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: $O(2^n)$

Сложность по времени: $O(2^n)$

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: $O(n^2)$

	МВиГ	АВБГ
Сложность по времени	$O(2^n)$	$O(n^2)$
Сложность по памяти	$O(2^n)$	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: $O(n^2)$

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

   Параметры:

   fileName - имя файла (str)

   matrix - матрица весов (list)

   Сложность по времени: $O(n^2)$

3. loadWightMatrix(fileName: str) - загружает матрицу из файла
   Параметры: fileName - имя файла (str)
   Возвращает: matrix - матрица весов (list)
   Сложность по времени: $O(n^2)$

4. 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: $O(V^2)$ где V - количество вершин

5. 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: $O(n^3)$

6. lowerBound2(matrix: list, path: list, visited: set) -> float - вторая нижняя оценка (вес МОД)
   Параметры: аналогично lowerBound1
   Возвращает: нижняя оценка (float)
   Сложность по времени: $O(n^2)$

7. 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)

8. 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: $O(2^n)$ в худшем случае

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

   Параметры: аналогично branchAndBoundV4

   Возвращает: path, cost - найденный путь и стоимость

   Сложность по времени: $O(n^3)$

Функция	Сложность по времени
generateWeightMatrix	$O(n^2)$
saveWeightMatrix	$O(n^2)$
loadWightMatrix	$O(n^2)$
primMST	$O(V^2)$
lowerBound1	$O(n^3)$
lowerBound2	$O(n^2)$

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	Метод ветвей и границ: Путь = [0, 1, 4, 2, 5, 3, 0], Стоимость = 14

	5 5 4 0 8 6 7 1 4 7 0 3 6 2 1 2 6 0	
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()
```