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

Студент гр. 3388 _____ Снигирев А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы:

Изучить алгоритмы, реализующие решение задачи коммивояжера и реализовать Метод Ветвей и Границ, а также алгоритм ближайшего соседа.

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

Алгоритма Литтла:

Задача коммивояжера состоит в построении наиболее выгодного гамильтонова пути по графу из городов. Между городами-вершинами проложены дороги-ребра разной стоимости. Алгоритм Литтла решает задачу, работая с матрицей смежности графа городов.

В начале алгоритм выполняет редукцию матрицы — вычитает из каждой строки значение ее минимального элемента. Тоже самое он делает и со столбцами. Сложив вычтенные значения, алгоритм получает так называемое значение нижней границы на текущем шаге. Путь, стоимостью меньше этого значения построить невозможно. Редукция будет выполняться в дальнейшем на каждом шаге алгоритма, получая новые значения нижней границы.

Далее алгоритм строит бинарное дерево матриц. Принцип таков: одна ветвь дерева содержит все пути, в которых будет определенное ребро, другая ветвь — все пути, где это ребро остутствует. Алгоритм выбирает ребро с максимальным штрафом и создает двух потомков первоначальной матрицы по этому ребру. К нижней границе ветви дерева, где было запрещено ребро, необходимо прибавить значение штрафа запрещенного ребра. В дальнейшей работе, при определенных условиях, ветвь решений можно будет «отсечь» как заведомо невыгодную.

Чтобы запретить ребро, алгоритм в нужной ячейке матрицы смежности устанавливает бесконечность.

В правой ветви дерева алгоритм, наоборот, добавляет ребро в буфер решения. По определению гамильтоного пути, добавление ребра в решение автоматически запрещает множество путей, соответствующих переходу в «конец» ребра и выходу из «начала» ребра.

Это означает, что в матрице смежности алгоритм обращает в бесконечность все элементы строки и столбца, в которых расположено ребро,

запрещая эти пути. После этого происходит редукция матрицы с обновлением нижней границы для этой ветви.

После этого происходит повтор: выбор ребра с максимальным штрафом и ветвление по нему.

Нижняя граница

Выше объясняется оценка нижней границы по сумме редукций. В модифицированном варианте алгоритма Литтла используется МОД — метод минимальных остовных деревьев.

Алгоритм Прима строит минимальное остовное дерево по текущей матрице смежности и стоимость прохождения по этому дереву может превосходить нижнюю границу, построенную по редукциям. Т.е такой подход может ускорить работу алгоритма, за счет более строгой оценки нижних границ.

Отслеживание циклов

На какой-то итерации алгоритм может выбрать ребро, которое замкнет гамильтонов цикл до прохода по всем вершинам. Чтобы избежать этой ошибки, и перед добавлением ребра происходит проверка цепочки ребер на корректность.

Таким образом алгоритм сохраняет все текущие матрицы и их нижние границы в очередь с приоритетом. В начале каждой итерации выбирается узел с минимальной нижней границей, чтобы продолжить ветвление с него.

В итоге цикл завершится, когда закончатся непосещенные вершины и будет получен первый гамильтонов цикл.

Далее алгоритм начинает искать лучший путь. Все ветви, нижние границы которых будут больше, чем стоимость прохода по текущему лучшему пути, отсекаются — они априори не дадут лучшее решение.

Если найдется путь с меньшей стоимостью, то он станет новым лучшим.

Алгоритм завершит свою работу, когда будет найден маршрут, чья стоимость не будет превосходить все оставшиеся нижние границы, т.е когда опустошится очередь с приоритетом.

Жадный алгоритм

Дополнительно реализован второй способ решения задачи, использующий жадный алгоритм — Алгоритм Ближайшего Соседа.

Алгоритм стартует в произвольной вершине и на каждом шаге выбирает непосещенную вершину, путь к которой даст наименьшие затраты.

После посещения всех вершин алгоритм проверяет возможность возвращения в стартовую вершину.

Жадный алгоритм редко возвращает оптимальный путь, однако результат не будет отличаться от оптимального более чем в два раза. При этом алгоритму требуется намного меньше времени для работы, что делает его очень даже применимым на практике.

