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: жадный алгоритм и алгоритм A*

Студент гр. 1303 _____ Смирнов Д. Ю.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2023

Цель работы.

Написать программу, которая решает задачу построения кратчайшего пути в ориентированном графе при помощи жадного алгоритма и алгоритма A*.

Задание.

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. Пример входных данных

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

abcde

Разработайте программу, которая решает задачу построения

кратчайшего пути в *ориентированном* графе **методом A***. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

ade

Вариант **6**. Реализация очереди с приоритетами, используемой в A*, через двоичную кучу.

Выполнение работы.

Весь код программ представлен в приложении А.

Описание алгоритма.

Жадный алгоритм:

Жадный алгоритм используется поиск в глубину. На каждом шаге выбирается смежное ребро с текущей вершиной имеющее наименьший вес, к текущему пути добавляем новую вершину и снова запускаем поиск в глубину, иначе поднимаемся к прошлой вершине. Если текущая вершина является конечной выходим из поиска сохранив найденный путь.

Алгоритм *A**:

В очередь приоритетом добавляем начальную вершину. Пока очередь не пуста, достаем из неё вершину с минимальной оценкой пути. Рассматриваем смежные с вершиной ребра рассчитываем временную цену пути в другие вершины по этим ребрам, если она меньше текущей цены или в вершину ещё не приходили обновляем её цену посещения и сохраняем как мы в неё пришли, также в очередь добавляем эту вершину и её стоимость с учетом эвристической функции (в данном случае: разница длин ASCII-кодов конечной и добавляемой вершины). Цена посещения вершины вычисляется, как сумма пути до текущей вершины + вес ребра из текущей вершины в вершину, которую хотят посетить + значение эвристической функции для вершины, которую хотят посетить. Если вершина на рассмотрении является конечной, поиск прекращаем. Если вершина является листом, то пропускаем её рассмотрение.

Минимальная куча:

Двоичная мин-куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: приоритет каждой вершины меньше приоритетов её потомков. Корень поддерева является минимальным из значений элементов поддерева. У каждой вершины есть не более двух потомков, а заполнение уровней вершин идет сверху вниз (в пределах одного уровня – слева направо). Куча хранится в виде

одномерного массива, причем левый потомок вершины с индексом *i* имеет индекс *2*i+1*, а правый *2*i+2*. Корень дерева – элемент с индексом *0*.

Новый элемент добавляется на последнее место в массиве, при этом возможно будет нарушено основное свойство кучи, так как новый элемент может быть меньше родителя. В таком случае следует «поднимать» новый элемент на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.

Доставая минимальный элемент (корень дерева) делаем следующее: ставим на его место последний элемент массива (будет максимальным) и восстанавливает основное свойство кучи для дерева, для этого необходимо «опускать» вершину (менять местами с наименьшим из потомков), пока основное свойство не будет восстановлено (процесс завершится, когда не найдется потомка, большего своего родителя).

Описание переменных, функций и минимальной кучи.

Для представления графа используется словарь *edges*, где ключом является вершина, из которой идут, а значениями список кортежей из двух элементов – цена ребра и буква вершины куда можно прийти.

Глобальные переменные:

- *DEBUG* – Флаг отвечающий за вывод дополнительной информации.

Функции:

Жадный алгоритм:

- *reader()* – Функция отвечает за считывание входных данных и сортировку ребер. Функция не принимает аргументов. Возвращает кортеж из 3-х элементов: стартовой вершины, конечной и словаря ребер графа.
- *modedDFS(path)* – внутренняя рекурсивная функция функции *greedyAlgorithm(startVertex, endVertex, edges)* осуществляет поиск в глубину. Аргумент *path* – путь на текущей итерации рекурсии.

Функция ничего не возвращает.

- g*reedyAlgorithm(startVertex, endVertex, edges)* – функция отвечает за жадный алгоритм описание было выше. Аргументы: *startVertex* – начальная вершина, *endVertex* – конечная вершина, *edges* – словарь ребер графа.

