается (не исследуется).

Когда маршрут в узле достигает длины n−1(посещены все вершины кроме одной), проверяется возможность замыкания цикла. Находится последняя вершина и проверяется, существует ли конечный путь от последней вершины маршрута к начальной. Если цикл гамильтонов (посещает все вершины ровно один раз и возвращается в начало), его стоимость вычисляется как сумма весов дуг. Если эта стоимость меньше текущей лучшей, она сохраняется как новый результат.

Алгоритм завершается, когда очередь узлов пуста или найден маршрут, чья стоимость не может быть улучшена (все оставшиеся узлы имеют большую нижнюю границу). Возвращается оптимальный маршрут и его длина.

Особенности алгоритма

- **Редукция**: Уменьшение значений в матрице позволяет сосредоточиться на нулевых ячейках, упрощая выбор дуг.
- **Штрафы**: Использование штрафов при исключении дуг помогает отсечь неперспективные ветви.
- **МОД**: Добавление оценки через минимальное остовное дерево улучшает точность нижних границ, ускоряя отсечение.

.

Описание функций и структур:

- *Node* – класс для представления подзадачи в методе ветвей и границ. Хранит состояние каждой подзадачи, включая матрицу, маршрут и оценку, для обработки и ветвления.
    1. *matrix:* Матрица расстояний (n×n) для текущего узла.
    2. *bound:* Нижняя граница стоимости маршрута.
    3. *route:* Список дуг текущего маршрута.
    4. *pieces:* Список кусков маршрута (подпоследовательностей вершин).
    5. *parent:* Ссылка на родительский узел для построения дерева.
    6. *depth:* Глубина узла в дереве поиска.
- *clone_matrix(matrix)* – метод, который создает копию матрицы расстояний, чтобы изменения в одной подзадаче не влияли на другие
- *row_mins(matrix)* - метод, который, находит минимальные элементы в каждой строке матрицы для редукции.
- *column_mins(matrix)* – метод, который находит минимальные элементы в каждом столбце матрицы для редукции.
- *reduce_rows(matrix, mins)* – метод, который вычитает минимальное значение строки из всех ее элементов.
- *reduce_columns(matrix, mins)* - метод, который вычитает минимальное значение столбца из всех ее элементов.
- *reduce(matrix)* – метод, который выполняет полную редукцию матрицы и вычисляет начальную нижнюю границу.
- *get_cell_with_max_penalty(self)* – метод, который определяет дугу для ветвления на основе максимального штрафа
- *get_lower_bounds(self)* – метод, который улучшает нижнюю границу с учетом остатка пути
- *get_acceptable_edges(self)* – метод, который формирует список ребер для построения графа МОД

- *build_mod_graph(self, edges)* – метод, который создает граф для вычисления минимального остова
- *calculate_mod_weight(self, mod_graph)* – метод, который определяет вес минимального остовного дерева
- *is_hamiltonian_cycle(route)* – функция, которая проверяет, является ли маршрут замкнутым циклом через все вершины
- *make_children(min_node)* – функция, которая генерирует узлы для ветвления
- *little(matrix)* – функия, которая реализует полный процесс поиска оптимального маршрута
- *hierarchy_pos(G, root=None, width=2., vert_gap=0.4, vert_loc=0, xcenter=0.5)* – функия, которая определяет координаты узлов для визуализации дерева поиска
- *visualize_tree(nodes)* – функия, которая создает графическое представление процесса поиска
- *Main()* – функция считывает из файла матрицу и вызывает функцию little. Выводит оптимальный путь и его длину.

Оценка сложности алгоритма:

Временная сложность: O(n^3·2^n) в худшем случае.

Редукция матрицы: O(n^2) для каждой строки и столбца (по n элементов).

Поиск ячейки с максимальным штрафом: O(n^3) для проверки всех ячеек и вычисления штрафов.

Построение МОД: O(n^2) для создания графа и O(n^2logn) для сортировки ребер (в худшем случае O(n^2) из-за числа ребер).

Количество узлов: В худшем случае алгоритм исследует все возможные подмножества вершин, что дает O(2^n) узлов.

Итог: O(n^3) операций на узел умножается на O(2^n) узлов, что дает O(n^3·2^n).

Однако отсечение по границам значительно сокращает количество исследуемых узлов в среднем случае, делая алгоритм эффективнее полного перебора (O(n!)).

