**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. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

## Цель работы.

Целью данной работы является изучение задачи о комммивояжёре, различных методов её решения, а также написание программы, реализующей точный метод решения на основе метода ветвей и границ и приближенный способ.

## Задание.

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

**Входные данные:**

* n - количество городов (5 ≤ n ≤ 15).
* Матрица расстояний между городами размером n×n , где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0 (и i ≠ j ), это означает, что прямого пути между городами нет.

**Выходные данные:**

* Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.
* Оптимальный путь в виде последовательности посещаемых городов, начинающегося и заканчивающегося в начальном городе.
* Если такого пути не существует, вывести "no path".

**Sample Input 1:**

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

**Sample Output 1:**

78

0 4 2 3 1 0

**Sample Input 2:**

3

0 1 0

1 0 1

0 1 0

**Sample Output 2:**

no path

## Индивидуализация.

8. Точный метод: динамическое программирование (не МВиГ), итеративная реализация.

Приближённый алгоритм: АЛШ-2.

Требование перед сдачей: прохождение кода в задании 3.1 на Stepik.

Замечание к варианту 8. АЛШ-2 начинать со стартовой вершины.

## Описание точного алгоритма.

**Идея алгоритма:**  
Используем динамическое программирование для поиска кратчайшего Гамильтонова цикла.

* dp[mask][i] — минимальная стоимость пути, проходящего через вершины маски mask и заканчивающегося в вершине i.
* parent[mask][i] — вершина, из которой пришли в i для маски mask (используется для восстановления пути).

**Параметры:**

* n — количество городов.
* graph — матрица смежности, где graph[i][j] — вес ребра между городами i и j.

**Инициализация:**

* size = 1 << n — количество возможных масок (2^n).
* dp[1][0] = 0 — база ДП: начинаем с города 0 (маска содержит только 0-й бит).

**Переходы между состояниями:**  
Для каждой маски mask и вершины i:

1. Если dp[mask][i] не вычислено (INF), переходим к следующей итерации.
2. Для всех вершин j, не входящих в маску mask:
   * Если существует ребро i → j (graph[i][j] > 0), вычисляем новую маску new\_mask = mask | (1 << j).
   * Обновляем dp[new\_mask][j] и parent[new\_mask][j], если найден более короткий путь.

**Восстановление пути:**

1. После заполнения таблицы dp проверяем возможность возврата в стартовый город (0) из всех вершин j.
2. Выбираем вершину best\_j, дающую минимальную стоимость полного цикла.
3. Восстанавливаем путь, двигаясь от best\_j к стартовой вершине с помощью массива parent.

**Сложность:**

* Время: O(2^n \* n^2) — три вложенных цикла (по маскам, вершинам i, вершинам j).
* Память: O(2^n \* n) — хранение массивов dp и parent.

**Описание приближенного алгоритма (метод ближайшего соседа + 2-opt)**

**Идея алгоритма:**

**Алгоритм ближайшего соседа:**

* + На каждом шаге выбирается город с минимальным расстоянием от текущего.
  + Построенный путь улучшается с помощью локальной оптимизации 2-opt.

**Шаги алгоритма:**

**Построение начального пути:**

* Стартуем из заданного города (start\_city).
* На каждом шаге:
  + Выбираем непосещенный город j с минимальным весом ребра current → j.
  + Добавляем j в путь.
* Проверяем возможность возврата в стартовый город.

**2. Оптимизация 2-opt:**

* Итеративно улучшаем путь, переставляя пары вершин:
  + Для всех пар (i, j) в пути проверяем, уменьшит ли стоимость перестановка подпути между i и j.
  + Если да, переворачиваем подпуть и обновляем стоимость.

**Условия завершения:**

* Если все города посещены и существует обратное ребро в стартовый город — путь найден.
* Если на любом этапе нет допустимых переходов — маршрут невозможен.

**Особенности реализации:**

* Фиксированный стартовый город (0 по умолчанию).
* Проверка на существование исходящих рёбер из стартового города.