Алгоритм *A**:

- *heuristic(currentVertex, endVertex)* – эврестическая функция. Аргументы: *currentVertex* – текущая вершина, *endVertex* – конечная вершина. Функция возвращает разницу значений по таблице ASCII между буквами вершине по модулю.
- *aStarAlgorithm(startVertex, endVertex, edges)* – функция отвечает за алгоритм *A** описанный выше. Аргументы: *startVertex* – начальная вершина, *endVertex* – конечная вершина, *edges* – словарь ребер графа. Функция возвращает словарь посещенных вершин.
- *recoverPath(map, endVertex)* – функция востанавливает найденный путь по словарю посещенных вершин. Аргументы: *map* – словарь посещенных вершин, *endVertex* – конечная вершина.
- *reader()* – Функция отвечает за считывание входных данных. Функция не принимает аргументов. Возвращает кортеж из 3-х элементов: стартовой вершины, конечной и словаря ребер графа.

Минимальная куча:

Является классом и имеет следующие методы:

- *getParent(index)* – Статический метод отвечает за вычисление индекса родителя узла. Аргумент *index* – индекс узла. Метод возвращает родительский индекс.
- *getLeft(index)* – Статический метод отвечает за вычисление индекса левого ребенка узла. Аргумент *index* – индекс узла. Метод возвращает индекс левого ребенка.
- *getRight(index)* – Статический метод отвечает за вычисление индекса правого ребенка узла. Аргумент *index* – индекс узла. Метод возвращает индекс правого ребенка.
- *__siftUp(self, index)* – Приватный метод отвечает за просеивание узла вверх по куче. Аргументы *self* – указатель на объект класса, *index* – индекс узла. Метод ничего не возвращает.
- *__siftDown(self, index)* – Приватный метод отвечает за просеивание узла вниз по куче. Аргументы *self* – указатель на объект класса, *index* – индекс узла. Метод ничего не возвращает.
- *extractMin(self)* – Метод отвечает за извлечения минимального элемента кучи. Аргументы *self* – указатель на объект класса, *index* – индекс узла. Метод возвращает минимальный элемент из кучи.
- *insert(self, element)* – Метод отвечает за вставку элемента в кучу. Аргументы *self* – указатель на объект класса, *element* – вставляемый элемент. Метод ничего не возвращает.
- *size(self)* – Метод отвечает за возврат размера кучи. Аргумент *self* – указатель на объект класса. Метод возвращает размер кучи.
- *__repr__(self)* – Метод отвечает за представления объекта класса ввиде строки. Аргумент *self* – указатель на объект класса. Метод возвращает представления объекта класса ввиде строки.

Оценка сложности алгоритмов.

Жадный алгоритм:

В худшем случае алгоритм проходит весь полный граф по ребрам |E| – кол-во ребер, |*V*| – кол-во вершин, так как в полном графе |*E*| = |*V*|*(|*V*|-1)/2 ребер, каждое ребро проходят по одному разу (после того как прошли ребро удаляется), то оценка будет иметь вид O(|*V*|^2).

По памяти - использует словарь, состоящий из ребер графа, его оценка по памяти O(|*E*|), где |*E*| – кол-во рёбер в графе.

Алгоритм *A**:

Временная сложность алгоритма A* зависит от выбранной эвристики и от разветвлённости графа. В худшем случае количество рассматриваемых вершин на каждом шаге растёт экспоненциально по сравнению с длиной оптимального пути, то есть алгоритм работает за O(b^d), где b – коэффициент ветвления, а d – длина кратчайшего пути. Чем лучше эвристика, тем меньшие будет эффективный коэффициент ветвления b, и следовательно, меньше временная сложность.

По памяти – в худшем случае ему приходится помнить экспоненциальное количество узлов O(e^|*V*|), где |*V*|– кол-во вершин.

Минимальная куча:

Оценка основных операций – нахождение минимального элемента за O(1), удаление и вставка за O(log *n*), где *n* – количество элементов кучи.

По памяти: O*(n),* где *n* – количество элементов в куче.

Тестирование.

Таблица 1 – тестирование жадного алгоритма.

№ п/п	Входные данные	Выходные данные	Комментарий
1	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	abdeag	Верно.
2	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	abdefg	Верно.

Таблица 2 – тестирование алгоритма *A**.

№ п/п	Входные данные	Выходные данные	Комментарий
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Верно.
2	a g a b 3.0 b d 2.0 b e 3.0 d e 4.0 e f 2.0 a g 60.0 f g 1.0	abefg	Верно.

Пример вывода дополнительной информации на примере из условия задачи для алгоритмов, представлен на рисунках 1-2 соответственно.