Сложность относительно памяти: O(2^n)

Матрица: O(n^2) для хранения матрицы в каждом узле.

Очередь узлов: В худшем случае O(2^n) узлов, каждый из которых хранит матрицу (O(n^2)), маршрут (O(n)), и куски (O(n)).

Итог: O(2^n).

Описание приближённого алгоритма: АБС:

Алгоритм ближайшего соседа — это жадный подход, который строит маршрут, на каждом шаге выбирая ближайшую непосещенную вершину, начиная с произвольной точки.

Алгоритм начинает с выбранной начальной вершины и итеративно добавляет к маршруту ближайшую непосещенную вершину, пока все вершины не будут включены. Затем он проверяет возможность возвращения в начальную точку, чтобы замкнуть цикл. Этот метод не гарантирует нахождение оптимального решения, но обеспечивает быстрый и приемлемый результат для многих практических случаев.

Исходные данные представляют собой квадратную матрицу n×n, где n — число вершин. Элемент matrix[i][j] обозначает стоимость пути из вершины i в вершину j, а диагональные элементы (i=j) равны бесконечности (∞), так как петли запрещены.

Выбирается начальная вершина, которая становится первым элементом маршрута.

Создается массив или список для отслеживания посещенных вершин, изначально все вершины, кроме начальной, отмечены как непосещенные.

Для каждой итерации (всего n−1 итераций, чтобы добавить оставшиеся вершины). Определяется текущая вершина — последняя добавленная в

маршрут. Из матрицы расстояний извлекаются все непосещенные вершины, доступные из текущей вершины, исключая те, пути к которым имеют бесконечную стоимость (∞). Среди доступных вершин выбирается та, расстояние до которой минимально. Если такой вершины нет (все пути бесконечны), алгоритм завершается с ошибкой, так как маршрут невозможен. Выбранная вершина добавляется в маршрут, и она отмечается как посещенная.

После добавления всех n вершин проверяется возможность возвращения из последней вершины маршрута в начальную. Если путь от последней вершины к начальной имеет бесконечную стоимость, алгоритм завершается с ошибкой, так как гамильтонов цикл невозможен. Если путь конечен, маршрут замыкается добавлением начальной вершины в конец.

Далее рассчитывается суммарная стоимость маршрута:

Для каждой пары последовательных вершин в маршруте берется соответствующее значение из матрицы расстояний.

Добавляется стоимость пути от последней вершины к начальной для завершения цикла.

Итоговая стоимость представляет длину найденного гамильтонова цикла.

Алгоритм возвращает построенный маршрут (список вершин) и его общую стоимость.

Если на каком-либо этапе маршрут не может быть построен (из-за бесконечных путей), возвращается индикация неудачи (None).

Особенности

- **Жадный подход**: Выбор ближайшего соседа на каждом шаге упрощает вычисления, но может привести к субоптимальному решению, так как не учитывает глобальную структуру графа.
- **Проверка бесконечностей**: Алгоритм явно обрабатывает случаи, когда пути имеют бесконечную стоимость, что важно для графов с отсутствующими ребрами.

- **Линейная последовательность**: Маршрут строится последовательно, без возврата к предыдущим решениям, что отличает его от методов полного перебора или ветвей и границ.

Описание функций и структур:

- *nearest_neighbor_tsp(distance_matrix, start_vertex=0)* – функция реализует алгоритм ближайшего соседа для построения гамильтонова цикла.
- *Main()* – функция считывает матрицу и стартовою вершину, запускает основной алгоритм, выводит длину пути и сам путь.

Оценка сложности алгоритма:

Временная сложность: O(n^2)

Инициализация: Создание структуры для отслеживания посещенных вершин занимает O(n).

Поиск ближайшего соседа: На каждой из n−1 итераций для текущей вершины проверяются все непосещенные вершины (в худшем случае n вершин), что требует O(n) операций.

Вычисление стоимости: Суммирование n n n дуг маршрута занимает O(n) O(n) O(n).

Итог: (n−1)·O(n)= O(n^2).

Сложность относительно памяти: O(n^2)

Матрица расстояний: Хранение матрицы n×n требует O(n^2).

Структура посещенных вершин: Список из n булевых значений занимает O(n).

