**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Никишин С.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить и реализовать алгоритм “Метод ветвей и границ” (МВиГ) и приближенный алгоритм “Алгоритм включения ближайшего города” (АВБГ), используемые при решении задачи коммивояжёра.

**Задание.**

Вариант 4.

4 МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — не в глубину, а по антиприоритету(S/k+L/N)(4N/(3N+k)). Приближённый алгоритм: АВБГ "улучшенный". Замечание к варианту 4 И МВиГ, и АВБГ "улучшенный" начинать со стартовой вершины.

**Описание алгоритма.**

1. Метод ветвей и границ (МВиГ):

Метод ветвей и границ — это точный алгоритм, который находит оптимальное решение задачи коммивояжёра. Он работает путём систематического перебора всех возможных путей, с использованием оценок для отсечения заведомо неперспективных ветвей.

Алгоритм:

1. Начинаем с начальной вершины. Создает стек и добавляем начальную ветвь.

2. Проверяем на завершенность пути. Если путь содержит все города:

2.1 Замыкаем путь, добавляя начальную вершину, если в нее есть путь.

2.2 Если текущее решение лучше обновляем лучшее решение.

3. Отсечение неперспективных ветвей. Вычисляем нижнюю границу для текущей ветви и отсекаем если стоимость + границу больше лучшей стоимости.

4. Производим ветвление. Для последней вершины в пути рассматриваем все возможные продолжения.

Для каждой непосещённой вершины:

4.1 Обновляем стоимость

4.2 Обновляем путь и список посещенных вершин

4.3 Добавляем новую ветвь в очередь.

5. Для оптимизации сохраняем промежуточные пути (длиной больше или равных 4) в хэш-таблицу.

6. Продолжаем пока очередь не пуста.

Сложность по памяти: O(2n)

Сложность по времени: O(2n)

2. Алгоритм включения ближайшего города (АВБГ):

Метод включения ближайшего города — это эвристический алгоритм, который быстро находит приближённое решение задачи коммивояжёра.

Алгоритм:

1. Начинаем с начальной вершины.

2. Осуществляем поиск ближайшей вершины (На каждом шаге находим вершину, которая ближе всего к любой из вершин в текущем пути).

3. Вставляем найденную вершину в оптимальную позицию в текущем пути.

3.1 В конец текущего пути

3.2 Между двумя другими вершинами текущего пути

4. Обновляем стоимость.

5. Продолжаем пока все вершины не будут пройдены.

6. Замыкаем путь, добавляя начальную вершину.

Сложность по памяти: O(n)

Сложность по времени: O(n2)

|  |  |  |
| --- | --- | --- |
|  | МВиГ | АВБГ |
| Сложность по времени | O(2n) | O(n2) |
| Сложность по памяти | O(2n) | O(n) |

**Описание функций и структур данных.**

Исходный код программы представлен в приложении (приложение 1).

Структуры данных:

* *matrix* - матрица весов (список списков)
* *priority\_queue* - приоритетная очередь (куча). Используется для хранения ветвей в функции branchAndBoundV4
* *visited* - множество посещенных вершин (set)
* *parts* - список кусков путей для вычисления нижних оценок

Функции:

1. generateWeightMatrix(n: int, symmetric: bool = True) - генерирует матрицу весов со случайными значениями

Параметры:

n - размер матрицы (int)

symmetric - флаг симметричности матрицы (bool)

Возвращает: matrix - сгенерированную матрицу весов

Сложность по времени: O(n²)

1. saveWeightMatrix(fileName: str, matrix: list) - сохраняет матрицу в файл

Параметры:

fileName - имя файла (str)

matrix - матрица весов (list)

Сложность по времени: O(n²)

1. loadWightMatrix(fileName: str) - загружает матрицу из файла

Параметры: fileName - имя файла (str)

Возвращает: matrix - матрица весов (list)

Сложность по времени: O(n²)

1. primMST(matrix: list, vertices: set) -> float - вычисляет вес минимального остовного дерева

Параметры:

matrix - матрица весов (list)

vertices - множество вершин (set)

Возвращает: вес МОД (float)

Сложность по времени: O(V²) где V - количество вершин

1. lowerBound1(matrix: list, path: list, visited: set) -> float - первая нижняя оценка (полусумма минимальных ребер)