Основные функции:

- *get_infinums(self)* – метод, который улучшает нижнюю границу, используя остовные деревья
- *get_acceptable_edges(self)* – метод, который формирует список ребер для построения графа МОД
- *build_mod_graph(self, edges)* – метод, который создает граф для вычисления минимальной границы
- *calculate_mod_weight(self, mod_graph)* – метод, который определяет вес минимального остовного дерева
- *is_hamilton_cycle(route)* – функция, которая проверяет, является ли маршрут замкнутым циклом через все вершины
- *create_branches(min_node)* – функция, которая генерирует узлы для ветвления
- *LittleAlgorithm(matrix)* – функия, которая реализует полный процесс поиска оптимального маршрута

- *NearestNeighborAlgorithm(distance_matrix, start_vertex=0)* – функция реализует алгоритм ближайшего соседа для построения гамильтонова цикла.
- *Main()* – функция считывает из файла матрицу и вызывает функцию *LittleAlgorithm и NearestNeighborAlgorithm*. Выводит оптимальный путь и его длину.

Оценка сложности алгоритма:

Временная сложность: O(n^2·2^n)

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

Поиск ячейки с максимальным штрафом: O(n^2) для проверки всех ячеек и вычисления штрафов.

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

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

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

Оценка сложности алгоритма:

Временная сложность: O(n^2)

n-1 итераций по вершинам, n проверок при поиске ближайшего соседа на каждой итерации.

Вывод

В ходе лабораторной работы было написано решение задачи коммивояжера двумя способами. Алгоритм Литтла, основанный на методе ветвей и границ, находит оптимальный путь за приемлемое время по сравнению с полным перебором.

Жадный алгоритм и аналитически и экспериментально проигрывает первому алгоритму в точности, однако имеет значительное преимущество в скорости работы.