Маршрут: Список из n+1 вершин (включая возврат) занимает O(n).

Итог: O(n^2) для матрицы плюс O(n) для маршрута и посещенных вершин, что упрощается до O(n^2 + n), где O(n^2) доминирует.

Дополнительные функции:

- *generate_matrix(size, max_weight=50)* – функция генерирует матрицу, заполняя её рандомными значениями.
- *generate_symmetric_matrix(size, max_weight=50)* - функция генерирует симметричную матрицу, заполняя её рандомными значениями.
- *save_matrix_to_file(matrix, filename)* – функция сохраняет матрицу в файл.
- *load_matrix_from_file(filename)* – функция считывает матрицу из файла.

Тестирование

Таблица 1. Тестирование.

Входные данные	Выходные данные Литтла	Выходные данные АБС
inf 37 27 8 inf 34 12 11 inf	Минимальный путь: [(0, 2), (1, 0), (2, 1)] Длина пути: 46.0	Маршрут: [2, 1, 0, 2] Общее расстояние: 46.0
inf 27 16 12 29 28 inf 23 36 10 12 18 inf 38 22 49 23 18 inf 10 16 1 26 36 inf	Минимальный путь: [(0, 3), (2, 0), (4, 1), (1, 2), (3, 4)] Длина пути: 58.0	Маршрут: [2, 0, 3, 4, 1] Общее расстояние: 58.0
inf 44 47 4 46 9 40 45 1 inf 6 26 46 42 37 46 8 37 inf 30 5 32 36 47 16 38 3 inf 40 6 34 14 20 49 8 25 inf 39 10 3 23 30 43 26 46 inf 18 2 25 20 47 33 18 34 inf 26 36 49 16 15 49 45 35 inf	Минимальный путь: [(6, 1), (2, 4), (1, 0), (4, 6), (5, 7), (0, 3), (3, 5), (7, 2)] Длина пути: 95.0	Маршрут: [2, 4, 7, 3, 5, 6, 1, 0, 2] Общее расстояние: 115.0
inf 23 48 20 16 6 23 44 1 23 inf 41 13 5 13 20 9 49 44 3 inf 47 40 45 36 32 41 26 39 40 inf 33 6 26 50 19 5 5 46 13 inf 19 16 24 29 5 23 17 36 39 inf 25 15 5	Минимальный путь: [(2, 1), (3, 5), (0, 8), (8, 2), (5, 0), (6, 7), (7, 4), (1, 6), (4, 3)] Длина пути: 95.0	Маршрут: [2, 1, 4, 0, 8, 5, 7, 6, 3, 2] Общее расстояние: 131.0

33 44 30 29 10 29 inf 10 42 44 22 28 26 13 18 25 inf 18 48 11 9 26 50 8 14 49 inf		
inf 10 37 49 44 10 inf 20 19 50 37 20 inf 22 50 49 19 22 inf 34 44 50 50 34 inf	Минимальный путь: [(0, 1), (3, 4), (4, 0), (1, 2), (2, 3)] Длина пути: 130.0	Маршрут: [2, 1, 0, 4, 3] Общее расстояние: 130.0
inf 14 6 31 32 1 43 50 6 14 inf 49 30 44 5 36 41 25 6 49 inf 43 38 38 18 17 33 31 30 43 inf 12 40 27 17 22 32 44 38 12 inf 49 7 32 28 1 5 38 40 49 inf 45 29 40 43 36 18 27 7 45 inf 3 8 50 41 17 17 32 29 3 inf 1 6 25 33 22 28 40 8 1 inf	Минимальный путь: [(2, 0), (5, 1), (0, 5), (3, 4), (4, 6), (8, 7), (1, 3), (6, 8), (7, 2)] Длина пути: 87.0	Маршрут: [2, 0, 5, 1, 8, 7, 6, 4, 3] Общее расстояние: 103.0