Параметры:

matrix - матрица весов (list)

path - текущий путь (list)

visited - посещенные вершины (set)

Возвращает: нижняя оценка (float)

Сложность по времени: O(n³)

1. lowerBound2(matrix: list, path: list, visited: set) -> float - вторая нижняя оценка (вес МОД)

Параметры: аналогично lowerBound1

Возвращает: нижняя оценка (float)

Сложность по времени: O(n²)

1. heuristicPriority(S: float, k: int, L: float, N: int) -> float - вычисляет приоритет по эвристике

Параметры:

S - средний вес ребра (float)

k - длина текущего пути (int)

L - средний вес из текущей вершины (float)

N - размер графа (int)

Возвращает: значение приоритета (float)

Сложность по времени: O(1)

1. branchAndBoundV4(matrix: list, start: int) - метод ветвей и границ

Параметры:

matrix - матрица весов (list)

start - начальная вершина (int)

Возвращает: bestPath, bestCost - лучший путь и его стоимость

Сложность по времени: O(2ⁿ) в худшем случае

1. improvedNearestInsertion(matrix: list, start: int) - улучшенный алгоритм АВБГ

Параметры: аналогично branchAndBoundV4

Возвращает: path, cost - найденный путь и стоимость

Сложность по времени: O(n³)

|  |  |
| --- | --- |
| Функция | Сложность по времени |
| generateWeightMatrix | O(n²) |
| saveWeightMatrix | O(n²) |
| loadWightMatrix | O(n²) |
| primMST | O(V²) |
| lowerBound1 | O(n³) |
| lowerBound2 | O(n²) |

**Применённые оптимизации.**

Применённые оптимизации в МВиГ:

Две нижние оценки - используются одновременно:

LB1: полусумма минимальных входящих и исходящих ребер по всем кускам пути

LB2: вес минимального остовного дерева для непосещенных вершин

Берется максимальная из двух оценок для более точного отсечения

Эвристика антиприоритета - вместо поиска в глубину используется приоритетная очередь с эвристикой:

Формула: (S/k + L/N) \* (4N/(3N+k))

Где S - средний вес ребра, k - длина пути, L - средний вес из текущей вершины, N - размер графа

Позволяет выбирать наиболее перспективные ветви для расширения. Последовательный рост пути - используется приоритетная очередь вместо стека, что обеспечивает более систематический поиск. Жадная сортировка кандидатов - при расширении пути кандидаты сортируются по возрастанию веса ребра

Применённые оптимизации в АВБГ:

Оптимальный выбор позиции вставки - для каждой новой вершины проверяются все возможные позиции вставки

Минимизация увеличения стоимости - выбирается позиция, дающая минимальное увеличение общей стоимости пути

Систематический перебор - гарантирует нахождение локально оптимального решения

**Тестирование.**

Тестирование методов представлено на таблице.

Таблица 1. Метод Ветвей и Границ

|  |  |  |
| --- | --- | --- |
| Номер теста | Входные данные | Вывод |
| 1 | 0 5  5 0 | Метод ветвей и границ: Путь = [0, 1, 0], Стоимость = 10 |
| 2 | 0 2 2 9 5 3  2 0 7 2 1 6  2 7 0 4 1 1  9 2 4 0 2 8  5 1 1 2 0 9  3 6 1 8 9 0 | Метод ветвей и границ: Путь = [0, 1, 3, 4, 2, 5, 0], Стоимость = 11 |
| 3 | 0 1 2 6 5 2  5 0 5 4 1 6  7 7 0 7 6 1  5 5 4 0 8 6  7 1 4 7 0 3  6 2 1 2 6 0 | Метод ветвей и границ: Путь = [0, 1, 4, 2, 5, 3, 0], Стоимость = 14 |
| 4 | 0 1 2 4 3 3 6 6  5 0 7 2 7 8 2 1  8 9 0 3 6 7 9 4  5 9 7 0 4 7 7 1  5 6 9 7 0 3 3 7  1 5 5 1 1 0 7 9  3 1 5 3 4 5 0 3  2 1 6 6 7 3 7 0 | Метод ветвей и границ: Путь = [0, 2, 3, 7, 1, 6, 4, 5, 0], Стоимость = 17 |
| 5 | 0 4 7 1 1 4 6 9 9 3 7 1  4 0 5 4 5 5 7 5 9 5 1 8  7 5 0 9 6 9 2 3 9 8 9 8  1 4 9 0 2 5 9 9 4 7 3 3  1 5 6 2 0 6 2 6 7 9 9 3  4 5 9 5 6 0 8 1 4 8 2 7  6 7 2 9 2 8 0 1 2 9 3 2  9 5 3 9 6 1 1 0 4 7 4 6  9 9 9 4 7 4 2 4 0 3 4 6  3 5 8 7 9 8 9 7 3 0 4 5  7 1 9 3 9 2 3 4 4 4 0 4  1 8 8 3 3 7 2 6 6 5 4 0 | Метод ветвей и границ: Путь = [0, 4, 3, 1, 10, 5, 7, 2, 6, 8, 9, 11, 0], Стоимость = 27 |

