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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Коммивояжер

Студент гр. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2025

Цель работы.

Написание алгоритма Коммивояжера на языке программирования методами динамического программирования, численного приближения и метода ветвей и границ.

Задание.

1) 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 ($5 \leq n \leq 15$).

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 $i \neq j$), 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, начинающейся и заканчивающейся 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

2) 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.

Пример входных данных

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

19.42 38.01 16.28 -1 21.02

3.61 17.0 21.19 21.02 -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. Для приведённых в примере входных данных ответом будет

91.92

2 3 0 4 1 2

Sample Input:

10

-1 9.22 23.32 8.49 16.76 30.0 26.02 13.89 3.61 13.15 21.4 8.06 4.12

9.22 -1 32.2 17.69 25.5 39.2 33.38 13.93 12.73 21.02 30.48 5.1 11.66

23.32 32.2 -1 15.23 15.52 12.17 25.24 27.29 19.72 12.21 3.16 29.0 20.62

8.49 17.69 15.23 -1 10.05 21.63 21.19 18.03 5.0 8.06 13.04 15.65 7.28

16.76 25.5 15.52 10.05 -1 15.26 11.66 28.07 14.14 16.12 12.37 24.74 17.03

30.0 39.2 12.17 21.63 15.26 -1 19.1 37.64 26.63 22.56 11.05 37.22 28.65

26.02 33.38 25.24 21.19 11.66 19.1 -1 38.83 24.33 27.78 22.2 34.0 27.46

13.89 13.93 27.29 18.03 28.07 37.64 38.83 -1 14.56 15.23 26.93 8.94 11.4

3.61 12.73 19.72 5.0 14.14 26.63 24.33 14.56 -1 10.0 17.8 10.77 3.16

13.15 21.02 12.21 8.06 16.12 22.56 27.78 15.23 10.0 -1 11.7 17.2 9.49

21.4 30.48 3.16 13.04 12.37 11.05 22.2 26.93 17.8 11.7 -1 27.66 19.1

8.06 5.1 29.0 15.65 24.74 37.22 34.0 8.94 10.77 17.2 27.66 -1 8.6

4.12 11.66 20.62 7.28 17.03 28.65 27.46 11.4 3.16 9.49 19.1 8.6 -1

Sample Output:

147.25

10 2 5 9 3 8 12 0 11 1 7 4 6 10

3)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 $M_{i,j}$ этой матрицы представляет собой затраты маны на перемещение из города i в городj.

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.

Sample Input 1:

3

-1 1 3

3 -1 1

1 2 -1

Sample Output 1:

0 1 2

3.0

Sample Input 2:

4

-1 3 4 1

1 -1 3 4

9 2 -1 4

8 9 2 -1

Sample Output 2:

0 3 2 1

6.0

Выполнение работы.

Для метода динамического программирования реализован алгоритм, находящийся в файле first.py.

В нем реализован класс *Matrix,* в котором происходит вся логика работы.

В методе *create_vertexes()* создаются связи, куда можно пройти из каждой вершины.

Метод *find_way()* инициализирует начало работы. Внутри создается массив *used* – посещенные вершины, и вызывается метод *find().*

В методе *find()* происходит рекурсивный поиск верного пути. Внутри происходит итерация по соседям поданной вершины графа, если новая найденная вершина не была использована и путь до нее с предыдущим путем меньше текущего минимального значения пути, то она добавляется к пути и для нее снова вызывается метод *find(),* до момента, пока ответ не будет найден.

Метод *print()* выводит результат на экран.

Функция *input_data()* производит ввод данных.

Тестирование приведено в табл. Б1.

Для метода 2-приближения разработан алгоритм, находящийся в файле second.py.

В нем реализован класс *Matrix*, в котором происходит вся логика работы.

В методе *create_vertexes()* создаются связи, куда можно пройти из каждой вершины.