Вывод

В ходе лабораторной работы была написана программа с использованием модифицированного алгоритма Литтла и алгоритма ближайшего соседа. На основании тестирования, можно сказать, что первый алгоритм более точный, нежели жадный алгоритм.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Little.py

```
import random
import matplotlib
matplotlib.use('Agg')
import math
import networkx as nx
import matplotlib.pyplot as plt
DEBUG = False


class Node:
    def __init__(self, matrix, bound, route, pieces, parent=None,
depth=0):
        self.matrix = matrix
        self.bound = bound
        self.route = route
        self.pieces = pieces
        self.parent = parent
        self.depth = depth

    @staticmethod
    def clone_matrix(matrix):
        return [row[:] for row in matrix]

    @staticmethod
    def row_mins(matrix):
        return [min(row) for row in matrix]

    @staticmethod
    def column_mins(matrix):
        return [min(matrix[i][j] for i in range(len(matrix))) for j in
range(len(matrix[0]))]

    @staticmethod
    def reduce_rows(matrix, mins):
        for i in range(len(matrix)):
            if math.isfinite(mins[i]):
                matrix[i] = [cell - mins[i] for cell in matrix[i]]

    @staticmethod
    def reduce_columns(matrix, mins):
        for j in range(len(matrix[0])):
            if math.isfinite(mins[j]):
                for i in range(len(matrix)):
                    matrix[i][j] -= mins[j]

    @staticmethod
    def reduce(matrix):
        row_mins = Node.row_mins(matrix)
        Node.reduce_rows(matrix, row_mins)

        col_mins = Node.column_mins(matrix)
        Node.reduce_columns(matrix, col_mins)
```

```
        return sum(val for val in row_mins if math.isfinite(val)) +
sum(val for val in col_mins if math.isfinite(val))

    def get_cell_with_max_penalty(self):
        max_penalty = -math.inf
        cell_with_max_penalty = None

        if DEBUG:
            print("Поиск ячейки с максимальной штрафной стоимостью:")

        for i in range(len(self.matrix)):
            for j in range(len(self.matrix[i])):
                if self.matrix[i][j] == 0:
                    row_min = min(
                        (self.matrix[i][k]       for       k       in
range(len(self.matrix[i])) if k != j),
                        default=math.inf
                    )
                    col_min = min(
                        (self.matrix[k][j]       for       k       in
range(len(self.matrix)) if k != i),
                        default=math.inf
                    )
                    penalty = row_min + col_min
                    if penalty > max_penalty:
                        max_penalty = penalty
                        cell_with_max_penalty = (i, j, max_penalty)

                    if DEBUG:
                        print(f"  Ячейка ({i}, {j}): штраф {penalty}")

        if DEBUG:
            print(f"    Ячейка с максимальной штрафной стоимостью:
{cell_with_max_penalty}\n")
        return cell_with_max_penalty

    def get_lower_bounds(self):
        if DEBUG:
            print("Вычисление нижней оценки на основе графа МОД:")

        dopustimye_dugi = self.get_acceptable_edges()
        mod_graph = self.build_mod_graph(dopustimye_dugi)
        bound2 = self.calculate_mod_weight(mod_graph)

        if DEBUG:
            print(f"  Нижняя оценка: {bound2}\n")
        return bound2

    def get_acceptable_edges(self):
        acceptable_edges = []
        for i in range(len(self.pieces)):
            for j in range(i + 1, len(self.pieces)):
                u, v = self.pieces[i][-1], self.pieces[j][0]
                if u != v and math.isfinite(self.matrix[u][v]):
                    acceptable_edges.append((u, v))
                u, v = self.pieces[j][-1], self.pieces[i][0]
                if u != v and math.isfinite(self.matrix[u][v]):
```

```
                    acceptable_edges.append((u, v))
        return acceptable_edges

    def build_mod_graph(self, edges):
        if DEBUG:
            print("Построение графа МОД:")

        mod_graph = {}
        for edge in edges:
            if edge[0] not in mod_graph:
                mod_graph[edge[0]] = []
            if edge[1] not in mod_graph:
                mod_graph[edge[1]] = []
            mod_graph[edge[0]].append((edge[1],
self.matrix[edge[0]][edge[1]]))
            mod_graph[edge[1]].append((edge[0],
self.matrix[edge[0]][edge[1]]))

        if DEBUG:
            print(f"  Граф МОД: {mod_graph}\n")
        return mod_graph

    def calculate_mod_weight(self, mod_graph):
        mod_weight = 0
        used_edges = set()
        edges = []
        for node in mod_graph:
            for edge in mod_graph[node]:
                if math.isfinite(edge[1]):
                    edges.append((edge[1], node, edge[0]))
        edges.sort()
        for edge in edges:
            if  (edge[1],  edge[2])  not  in  used_edges  and  (edge[2],
edge[1]) not in used_edges:
                mod_weight += edge[0]
                used_edges.add((edge[1], edge[2]))
                used_edges.add((edge[2], edge[1]))
        return mod_weight


def is_hamiltonian_cycle(route):
    if DEBUG:
        print("Проверка на гамильтонов цикл:")

    if len(route) != len(set(route)):
        if DEBUG:
            print("     Не  гамильтонов  цикл:  длина  маршрута  не
соответствует количеству уникальных вершин.")
        return False
    graph = {}
    for u, v in route:
        graph[u] = v
    visited = set()
    current = route[0][0]
    while current not in visited:
        visited.add(current)
        if current not in graph:
            if DEBUG:
```

```
                print("  Не гамильтонов цикл: не все вершины посещены.")
            return False
        current = graph[current]

    if len(visited) == len(graph):
        if DEBUG:
            print("  Гамильтонов цикл найден.\n")
        return True
    else:
        if DEBUG:
            print("  Не гамильтонов цикл: не все вершины посещены.\n")
        return False


def make_children(min_node):
    if DEBUG:
        print("Создание потомков:")

    row, column, left_penalty = min_node.get_cell_with_max_penalty()
    if row is None or column is None:
        return []
    left_matrix = Node.clone_matrix(min_node.matrix)
    left_matrix[row][column] = math.inf
    left_route = min_node.route[:]
    left_pieces = [piece[:] for piece in min_node.pieces]

    forbidden_row, forbidden_col = None, None
    if min_node.depth == 0:
