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 Пивоев Н. М.

Преподаватель Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Изучить принцип работы алгоритмов Литтла и АДО МОД и реализовать их на практике.

Задание.

Вариант 1

Метод Ветвей и Границ: Алгоритм Литтла. Приближённый алгоритм: 2-приближение по МиД (Алгоритм двойного обхода минимального остовного дерева). Замечание к варианту 1 АДО МОД является 2-приближением только для евклидовой матрицы. Начинать обход МОД со стартовой вершины.

Независимо от варианта, при сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную), сохранять её в файл и использовать в качестве входных данных.

Алгоритм Литтла

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 0, 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.

АДО МОД

Разработайте программу, которая решает задачу коммивояжера при помощи 2-приближенного алгоритма. В данной постановке задачи нужно вернуться в исходную вершину после прохождения всех остальных вершин. При обходе остовного дерева (MST) необходимо идти по минимальному допустимому ребру из текущего. Каждая вершина в графе обозначается неотрицательным числом, начиная с 0, каждое ребро имеет неотрицательный вес. В графе **нет рёбер из вершины в саму себя,** в матрице весов на месте таких ***отсутствующих рёбер*** стоит значение ***-1.***

В первой строке указывается начальная вершина. Далее идёт матрица весов.

В качестве выходных данных необходимо представить длину пути, полученного при помощи алгоритма. Следующей строкой необходимо представить путь, в котором перечислены вершины, по которым необходимо пройти от начальной вершины.

Выполнение работы

Описание алгоритма Литтла

Алгоритм Литтла – алгоритм решения задачи коммирояжёра, основанный на методе ветвей и границ. Он позволяет находить наиболее оптимальный путь по матрице расстояний. Алгоритм состоит из следующих шагов:

1) Сначала происходит вычитание минимального значения строки из всех элементов этой строки. То же самое повторяется для всех строк, а затем для столбцов.

2) Затем идёт поиск нулевых элементов матрицы, среди них выбирается тот, в строке и ряде которого находятся минимальные элементы.

3) Далее идёт ветвление на два разных случая. В первом удаляется строка и столбик, содержащие выбранный нулевой элемент, а также запрещаются рёбра, которые могут привести к недопустимым решениям и алгоритм начинается заново для новой матрицы. Во втором рассматриваемый нулевой элемент заменяется на бесконечность и алгоритм начинается заново.

Каждый раз полученная стоимость сравнивается с рекордной. Если полученная стоимость больше минимальной, то текущая ветвь решения не рассматривается.

Описание АДО МОД

Алгоритм двойного обхода минимального остовного дерева – алгоритм поиска приближённого решения задачи коммивояжёра. Сначала находится минимальное остовное дерево, с помощью алгоритма Крускала и строится новый граф, в котором каждая вершина исходного соединена с ближайшей вершиной в МОД. Затем поиском в глубину, начиная с заданной вершины, составляется гамильтонов цикл, составляющий замкнутый цикл, стоимость которого вычисляется как сумма весов рёбер в нём.

Оценка сложности алгоритма Литтла

Временная сложность:

Вычитание минимальных значений происходит за $O(n^2)$, где n – длина одной из сторон матрицы.

Обработка нулевых коэффициентов происходит за $O(n^2)$.

Ветвление происходит за $O(2^{n-1} \cdot n^2)$, поскольку в худшем случае необходимо обойти полное бинарное дерево, а на каждом этапе необходимо вычитать минимальные значения и обрабатывать нулевые коэффициенты.

Временная сложность - $O(2^{n-1} \cdot n^2)$.

Пространственная сложность – тоже $O(2^{n-1} \cdot n^2)$, поскольку на каждом этапе создаётся копия матрицы, в худшем случае их $O(2^{n-1})$.

Оценка сложности АДО МОД

Временная сложность:

В методе Крускала происходит обход половины матрицы за $O(E^2)$, сортировка за $O(ElogE)$, поиск ближайших вершин за $O(E^2)$, где E – количество рёбер в графе для одной вершины (E = n-1).

Поиск в глубину рассчитывается за $O(E + V)$, поскольку нужно обойти все вершины и рёбра, где V – все вершины.

Временная сложность - $O(E^2 + V)$.

Пространственная сложность – $O(E^2)$, поскольку составляется массив рёбер, размером E^2, остальные массивы имеют размер E.

