**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: Коммивояжер. Вариант 3.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3343 |  | Ермолаева В. А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Реализовать алгоритм для нахождения пути коммивояжера и его стоимости двумя методами.

## Задание

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны заклятием процветания. Для этого ему необходимо посетить каждый город ровно один раз, создавая тропу благополучия, и вернуться обратно в столицу, используя минимум своих чародейских сил. Вашей задачей является помощь в прокладывании маршрута с помощью древнего и могущественного алгоритма ветвей и границ.

Карта дорог Алгоритмии перед Гамильтоном представляет собой полный граф, где каждый город соединён магическими порталами с каждым другим. Стоимость использования портала из города в город занимает определённое количество маны, и Гамильтон стремится минимизировать общее потребление магической энергии для закрепления проклятия.

**Входные данные:**

Первая строка содержит одно целое число N (N — количество городов). Города нумеруются последовательными числами от 0 до N−1.

Следующие N строк содержат по N чисел каждая, разделённых пробелами, формируя таким образом матрицу стоимостей M. Каждый элемент Mi,j​ этой матрицы представляет собой затраты маны на перемещение из города i в город j.

**Выходные данные:**

Первая строка: Список из N целых чисел, разделённых пробелами, обозначающих оптимальный порядок городов в магическом маршруте Гамильтона. В начале идёт город, с которого начинается маршрут, затем последующие города до тех пор, пока все они не будут посещены.

Вторая строка: Число, указывающее на суммарное количество израсходованной маны для завершения пути.

Вариант 3. 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 — 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.

## Выполнение работы

Для решения задачи был применен метод ветвей и границ с использованием эвристик. На каждом шаге выбирается ребро, добавление которого в итоге даст минимальную сумму пути. Для оценки оставшегося пути используются две эвристики:

1) Минимальное остовное дерево (МОД). Оценивает минимальную стоимость соединения оставшихся городов.

2) Полусумма легчайших рёбер. Для каждого куска вычисляется полусумма минимальных входящих и исходящих рёбер, что даёт нижнюю оценку стоимости.

Второй метод, реализованный для решения задачи - алгоритм модификации решения АМР. Он заключается в поиске решения через локальные модификации. Город переставляется в другое место, если вычисленная новая стоимость маршрута оказывается меньше текущей.

Временная сложность МВиГ в худшем случае будет экспоненциальной из-за полного перебора О(n!). Затраты по памяти будут составлять O(n) для хранения текущего маршрута. Для АМР временная сложность - O(n3), т. к. для каждой модификации перебирается n2 вариантов, а пространственная - O(n).

Описание реализованных функций и структур:

* def generate\_random\_matrix(n): Генерирует квадратную матрицу стоимостей размером n x n.
* def save\_matrix\_to\_file(matrix, filename): Сохраняет матрицу стоимостей в файл.
* load\_matrix\_from\_file(filename): Загружает матрицу стоимостей из файла.
* calculate\_total\_cost(path, cost\_matrix): Вычисляет общую стоимость маршрута.
* amr\_algorithm(initial\_path, cost\_matrix): Реализует алгоритм модификации решения (АМР) для поиска оптимального пути.
* get\_allowed\_edges(path, remaining\_cities): Возвращает список допустимых рёбер для продолжения пути.
* calculate\_mst(cost\_matrix, path, remaining\_cities): Вычисляет минимальное остовное дерево (МОД) для оставшихся городов.
* calculate\_half\_sum(cost\_matrix, path, remaining\_cities): Вычисляет полусумму весов двух легчайших рёбер для каждого куска.
* calculate\_lower\_bound(path, current\_cost, remaining\_cities, cost\_matrix): Вычисляет нижнюю оценку стоимости оставшегося пути.
* branch\_and\_bound(cost\_matrix): Реализует алгоритм ветвей и границ для поиска оптимального пути.
* print\_matrix(matrix): Выводит матрицу стоимостей на экран.
* print\_path(path): Преобразует путь в строку для вывода.

Исходный код программы смотреть в приложении А.

## Тестирование

Результаты тестирования представлены в таблице 1.