Таблица 2. Алгоритм включения ближайшего города

|  |  |  |
| --- | --- | --- |
| Номер теста | Входные данные | Вывод |
| 1 | 0 5  5 0 | Метод АВБГ: Путь = [0, 1, 0], Стоимость = 10 |
| 2 | 0 2 2 9 5 3  2 0 7 2 1 6  2 7 0 4 1 1  9 2 4 0 2 8  5 1 1 2 0 9  3 6 1 8 9 0 | Метод АВБГ: Путь = [0, 1, 3, 4, 2, 5, 0], Стоимость = 11 |
| 3 | 0 1 2 6 5 2  5 0 5 4 1 6  7 7 0 7 6 1  5 5 4 0 8 6  7 1 4 7 0 3  6 2 1 2 6 0 | Метод АВБГ: Путь = [0, 2, 5, 3, 1, 4, 0], Стоимость = 18 |
| 4 | 0 1 2 4 3 3 6 6  5 0 7 2 7 8 2 1  8 9 0 3 6 7 9 4  5 9 7 0 4 7 7 1  5 6 9 7 0 3 3 7  1 5 5 1 1 0 7 9  3 1 5 3 4 5 0 3  2 1 6 6 7 3 7 0 | Метод АВБГ: Путь = [0, 5, 4, 2, 1, 6, 3, 7, 0], Стоимость = 30 |
| 5 | 0 4 7 1 1 4 6 9 9 3 7 1  4 0 5 4 5 5 7 5 9 5 1 8  7 5 0 9 6 9 2 3 9 8 9 8  1 4 9 0 2 5 9 9 4 7 3 3  1 5 6 2 0 6 2 6 7 9 9 3  4 5 9 5 6 0 8 1 4 8 2 7  6 7 2 9 2 8 0 1 2 9 3 2  9 5 3 9 6 1 1 0 4 7 4 6  9 9 9 4 7 4 2 4 0 3 4 6  3 5 8 7 9 8 9 7 3 0 4 5  7 1 9 3 9 2 3 4 4 4 0 4  1 8 8 3 3 7 2 6 6 5 4 0 | Метод АВБГ: Путь = [0, 9, 11, 6, 8, 2, 7, 5, 10, 1, 4, 3, 0], Стоимость = 36 |

**Выводы.**

Были изучены и реализованы алгоритм “Метод ветвей и границ” (МВиГ) и приближенный алгоритм “Алгоритм включения ближайшего города” (АВБГ), используемые при решении задачи коммивояжёра. Полученные при тестировании ответы подтвердили корректность работы алгоритмов.

**Приложение**

Приложение 1

import random

import sys

import heapq

def generateWeightMatrix(n: int, symmetric: bool = True):

