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 (TSP).

Студент гр. 1304 _____ Сулименко М.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Разработать программу, которая решает задачу нахождения минимального гамильтонова цикла в графе.

Задание.

Дана карта городов в виде ассиметричного, неполного графа $G = (V, E)$, где $V(|V|=n)$ – это вершины графа, соответствующие городам; $E(|E|=m)$ – это ребра между вершинами графа, соответствующие путям сообщения между этими городами.

Каждому ребру m_{ij} (переезд из города i в город j) можно сопоставить критерий выгодности маршрута (вес ребра) равный w_i (натуральное число [1, 1000]), $m_{ij}=inf$, если $i=j$.

Если маршрут включает в себя ребро m_{ij}, то $x_{ij}=1$, иначе $x_{ij}=0$.

Требуется найти минимальный маршрут (минимальный гамильтонов цикл):

Входные данные

Матрица графа из текстового файла.

inf 1 2 2

- inf 1 2
- 1 inf 1

1 1 – inf

Выходные данные:

Кратчайший путь, вес кратчайшего пути, скорость решения задачи.

[1, 2, 3, 4, 1], 4, 0mc

Выполнение работы.

Для реализации программы был выбран алгоритм ветвей и границ. Для хранения информации реализован класс graph и необходимые методы для реализации и запуска алгоритма. Было проведено исследование методов оптимизации алгоритма для достижения среднего времени работы не более 3 минут на графе из 20 вершин. Была разработана программа на языка python. Подробнее о реализованных классах и функциях:

Класс *Graph*:

Данный класс описывает граф. В данном классе содержатся поля для хранения матрицы смежности ребер графа, количества вершин, записи вершин, через который будет проходить путь, его длину и булевая переменная, отвечающая за то, был ли путь найден на данный момент. Класс не содержит никаких методов

Функция *read_input_from_file(filename)*:

Данная функция отвечает за считывание информации из файла и её преобразование в объект класса Graph. Последовательно проходя по всем строкам, создаются соответствующие значения в матрице смежности, которая затем передаётся в конструктор класса.

Функция *iterate_search(graph, currentPath):*

Рекурсивная функция, в которой реализуется поиск оптимального пути методом перебора с возвратом. На вход подается объект класса graph и текущий путь, полученный на предыдущем уровне рекурсии. Изначально происходит проверка, не хуже ли текущий путь уже найденного, затем проверяются все вершины графа на вхождение в путь. Если оба эти условия выполнены, то лучший путь перезаписывается. Затем идет перебор всех соседних вершин для нахождения дальнейшего пути.

Функция calculate_lower_bound(graph, path):

Рекурсивная функция, в которой проверка существования пути методом жадного алгоритма. На вход подается объект класса graph и текущий путь, полученный на предыдущем уровне рекурсии. Если путь существует, то вычисляется его стоимость, которая будет являться границей. Это необходимо для дальнейшей оптимизации поиска лучшего пути, чтобы быстро отсекать заведомо плохие варианты.

Функция start_algorithm(fileName):

Функция, необходимая для запуска всех нужных методов в необходимом порядке. На вход подается имя файла, из которого берется информация для решения задачи. Помимо этого, происходит замер времени работы алгоритма.

Разработанный программный код см. в приложении А.Тестирование см. в приложении Б.

Выводы.

Была разработана программа на языке Python, решающая задачу нахождения минимального гамильтонова цикла в графе.

Для решения данной задачи было использован метод ветвей и границили же поиск с возвратом. Помимо этого, применялся жадный алгоритм.