Код программы содержит реализацию следующих функций:

Алгоритм Литтла

- *subtract_min_from_matrix(self)* – вычитает из каждой ячейки минимальное значение для данной строки и столбца, потому что это значение и так будет заложено в минимальную стоимость.

- *coef_finder(self, row, column)* – находит минимальный элемент в выбранной строке и столбце, возвращает их сумму.
- *find_zero_coefs* – поиск нулевого элемента, у которого сумма двух элементов из той же строки и столбца минимальна.
- *find_longest_path(self, path, edge)* – находит самый длинный путь в графе основываясь на текущем ребре.
- *process_path(self, path, x_ind, y_ind)* – запрещает ребро, оканчивающее цикл в графе.
- *reduce_matrix(self, coordinate, path, x_ind, y_ind)* – удаляет строку и столбец матрицы, в которых находился выбранный нулевой элемент.
- *check_solution(self, path, current_cost, x_ind, y_ind)* – обработка матрицы размером 2x2 для нахождения более выгодной стоимости.
- *solve(self, matrix, path, lower_limit_ x_ind, y_ind, iteration)* – главная функция, объединяющая все остальные для нормальной работы алгоритма. Также отвечает за ветвление матрицы.

АДО МОД

- *find(parent, node)* – находит ближайшего соседа.
- *kruskal_mod(self)* – строит минимальное остовное дерево.
- *dfs(self, node, adjacent, visited, path)* – поиск в глубину для остовного дерева и построения пути
- *solve(self, start)* – главная функция, объединяющая все остальные.

Тестирование

Программа была протестирована на различных входных данных. Для удобства тестирования, создан генератор матриц, подходящий для обоих алгоритмов. Рёбра на главной диагонали – inf, а остальные – float числа. Соответственно для алгоритма Литтла float значения округляются, а для АДО МОД inf обрабатываются как -1.

Таблица 1.

Входные данные	Выходные данные	Комментарий
3 -1 1 3 3 -1 1 1 2 -1	0 1 2 3.0	Алгоритм Литтла
4 -1 3 4 1 1 -1 3 4 9 2 -1 4 8 9 2 -1	0 3 2 1 6.0	Алгоритм Литтла
3 -1 1 1 1 -1 1 1 1 -1	0 1 2 3.0	Алгоритм Литтла
1 -1 50.1 5.45 58.91 -1 4.36 19.0 66.71 -1	73.46 1 2 0 1	АДО МОД
2 -1 18.97 22.36 19.42 3.61 18.97 -1 35.61 38.01 17.0 22.36 35.61 -1 16.28 21.19	91.92 2 3 0 4 1 2	АДО МОД

19.42 38.01 16.28 -1 21.02 3.61 17.0 21.19 21.02 -1		
-1 1 1 1 -1 1 1 1 -1	3.0 2 0 1 2	АДО МОД

```
1 Сгенерировать обычную матрицу
2 Сгенерировать симметричную матрицу
3 Сгенерировать евклидову матрицу
4 Загрузить матрицу
5 Сохранить матрицу
6 Метод Литтла
7 Метод АДО МОД
8 Выход
1

Введи количество городов (размер матрицы)
3

[inf, 64.16, 6.24]
[33.11, inf, 5.67]
[1.6, 60.92, inf]

1 Сгенерировать обычную матрицу
2 Сгенерировать симметричную матрицу
3 Сгенерировать евклидову матрицу
4 Загрузить матрицу
5 Сохранить матрицу
6 Метод Литтла
7 Метод АДО МОД
8 Выход
6

0 1 2
70.0
```