        forbidden_row, forbidden_col = column, row
    else:
        for piece in left_pieces:
            if piece[-1] == row:
                piece.append(column)
                break
        else:
            left_pieces.append([row, column])

        for piece in left_pieces:
            if len(piece) >= 3:
                if piece[-2] == row and piece[-1] == column:
                    for other_piece in left_pieces:
                        if other_piece != piece and other_piece[0] !=
piece[-1]:
                            forbidden_row, forbidden_col = piece[-1],
other_piece[0]
                            break
                    if forbidden_row is None:
                        forbidden_row,  forbidden_col  =  piece[-1],
piece[0]
                    break

        if forbidden_row is None or forbidden_col is None:
            forbidden_row, forbidden_col = column, row

    if forbidden_row is not None and forbidden_col is not None:
        left_matrix[forbidden_row][forbidden_col] = math.inf

    Node.reduce(left_matrix)
```

```
    left_bound = min_node.bound + left_penalty
    left_child = Node(left_matrix, left_bound, left_route, left_pieces,
parent=min_node, depth=min_node.depth + 1)

    right_matrix = Node.clone_matrix(min_node.matrix)
    right_matrix[column][row] = math.inf
    for i in range(len(right_matrix)):
        right_matrix[row][i] = math.inf
        right_matrix[i][column] = math.inf

    right_route = min_node.route + [(row, column)]
    right_penalty = Node.reduce(right_matrix)
    right_bound = min_node.bound + right_penalty
    right_pieces = [piece[:] for piece in min_node.pieces]
    for piece in right_pieces:
        if piece[-1] == row:
            piece.append(column)
            break
    else:
        right_pieces.append([row, column])

    right_child   =   Node(right_matrix,   right_bound,   right_route,
right_pieces, parent=min_node, depth=min_node.depth + 1)

    if DEBUG:
        print(f"     Левый потомок: граница {left_bound}, маршрут
{left_route}, запрещена дуга ({forbidden_row}, {forbidden_col})")
        print(f"    Правый потомок: граница {right_bound}, маршрут
{right_route}")

    return [left_child, right_child]


def little(matrix):
    if DEBUG:
        print("Запуск алгоритма Little:")

    root_matrix = Node.clone_matrix(matrix)
    min_bound = Node.reduce(root_matrix)
    root = Node(root_matrix, min_bound, [], [[0]])
    priority_queue = [root]
    record = None
    nodes_for_graph = []

    while priority_queue:
        if DEBUG:
            print("Выбор узла с минимальной границей:")

        min_node = min(priority_queue, key=lambda node: node.bound)
        priority_queue.remove(min_node)
        nodes_for_graph.append(min_node)

        if DEBUG:
            print(f"      Узел с минимальной границей: граница
{min_node.bound}, маршрут {min_node.route}\n")

        if record is not None and record['length'] <= min_node.bound:
            if DEBUG:
```

```
                print("Остановка алгоритма: найден оптимальный
маршрут.")
            break

        if len(min_node.route) == len(matrix) - 1:
            if DEBUG:
                print("Построение полного маршрута:")

            for row in range(len(matrix)):
                for column in range(len(matrix)):
                    if math.isfinite(min_node.matrix[row][column]):
                        min_node.bound += min_node.matrix[row][column]
                        min_node.route.append((row, column))

            if is_hamiltonian_cycle(min_node.route):
                if record is None or record['length'] > min_node.bound:
                    record = {'length': min_node.bound, 'route':
min_node.route}
                    if DEBUG:
                        print(f"Найден оптимальный маршрут: длина
{record['length']}, маршрут {record['route']}\n")


        else:
            if DEBUG:
                print("Вычисление нижней оценки:")

            lower_bound = min_node.get_lower_bounds()
            if lower_bound > min_node.bound:
                min_node.bound = lower_bound
                if DEBUG:
                    print(f"          Обновлена граница узла:
{min_node.bound}\n")

            left_child, right_child = make_children(min_node)
            priority_queue.append(left_child)
            priority_queue.append(right_child)

    return record, nodes_for_graph


def hierarchy_pos(G, root=None, width=2., vert_gap=0.4, vert_loc=0,
xcenter=0.5):

    if not nx.is_tree(G):
        raise TypeError('cannot use hierarchy_pos on a graph that is not
a tree')

    if root is None:
        if isinstance(G, nx.DiGraph):
            root = next(iter(nx.topological_sort(G)))
        else:
            root = random.choice(list(G.nodes))

    def _hierarchy_pos(G, root, width=1., vert_gap=0.2, vert_loc=0,
xcenter=0.5, pos=None, parent=None):
```

```
        if pos is None:
            pos = {root: (xcenter, vert_loc)}
        else:
            pos[root] = (xcenter, vert_loc)
        children = list(G.neighbors(root))
        if not isinstance(G, nx.DiGraph) and parent is not None:
            children.remove(parent)
        if len(children) != 0:
            dx = width / len(children)
            nextx = xcenter - width / 2 - dx / 2
            for child in children:
                nextx += dx
                pos     =     _hierarchy_pos(G,     child,     width=dx,
vert_gap=vert_gap,
                                    vert_loc=vert_loc    -    vert_gap,
xcenter=nextx, pos=pos,
                                    parent=root)
        return pos

    return _hierarchy_pos(G, root, width, vert_gap, vert_loc, xcenter)


def visualize_tree(nodes):
    G = nx.DiGraph()
    root_node = None

    for node in nodes:
        G.add_node(id(node),                label=f"B:{node.bound:.0f},
R:{node.route}")
        if node.parent:
            G.add_edge(id(node.parent), id(node))
        else:
            root_node = node

    pos = hierarchy_pos(G, root=id(root_node), width=4, vert_gap=0.8)

    labels   =   {node_id:   G.nodes[node_id]['label']   for   node_id   in
G.nodes()}

    plt.figure(figsize=(20, 12))  # Increased figure size
    nx.draw(G,       pos,       with_labels=False,       node_size=2000,
node_color='lightblue', font_size=10,
            arrowsize=20)
    nx.draw_networkx_labels(G, pos, labels=labels, font_size=10)

    plt.title("Search Tree", fontsize=16)
    plt.savefig("search_tree.png")
    print("График сохранен в search_tree.png")
```