**Сложность:**

* **Ближайший сосед:** O(n^2) — для каждого города перебираем всех соседей.
* **2-opt:** O(k \* n^2), где k — количество итераций улучшения.

## Описание функций и структур данных.

**Точное решение (метод динамического программирования):**

Функция tsp - Реализует точное решение задачи коммивояжёра (TSP) методом динамического программирования с оптимизацией битовых масок.

Параметры: graph - матрица смежности графа (n×n), n - количество вершин, debug - флаг вывода отладочной информации (по умолчанию False)

Возвращает: Кортеж (min\_cost, path) где: min\_cost - минимальная стоимость цикла (INF при отсутствии решения), path - список вершин пути (None при отсутствии решения)

**Приближенное решение (метод ближайшего соседа + 2-opt):**

Функция approx - Реализует приближённое решение TSP комбинацией алгоритма ближайшего соседа и оптимизации 2-opt

Параметры: graph - матрица смежности графа (n×n), start\_city - стартовая вершина (по умолчанию 0), debug - флаг вывода отладочной информации (по умолчанию False)

Возвращает: Кортеж (path, cost) где: path - список вершин пути ("no path" при отсутствии решения), cost - суммарная стоимость маршрута (0 при отсутствии решения)

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

## Выводы.

В результате выполнения данной лабораторной работы была изучена задача о коммивояжёре и различные методы её решения. На языке С++ написаны алгоритмы, реализующие точное решение методом динамического программирования (итеративная реализация) и приближенное решение методом АЛШ-2.

# Приложение А Исходный код программы

Название файла: *main.py*

# Импорт модуля с реализацией алгоритмов

import src.includes as inc

# Основной цикл программы для взаимодействия с пользователем

while True:

    # Запрос типа решения у пользователя

    print("Input type of solution (tsp/approx/exit)")

    user\_input = input().strip().lower()  # Приводим к нижнему регистру и убираем пробелы

    # Выход из программы

    if user\_input == "exit":

        break  # Прерываем цикл while

    # Запрос режима отладки

    print("Use debug? (y/n)")

    debug\_input = input().strip().lower()

    debug = debug\_input != "n"  # debug=True для любого ввода, кроме 'n'

    # Обработка точного решения (метод динамического программирования)

    if user\_input == "tsp":

        print("Set size")

        try:

            n = int(input())  # Получаем размер матрицы

            if n <= 0:

                print("Size must be positive integer")

                continue

            # Ввод матрицы смежности

            print(f"Input {n}x{n} matrix (row by row, values separated by spaces):")

            graph = []

            valid\_matrix = True

            for \_ in range(n):

                row = list(map(int, input().split()))

                if len(row) != n:

                    print(f"Error: expected {n} values, got {len(row)}")

                    valid\_matrix = False

                    break

                graph.append(row)

            if not valid\_matrix:

                continue  # Возвращаемся к началу цикла при ошибке ввода

            # Вызов точного алгоритма

            cost, path = inc.tsp(graph, n, debug)

            # Вывод результатов

            if cost == inc.INF:

                print("No valid path exists")

            else:

                print(f"Minimum cost: {cost}")

                print("Optimal path:")

                print(" → ".join(map(str, path)))

        except ValueError:

            print("Invalid input - please enter integers only")

            continue

    # Обработка приближенного решения (метод ближайшего соседа + 2-opt)

    elif user\_input == "approx":

        print("Set size")

        try:

            n = int(input())

            if n <= 0:

                print("Size must be positive integer")

                continue

            # Запрос стартового города

            print("Set start city (default=0):")

            start\_input = input().strip()

            start = 0 if start\_input == "" else int(start\_input)