Рисунок 1 – Результат работы программы

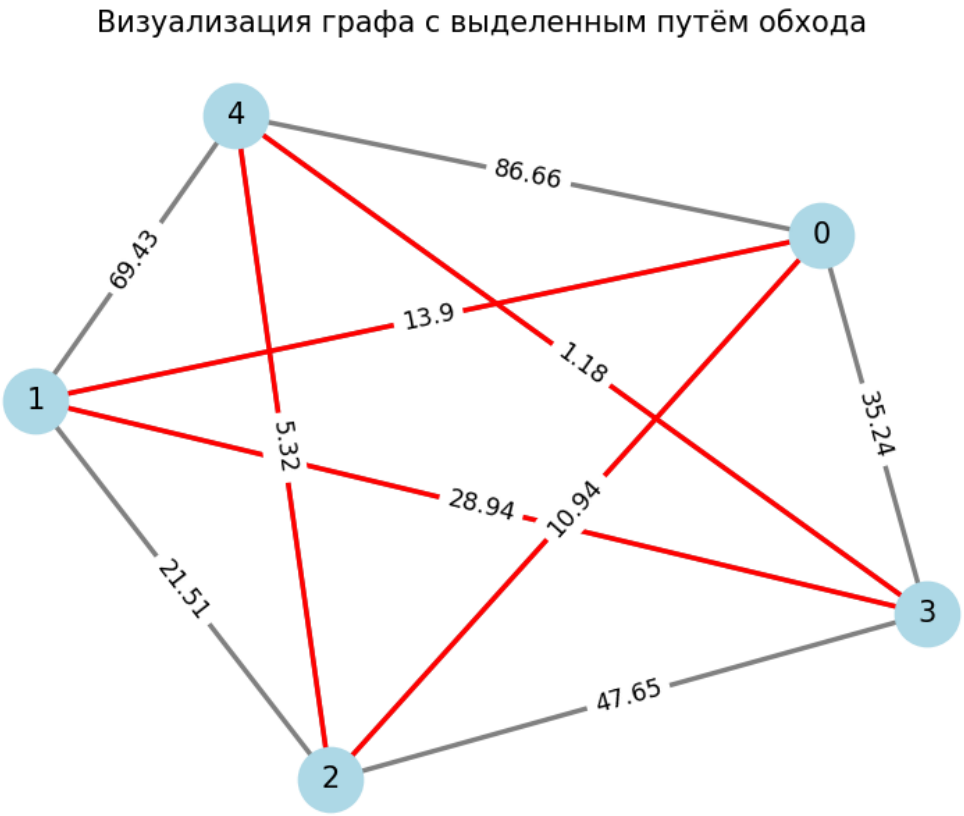

Рисунок 2 – Визуализация для АДО МОД

Исследование

Исследуем эффективность обоих алгоритмов на входных данных разного размера.

Таблица 2. Исследование эффективности по времени.

Алгоритм Литтла	
N	Время в секундах
5	0.000177
10	0.004046
20	0.048956
30	0.412179
40	0.863758
АДО МОД	
N	Время в секундах
100	0.004695
250	0.020719
500	0.130701
750	0.418319
1000	0.707339

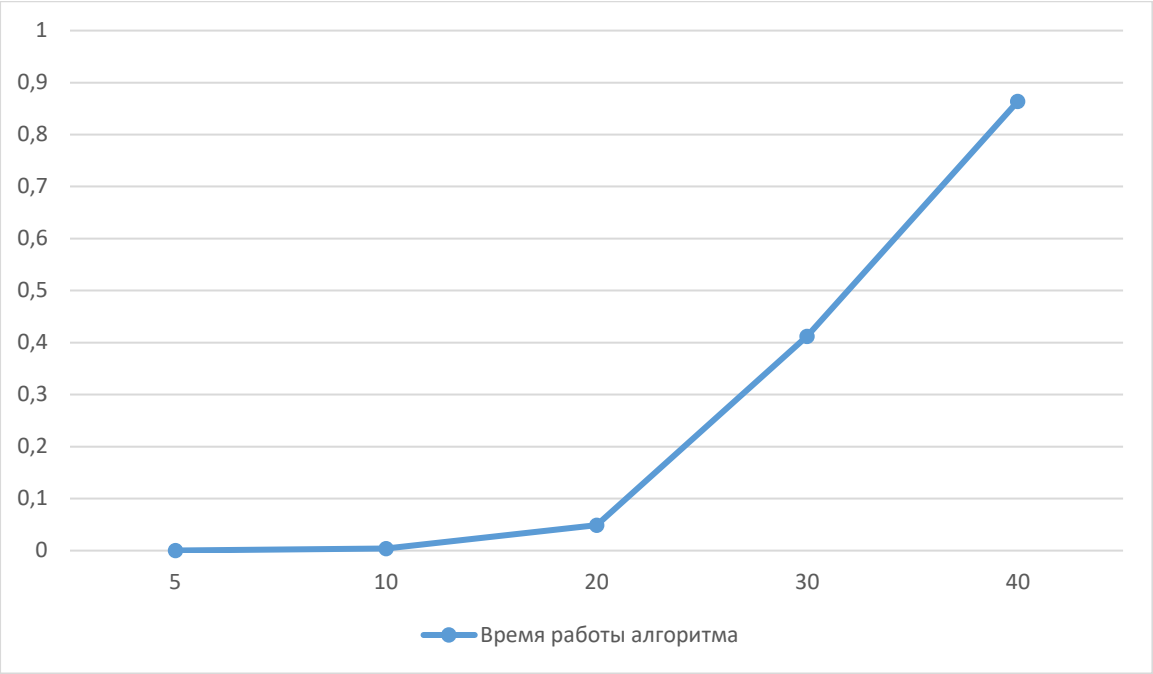

Рисунок 3 – График зависимости затраченного времени от размера матрицы для алгоритма Литтла