Метод *find_min_edges()* ищет минимальный путь и возвращает вершины, из которых он начинается и где он заканчивается. Это нужно для начала построения оставного дерева

В методе *create_spanning_tree()* строится минимальное оставное дерево по алгоритму Прима, создающееся при инициализации класса.

В методе *create_way()* происходит поиск приближенного лучшего пути на основе добавления путей из оставного дерева, если вершина, в которую нужно пройти уже была использована, то берется путь до следующей после нее. Так же внутри создается массив *self.__way,* в котором хранится итоговый путь, который и возвращается.

Метод *calculate_sum()* считает длину найденного пути и возвращает ее.

В функции *input_data()* происходит ввод данных.

В функции *output_data()* происходит вывод данных.

Тестирование приведено в табл. Б2.

Для метода ветвей и границ реализован алгоритм, находящийся в файле third.py.

В нем используются функции для поиска оптимального пути в графе.

В функции *find_way()* происходит рекурсивный поиск пути. Если все города посещены, вызывается функция *complete_path(),* которая завершает путь и проверяет, является ли он лучшим. Если нет, то для каждого непосещенного города вызывается функция *explore_next_city().*

Функция *complete_path()* добавляет возврат в начальный город и проверяет, является ли текущий путь лучше найденного ранее.

Функция *explore_next_city()* проверяет, можно ли перейти в следующий город без превышения текущего лучшего пути. Если условие выполняется, город отмечается как посещенный, и рекурсивно вызывается *find_way().* После возврата из рекурсии город снова помечается как непосещенный.

Функция *find_with_branch_and_bound()* инициализирует поиск, отмечая начальный город как посещенный и запуская рекурсивный обход.

Функция *input_data()* считывает входные данные, заменяя значения -1 на INT_MAX для корректной работы алгоритма.

Функция *output_data()* выводит найденный путь и его стоимость с округлением до одного знака после запятой.

Тестирование приведено в табл. Б3.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

В ходе работы были реализованы алгоритмы для решения задачи Коммивояжера. Разработаны 3 алгоритма, реализующие различные методы поиска оптимального пути: метод динамического программирования, метод 2-приближения и метод ветвей и границ.

Метод динамического программирования является методом полного перебора, вследствие чего он выдает точное решение, но при больших размерах графа он будет нерационален в использовании.

Метод 2-приближения выдает путь не более чем в 2 раза хуже оптимального пути. Это обусловлено неравенством треугольника (если есть 2 кратчайших пути между тремя вершинами, то 3 путь будет меньше, чем сумма двух предыдущих.

Метод ветвей и границ выдает лучшее решение, отсекая ненужные ветви и исходы, используя общие ограничения веток, для того чтобы не рассматривать варианты которые точно не приведут к оптимальному решению.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: first.py

```
import copy
INT_MAX = 2147483647

class Matrix:
    def __init__(self, matrix):
        self.__matrix = matrix
        self.__size = len(matrix)
        self.__vertexes = []
        self.create_vertexes()

    def create_vertexes(self):
        self.__vertexes = []
        for i in range(self.__size):
            next_vertexes = []
            for j in range(self.__size):
                if (i != j and self.__matrix[i][j] != 0):
                    next_vertexes.append(j)
            self.__vertexes.append(next_vertexes)

    def find_way(self):
        used = [1] + [0 for x in range(self.__size - 1)]
        path = [0]

        path, min_value, flag = self.find(used, path, 0, 0, INT_MAX)
        self.print(path, min_value, flag)

    def print(self, path, min_value, flag):
        if flag:
            print(min_value)
            print(' '.join(map(str, path)))
        else:
            print("no path")
```

```
        def find(self, used, path, idx, cur_value, min_value):
            flag = False
            return_path = path
            for j in (self.__vertexes[idx]):
                if j == 0 and used.count(0) == 0:
                    if cur_value + self.__matrix[idx][j] < min_value:
                        return path + [0], cur_value + self.__matrix[idx][j], True
                    return path, cur_value + self.__matrix[idx][j], False
                if used[j] == 0 and cur_value + self.__matrix[idx][j] < min_value:
                    used[j] = 1
                    path.append(j)
                    cur_value += self.__matrix[idx][j]
                    new_path, new_min_value, new_flag = self.find(used, path, j, cur_value, min_value)
                    if new_flag:
                        return_path = copy.copy(new_path)
                        min_value = new_min_value
                        flag = True
                    used[j] = 0
                    path.remove(j)
                    cur_value -= self.__matrix[idx][j]
            return return_path, min_value, flag

    def input_data():
        number = int(input())
        matrix = []
        for i in range(number):
            matrix.append(list(map(int, input().split())))
        return matrix

    def main():
        matrix = input_data()
        ways = Matrix(matrix)
        ways.find_way()

    if __name__ == '__main__':
```

```
main()
```