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

for y in range(n):

for x in range(n):

if x == y:

matrix[y][x] = 0

elif symmetric:

if x > y:

weight = random.randint(1, 10)

matrix[y][x] = weight

matrix[x][y] = weight

else:

if x != y:

matrix[y][x] = random.randint(1, 10)

return matrix

def saveWeightMatrix(fileName: str, matrix: list):

with open(fileName, "w") as file:

for line in matrix:

file.write(" ".join(map(str, line)) + "\n")

def loadWightMatrix(fileName: str):

with open(fileName, "r") as file:

matrix = [list(map(int, line.split(" "))) for line in file.readlines()]

return matrix

def printMatrix(matrix: list, title: str = ""):

if title:

print(f"\n{title}")

n = len(matrix)

print(" ", end="")

for i in range(n):

print(f"{i:3}", end="")

print()

for i in range(n):

print(f"{i:2} ", end="")

for j in range(n):

print(f"{matrix[i][j]:3}", end="")

print()

def primMST(matrix: list, vertices: set) -> float:

if not vertices:

return 0

n = len(matrix)

visited = set()

mst\_weight = 0

start = next(iter(vertices))

visited.add(start)

while len(visited) < len(vertices):

min\_edge = float('inf')

next\_vertex = -1

for v in visited:

for u in vertices - visited:

if matrix[v][u] > 0 and matrix[v][u] < min\_edge:

min\_edge = matrix[v][u]

next\_vertex = u

if next\_vertex == -1:

break

mst\_weight += min\_edge

visited.add(next\_vertex)

return mst\_weight

def lowerBound1(matrix: list, path: list, visited: set) -> float:

n = len(matrix)

bound = 0

parts = [path] + [[i] for i in range(n) if i not in visited]

for part in parts:

start = part[0]

end = part[-1]

incoming = []

outgoing = []

for otherPart in parts:

if otherPart != part:

otherPartStart = otherPart[0]

otherPartEnd = otherPart[-1]

if matrix[otherPartEnd][start] != 0:

incoming.append(matrix[otherPartEnd][start])

if matrix[end][otherPartStart] != 0:

outgoing.append(matrix[end][otherPartStart])

incomingMin = min(incoming) if incoming else 0

outgoingMin = min(outgoing) if outgoing else 0

bound += (incomingMin + outgoingMin) / 2

return bound

def lowerBound2(matrix: list, path: list, visited: set) -> float:

unvisited = set(range(len(matrix))) - visited

return primMST(matrix, unvisited)

def heuristicPriority(S: float, k: int, L: float, N: int) -> float:

if k == 0:

return float('inf')

return (S/k + L/N) \* (4\*N/(3\*N + k))

def branchAndBoundV4(matrix: list, start: int):

n = len(matrix)

bestPath = None

bestCost = float('inf')

print("\nМетод ветвей и границ (вариант 4)")

print(f"Начальная вершина: {start}")

print(f"Размер графа: {n}")

total\_edges = 0

total\_weight = 0

for i in range(n):

for j in range(n):

if matrix[i][j] > 0:

total\_edges += 1

total\_weight += matrix[i][j]

S = total\_weight / total\_edges if total\_edges > 0 else 0

L = sum(matrix[start][j] for j in range(n) if matrix[start][j] > 0) / n

priority\_queue = []

initial\_path = [start]

initial\_visited = set([start])

initial\_cost = 0

initial\_priority = heuristicPriority(S, 1, L, n)

heapq.heappush(priority\_queue, (initial\_priority, initial\_cost, initial\_path, initial\_visited))

iteration = 0

while priority\_queue:

iteration += 1

priority, cost, path, visited = heapq.heappop(priority\_queue)

print(f"\nИтерация {iteration}: путь {path}, стоимость {cost}")

if len(path) == n:

if matrix[path[-1]][start] > 0:

total\_cost = cost + matrix[path[-1]][start]

if total\_cost < bestCost:

bestCost = total\_cost

bestPath = path + [start]

print(f"Найден новый лучший путь: стоимость {bestCost}")

continue

lb1 = lowerBound1(matrix, path, visited)

lb2 = lowerBound2(matrix, path, visited)

lb = max(lb1, lb2)

print(f"Нижние оценки: LB1={lb1:.2f}, LB2={lb2:.2f}, максимум={lb:.2f}")

if cost + lb >= bestCost:

print(f"Отсечение: {cost + lb:.2f} >= {bestCost}")

continue

last\_vertex = path[-1]

candidates = []

for next\_vertex in range(n):

if next\_vertex not in visited and matrix[last\_vertex][next\_vertex] > 0:

candidates.append(next\_vertex)

candidates.sort(key=lambda v: matrix[last\_vertex][v])

for next\_vertex in candidates:

new\_path = path + [next\_vertex]

new\_visited = visited | {next\_vertex}

new\_cost = cost + matrix[last\_vertex][next\_vertex]

k = len(new\_path)

