САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Кратчайшие пути в графе: коммивояжёр. Вариант 3.

Студент гр. 3343 _____ Иванов П. Д.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Разработать и реализовать два алгоритма для решения задачи коммивояжёра: точный метод ветвления с отсечением (МВиГ) и приближённый метод модификации решения (АМР), с использованием эвристик для ускорения поиска.

Задание

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 — поиск в глубину с учётом веса добавляемой дуги и нижней оценки веса остатка пути. Приближённый алгоритм: АМР. Замечание к варианту 3 Начинать МВиГ со стартовой вершины.

АМР(алгоритм модификации решения) описывается так:

● Получаем некоторое решение. При выполнении ЛР — берём решение 1-2-3-...-N.

● Запускаем цикл модификации решения. Под единичной модификацией будем понимать взятие некоторого города и перемещение его в другое место в цепочке.

Вид цикла:

i=0; //счётчик выполненных модификаций

m=true; //флаг обнаружения успешной модификации

Пока (m==true && i<F){ //не более F модификаций, взять F=N

m=false;

По множеству вариантов модификации* {

Если вес модифицированного решения получился меньше рекорда{

m=true; i++;

Сохранить модификацию в качестве текущего решения;

break;

}

}

}

Разработать и реализовать приближённый алгоритм в соответствии с вышеприведённой схемой, с учётом следующих ограничений:

- Алгоритм должен иметь полиномиальную сложность;

- Поиск модификации должен вестись не случайным подбором: при выборе перемещаемого города и/или при выборе места назначения должна

использоваться какая-то простая эвристика для повышения шанса побыстрее найти успешную модификацию.

- Множество вариантов модификации не обязательно должно включать в себя ВСЕ возможные модификации, оно может быть ограничено эвристиками.

Должна быть возможность генерировать матрицу весов (произвольную или симметричную), сохранять её в файл и использовать в качестве входных данных.

Выполнение работы

В работе реализованы два подхода к решению задачи коммивояжёра:

1. МВиГ – ветвление с отсечением: Алгоритм начинается с фиксированной стартовой вершины и последовательно строит частичные маршруты. На каждом шаге для каждого кандидата вычисляется нижняя оценка оставшейся части пути с использованием двух методов:

   1 - Оценки на основе полусуммы двух минимальных допустимых дуг для каждого «куска».

   2 - Оценки на основе минимального остовного дерева (МОД) для набора «кусков» (текущая цепочка и оставшиеся вершины).

   Из двух оценок выбирается максимум, который используется как гарантированно нижняя граница. Если сумма накопленной стоимости, стоимости выбранной дуги и нижней оценки превышает текущую лучшую стоимость найденного маршрута, ветка отсекается. При этом кандидаты сортируются по сумме (стоимость дуги + нижняя оценка), что позволяет эвристически выбирать более перспективные варианты.

2. Приближённый метод (АМР – алгоритм модификации решения): Исходное решение формируется как простой обход вершин (начало → 1 → 2 → … → n-1 → начало). Затем алгоритм пытается улучшить решение посредством локальных модификаций: для каждого города (исключая стартовую вершину) вычисляется его вклад в стоимость тура, после чего город переустанавливается в другое место, если перемещение приводит к уменьшению общей стоимости. Выбор модификации осуществляется по эвристике, оценивающей потенциальное снижение стоимости, что позволяет быстрее находить улучшения.

Были написаны следующие функции:

1. Функции работы с матрицей весов

- *generate_matrix(n, symmetric=True, max_weight=100)* - Генерирует матрицу смежности для полного графа с *n* вершинами.

  Логика*:*

  - Для каждой пары вершин (i, j) генерируется случайное значение; если i = j, вес равен 0.
  - Если параметр *symmetric* установлен в True, для всех пар i < j значение присваивается и симметричной позиции [j][i].
- *save_matrix(matrix, filename)* - Сохраняет матрицу в текстовый файл.

  Логика*:*

  - Первая строка файла содержит число вершин.
  - Каждая последующая строка содержит веса строки матрицы, разделённые пробелами.
- *load_matrix(filename)* - Загружает матрицу весов из файла.

  Логика*:*

  - Считывается первая строка (число вершин).
  - Далее построчно считываются значения и формируется матрица.