Название файла: second.py

```
import heapq
INT_MAX = 2147483647

class Matrix:
    def __init__(self, matrix, start_vertex):
        self.__matrix = matrix
        self.__start_vertex = start_vertex
        self.__size = len(matrix)
        self.__spanning_tree = []
        self.__way = []
        self.create_spanning_tree()

    def find_min_edges(self):
        min_weight = INT_MAX
        first = self.__start_vertex
        second = 0
        for i in range(self.__size):
            if self.__matrix[first][i] < min_weight and
self.__matrix[first][i] != -1:
                min_weight = self.__matrix[first][i]
                second = i
        return first, second

    def create_spanning_tree(self):
        first, second = self.find_min_edges()
        self.__spanning_tree = [(first, second), (second, first)]
        heap = []
        used = [False for i in range(self.__size)]
        used[first] = True
        used[second] = True

        for i in range(self.__size):
            if self.__matrix[first][i] != -1 and i != second:
                heapq.heappush(heap, (self.__matrix[first][i],
(first, i)))
```

```
                if self.__matrix[second][i] != -1 and i != first:
                    heapq.heappush(heap,      (self.__matrix[second][i],
(second, i)))

            while used.count(False) > 0:
                current_value = heapq.heappop(heap)
                current_vertex = current_value[1][1]

                if not used[current_vertex]:
                    used[current_vertex] = True
                    self.__spanning_tree.append(current_value[1])
                    self.__spanning_tree.append(current_value[1][::-1])

                    for i in range(self.__size):
                        if  self.__matrix[current_vertex][i]  !=  -1  and
(not used[i]):
                            heapq.heappush(heap,
(self.__matrix[current_vertex][i], (current_vertex, i)))

        def create_vertexes(self):
            vertexes = []
            for i in range(self.__size):
                next_vertexes = []
                for j in (self.__spanning_tree):
                    if j[0] == i:
                        next_vertexes.append(j[1])
                vertexes.append(next_vertexes)
            return vertexes

        def create_way(self):
            count = len(self.__spanning_tree)
            vertexes = self.create_vertexes()
            idx = self.__start_vertex
            way = [idx]
            self.__way = []

            while count > 0:
                flag = True
                for i in vertexes[idx]:
```

```
                if idx in vertexes[i]:
                    way.append(i)
                    flag = False
                    vertexes[idx].remove(i)
                    idx = i
                    break
            if flag:
                add_vertex = vertexes[idx][0]
                way.append(add_vertex)
                vertexes[idx].remove(add_vertex)
                idx = add_vertex
            count -= 1

        for i in way:
            if not i in self.__way:
                self.__way.append(i)
        self.__way += [self.__start_vertex]
        return self.__way

    def calculate_sum(self):
        sum = 0
        for i in range(len(self.__way) - 1):
            sum += self.__matrix[self.__way[i]][self.__way[i + 1]]
        return sum

def input_data():
    start_vertex = int(input())
    matrix = []
    add_data = list(map(float, input().split()))
    matrix.append(add_data)
    for i in range(len(add_data) - 1):
        matrix.append(list(map(float, input().split())))
    return matrix, start_vertex

def output_data(way, sum):
    print("{:.2f}".format(sum))
    print(' '.join(map(str, way)))

def main():
```

```
        matrix, start_vertex = input_data()
        ways = Matrix(matrix, start_vertex)
        way = ways.create_way()
        sum = ways.calculate_sum()
        output_data(way, sum)

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

Название файла: third.py

