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: Задача Коммивояжёра

Студент гр. 1381 _____ Возмитель В. Е

Преподаватель _____ Токарев А. П.

Санкт-Петербург

2023

Цель работы.

Изучить метод ветвей и границ на практике на примере решения задачи о нахождении пути коммивояжёра и его стоимости.

Задание.

Необходимо решить задачу коммивояжёра.

Входные данные: матрица весов графа, все веса неотрицательны; стартовая вершина.

Выходные данные: путь коммивояжёра (последовательность вершин) и его стоимость.

При сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную; для варианта 4 - симметричную), сохранять её в файл и использовать в качестве входных данных.

Вариант 2ц:

1) МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути:
   - полусуммы весов двух легчайших рёбер по всем вершинами
   - веса МОД
2) Приближённый алгоритм: АВБГ

Выполнение работы.

Для решения поставленных задач был реализован класс Graph(), содержащий следующие поля: matrix - двумерный массив, хранящий матрицу смежности графа, length - количество вершин в графе, start - стартовая вершина, chain - путь коммивояжёра, ok - переменная, отвечающая за нахождение ответа в задаче, weight - стоимость пути.

1) АВБГ.

Для реализации метода АВБГ были написаны следующие функции:

- *my_input(self)* - функция считывания входных данных. Запрашивает у пользователя матрицу, которую можно предварительно вписать в файл, и начальную вершину.

- *weight_count(self)* - подсчитывает текущую стоимость пути.
- *next_vertex(self, chain)* - функция поиска вершины для вставки в текущую цепочку. В цикле просматриваются все вершины-соседи цепочки. Вершина с наименьшим весом ребра, добавляется в цепочку.
- *chain_search(self, chain, edge)* - данная функция принимает аргументом цепочку и вершину для вставки. Идет проверка на достижение полной длины цепочки, если цепочка не достаточной дины, то функция рекурсивно вызывает саму себя.
- *my_print(self)* - форматированный вывод результата работы программы.

2) МВиГ.

Для оценки стоимости текущего пути были введены 2 нижние границы: оценка с помощью каркаса графа и с помощью полусуммы весов двух легчайших рёбер по всем вершинам.

Реализованы следующие функции:

- *my_input(self)* - функция считывания входных данных. Запрашивает у пользователя матрицу, которую можно предварительно вписать в файл, и начальную вершину.
- mod(self, viewed, tmp_ver, path_weight) - вес минимального остовного дерева. В остов последовательно добавляются минимальные ребра для вершин в текущей цепочке. После добавления ребер вершины, связанные с ними, добавляются в цепочку, параллельно суммируя вес всех ребер. К пройденному пути добавляется полученная сумма, она же и возвращается.
- *lightest(self, viewed, tmp_ver, path_weight)* - полусумма двух легчайших ребер по всем вершинам графа. К весу значение включаемого ребра прибавляются минимальные веса не просмотренных вершин. Отдельно считаются минимальные ребра для первой и последней вершин пути. К полученной полусумме добавляется стоимость уже пройденного пути. Функция возвращает значение границы.

- *chain_search(self, tmp_ver, viewed1, chain1, chain_weight, edge_weight)* - Основная рекурсивная функция программы. Если стоимость текущего пути превышает рекорд или найден новый рекорд, то происходит выход из рекурсии и во втором случае еще - обновление полей. В цикле while для текущей вершины составляется список соседей. Далее начинается перебор не рассмотренных соседей - если нижние оценки после включения соседа меньше рекорда, рекурсивно вызывается функция поиска пути с новой включенной вершиной.

Также для решения задач была реализована функция *init_matrix(n, lim)*, генерирующая случайную матрицу. Сгенерированная матрица записывается в файл mtrx.txt.

Кода программ расположены в приложении А.

Тестирование.

Для проверки работы программ были разработаны коды тестовых программ.

1) АВБГ.

- *test_1*. Данный тест был взят для проверки работы программы с пустой матрицей.

- *test_2*. Тест для проверки работы программы с разряженным графом.

- test_3. Тест для проверки работы программы с нормальными условиями.

2) МВиГ.

- test_4. Данный тест был взят для проверки работы программы с пустой матрицей.

- test_5. Данный тест был взят для проверки работы программы с нормальными условиями.

- test_6. Данный тест был взят для проверки работы программы с графом, имеющим единственный гамильтонов путь и равные ребра.

Код файла с тестами расположен в приложении А.

Выводы.