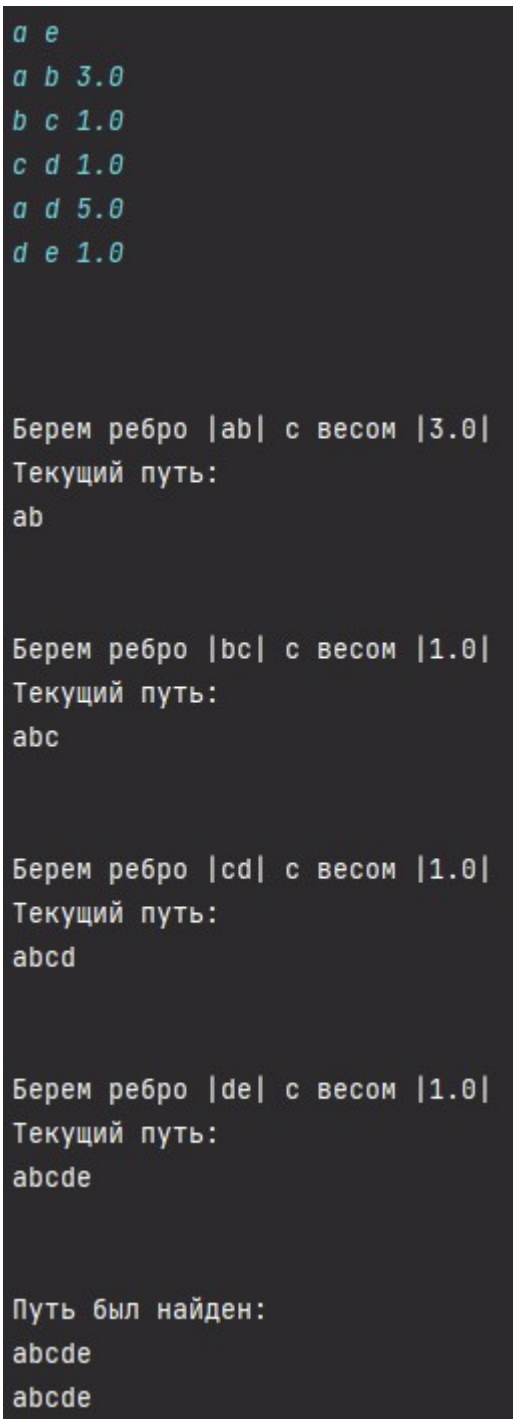

Рисунок 1 - Вывод дополнительной информации на примере из условия для жадного алгоритма

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

Очередь на текущей итеарции имеет вид:
    (0, 'a')

Обновляется решение!
    Словарь посещенных вершин:
     {'a': None, 'b': 'a'}
    Словарь цен пути в вершины:
     {'a': 0, 'b': 3.0}
    Очередь имеет вид:
    (6.0, 'b')

Обновляется решение!
    Словарь посещенных вершин:
     {'a': None, 'b': 'a', 'd': 'a'}
    Словарь цен пути в вершины:
     {'a': 0, 'b': 3.0, 'd': 5.0}
    Очередь имеет вид:
    (6.0, 'b')
    (6.0, 'd')

Очередь на текущей итеарции имеет вид:
    (6.0, 'b')
    (6.0, 'd')
```

Рисунок 2 – Отрывок вывода дополнительной информации на примере из условия для алгоритма $A*$

Вывод.

Написана программа, которая решает задачу построения кратчайшего пути в ориентированном графе при помощи жадного алгоритма и алгоритма A*. Также была оценена сложность каждого алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММ

Название файла: greedy.py

```
import sys

DEBUG = False


def reader():
    """
    Функция считывает стартовую и конечную вершины. После чего считывает ребра из
    стандартного потока ввода, пока в него подают.
    Разбивает ввод и заносить в словарь ребер ребра.
    После чего происходит сортировка ребер, по их весу.
    :return: кортеж из 3х элементов: стартовая вершина, конечная вершина, словарь ребер графа
    """
    global DEBUG
    start, end = input().split()
    edges = dict()

    for line in sys.stdin:
        if line == '\n':
            break
        source, destination, weight = line.split()
        if source not in edges.keys():
            edges[source] = list()
        edges[source].append((destination, float(weight)))

    for key in edges.keys():
        edges[key].sort(key=lambda edge: edge[1])
    return start, end, edges


def greedyAlgorithm(startVertex, endVertex, edges):
    """
    Жадный поиск пути от стартовой вершины к конечной. Используется поиск в глубину.
    :param startVertex: стартовая вершина
    :param endVertex: конечная
    :param edges: словарь ребер
    :return: Возвращает путь от стартовой до конечной вершины.
    """
    result = ""  # В данной переменной после поиска в глубину будет храниться конечное решение.
    found = False  # Переменная 'флаг', необходима для приостановки поиска в глубину, когда решение было найдено.