NN.py

```
import                                                          math

DEBUG                                =                          False

def       nearest_neighbor_tsp(distance_matrix,       start_vertex=0):
    n                       =                       len(distance_matrix)
    visited            =            [False]            *            n
```

```
    route = [start_vertex]
    visited[start_vertex] = True

    if DEBUG:
        print("Начало поиска маршрута ближайшего соседа.")
        print(f"Матрица расстояний: {distance_matrix}")
        print(f"Начальный город: {start_vertex}")

    for _ in range(n - 1):
        last_city = route[-1]

        if DEBUG:
            print(f"\nИщем ближайшего соседа для города {last_city}...")
        eligible_neighbors = [(i, distance_matrix[last_city][i]) for i
in range(n) if not visited[i] and distance_matrix[last_city][i] !=
math.inf]

        if DEBUG:
            print(f"Возможные соседи: {eligible_neighbors}")

        nearest_city = min(eligible_neighbors, key=lambda x: x[1],
default=(None, math.inf))

        if nearest_city[0] is None:
            print("Невозможно найти путь без бесконечностей.")
            return None, None

        if DEBUG:
            print(f"Ближайший город: {nearest_city[0]} (расстояние:
{nearest_city[1]})")

        route.append(nearest_city[0])
        visited[nearest_city[0]] = True

    if distance_matrix[route[-1]][start_vertex] == math.inf:
        print("Невозможно вернуться в начальный город без
бесконечностей.")
        return None, None

    total_distance = 0
    for i in range(n - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    total_distance += distance_matrix[route[-1]][start_vertex]

    if DEBUG:
        print(f"\nЗавершен поиск маршрута.")

    return route, total_distance
```