2. Функции для вычисления нижних оценок (эвристики для метода ветвления)

- *get_pieces(chain, remaining)* - Формирует список «кусков», на основе которых рассчитываются нижние оценки.

  Логика*:*

  - Первый «кусок» – это текущая цепочка маршрута, представляемая парой (первая вершина, последняя вершина).
  - Каждый оставшийся город считается отдельным куском (пара, где начало и конец совпадают).
- *lower_bound_half_sum(matrix, pieces)* - Вычисляет нижнюю оценку остатка пути на основе полусуммы минимальных допустимых дуг для каждого куска.

Логика*:*

- Для каждого куска определяется минимальный вес дуги, исходящей из его конечной вершины (min_out) и минимальный вес дуги, входящей в его начальную вершину (min_in).
- По каждому куску сумма (min_out + min_in) накапливается, затем итоговая сумма делится на 2.

- *lower_bound_MST(matrix, pieces)* - Вычисляет нижнюю оценку остатка пути как вес минимального остовного дерева (МОД), построенного для «кусков».

  Логика*:*

  - На вход подаётся список кусков, и для каждой пары кусков вес ребра определяется как минимум из двух возможных дуг (из конца одного куска в начало другого и наоборот).
  - Используется алгоритм Прима для построения МОД с суммированием весов выбранных ребер.
- *compute_lower_bound(matrix, chain, remaining)* - Объединяет обе оценки нижней стоимости остатка пути.

  Логика*:*

  - Если remaining пуст, возвращается 0 (так как дополнительных затрат завершения пути нет).
  - Иначе формируется список кусков с помощью **get_pieces**, затем рассчитываются обе оценки: lb1 (half_sum) и lb2 (MST).
  - Возвращается максимум из lb1 и lb2, чтобы оценка не была заниженной.

3. Реализация МВиГ

- *tsp_branch_and_bound(matrix, start=0)* - Находит оптимальное решение задачи коммивояжёра с помощью рекурсивного перебора (ветвления) с отсечением, основанного на нижней оценке остатка пути.

  Логика*:*

- Алгоритм стартует с начальной вершины; оставшиеся вершины формируются в список.
- Рекурсивная функция *search(chain, current_cost, remaining):*
  - Базовый случай: Если длина цепочки равна числу вершин, добавляется дуга возврата к стартовой, и, если полученный тур лучше текущего, он сохраняется.
  - Формирование кандидатов: Для каждой вершины из remaining вычисляется:
    - Стоимость дуги от последней вершины текущей цепочки к выбранной вершине.
    - Новая цепочка (с добавлением вершины) и обновлённый список remaining.
    - Нижняя оценка остатка пути, вычисляемая через compute_lower_bound.
  - Кандидаты сортируются по сумме (стоимость дуги + нижняя оценка), что обеспечивает эвристический выбор — сначала рассматриваются более перспективные ветви.
  - Если сумма (текущая стоимость + стоимость дуги + нижняя оценка) превышает уже найденное лучшее решение, ветка отсеивается.
- Рекурсия продолжается до перебора всех перспективных вариантов.

4. Реализация приближённого алгоритма

- *tour_cost(matrix, tour)* - Вычисляет суммарную стоимость заданного тура.
  Логика*:*
  - Проходит по последовательности вершин и суммирует веса дуг между последовательными вершинами.
- *tsp_approx(matrix, start=0)* - Находит приближённое решение задачи коммивояжёра методом локальных модификаций.
  Логика*:*

- Начальное решение: Формируется тур, где стартовая вершина идёт первой, далее следуют все остальные вершины (в порядке их номеров), и в конце снова стартовая вершина.
- Эвристика модификаций:
  - Для каждого города (исключая старт) рассчитывается «удалительный вклад», показывающий, насколько текущая позиция увеличивает стоимость тура, по формуле: removal_cost = matrix[tour[i-1]][v] + matrix[v][tour[i+1]] – matrix[tour[i-1]][tour[i+1]]
  - Если вклад положительный, рассматривается перемещение города в другие позиции.
  - Перебираются все возможные позиции для вставки, вычисляется новая стоимость тура и определяется величина улучшения.
  - Из всех вариантов выбирается тот, который даёт максимальное снижение стоимости. Если найдено улучшение, тур обновляется.
  - Процесс повторяется, пока находятся улучшения или не достигнут лимит модификаций (F, обычно равное n).