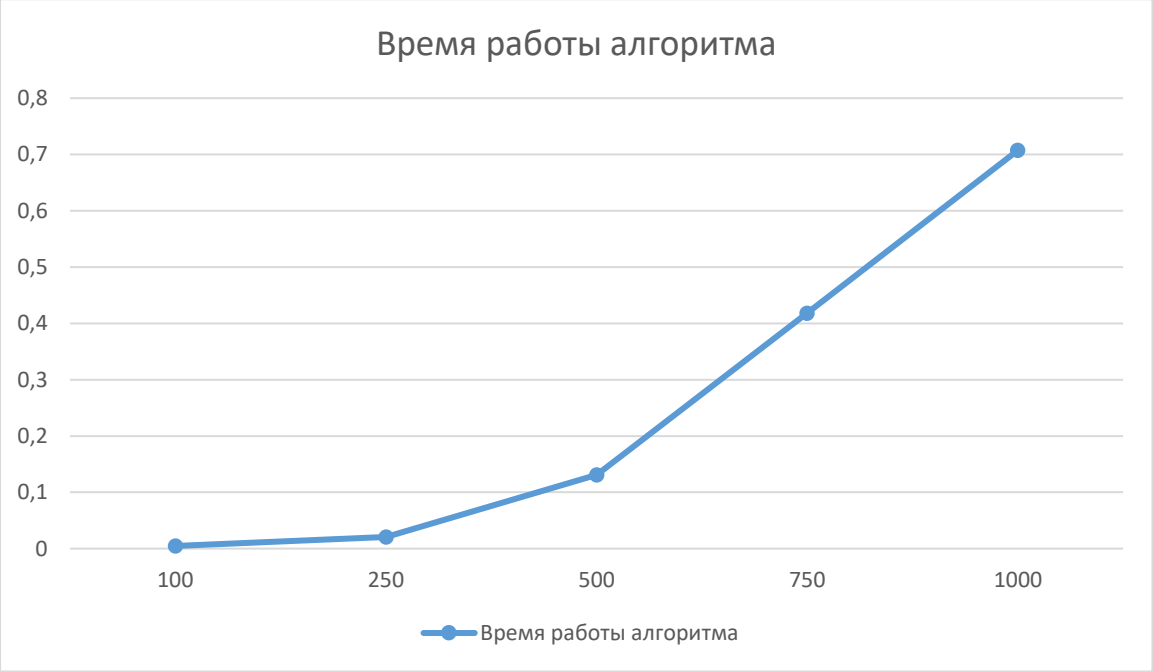

Рисунок 4 – График зависимости затраченного времени от размера матрицы для АДО МОД

Полученные данные доказывают, что время работы алгоритма Литтла возрастает крайне быстро (экспоненциально), поэтому он неэффективен даже для относительно небольших n.

Приближённый АДО МОД выполняется намного быстрее, поэтому его можно использовать для нахождения приближённого решения при больших n, а алгоритм Литтла для точного результата при малых n.

Выводы

Во время выполнения лабораторной работы, была изучена работа алгоритма Литтла и АДО МОД. Решены задачи поиска точного и приближённого расстояния коммивояжёра.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.py

```
from levenshtein import *


def get_costs(is_special):
    try:
        costs = list(map(int, input().split()))
        if is_special:
            if len(costs) != 2:
                raise ValueError
        else:
            if len(costs) != 3:
                raise ValueError

        return costs
    except:
        print("Стоимости некорректны")
        exit(1)


def main():
    print("Введи стоимость замены, вставки и удаления символов")
    costs = Costs(*get_costs(0))

    print("\nВведи первую строку")
    s1 = input()

    print("\nВведи вторую строку")
    s2 = input()
    print()

    print("Добавить особо заменяемый и добавляемый символы? y")
    if input() == 'y':
        print("Введи цены, затем символы")
        special_costs = Special_Costs(*get_costs(1), *input().split())

        d = get_distance_matrix(s1, s2, costs, special_costs)
        solution = traceback_operations(d, s1, s2, costs, special_costs)
    else:
        d = get_distance_matrix(s1, s2, costs)
        solution = traceback_operations(d, s1, s2, costs)

    check_solution(s1, s2, solution)

    print('    ', *[c for c in s2], sep='  ')
    for i, column in enumerate(d):
        if i == 0:
            print(' ', end='  ')
        else:
            print(s1[i - 1], end='  ')

        for j in column:
            if j >= 10:
                print(j, end=' ')
                continue
            print(j, end='  ')
        print()
```

```
        print()

        print("Расстояние Левенштейна =", d[len(s1)][len(s2)])
        print(solution, s1, s2, sep='\n')


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

Имя файла: levenshtein.py