L\_new = sum(matrix[last\_vertex][j] for j in range(n) if matrix[last\_vertex][j] > 0) / n

new\_priority = heuristicPriority(S, k, L\_new, n)

heapq.heappush(priority\_queue, (new\_priority, new\_cost, new\_path, new\_visited))

print(f" Добавлен путь: {new\_path}, стоимость: {new\_cost}")

return bestPath, bestCost

def improvedNearestInsertion(matrix: list, start: int):

n = len(matrix)

path = [start]

cost = 0

visited = set([start])

print("\nУлучшенный алгоритм включения ближайшего города")

print(f"Начальный путь: {path}")

step = 0

while len(path) < n:

step += 1

best\_improvement = float('inf')

best\_vertex = -1

best\_position = -1

best\_cost\_change = 0

for candidate in range(n):

if candidate in visited:

continue

for pos in range(len(path)):

current\_vertex = path[pos]

next\_vertex = path[(pos + 1) % len(path)]

current\_edge\_cost = matrix[current\_vertex][next\_vertex]

new\_edges\_cost = matrix[current\_vertex][candidate] + matrix[candidate][next\_vertex]

cost\_change = new\_edges\_cost - current\_edge\_cost

if cost\_change < best\_improvement:

best\_improvement = cost\_change

best\_vertex = candidate

best\_position = pos

best\_cost\_change = cost\_change

if best\_vertex != -1:

cost += best\_cost\_change

path.insert(best\_position + 1, best\_vertex)

visited.add(best\_vertex)

print(f"Шаг {step}: добавлена вершина {best\_vertex} в позицию {best\_position}, стоимость {cost}")

else:

return None, float('inf')

if matrix[path[-1]][start] > 0:

closing\_cost = matrix[path[-1]][start]

cost += closing\_cost

path.append(start)

print(f"Замыкание цикла: стоимость {cost}")

return path, cost

return None, float('inf')

def main():

fileName = "matrix.txt"

startVertex = 0

while True:

print("\nМеню:")

print("1 - Сгенерировать несимметричную матрицу")

print("2 - Сгенерировать симметричную матрицу")

print("3 - Выбрать начальную вершину")

print("4 - Загрузить матрицу и запустить алгоритмы")

print("5 - Выход")

option = input("Выберите опцию: ").strip()

if option == "1":

numberOfVertices = int(input("Количество вершин: "))

matrix = generateWeightMatrix(numberOfVertices, False)

saveWeightMatrix(fileName, matrix)

print(f"Матрица сохранена в {fileName}")

elif option == "2":

numberOfVertices = int(input("Количество вершин: "))

matrix = generateWeightMatrix(numberOfVertices, True)

saveWeightMatrix(fileName, matrix)

print(f"Матрица сохранена в {fileName}")

elif option == "3":

startVertex = int(input("Начальная вершина: "))

print(f"Установлена вершина {startVertex}")

elif option == "4":

try:

matrix = loadWightMatrix(fileName)

n = len(matrix)

print(f"Загружена матрица {n}x{n}")

break

except FileNotFoundError:

print(f"Файл {fileName} не найден")

elif option == "5":

return

else:

print("Неверная опция")

bbPath, bbCost = branchAndBoundV4(matrix, startVertex)

niPath, niCost = improvedNearestInsertion(matrix, startVertex)

print("\nРезультаты:")

if bbPath is not None and bbCost != float('inf'):

print(f"Метод ветвей и границ: путь {bbPath}, стоимость {bbCost}")

else:

print("МВиГ: путь не найден")

if niPath is not None and niCost != float('inf'):

print(f"Улучшенный АВБГ: путь {niPath}, стоимость {niCost}")

if bbPath is not None:

difference = niCost - bbCost

if difference > 0:

print(f"АВБГ хуже на {difference:.2f}")

elif difference < 0:

print(f"АВБГ лучше на {abs(difference):.2f}")

else:

print(f"Результаты одинаковы")

else:

print("АВБГ: путь не найден")

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

main()