Оценка сложности алгоритмов

- Метод ветвей и границ:
  - Время**:** В худшем случае O((n-1)!), так как алгоритм потенциально перебирает все маршруты.
  - Память**:** O(n) – стек рекурсии и хранение текущего маршрута.
- Приближённый метод:
  - Время**:** В худшем случае $O(n^3)$ – при каждой из F модификаций перебирается $O(n^2)$ вариантов.
  - Память**:** O(n)– хранение текущего маршрута.

Тестирование

Результаты тестирования представлены в таблице 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	--n 5 --method vig	Матрица весов: [0, 56, 12, 62, 77] [70, 0, 77, 94, 26] [52, 36, 0, 67, 20] [17, 59, 55, 0, 27] [79, 59, 72, 96, 0] Решение МВиГ (ветвление + отсечение): Путь: [0, 2, 1, 4, 3, 0] Стоимость: 187	Работа флагов указания размеров матрицы и указания метода решения
2.	--matrix_file "last_matrix" --method amr	Матрица весов: [0, 56, 12, 62, 77] [70, 0, 77, 94, 26] [52, 36, 0, 67, 20] [17, 59, 55, 0, 27] [79, 59, 72, 96, 0] Решение АМР(приближённый метод): Путь: [0, 2, 1, 4, 3, 0] Стоимость: 187	Работа флагов загрузки матрицы из файла и проверка работы приближенного метода
3.	--n 10 --symmetric --method vig	Матрица весов: [0, 42, 66, 16, 47, 4, 28, 82, 4, 98] [42, 0, 8, 73, 72, 37, 19, 5, 22, 98] [66, 8, 0, 72, 60, 37, 34, 95, 22, 32] [16, 73, 72, 0, 44, 57, 13, 51, 96, 69] [47, 72, 60, 44, 0, 68, 26, 83, 6, 87] [4, 37, 37, 57, 68, 0, 36, 62, 73, 97] [28, 19, 34, 13, 26, 36, 0, 83, 92, 10] [82, 5, 95, 51, 83, 62, 83, 0, 29, 83] [4, 22, 22, 96, 6, 73, 92, 29, 0, 10] [98, 98, 32, 69, 87, 97, 10, 83, 10, 0] Решение МВиГ (ветвление + отсечение): Путь: [0, 5, 7, 1, 2, 9, 8, 4, 6, 3, 0] Стоимость: 182	Возможность задать симметричную матрицу

4.	--n 7 --symmetric --method amr --debug	Матрица весов: [0, 96, 4, 29, 97, 95, 50] [96, 0, 39, 84, 1, 60, 2] [4, 39, 0, 26, 56, 63, 76] [29, 84, 26, 0, 43, 96, 69] [97, 1, 56, 43, 0, 69, 92] [95, 60, 63, 96, 69, 0, 26] [50, 2, 76, 69, 92, 26, 0] Первая модификация: [0, 1, 2, 3, 4, 5, 6, 0] с ценой 349 Найдена улучшенная модификация: [0, 2, 1, 3, 4, 5, 6, 0] с ценой 315 Найдена улучшенная модификация: [0, 2, 3, 1, 4, 5, 6, 0] с ценой 260 Найдена улучшенная модификация: [0, 3, 2, 1, 4, 5, 6, 0] с ценой 240 Найдена улучшенная модификация: [0, 2, 1, 4, 5, 6, 3, 0] с ценой 237 Решение AMP (приближённый метод): Путь: [0, 2, 1, 4, 5, 6, 3, 0] Стоимость: 237	Работа флага режима отладки

Табл. 1. – Результаты тестирования

Выводы

В результате проделанной работы были разработаны и реализованы два алгоритма для решения задачи коммивояжёра: точный метод ветвления с отсечением (МВиГ) и приближённый метод модификации решения (АМР). Точный метод, основанный на рекурсивном переборе с использованием нижних оценок остатка пути, гарантирует нахождение оптимального решения, однако его экспоненциальная сложность делает его непрактичным для больших экземпляров задачи. При этом приближённый метод, использующий эвристику локальных модификаций для улучшения начального решения, демонстрирует полиномиальную сложность и позволяет быстро получать хорошие приближения оптимального маршрута, хотя и не всегда гарантирует точное решение. Таким образом, выбор метода зависит от конкретных требований к точности и объёму обрабатываемых данных: для небольших задач можно использовать точный метод, а для более крупных – приближённый алгоритм, осознавая, что он может давать решения, отличающиеся от оптимальных.