            # Проверка корректности стартового города

            if start < 0 or start >= n:

                print(f"Start city must be in range 0-{n-1}")

                continue

            # Ввод матрицы смежности

            print(f"Input {n}x{n} matrix (row by row, values separated by spaces):")

            graph = []

            valid\_matrix = True

            for \_ in range(n):

                row = list(map(int, input().split()))

                if len(row) != n:

                    print(f"Error: expected {n} values, got {len(row)}")

                    valid\_matrix = False

                    break

                graph.append(row)

            if not valid\_matrix:

                continue

            # Вызов приближенного алгоритма

            path, cost = inc.approx(graph, start, debug)

            # Вывод результатов

            if path == "no path":

                print("No valid path exists")

            else:

                print(f"Approximate cost: {cost}")

                print("Constructed path:")

                print(" → ".join(map(str, path)))

        except ValueError:

            print("Invalid input - please enter integers only")

            continue

    # Обработка некорректного ввода

    else:

        print("Incorrect input, please try again")

        print("Available options: tsp, approx, exit")

Название файла: includes.py

# Глобальная константа для представления бесконечности

global INF

INF = float('inf')

def tsp(graph, n, debug=False):

"""

Точное решение задачи коммивояжера методом динамического программирования

с использованием битовых масок.

Параметры:

graph - матрица смежности графа (n x n)

n - количество городов

debug - флаг отладочного вывода (по умолчанию False)

Возвращает:

tuple: (минимальная стоимость, список городов пути) или (INF, None) если путь не найден

"""

# Инициализация таблицы динамического программирования

size = 1 << n # Всего возможных масок состояний (2^n)

dp = [[INF] \* n for \_ in range(size)] # Таблица минимальных стоимостей

parent = [[-1] \* n for \_ in range(size)] # Таблица для восстановления пути

# Базовый случай: начало из города 0 (маска 000..001)

dp[1][0] = 0 # Стоимость пути из 0 в 0 равна 0

if debug:

print("\n--- Начало алгоритма точного решения (метод динамического программирования) ---")

print(f"Инициализирована таблица DP размером {size} x {n}")

print(f"Начальное состояние: dp[1][0] = 0 (маска 000..001, город 0)")

# Основной цикл динамического программирования

for mask in range(size):

if debug and mask % 100 == 0: # Периодический вывод для больших n

print(f"\nОбрабатываю маску: {bin(mask)}")

for i in range(n):

if dp[mask][i] == INF: # Пропускаем непосещенные комбинации

continue

if debug:

print(f" Город {i}: текущая стоимость = {dp[mask][i]}")

# Перебираем все возможные переходы

for j in range(n):

# Условия перехода:

# - города разные

# - город j еще не посещен в текущей маске

# - существует ребро i->j

if i != j and (mask & (1 << j)) == 0 and graph[i][j] > 0:

new\_mask = mask | (1 << j) # Новая маска с посещенным городом j

new\_cost = dp[mask][i] + graph[i][j] # Новая стоимость

# Если нашли более оптимальный путь

if new\_cost < dp[new\_mask][j]:

if debug:

print(f" Обновляю dp[{bin(new\_mask)}][{j}]: {dp[new\_mask][j]} -> {new\_cost} (из города {i})")

dp[new\_mask][j] = new\_cost

parent[new\_mask][j] = i # Запоминаем предка для восстановления пути

# Поиск минимального гамильтонова цикла

full\_mask = (1 << n) - 1 # Маска всех посещенных городов (111..111)

min\_cost = INF

best\_j = -1 # Будет хранить последний город перед возвратом в 0

if debug:

print("\n--- Поиск оптимального маршрута ---")

print(f"Полная маска всех городов: {bin(full\_mask)}")

# Проверяем все возможные завершения цикла

for j in range(n):

if j != 0 and graph[j][0] > 0: # Должен существовать путь из j в 0

total = dp[full\_mask][j] + graph[j][0] # Полная стоимость цикла

if debug:

print(f" Город {j}: стоимость полного цикла = {total} (dp[{bin(full\_mask)}][{j}] = {dp[full\_mask][j]} + graph[{j}][0] = {graph[j][0]})")

if total < min\_cost:

min\_cost = total

best\_j = j

# Восстановление пути

if min\_cost == INF:

if debug:

print("Не найден допустимый маршрут")

return INF, None

else:

path = []

current = best\_j

current\_mask = full\_mask

if debug:

print(f"\nВосстановление пути (начало с города {current}, маска {bin(current\_mask)})")