```
    INT_MAX = 2147483647


    def find_way(city, path, cost, count, visited, matrix, best_cost,
best_path):
        if count == len(matrix):
            return complete_path(city, path, cost, best_cost, best_path,
matrix)

        for next_city in range(len(matrix)):
            if not visited[next_city]:
                best_cost, best_path = explore_next_city(city,
next_city, path, cost, count, visited, matrix, best_cost, best_path)

        return best_cost, best_path


    def complete_path(city, path, cost, best_cost, best_path, matrix):
        cost += matrix[city][path[0]]
        if cost < best_cost:
            best_cost = cost
            best_path = path + [path[0]]
        return best_cost, best_path


    def explore_next_city(city, next_city, path, cost, count, visited,
matrix, best_cost, best_path):
        new_cost = cost + matrix[city][next_city]
```

```
            if new_cost < best_cost:
                visited[next_city] = True
                best_cost, best_path = find_way(next_city, path +
[next_city], new_cost, count + 1, visited, matrix, best_cost, best_path)
                visited[next_city] = False
            return best_cost, best_path


        def find_with_branch_and_bound(matrix):
            matrix_size = len(matrix)
            visited = [False] * matrix_size
            best_cost = INT_MAX
            best_path = []

            visited[0] = True
            best_cost, best_path = find_way(0, [0], 0, 1, visited, matrix,
best_cost, best_path)

            return best_path, best_cost


        def input_data():
            number = int(input())
            matrix = []
            for i in range(number):
                matrix.append(list(map(int, input().split())))
                matrix[i][matrix[i].index(-1)] = INT_MAX
            return matrix


        def output_data(answer_way, answer_value):
            print(' '.join(map(str, answer_way[:-1])))
            print("{:.1f}".format(answer_value))


        def main():
            matrix = input_data()
            path, cost = find_with_branch_and_bound(matrix)
            output_data(path, cost)
```

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

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Тесты для первого задания представлены в табл. Б1.

Таблица Б.1 - Примеры тестовых случаев задания 1.

№ 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	Ok
2.	3 0 1 0 1 0 1 0 1 0	no path	Ok
3.	2 0 1 2 0	3 0 1 0	Ok

Тесты для второго задания представлены в табл. Б2.

Таблица Б.2 - Примеры тестовых случаев задания 2.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	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 19.42 38.01 16.28 -1 21.02 3.61 17.0 21.19 21.02 -1	91.92 2 3 0 4 1 2	Ok
2.	0 -1 10 10 -1	20.00 0 1 0	Ok
3.	0 -1 5 5 5 5 -1 5 5 5 5 -1 5 5 5 5 -1	20.00 0 1 2 3 0	Ok
4.	1 -1 2 3 4 5 2 -1 6 7 8 3 6 -1 9 10 4 7 9 -1 11 5 8 10 11 -1	33.00 1 0 2 3 4 1	Ok

Тесты для третьего задания представлены в табл. Б3.

Таблица Б.3 - Примеры тестовых случаев задания 3.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	3 -1 1 3 3 -1 1 1 2 -1	0 1 2 3.0	Ok
2.	4 -1 3 4 1 1 -1 3 4 9 2 -1 4 8 9 2 -1	0 3 2 1 6.0	Ok
3.	5 -1 3 3 3 3 3 -1 3 3 3 3 3 -1 3 3 3 3 3 -1 3 3 3 3 3 -1	0 1 2 3 4 15.0	Ok
4.	3 -1 10 1 1 -1 10 10 100 -1	0 1 2 30.0	Ok
5.	2 -1 1 1 -1	0 1 2.0	Ok