В ходе лабораторной работы изучен метод ветвей и границ на практике на примере решения задачи о нахождении пути коммивояжёра и его стоимости.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЕ КОДЫ ПРОГРАММЫ И ТЕСТЫ

Название файла: *PIAA_AVBG.py*

```
from generator import init_matrix
class Graph:
    def __init__(self):
        self.start = 0
        self.matrix = []
        self.chain = []
        self.weight = 0
        self.ok = 0
        self.length = 0

    def my_input(self):
        key = int(input("0 - generate random matrix, 1 - use file's matrix\n"))
        if key == 0:
            n = int(input("how many vertexes: "))
            limit = int(input("What is the limit for edge's weight: "))
            init_matrix(n, limit)
        self.start = int(input("Start vertex: "))
        f = open('mtrx.txt')
        for line in f:
            tmp = list(map(int, line.split()))
            self.matrix.append(tmp)
        self.length = len(self.matrix)
        if self.length > 0:
            print("\nmatrix:")
            for i in range(self.length):
                print(*self.matrix[i])

    def weight_count(self):
        for i in range(len(self.chain) - 1):
            self.weight += self.matrix[self.chain[i]][self.chain[i + 1]]

    def next_vertex(self, chain):
        mb_vertex = []
        item = (0, 0, 0)
        for elem in chain[:len(chain) - 1:]:
            ind = chain.index(elem)
            for j in range(self.length):
                if j not in chain and self.matrix[elem][j] != 0 and self.matrix[j][chain[ind + 1]]:
                    mb_vertex.append((elem, j, self.matrix[elem][j]))
        if mb_vertex:
            item = min(mb_vertex, key=lambda x: x[2])
        return item

    def chain_search(self, chain, edge):
        if edge != (0, 0, 0):
            ind = chain.index(edge[0])
            chain.insert(ind + 1, edge[1])
        if len(chain) == self.length + 1:
            self.chain = chain
            self.weight_count()
```

```
            self.ok = 1
            return

        mb_vertex = self.next_vertex(chain)
        if mb_vertex == (0, 0, 0):
            return
        self.chain_search(chain, mb_vertex)

    def my_print(self):
        if self.ok == 1:
            print("Chain: ", self.chain, "\nCost: ", self.weight)
        else:
            print("There's no the Hamilton's path in the matrix.")


if __name__ == '__main__':
    graph = Graph()
    graph.my_input()
    graph.chain_search([graph.start, graph.start], (0, 0, 0))
    graph.my_print()
```