ПРИЛОЖЕНИЕ А

```
import math
import random
import argparse
from copy import deepcopy


DEBUG = False


def debug_print(*args, **kwargs):
    if DEBUG:
        print(*args, **kwargs)


def generate_matrix(n, symmetric=True, max_weight=100):
    """
    Генерирует матрицу весов для полного графа с n вершинами.
    Если symmetric=True, матрица делается симметричной.
    """
    matrix = [[0 if i == j else random.randint(1, max_weight) for j in
range(n)] for i in range(n)]
    if symmetric:
        for i in range(n):
            for j in range(i + 1, n):
                matrix[j][i] = matrix[i][j]
    return matrix


def save_matrix(matrix, filename):
    """
    Сохраняет матрицу в файл.
    """
    with open(filename, 'w') as f:
        n = len(matrix)
        f.write(str(n) + "\n")
        for row in matrix:
            f.write(" ".join(map(str, row)) + "\n")


def load_matrix(filename):
    """
```

```
        Загружает матрицу из файла.
        """
        with open(filename, 'r') as f:
            n = int(f.readline())
            matrix = []
            for _ in range(n):
                row = list(map(int, f.readline().split()))
                matrix.append(row)
        return matrix


    def get_pieces(chain, remaining):
        """
        Формирует список кусков.
        Первый кусок – текущая цепочка (представлена парой: (начало, конец)).
        Остальные куски – одиночные вершины из remaining, где начало = конец.
        """
        pieces = [(chain[0], chain[-1])]
        for v in remaining:
            pieces.append((v, v))
        return pieces


    def lower_bound_half_sum(matrix, pieces):
        """
        Вычисляет нижнюю оценку на основе полусуммы двух легчайших допустимых
дуг для каждого куска.
        """
        total = 0
        for i, (s_i, e_i) in enumerate(pieces):
            min_out = math.inf
            min_in = math.inf
            for j, (s_j, e_j) in enumerate(pieces):
                if i == j:
                    continue
                weight_out = matrix[e_i][s_j]
                if weight_out < min_out:
                    min_out = weight_out
                weight_in = matrix[e_j][s_i]
                if weight_in < min_in:
                    min_in = weight_in
            total += (min_out + min_in)
        return total / 2
```

```
    def lower_bound_MST(matrix, pieces):
        """
        Вычисляет нижнюю оценку на основе веса минимального остовного дерева
(МОД).
        """
        n = len(pieces)
        if n == 0:
            return 0
        in_mst = [False] * n
        key = [math.inf] * n
        key[0] = 0
        total_weight = 0
        for _ in range(n):
            u = None
            min_val = math.inf
            for i in range(n):
                if not in_mst[i] and key[i] < min_val:
                    min_val = key[i]
                    u = i
            if u is None:
                break
            in_mst[u] = True
            total_weight += key[u]
            for v in range(n):
                if not in_mst[v]:
                    w1 = matrix[pieces[u][1]][pieces[v][0]]
                    w2 = matrix[pieces[v][1]][pieces[u][0]]
                    w = min(w1, w2)
                    if w < key[v]:
                        key[v] = w
        return total_weight


    def compute_lower_bound(matrix, chain, remaining):
        """
        Вычисляет нижнюю оценку остатка пути.
        Если remaining пуст, возвращает 0.
        """
        if not remaining:
            return 0
        pieces = get_pieces(chain, remaining)
```

```
        lb1 = lower_bound_half_sum(matrix, pieces)
        lb2 = lower_bound_MST(matrix, pieces)
        return max(lb1, lb2)


    def tsp_branch_and_bound(matrix, start=0):
        """
        Решение задачи коммивояжёра методом МВиГ (ветвление с отсечением).
        """
        n = len(matrix)
        best = {'cost': math.inf, 'path': None}

        def search(chain, current_cost, remaining):
            nonlocal best
            if len(chain) == n:
                tour_cost = current_cost + matrix[chain[-1]][start]
                if tour_cost < best['cost']:
                    best['cost'] = tour_cost
                    best['path'] = chain + [start]