Табл. 1. – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | 4  -1 8 1 11  1 -1 7 7  12 3 -1 6  2 15 2 -1 | 0 2 1 3  13.0 | Результат соответствует ожиданиям. |
| 2. | 5  -1 12 14 24 87  29 -1 70 12 92  36 98 -1 6 17  12 55 47 -1 62  146 80 55 77 -1 | 0 2 4 1 3  135.0 | Результат соответствует ожиданиям. |
| 3. | 6  -1 40 39 2 16 35  77 -1 84 2 47 59  7 19 -1 54 6 17  68 80 83 -1 90 23  11 82 88 81 -1 90  39 11 35 89 65 -1 | 0 1 3 5 2 4  117.0 | Результат соответствует ожиданиям. |

## Выводы

В ходе выполнения лабораторной работы был реализован метод ветвей и границ для решения задачи коммивояжера. Для оптимизации алгоритма были применены эвристики для оценки оставшегося пути и выбора лучшего варианта. Также был написан и применен алгоритм модификации решения для улучшения найденного решения.

# Приложение А

import random

def generate\_matrix(n):

matrix = [[-1] \* n for \_ in range(n)]

for i in range(n):

for j in range(n):

if i != j:

matrix[i][j] = random.randint(1, 100)

return matrix

def save\_matrix\_to\_file(matrix, filename):

with open(filename, 'w') as file:

file.write(f"{len(matrix)}\n")

for row in matrix:

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

def load\_matrix\_from\_file(filename):

with open(filename, 'r') as file:

n = int(file.readline())

matrix = []

for \_ in range(n):

row = list(map(int, file.readline().split()))

matrix.append(row)

return matrix

def calculate\_total\_cost(path, cost\_matrix):

total\_cost = 0.0

n = len(path)

for i in range(n):

total\_cost += cost\_matrix[path[i]][path[(i + 1) % n]]

return total\_cost

def amr\_algorithm(cost\_matrix):

n = len(cost\_matrix)

initial\_path = list(range(n))

best\_path = initial\_path.copy()

best\_cost = calculate\_total\_cost(initial\_path, cost\_matrix)

m = True

iterations = 0

F = n

print(f"Начальный путь: {print\_path(best\_path)} со стоимостью: {best\_cost}.")

while m and iterations < F:

m = False

for i in range(1, n):

for j in range(1, n):

new\_path = best\_path[:]

new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

new\_cost = calculate\_total\_cost(new\_path, cost\_matrix)

if new\_cost < best\_cost:

print(f"\tМеняем местами города {best\_path[i]} и {best\_path[j]}.")

print(f"Было обнаружено лучшее решение {print\_path(new\_path)} со стоимостью {new\_cost} (улучшение на {best\_cost - new\_cost}).")

best\_path = new\_path

best\_cost = new\_cost

m = True

iterations += 1

break

print(f"Все города были посещены в порядке: {print\_path(best\_path)}. Стоимость найденного пути = {best\_cost}.")

return best\_path, best\_cost

def get\_allowed\_edges(path, remaining\_cities):

allowed\_edges = []

last\_city = path[-1]

for city in remaining\_cities:

allowed\_edges.append((last\_city, city))

return allowed\_edges

def calculate\_mst(cost\_matrix, path, remaining\_cities):

chunks = [path] + [[city] for city in remaining\_cities]

print("\tОценим оставшийся путь с помощью МОД для оставшихся кусков:")

print(f"\t{" | ".join([", ".join(map(str, chunk)) for chunk in chunks])}.")

print(f"\tВсе доступные ребра:")

edges = []

for i in range(len(chunks)):

for j in range(len(chunks)):

if i != j:

start = chunks[i][-1]

end = chunks[j][0]

cost = cost\_matrix[start][end]

if cost != -1:

print(f"\t\t{start} -> {end}, стоимость = {cost}")

edges.append((cost, start, end))

edges.sort()

parent = {city: city for chunk in chunks for city in chunk}

def find(u):

while parent[u] != u:

parent[u] = parent[parent[u]]

u = parent[u]

return u

mst\_weight = 0

for cost, u, v in edges:

root\_u = find(u)

root\_v = find(v)

if root\_u != root\_v:

print(f"\tДобавляем к каркасу ребро {u} -> {v} со стоимостью {cost}.")

mst\_weight += cost

parent[root\_v] = root\_u

return mst\_weight

def calculate\_half\_sum(cost\_matrix, path, remaining\_cities):

chunks = [path] + [[city] for city in remaining\_cities]

half\_sum = 0

print("\n\tОценим оставшийся путь с помощью полусуммы весов двух легчайших рёбер по всем кускам:")

print(f"\t{" | ".join([", ".join(map(str, chunk)) for chunk in chunks])}.")