Программа работает достаточно быстро, не смотря на перебор всех возможных вариантов пути. Для графов с 20 вершинами алгоритм работает в среднем 2-3 минуты, в зависимости от значений ребер.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lb3.py

```
import time, queue
inf = 99999999


# Класс, в котором содержится основная информация о графе:
# матрица смежностей, количество вершин, путь, его длина, и переменная
# отвечающая за то, был ли уже найден путь
class Graph:
    def __init__(self, matrix):
        self.matrix = matrix
        self.numberOfNodes = len(matrix)
        self.path = []
        self.pathLength = inf
        self.isPathFound = False


#    Метод для считывания данных из файла и записи их в виде
графа(списка из списков)
def read_input_from_file(fileName):
    file = open(fileName, 'r')
    matrix = []

    for line in file.readlines():
        row = []
        for elem in line.strip().split():
            if elem == "inf" or elem == '-':
                row.append(inf)
            else:
                row.append(int(elem))
        matrix.append(row)

    graph = Graph(matrix)

    return graph


# Функция реализует рекурсивный поиск оптимального пути методом
перебора с возвратом
# На вход подаётся элемент типа Graph и путь полученный на прошлом
уровне реккурсии
def iterate_search(graph, currentPath):
    if currentPath[0] >= graph.pathLength:
        return

    if len(currentPath[1]) == graph.numberOfNodes:
        if  (graph.matrix[currentPath[1][-1]][graph.path[0]]  !=  inf)
and \
                ((currentPath[0]    +    graph.matrix[currentPath[1][-
1]][graph.path[0]]) < graph.pathLength):
            graph.path.clear()

            for element in currentPath[1]:
```

```
                graph.path.append(element)
            graph.path.append(graph.path[0])
            graph.pathLength = currentPath[0] +
graph.matrix[currentPath[1][-1]][graph.path[0]]
            return

    for count, element in enumerate(graph.matrix[currentPath[1][-1]]):
        if element != inf and count not in currentPath[1]:
            if currentPath[0] + element >= graph.pathLength:
                continue
            if len(currentPath) == graph.numberOfNodes - 1 and
currentPath[0] + \
                    element + graph.matrix[count][graph.path[0]] >=
graph.pathLength:
                continue

            currentPath[0] += element
            currentPath[1].append(count)

            iterate_search(graph, currentPath)

            currentPath[1].pop()
            currentPath[0] -= element
    return


# Метод для роверки существования и вычисления нижней границы
гамильтонова цикла при помощи жадного алгоритма
def calculate_lower_bound(graph, path):
    graphQueue = queue.PriorityQueue()

    for node in range(graph.numberOfNodes):
        if graph.matrix[path[1][-1]][node] != inf:
            graphQueue.put((graph.matrix[path[1][-1]][node], node))

    while not graphQueue.empty():
        currentNode = graphQueue.get()[1]
        if currentNode != inf and currentNode not in path[1]:
            path[1].append(currentNode)
            path[0] += graph.matrix[path[1][-2]][currentNode]

            if len(path[1]) == graph.numberOfNodes:
                if graph.matrix[currentNode][path[1][0]] != inf:
                    graph.path = path[1]
                    graph.path.append(path[1][0])
                    graph.pathLength = path[0] +
graph.matrix[currentNode][path[1][0]]
                    graph.isPathFound = True
                    break
                else:
                    path[1].pop()
                    path[0] -= graph.matrix[path[1][-1]][currentNode]
                    continue

            calculate_lower_bound(graph, path)

            if graph.isPathFound:
                break
```

```
            path[1].pop()
            path[0] -= graph.matrix[path[1][-1]][currentNode]
    return


# Метод, который вызывает необходимые функции и решает задачу о
нахождении
# кратчайшего пути, а после этого выводит ответ в консоль.
# Помимо этого, происходит замер времени работы алгоритма
def start_algorithm(fileName):
    startTime = time.time()

    graph = read_input_from_file(fileName)
    path = [0, [0]]
    calculate_lower_bound(graph, path)

    if not graph.isPathFound:
        return ['Гамильтонов путь не найден']

    path = [0, [0]]
    iterate_search(graph, path)

    for i in range(len(graph.path)):
        graph.path[i] += 1

    endTime = time.time()

    print(graph.path, graph.pathLength, endTime - startTime)


start_algorithm("test.txt")
```

ПРИЛОЖЕНИЕ Б.

ТЕСТИРОВАНИЕ

Таблица 1. Тестирование кода программы.

№	Входные данные	Вывод программы
1.	inf 1 2 2 - inf 1 2 - 1 inf 1 1 1 - inf	[1, 2, 3, 4, 1] 4 0.0
2.	inf - 1 1 1 - 1 inf - - 1 1 1 - inf 1 1 1 1 - 1 inf 1 1 1 - - 1 inf 1 1 - 1 - 1 inf	Гамильтонов путь не найден
3.	inf 7 9 - 3 1 - - - - 16 inf 4 46 - - 1 19 - - 13 34 inf 1 - - 15 - - - - - 14 inf - - 67 1 - - 17 - - - inf----13 1 - 19 1------ nf 78 91 12 - - 178 1 82---- nf 12 56 32 - 45 --- 1 - 37 inf 14 - - - - - 56 93 10 27 inf 1 1 23 76----- 45 76 34 inf	[1, 6, 2, 7, 3, 4, 8, 5, 9, 10, 1] 10 0.0
4.	inf 2 2 2 2 2 inf 2 2 2 2 2 inf 2 2 2 2 1 inf 2 2 2 2 1 inf	[1, 2, 5, 4, 3, 1] 8 0.000990152359 008789
5.	inf - 65 - 18 - 21 67 48 17 91 74 - 21 35 - 85 87 21 43 35 inf 88 - 24 43 46 75 - - 3 27 87 55 50 - 65 - 10 68 3 68 inf 24 44 28 19 17 13 66 43 93 38 - 42 34 58 - 91 36 12 50 75 inf 87 62 89 21 - 41 45 89 68 35 32 9 16 88 23 75 84 19 89 90 inf 93 69 52 - 3 62 62 23 - 77 93 68 24 20 38 17 77 48 19 70 inf - 43 - - 43 - 10 - 91 - 89 79 35 50 - - 93 19 94 - inf 20 - 79 43 82 - 7 61 - 49 - - - 7 1 76 - 64 20 1 inf 12 4 42 - 75 - 34 - 9 35 69 79 7 41 90 38 88 68 - 49 inf 91 87 50 58 81 - 47 48 - - - 21 20 72 97 90 - - - 50 inf - 47 - - 72 59 11 - - 41 - - 98 97 34 45 7 55 1 47 inf - 47 38 35 97 - 53 61 95 64 51 21 64 55 92 64 41 68 66 56 inf 70 - 77 84 55 87 82 48 95 23 49 54 88 34 - 97 18 76 43 40 inf 54 46 - 77 1 84 42 50 - 93 4 73 53 79 66 73 17 95 10 - inf 1 27 - 11 85 - 69 80 81 11 76 68 83 28 67 16 45 74 1 84 inf 74 81 - - - 20 54 97 47 16 - 56 80 42 84 20 83 76 62 61 inf 84 - 74 64 27 12 61 96 41 46 12 83 96 37 34 - 46 53 36 11 inf 13 87 49 94 70 50 4 75 58 96 - 24 9 - 76 10 61 16 98 - inf - 4 - 85 47 77 49 32 4 - 16 50 82 11 76 - - 92 70 - inf - 91 - 72 - 36 43 55 - 95 - 87 52 - 40 - - - 41 16 inf	[1, 7, 14, 15, 13, 18, 20, 19, 12, 3, 8, 6, 4, 16, 5, 10, 17, 2, 11, 9, 1] 201 36.2131829261779 8