                    debug_print(f"Найден новый тур: {best['path']} с ценой
{best['cost']}")
                return

            candidates = []
            for v in remaining:
                edge_cost = matrix[chain[-1]][v]
                new_chain = chain + [v]
                new_remaining = remaining.copy()
                new_remaining.remove(v)
                lb = compute_lower_bound(matrix, new_chain, new_remaining)
                candidates.append((v, edge_cost, lb))
                debug_print(f"Кандидат: добавляем {v}, edge_cost={edge_cost},
lb={lb}, chain={chain}")

            candidates.sort(key=lambda x: x[1] + x[2])
            for v, edge_cost, lb in candidates:
                total_estimate = current_cost + edge_cost + lb
                debug_print(
                    f"Проверка: текущая стоимость={current_cost},
edge_cost={edge_cost}, lb={lb}, total_estimate={total_estimate},
best={best['cost']}")
                if total_estimate > best['cost']:
                    debug_print("Отсекаем ветку")
```

```
                continue
            new_chain = chain + [v]
            new_remaining = remaining.copy()
            new_remaining.remove(v)
            search(new_chain, current_cost + edge_cost, new_remaining)

    remaining = [i for i in range(n) if i != start]
    search([start], 0, remaining)
    return best['path'], best['cost']


def tour_cost(matrix, tour):
    """
    Вычисляет стоимость данного тура.
    """
    cost = 0
    for i in range(len(tour) - 1):
        cost += matrix[tour[i]][tour[i + 1]]
    return cost


def tsp_approx(matrix, start=0):
    """
    Приближённый алгоритм (АМР).
    """
    n = len(matrix)
    tour = [start] + [i for i in range(n) if i != start] + [start]
    best_cost = tour_cost(matrix, tour)
    debug_print(f"Первая модификация: {tour} с ценой {best_cost}")
    F = n
    modifications = 0
    improved = True
    while improved and modifications < F:
        improved = False
        for idx in range(1, n):
            for j in range(1, n):
                if j == idx:
                    continue
                new_tour = tour[:]
                city = new_tour.pop(idx)
                new_tour.insert(j, city)
                new_cost = tour_cost(matrix, new_tour)
                if new_cost < best_cost:
```

```
                            tour = new_tour
                            best_cost = new_cost
                            improved = True
                            modifications += 1
                            debug_print(f"Найдена улучшенная модификация: {tour} с
ценой {best_cost}")
                            break
                    if improved:
                        break
            return tour, best_cost


        def main():
            global DEBUG
            parser    =    argparse.ArgumentParser(description="Решение    задачи
коммивояжёра методами МВиГ и АМР")
            parser.add_argument("--n",   type=int,   default=5,   help="Количество
вершин")
            parser.add_argument("--symmetric",                action="store_true",
help="Симметричная матрица")
            parser.add_argument("--matrix_file", type=str, help="Файл с матрицей
весов")
            parser.add_argument("--method",   type=str,   choices=["vig",   "amr"],
default="vig", help="Метод решения: vig или amr")
            parser.add_argument("--debug",   action="store_true",   help="Включить
режим отладки")
            args = parser.parse_args()

            DEBUG = args.debug

            if args.matrix_file:
                matrix = load_matrix(args.matrix_file)
            else:
                matrix = generate_matrix(args.n, symmetric=args.symmetric)
                save_matrix(matrix, "last_matrix")

            print("Матрица весов:")
            for row in matrix:
                print(row)

            start = 0
            if args.method == "vig":
                path, cost = tsp_branch_and_bound(matrix, start)
```

```
        print("\nРешение МВиГ (ветвление + отсечение):")
    else:
        path, cost = tsp_approx(matrix, start)
        print("\nРешение АМР (приближённый метод):")
    print("Путь:", path)
    print("Стоимость:", cost)


if __name__ == "__main__":
    main()
```