# Обратный проход от конечного города к начальному

while current != 0:

path.append(current)

prev = parent[current\_mask][current]

if debug:

print(f" Текущий город: {current}, предыдущий: {prev}, маска: {bin(current\_mask)}")

current\_mask ^= (1 << current) # Удаляем текущий город из маски

current = prev

path.append(0) # Добавляем начальный город в конец

path = path[::-1] # Разворачиваем путь (чтобы шел от 0)

path.append(0) # Добавляем возврат в начальный город для завершения цикла

if debug:

print(f"Финальный путь: {path}, стоимость: {min\_cost}")

return min\_cost, path

def approx(graph, start\_city=0, debug=False):

"""

Приближенное решение задачи коммивояжера методом ближайшего соседа

с последующей оптимизацией 2-opt.

Параметры:

graph - матрица смежности графа (n x n)

start\_city - начальный город (по умолчанию 0)

debug - флаг отладочного вывода (по умолчанию False)

Возвращает:

tuple: (список городов пути, стоимость) или ("no path", 0) если путь не найден

"""

n = len(graph)

if n == 0:

if debug:

print("Граф пуст")

return "no path", 0

# Проверка на существование исходящих путей из стартового города

if all(graph[start\_city][j] <= 0 for j in range(n) if j != start\_city):

if debug:

print(f"Нет исходящих путей из стартового города {start\_city}")

return "no path", 0

if debug:

print("\n--- Начало приближенного алгоритма (метод ближайшего соседа с оптимизацией 2-opt) ---")

print(f"Стартовый город: {start\_city}")

# Алгоритм ближайшего соседа с фиксированным стартом

def nearest\_neighbor(start):

"""Построение начального маршрута методом ближайшего соседа"""

path = [start]

visited = set([start])

total\_cost = 0

current = start

for step in range(n - 1): # Нужно посетить n-1 оставшихся городов

if debug:

print(f"\nШаг {step+1}: текущий город {current}, посещенные города: {visited}")

next\_node = -1

min\_dist = float('inf')

# Поиск ближайшего непосещенного города

for j in range(n):

if j not in visited and graph[current][j] > 0 and graph[current][j] < min\_dist:

min\_dist = graph[current][j]

next\_node = j

if next\_node == -1: # Если не нашли подходящий город

if debug:

print("Не найден следующий город - маршрут прерван")

return None, float('inf')

path.append(next\_node)

visited.add(next\_node)

total\_cost += min\_dist

if debug:

print(f" Выбран город {next\_node} (расстояние {min\_dist}), общая стоимость: {total\_cost}")

current = next\_node

# Попытка вернуться в стартовый город

if graph[current][start] > 0:

total\_cost += graph[current][start]

path.append(start)

if debug:

print(f"\nВозврат в стартовый город {start} (расстояние {graph[current][start]})")

print(f"Исходный маршрут: {path}, общая стоимость: {total\_cost}")

return path, total\_cost

else:

if debug:

print("\nНевозможно вернуться в стартовый город")

return None, float('inf')

# Построение начального маршрута

path, cost = nearest\_neighbor(start\_city)

if path is None:

if debug:

print("Не удалось построить маршрут методом ближайшего соседа")

return "no path", 0

# Оптимизация 2-opt (без изменения стартового города)

def two\_opt(path, cost):

"""Оптимизация маршрута методом 2-opt"""

if debug:

print("\n--- Начало оптимизации 2-opt ---")

print(f"Начальный маршрут: {path}, стоимость: {cost}")

improved = True

iteration = 0

while improved:

iteration += 1

improved = False

# Перебираем все возможные пары ребер для перестановки

# Не изменяем стартовый город (первый и последний элементы)

for i in range(1, len(path) - 2):

for j in range(i + 1, len(path) - 1):