matrix.py

```
import random
import math
DEBUG = 0

def generate_matrix(size, max_weight=50):
    matrix = [[math.inf if i == j else random.randint(1, max_weight) for
j in range(size)] for i in range(size)]
    return matrix
```

```
def generate_symmetric_matrix(size, max_weight=50):
    matrix = [[math.inf if i == j else 0 for j in range(size)] for i in
range(size)]
    for i in range(size):
        for j in range(i + 1, size):
            weight = random.randint(1, max_weight)
            matrix[i][j] = weight
            matrix[j][i] = weight
    return matrix


def save_matrix_to_file(matrix, filename):
    with open(filename, 'w') as file:
        for row in matrix:
            file.write(' '.join(map(str, row)) + '\n')


def load_matrix_from_file(filename):
    matrix = []
    try:
        with open(filename, 'r') as file:
            for line in file:
                row = list(map(lambda x: float(x) if x != 'inf' else
math.inf, line.split()))
                matrix.append(row)

        n = len(matrix)
        if not all(len(row) == n for row in matrix):
            print("Ошибка: Матрица не квадратная (число столбцов не
равно числу строк).")
            return None

        for i in range(n):
            for j in range(n):
                if i == j:
                    if not math.isinf(matrix[i][j]):
                        print(
                            f"Ошибка: Элемент на диагонали ({i}, {j})
должен быть бесконечностью, а не {matrix[i][j]}.")
                        return None
                else:
                    if matrix[i][j] < 0:
                        print(
                            f"Ошибка: Элемент ({i}, {j}) =
{matrix[i][j]} отрицательный, ожидается неотрицательное значение.")
                        return None
                    if math.isnan(matrix[i][j]):
                        print(f"Ошибка: Элемент ({i}, {j}) содержит NaN,
что недопустимо.")
                        return None

        if DEBUG:
            print(f"Матрица успешно загружена из файла {filename} и
проверена на корректность.")
        return matrix

    except FileNotFoundError:
        print(f"Ошибка: Файл {filename} не найден.")
```

```
        return None
    except ValueError:
        print("Ошибка: В файле содержатся некорректные значения (не числа).")
        return None
```

main.py

```
from Little import little,visualize_tree
from matrix import generate_matrix, generate_symmetric_matrix,save_matrix_to_file,load_matrix_from_file
from NN import nearest_neighbor_tsp

print("Введите размер матрицы:")
size = int(input())
print("1 - Симметричная")
print("2 - Обычная")
type = int(input())

if type == 1:
    matrix = generate_symmetric_matrix(size)
elif type == 2:
    matrix = generate_matrix(size)

print("Сгенерированная матрица:")
for row in matrix:
    print(row)
filename = 'matrix.txt'
save_matrix_to_file(matrix, filename)
print(f"Матрица сохранена в файл {filename}")

print("")
matrix = load_matrix_from_file('matrix.txt')
result, nodes = little(matrix)
print("Модифицированный алгоритм Литтла:")
print(f"Минимальный путь: {result['route']}")
print(f"Длина пути: {result['length']}")
visualize_tree(nodes)

print("")
print("Алгоритм АБС:")
print("Введите стартовую вершину:")
start_vertex = int(input())
route, distance = nearest_neighbor_tsp(matrix, start_vertex)
print(f"Маршрут: {route}")
print(f"Общее расстояние: {distance}")
```