Исходный код см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

little_algorithm.py

```
import math
DEBUG = True
class Node:
       def __init__(self, matrix, infinum, way, edges, parent=None,
height=0):
        self.matrix = matrix
        self.infinum = infinum
        self.way = way
        self.edges = edges
        self.parent = parent
        self.height = height

    @staticmethod
    def copy_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):
        if DEBUG:
            print("Редукция матрицы...")
        row_mins = Node.row_mins(matrix)
        Node.reduce_rows(matrix, row_mins)

        col_mins = Node.column_mins(matrix)
        Node.reduce_columns(matrix, col_mins)
        result = sum(val for val in row_mins if math.isfinite(val)) +
sum(val for val in col_mins if math.isfinite(val))
        if DEBUG:
                    print(f"Нижняя граница, полученная при редукции:
{result}")
        return result

    def find_edge_max_penalty(self):
        max_penalty = -math.inf
        edge_max_penalty = None

        if DEBUG:
            print("Поиск ребра с максимальным штрафом")

        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
                        edge_max_penalty = (i, j, max_penalty)
```

```
                    if DEBUG:
                        print(f"Ребро ({i}, {j}): Штраф {penalty}")

        if DEBUG:
             print(f"Ребро с максимальным штрафом: {edge_max_penalty}\
n")
        return edge_max_penalty

    def get_infinums(self):
        if DEBUG:
              print("Вычисление нижней оценки с помощью Минимального
Остовного Дерева:")

                                                    mod_infinum      =
self.calculate_mod_weight(self.make_mod_graph(self.get_acceptable_edge
s()))

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

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

    def make_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"Минимальное остовное дерево: {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_hamilton_cycle(way):
    if DEBUG:
        print("Проверка на гамильтонов цикл:")

    if len(way) != len(set(way)):
        if DEBUG:
                    print("Не гамильтонов цикл: длина маршрута не
соответствует количеству уникальных вершин.")
        return False
    graph = {}
    for u, v in way:
        graph[u] = v
    visited = set()
    current = way[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 create_branches(min_node):
    if DEBUG:
        print("Создание потомков:")

    row, column, left_penalty = min_node.find_edge_max_penalty()
    if row is None or column is None:
        return []
    left_matrix = Node.copy_matrix(min_node.matrix)
    left_matrix[row][column] = math.inf
    left_way = min_node.way[:]

    left_edges = [edge[:] for edge in min_node.edges]

    cyclic_row, cyclic_col = None, None
```

```
    if min_node.height == 0:
        cyclic_row, cyclic_col = column, row
    else:
        for edge in left_edges:
            if edge[-1] == row:
                edge.append(column)
                break
        else:
            left_edges.append([row, column])

        for edge in left_edges:
            if len(edge) >= 3:
                if edge[-2] == row and edge[-1] == column:
                    for other_edge in left_edges:
                            if other_edge != edge and other_edge[0] !=
edge[-1]:
                                    cyclic_row, cyclic_col = edge[-1],
other_edge[0]
                            break
                    if cyclic_row is None:
                        cyclic_row, cyclic_col = edge[-1], edge[0]
                    break

        if cyclic_row is None or cyclic_col is None:
            cyclic_row, cyclic_col = column, row

    if cyclic_row is not None and cyclic_col is not None:
        left_matrix[cyclic_row][cyclic_col] = math.inf

    Node.reduce(left_matrix)
    left_infinum = min_node.infinum + left_penalty
     left_child = Node(left_matrix, left_infinum, left_way, left_edges,
parent=min_node, height=min_node.height + 1)

    right_matrix = Node.copy_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_way = min_node.way + [(row, column)]
    right_penalty = Node.reduce(right_matrix)
    right_infinum = min_node.infinum + right_penalty
    right_edges = [edge[:] for edge in min_node.edges]
    for edge in right_edges:
        if edge[-1] == row:
            edge.append(column)
            break
    else:
        right_edges.append([row, column])

         right_child  =  Node(right_matrix,  right_infinum,  right_way,
right_edges, parent=min_node, height=min_node.height + 1)

    if DEBUG:
          print(f"Левый потомок: нижняя граница: {left_infinum}, путь:
{left_way}, запрещенная дуга: ({cyclic_row}, {cyclic_col})")
```

```
          print(f"Правый потомок: нижняя граница {right_infinum}, путь
{right_way}")

    return [left_child, right_child]


def LittleAlgorithm(matrix):
    root_matrix = Node.copy_matrix(matrix)
    min_infinum = Node.reduce(root_matrix)
    root = Node(root_matrix, min_infinum, [], [[0]])
    priority_queue = [root]
    best_way = None
    nodes_for_graph = []

    while priority_queue:
        if DEBUG:
                print("Выбираем из очереди узел с минимальной нижней
границей")

        min_node = min(priority_queue, key=lambda node: node.infinum)
        # Удаляем этот узел из очереди
        priority_queue.remove(min_node)
        nodes_for_graph.append(min_node)

        if DEBUG:
             print(f"Граница: {min_node.infinum}, Путь: {min_node.way}\
n")

               if best_way is not None and best_way['length'] <=
min_node.infinum:
            if DEBUG:
                 print("Больше нет конкурентоспособных путей. Алгоритм
завершен")
            break

        if len(min_node.way) == 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.infinum += min_node.matrix[row]
[column]
                        min_node.way.append((row, column))
                        # Добавление замыкающего ребра

            if is_hamilton_cycle(min_node.way):
                         if best_way is None or best_way['length'] >
min_node.infinum:
                        best_way = {'length': min_node.infinum, 'way':
min_node.way}
                    if DEBUG:
                                  print(f"Найден новый маршрут: Длина
{best_way['length']}, Путь {best_way['way']}\n")
        else:
            if DEBUG:
                print("Вычисление нижней оценки:")
```

```
            # Применение МОД для уточнения минимальной нижней границы
            lower_infinum = min_node.get_infinums()
            if lower_infinum > min_node.infinum:
                min_node.infinum = lower_infinum
                if DEBUG:
                                        print(f"Обновлена граница узла:
{min_node.infinum}\n")
            # Создание потомков
            left_child, right_child = create_branches(min_node)
            # Добавление потомков в очередь
            priority_queue.append(left_child)
            priority_queue.append(right_child)

    return best_way, nodes_for_graph
```

nearest_neighbour.py

```
import math

DEBUG = False

def NearestNeighborAlgorithm(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_algorithm import LittleAlgorithm
from             matrix             import             generate_matrix,
generate_symmetric_matrix,save_matrix_to_file,load_matrix_from_file
from nearest_neighbour import NearestNeighborAlgorithm

print("Считать матрицу из файла? (Y/n)")
answer = input()
if answer == 'Y':
    print("Введите имя файла")
    filename = input()
    matrix = load_matrix_from_file(filename)
else:
    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 = LittleAlgorithm(matrix)
print("Алгоритм Литтла:")
print(f"Минимальный путь: {result['way']}")
print(f"Длина пути: {result['length']}")

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