    def modedDFS(path):
        """
        Модифицированный поиск в глубину.
        Смотрим текущую смежные ребра с текущей вершиной.
        Пока они есть удаляем из них минимальное и добавляем к текущему пути
```

```
        букву вершины куда ведет эторебро.
        :param path: путь на текущей интерации рекурсии
        :return: ничего не возвращает
        """
        nonlocal result, found, endVertex, edges
        global DEBUG
        if found:  # выход из рекурси если путь был найден
            return
        if path[-1] == endVertex:  # нашли путь от стартовой до
конечной вершины
            result = path  # сохраняем его
            found = True  # поднимаем флаг, ибо нашли путь
            if DEBUG:
                print('\n')
                print("Путь был найден:")
                print(f"{result}")
            return
        if edges.get(path[-1]) is None and DEBUG:
            print('\n')
            print(f"Вершина |{path[-1]}| является листом графа")
        while edges.get(path[-1]) and edges[path[-1]] is not None:
# пока есть смежные ребра с текущей вершиной
            newEdge = edges[path[-1]].pop(0)  # берем минимальное из
них
            if DEBUG and not found:
                print('\n')
                print(f"Берем ребро |{path[-1]}{newEdge[0]}| с весом
|{newEdge[1]}|")
                print("Текущий путь:")
                print(path + newEdge[0])
            modedDFS(path + newEdge[0])  # запускаем поиск добавив к
текущему пути вершину куда это ребро ведет.

    modedDFS(startVertex)  # вызываем поиск в глубину из стартовой
вершины
    return result  # возвращаем найденный путь


def main():
    data = reader()  # Считываем данные
    print(greedyAlgorithm(*data))  # Передаем их алгоритму, после
чего печатаем ответ.


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

Название файла: binaryHeap.py

```
class Heap:
    def __init__(self, arr=None):
        "Инициализация объектс класса"
        if arr is None:
            arr = []
        self.__heap = []
        for el in arr:
            self.insert(el)
```

```
    @staticmethod
    def getParent(index):
        "Получение индекса родителя текущей вершины"
        return (index - 1) // 2

    @staticmethod
    def getLeft(index):
        "Получение индекса левого ребенка"
        return 2 * index + 1

    @staticmethod
    def getRight(index):
        "Получение индекса правого ребенка"
        return 2 * index + 2

    def __siftUp(self, index):
        """
        Просеивает вверх узел
        :param index: индекс вершины которую хотят просеить вверх
        :return: ничего возвращает
        """
        if index < 0 or index >= len(self.__heap):
            return
        parent = self.getParent(index)
                    while index and not self.__heap[parent] <
self.__heap[index]:
                            self.__heap[parent], self.__heap[index] =
self.__heap[index], self.__heap[parent]
            index, parent = parent, self.getParent(index)

    def __siftDown(self, index):
        """
        Просеивает вниз узел
        :param index: индекс вершины которую хотят просеить вниз
        :return: ничего возвращает
        """
        if index < 0 or index >= len(self.__heap):
            return
        minIndex = index
        while True:
            left, right = self.getLeft(index), self.getRight(index)
                if right < len(self.__heap) and self.__heap[right] <
self.__heap[minIndex]:
                minIndex = right
                 if left < len(self.__heap) and self.__heap[left] <
self.__heap[minIndex]:
                minIndex = left
            if minIndex == index:
                return
            else:
                            self.__heap[index], self.__heap[minIndex] =
self.__heap[minIndex], self.__heap[index]
                index = minIndex

    def extractMin(self):
        """
            Достает минимальеый ставит максимальный на его место и
```

```
просеивает вниз.
        :return: возвращает минимальный элемент
        """
        if not self.__heap:
            return
        min_element = self.__heap[0]
        self.__heap[0] = self.__heap[-1]
        del self.__heap[-1]
        self.__siftDown(0)
        return min_element

    def insert(self, element):
        """
        Добавляет элемент ставит в конец и просеивает вверх его.
        :return: ничего возвращает
        """
        self.__heap.append(element)
        self.__siftUp(len(self.__heap) - 1)

    def size(self):
        """
        :return: возвращает размер кучи
        """
        return len(self.__heap)

    def __repr__(self):
        representation = ""
        for item in self.__heap:
            representation += '\t' + str(item) + "\n"
        return representation
```