for chunk in chunks:

incoming\_edges = []

for other\_chunk in chunks:

if other\_chunk != chunk:

start = other\_chunk[-1]

end = chunk[0]

cost = cost\_matrix[start][end]

if cost != -1:

incoming\_edges.append(cost)

min\_incoming = min(incoming\_edges) if incoming\_edges else 0

outgoing\_edges = []

for other\_chunk in chunks:

if other\_chunk != chunk:

start = chunk[-1]

end = other\_chunk[0]

cost = cost\_matrix[start][end]

if cost != -1:

outgoing\_edges.append(cost)

min\_outgoing = min(outgoing\_edges) if outgoing\_edges else 0

print(f"\tРассматриваем кусок {chunk}. Легчайшее входящее ребро = {min\_incoming}, а исходящее = {min\_outgoing}.")

half\_sum += (min\_incoming + min\_outgoing) / 2

return half\_sum

def calculate\_lower\_bound(path, remaining\_cities, cost\_matrix):

mst\_estimate = calculate\_mst(cost\_matrix, path, remaining\_cities)

half\_sum\_estimate = calculate\_half\_sum(cost\_matrix, path, remaining\_cities)

print(f"\tДля оставшегося пути вес минимального каркаса = {mst\_estimate}, минимальная полусумма = {half\_sum\_estimate}\n\t=> Берем максимальную из двух оценок = {max(mst\_estimate, half\_sum\_estimate)}.\n")

return max(mst\_estimate, half\_sum\_estimate)

def branch\_and\_bound(cost\_matrix):

n = len(cost\_matrix)

best\_path = None

best\_cost = float('inf')

def backtrack(path, current\_cost, remaining\_cities):

nonlocal best\_path, best\_cost

if not remaining\_cities:

total\_cost = current\_cost + cost\_matrix[path[-1]][path[0]]

print(f"Все города были посещены в порядке: {print\_path(path)}. Стоимость найденного пути = {total\_cost}.\n-------------------------")

if total\_cost < best\_cost:

best\_cost = total\_cost

best\_path = path[:]

return

for u, v in get\_allowed\_edges(path, remaining\_cities):

new\_cost = current\_cost + cost\_matrix[u][v]

lower\_bound = calculate\_lower\_bound(path + [v], remaining\_cities - {v}, cost\_matrix)

if new\_cost + lower\_bound < best\_cost:

backtrack(path + [v], new\_cost, remaining\_cities - {v})

else:

print("\t=> Данное решение заведомо плохое и не подходит. Обрубаем ветку.\n")

backtrack([0], 0, set(range(1, n)))

return best\_path, best\_cost

def print\_matrix(matrix):

print("Матрица стоимости путей:")

for i in range(len(matrix[0])):

print("\t".join(map(str, matrix[i])))

print()

def print\_path(path):

p = ""

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

p += f"{path[i]} -> "

return p + f"{path[-1]}"

cost\_matrix = []

opt = int(input("Хотите ли вы:\n\t1. Сгенерировать матрицу и сохранить ее в файл;\n\t2. Загрузить матрицу из файла;\n\t3. Ввести матрицу вручную.\n"))

if opt == 1:

n = int(input("Введите размер матрицы стоимости путей: "))

cost\_matrix = generate\_matrix(n)

save\_matrix\_to\_file(cost\_matrix, "matrix.txt")

elif opt == 2:

try:

cost\_matrix = load\_matrix\_from\_file("matrix.txt")

except FileNotFoundError:

print("Файл с матрицей не существует, генерируем матрицу размера 3.")

cost\_matrix = generate\_matrix(3)

save\_matrix\_to\_file(cost\_matrix, "matrix.txt")

else:

n = int(input("Введите размер матрицы стоимости путей: "))

cost\_matrix = []

for i in range(n):

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

cost\_matrix.append(row)

opt = int(input("Какой из методов решения использовать?\n\t1. МВиГ;\n\t2. АМР.\n"))

best\_path, best\_cost = branch\_and\_bound(cost\_matrix) if opt == 1 else amr\_algorithm(cost\_matrix)

print(f"Лучшее решение:\nВсе города были посещены в порядке: {print\_path(best\_path)}. Стоимость найденного пути = {best\_cost}.")