```
    class Costs:
        def __init__(self, replacement_cost, insertion_cost, deletion_cost):
            self.replacement_cost = replacement_cost
            self.insertion_cost = insertion_cost
            self.deletion_cost = deletion_cost


    class Special_Costs:
        def __init__(self, replacement_cost, insertion_cost, replacement_symbol,
insertion_symbol):
            self.replacement_cost = replacement_cost
            self.insertion_cost = insertion_cost
            self.replacement_symbol = replacement_symbol
            self.insertion_symbol = insertion_symbol


    def decide_costs(symbol_1, symbol_2, costs, special_costs = None):
        if special_costs and symbol_1 == special_costs.replacement_symbol:
            replacement_cost = special_costs.replacement_cost
        else:
            replacement_cost = costs.replacement_cost

        if special_costs and symbol_2 == special_costs.insertion_symbol:
            insertion_cost = special_costs.insertion_cost
        else:
            insertion_cost = costs.insertion_cost
        return replacement_cost, insertion_cost


    def get_distance_matrix(s1, s2, costs, special_costs = None):
        m, n = len(s1), len(s2)

        d = [[-1] * (n+1) for _ in range(m+1)]
        d[0][0] = 0

        for j in range(1, n + 1):
            if special_costs and s2[j - 1] == special_costs.insertion_symbol:
                d[0][j] = d[0][j - 1] + special_costs.insertion_cost
                continue
            d[0][j] = d[0][j - 1] + costs.insertion_cost

        for i in range(1, m + 1):
            d[i][0] = d[i - 1][0] + costs.deletion_cost
            for j in range(1, n + 1):
                if s1[i-1] == s2[j-1]:
                    d[i][j] = d[i - 1][j - 1]
                    continue

                replacement_cost, insertion_cost = decide_costs(s1[i - 1], s2[j
- 1], costs, special_costs)
```

```
                d[i][j] = min(
                    d[i - 1][j - 1] + replacement_cost,
                    d[i][j - 1] + insertion_cost,
                    d[i - 1][j] + costs.deletion_cost
                )
        return d


    def traceback_operations(d, s1, s2, costs, special_costs = None):
        m, n = len(s1), len(s2)
        solution = ''
        i, j = m, n
        while i > 0 or j > 0:
            if i > 0 and j > 0 and s1[i - 1] == s2[j - 1]:
                solution += "M"
                i -= 1
                j -= 1
                continue

            replacement_cost, insertion_cost = decide_costs(s1[i - 1], s2[j -
1], costs, special_costs)

            if i > 0 and j > 0 and d[i - 1][j - 1] + replacement_cost == d[i][j]:
                solution += 'R'
                i -= 1
                j -= 1
            elif j > 0 and d[i][j - 1] + insertion_cost == d[i][j]:
                solution += 'I'
                j -= 1
            elif i > 0 and d[i - 1][j] + costs.deletion_cost == d[i][j]:
                solution += 'D'
                i -= 1
        return solution[::-1]


    def check_solution(s1, s2, solution):
        s = list(s1)
        ptr = 0
        for option in solution:
            print(''.join(s), option, ptr)
            if option == 'R':
                s[ptr] = s2[ptr]
            elif option == 'I':
                s.insert(ptr, s2[ptr])
            elif option == 'D':
                del s[ptr]
                ptr -= 1
            ptr += 1
            print(''.join(s), option, ptr, "\n")

        print("Изменённая s1 по сравнению с s2")
        print(''.join(s))
        print(s2)
        if ''.join(s) == s2:
            print("Совпадение\n")
```