Название файла: aStar.py

```
import sys
from binaryHeap import Heap

DEBUG = False


def heuristic(currentVertex, endVertex):
    """
    Эврестическая функция - близость символов,
    обозначающих вершины графа, в таблице ASCII.
    Считается следующим образом: разница кодов конечной вершины с
текущей по модолю.
    :param currentVertex: текущая вершина
    :param endVertex: конечная вершина
    :return: возвращает значение эврестической функции
    """
    return abs(ord(endVertex) - ord(currentVertex))


def aStarAlgorithm(startVertex, endVertex, edges):
    """
    Функция реализует алгоритм A*. Создает словарь расстояний, и
словарь корней.
    Создает очередь с приоритетом на куче.
    Пока куча не пустая, достаем вершину с минимальной ценной из
```

```
очереди. Если это конечная прекращаем поиск,
    если эта вершина лист пропускаем рассмотрение этой вершины,
    иначе рассматриваем ребра смежные с этой вершиной, берем вершину
куда ведет это ребро, считаем вес пути в данную вершину
    если такую вершину не рассматривали или вес получился более
оптимальный  обновляем словари и добавляем в очередь эту вершину.
    :param startVertex: начальная вершина
    :param endVertex: конечная вершина
    :param edges: словарь ребер графа
    :return: словарь корней (ключ - куда пришли, значение - откуда)
    """
    global DEBUG
    distances = dict({startVertex: 0})
    roots = dict({startVertex: None})
    queue = Heap([(0, startVertex)])
    while queue.size() != 0:
        if DEBUG:
            print("Очередь на текущей итеарации имеет вид:")
            print(queue)
        current = queue.extractMin()[1]
        if current == endVertex:
            if DEBUG:
                print('Дошли до конечной вершины!')
            break
        if current not in edges:
            if DEBUG:
                print(f"Вершина |{current}| является листом графа")
            continue
        for nextVertex, weight in edges[current]:
            temp_dist = distances[current] + weight
            if nextVertex not in distances or temp_dist <
distances[nextVertex]:
                roots[nextVertex] = current
                distances[nextVertex] = temp_dist
                queue.insert((temp_dist + heuristic(nextVertex,
endVertex), nextVertex))
                if DEBUG:
                    print(f"Обновляется решение!")
                    print("\tСловарь посещенных вершин:")
                    print("\t", roots)
                    print("\tСловарь цен пути в вершины:")
                    print("\t", distances)
                    print("\tОчередь имеет вид:")
                    print(queue)
    return roots


def recoverPath(map, endVertex):
    """
    Функция востанавливает путь пройденный A*. Начинает
востанавливать с конечной вершины по старт.
    :param map: словарь путей
    :param endVertex: конечная вершина
    :return: возвращает путь от стартовой вершины до конечной
    """
    path = ''
    current = endVertex
```

```
    while current:  # пока существует вершина из которой пришли
        path += current  # к пути добавляем текущую
        current = map[current]  # берем вершину откуда пришли в
текущую
    return path[::-1]  # переворачиваем путь


def reader():
    """
    Функция считывает стартовую и конечную вершины. После чего
считывает ребра из
    стандартного потока ввода, пока в него подают.
    Разбивает ввод и заносить в словарь ребер ребра.
    :return: кортеж из 3х элементов: стартовая вершина, конечная
вершина, словарь ребер графа
    """
    start, end = input().split()
    edges = dict()

    for line in sys.stdin:
        if line == '\n':
            break
        source, destination, weight = line.split()
        if source not in edges.keys():
            edges[source] = list()
        edges[source].append((destination, float(weight)))
    return start, end, edges


if __name__ == "__main__":
    data = reader()  # считываем данные
    """
    Предаем данные в алгоритм A*,
    после чего востанавливаем путь проёденный алгоритмом,
    печатаем результат
    """
    print(recoverPath(aStarAlgorithm(*data), data[1]))
```