Название файла: *PIAA_MVIG.py*

```
import copy
from generator import *
class Graph:
    def __init__(self):
        self.start = 0
        self.matrix = []
        self.chain = []
        self.weight = float('inf')
        self.ok = 0
        self.length = 0


    def my_input(self):
        key = int(input("0 - generate random matrix, 1 - use file's matrix\n"))
        if key == 0:
            n = int(input("how many vertexes: "))
            limit = int(input("What is the limit for edge's weight: "))
            init_matrix(n, limit)
        self.start = int(input("Start vertex: "))
```

```
    f = open('mtrx.txt')
    for line in f:
        tmp = list(map(int, line.split()))
        self.matrix.append(tmp)
    self.length = len(self.matrix)
    if self.length > 0:
        print("\nmatrix:")
        for i in range(self.length):
            print(*self.matrix[i])


def mod(self, viewed, tmp_ver, path_weight):
    viewed = copy.deepcopy(viewed)
    viewed.insert(0, self.start)
    edge = self.matrix[viewed[-1]][tmp_ver]
    viewed.append(tmp_ver)

    while len(viewed) < self.length:
        min_weight = float('inf')
        for ver in viewed:
            for i in range(self.length):
                if i not in viewed and self.matrix[ver][i] != 0 and self.matrix[ver][i] < min_weight:
                    tmp_ver = i
                    min_weight = self.matrix[ver][i]

        edge += min_weight
        viewed.append(tmp_ver)

    path_weight += edge
    return path_weight


def lightest(self, viewed, tmp_ver, path_weight):
    viewed = copy.deepcopy(viewed)
    viewed.insert(0, self.start)
    edge = self.matrix[viewed[-1]][tmp_ver]
    bound = edge
    viewed.append(tmp_ver)
```

```
    min_s, min_f = float('inf'), float('inf')
    for i in range(self.length):
        if (self.matrix[viewed[0]][i] > 0) and (min_s > self.matrix[viewed[0]][i]):
            min_s = self.matrix[viewed[0]][i]
        if (self.matrix[viewed[-1]][i] > 0) and (min_f > self.matrix[viewed[-1]][i]):
            min_f = self.matrix[viewed[-1]][i]


        if i not in viewed:
            min_weight1, min_weight2 = float('inf'), float('inf')
            for j in range(self.length):
                if min_weight1 > self.matrix[i][j] > 0:
                    min_weight1 = self.matrix[i][j]
                if min_weight1 < min_weight2:
                    tmp = min_weight1
                    min_weight1 = min_weight2
                    min_weight2 = tmp
            bound += min_weight1 + min_weight2
    bound += min_s + min_f
    bound /= 2
    bound += path_weight
    return bound


def chain_search(self, tmp_ver, viewed1, chain1, chain_weight, edge_weight):
    if tmp_ver == self.start and 1 < len(chain1) < self.length:
        return


    viewed = copy.deepcopy(viewed1)
    chain = copy.deepcopy(chain1)
    if tmp_ver != self.start:
        viewed.append(tmp_ver)
        chain.append(tmp_ver)
        chain_weight += edge_weight


    if chain_weight >= self.weight:
        return
```

```
        if tmp_ver == self.start and len(chain) == self.length and chain_weight < self.weight:
            self.ok = 1
            chain.append(self.start)
            self.chain = chain
            self.weight = chain_weight + edge_weight
            return


        while True:
            closest = []
            for i in range(self.length):
                if self.matrix[tmp_ver][i] > 0:
                    closest.append(i)
            for ver in closest:
                if ver not in viewed:
                    if self.ok == 0 or (self.mod(viewed, ver, chain_weight) < self.weight
                                        and self.lightest(viewed, ver, chain_weight) < self.weight):
                        self.chain_search(ver, viewed, chain, chain_weight, self.matrix[tmp_ver][ver])
            return


    def my_print(self):
        if self.ok == 1:
            print("Chain: ", self.chain, "\nCost: ", self.weight)
        else:
            print("There's no the Hamilton's path in the matrix.")



if __name__ == '__main__':
    graph = Graph()
    graph.my_input()
    graph.chain_search(graph.start, [], [graph.start], 0, 0)
    graph.my_print()
```

Название файла: PIAA_TESTS.py

```
import unittest
from PIAA_AVBG import Graph as AVBG
from PIAA_MVIG import Graph as MVIG
```

```
class TestMethods(unittest.TestCase):
    def test_1(self):
        graph = AVBG()
        graph.matrix = []
        graph.length = 0
        graph.start = 0
        graph.chain_search([graph.start, graph.start], (0, 0, 0))
        self.assertEqual(graph.weight, 0)

    def test_2(self):
        graph = AVBG()
        graph.matrix = [[0, 1, 2, 2], [0, 0, 1, 2], [0, 1, 0, 1], [1, 1, 0, 0]]
        graph.start = 2
        graph.length = 4
        graph.chain_search([graph.start, graph.start], (0, 0, 0))
        self.assertEqual((graph.chain, graph.weight), ([2, 3, 0, 1, 2], 4))

    def test_3(self):
        graph = AVBG()
        graph.matrix = [[0, 0, 5, 1, 0], [0, 0, 2, 0, 3], [5, 2, 0, 2, 2], [1, 0, 2, 0, 4], [0, 3, 2, 4, 0]]
        graph.start = 2
        graph.length = 5
        graph.chain_search([graph.start, graph.start], (0, 0, 0))
        self.assertEqual((graph.chain, graph.weight), ([2, 0, 3, 4, 1, 2], 15))

    def test_4(self):
        graph = MVIG()
        graph.matrix = []
        graph.length = 0
        graph.start = 0
        graph.chain_search(graph.start, [], [graph.start], 0, 0)
        self.assertEqual((graph.chain, graph.weight), ([], float('inf')))

    def test_5(self):
```

```
        graph = MVIG()
        graph.matrix = [[0, 3, 2, 9], [3, 0, 5, 5], [2, 5, 0, 2], [9, 5, 2, 0]]
        graph.length = 4
        graph.start = 3
        graph.chain_search(graph.start, [], [graph.start], 0, 0)
        self.assertEqual((graph.chain, graph.weight), ([3, 1, 0, 2, 3], 12))


    def test_6(self):
        graph = MVIG()
        graph.matrix = [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0]]
        graph.length = 4
        graph.start = 0
        graph.chain_search(graph.start, [], [graph.start], 0, 0)
        self.assertEqual((graph.chain, graph.weight), ([0, 1, 2, 3, 0], 4))



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