# Текущие ребра: a-b и c-d

a, b, c, d = path[i-1], path[i], path[j], path[j+1]

# Если перестановка дает улучшение

if graph[a][b] + graph[c][d] > graph[a][c] + graph[b][d]:

old\_cost\_segment = graph[a][b] + graph[c][d]

new\_cost\_segment = graph[a][c] + graph[b][d]

improvement = old\_cost\_segment - new\_cost\_segment

new\_cost = cost - improvement

if debug:

print(f"\nИтерация {iteration}: улучшение найдено на сегменте {i}-{j}")

print(f" Старый сегмент: {a}-{b} ({graph[a][b]}) + {c}-{d} ({graph[c][d]}) = {old\_cost\_segment}")

print(f" Новый сегмент: {a}-{c} ({graph[a][c]}) + {b}-{d} ({graph[b][d]}) = {new\_cost\_segment}")

print(f" Улучшение: {improvement}, новая стоимость: {new\_cost}")

# Переворачиваем подотрезок пути между i и j

path[i:j+1] = path[i:j+1][::-1]

cost = new\_cost

improved = True

if debug:

print(f" Новый маршрут: {path}")

if debug:

print(f"\nЗавершено после {iteration} итераций")

print(f"Оптимизированный маршрут: {path}, стоимость: {cost}")

return path, cost

# Применяем оптимизацию 2-opt

optimized\_path, optimized\_cost = two\_opt(path, cost)

if optimized\_cost == float('inf'):

if debug:

print("Оптимизированный маршрут недействителен")

return "no path", 0

else:

if debug:

print(f"\nФинальный результат:")

print(f"Маршрут: {optimized\_path}")

print(f"Общая стоимость: {optimized\_cost}")

return optimized\_path, optimized\_cost

# Приложение Б

Результаты тестирования точного метода дп (итеративный) приведены в табл.1.

Таблица 1 – Тестирование точного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | 5  0 1 13 23 7  12 0 15 18 28  21 29 0 33 28  23 19 34 0 38  5 40 7 39 0 | 78  0 4 2 3 1 0 | Верно |
| 2. | 3  0 1 0  1 0 1  0 1 0 | no path | Верно |
| 3. | 3  0 27 7  15 0 28  25 21 0 | 43  0 2 1 0 | Верно |
| 4. | 4  0 25 23 11  19 0 37 18  20 31 0 24  15 39 2 0 | 63  0 3 2 1 0 | Верно |

Результаты тестирования приближенного алгоритма АЛШ-2 приведены в табл.2.

Таблица 2 – Тестирование приближенного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | 9  3  0 78 62 27 77 15 64 80 95  78 0 65 39 48 68 58 85 88  62 65 0 28 33 43 53 9 98  27 39 28 0 22 6 25 8 46  77 48 33 22 0 82 46 30 80  15 68 43 6 82 0 40 42 58  64 58 53 25 46 40 0 68 60  80 85 9 8 30 42 68 0 14  95 88 98 46 80 58 60 14 0 | 285 | +0.00% |
| 2. | 3  1  0 48 14  48 0 58  14 58 0 | 120 | +0.00% |
| 3. | 7  5  0 0 100 96 85 68 26  0 0 0 0 0 0 0  100 0 0 50 83 30 66  96 0 50 0 96 9 47  85 0 83 96 0 32 90  68 0 30 9 32 0 7  26 0 66 47 90 7 0 | -1 | +0.00% |
| 4. | 9  4  0 57 56 50 3 99 66 77 6  57 0 22 39 78 74 32 84 81  56 22 0 39 52 49 24 23 23  50 39 39 0 66 43 28 93 20  3 78 52 66 0 98 85 44 3  99 74 49 43 98 0 76 2 96  66 32 24 28 85 76 0 5 95  77 84 23 93 44 2 5 0 53  6 81 23 20 3 96 95 53 0 | 179 | +3.00% |