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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Жадный алгоритм и A*

Студент гр. 8382 _____ Нечепуренко Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Разработать программу для поиска минимального пути в графе между двумя заданными вершинами, используя жадный алгоритм и алгоритм A*.

Постановка задачи.

Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма (алгоритма A*). 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 (ade)

Индивидуальное задание: Вар. 2. В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Описание алгоритма.

Жадный алгоритм:

На каждом шаге выбирается последняя посещенная вершина, выбирается соседняя не посещённая вершина с минимальным весом ребра. Процесс повторяется до тех пор, пока не будет обработана конечная вершина или вершины закончатся. Такой алгоритм не гарантирует нахождение оптимального решения при его существовании.

Алгоритм A*:

Данный алгоритм во многом схож с алгоритмом Дейкстры. На каждом шаге проверяем, меньше ли расстояние до соседа через текущую вершину, и обновляем наименьший путь до соседней вершины. Текущая вершина на каждой итерации должна иметь минимальную дистанцию от начальной вершины, т.е. должна поддерживаться очередь с приоритетами с возможностью извлечения минимума. Отличие алгоритма A* от алгоритма Дейкстры заключается в том, что к приоритету вершины прибавляется значение некоторой эвристической функции от этой вершины до целевой. Эвристическая функция должна быть монотонна и допустима, иначе алгоритм A* будет асимптотически хуже алгоритма Дейкстры. Правильно подобранная эвристическая функция в ряде прикладных задач позволяет значительно ускорить поиск пути, уменьшив фронт вершин поиска. Алгоритм A* гарантирует нахождение оптимального решения, если оно существует, при условии, что эвристическая функция допустима, монотонна и определена в тех же единицах измерений, что и веса ребер. A* отличается от жадного алгоритма тем, что учитывает уже построенный путь и некоторую топологическую оценку нахождения целевой вершины.

Особенности реализации алгоритма.

Будем хранить граф в виде словаря, где ключом будет вершина из которой выходят ребра, а значением – массив упорядоченных пар: в какую вершину ведет ребро и вес данного ребра.

Для реализации жадного алгоритма сначала для каждой вершины отсортируем ребра по неубыванию. Затем в рекурсивной функции dfs с помощью обхода в глубину обойдем граф и, если дошли до целевой вершины, выведем ответ.

Для реализации алгоритма A* потребуется очередь с приоритетами. Воспользуемся стандартной библиотекой языка python. Из модуля queue импортируем PriorityQueue. Данная структура данных оперирует кортежами и сравнивает элементы по первому элементу кортежа, позволяя осуществлять добавление и получение минимума за логарифмическое время. Будем хранить несколько словарей: visited для обработанных вершин, distance для длины пути до вершины (изначально равны ∞), path для восстановления пути. Для выполнения задания на stepik была использована эвристическая функция разности между кодами символов. Для индивидуального задания значения эвристической функции задаются пользователем. Вариант не предполагает проверку данной функции на допустимость и монотонность.

Полный исходный код с комментариями находится в Приложении А.

Функции и структуры данных.

Для хранения графа используется встроенная структура данных dict (см. Особенности реализации алгоритма). Значением является пара float, string:

```
graph = dict()
[(cost, v)]
```

Для хранения посещенных вершин, длины пути до вершины, предыдущей вершины в пути для заданной, а также для таблицы эвристической функции используется dict:

```
heuristic = dict()
path = dict()
visited = dict()
distance = {key: inf for key in vertex}
```

В переменной vertex хранится множество всех вершин графа:

```
vertex = set()
```

Очередь с приоритетами возьмем из стандартной библиотеки:

```
pQueue = queue.PriorityQueue()
```

Для реализации жадного алгоритма воспользуемся функцией dfs:

```
def dfs(cur, to)
```

Первым аргументом идет текущая обрабатываемая вершина, вторым целевая вершина. Результатом работы функции является вывод в stdout строки, содержащей путь до конечной вершины.

Для восстановления пути в алгоритме A* используется функция reconstructPath:

```
def reconstructPath(path, currentNode)
```

Она принимает первым аргументом словарь предыдущих для некоторой вершины вершин, вторым аргументом текущую вершину. Результатом работы функции является строка, содержащая путь от начальной вершины до текущей.

Для эвристической функции задания на stepik была реализована функция heuristic:

```
def heuristic(currentNode, to)
```

Принимающая некоторую вершину и целевую вершину и возвращающая модуль разности кодов символов вершинах.

Тестирование программы.

Проведем тестирование алгоритма A* с выводом отладочной информации и жадного алгоритма (см. табл. 1 и 2).

Таблица 1 – Тестирование алгоритма A*.

Входные данные	**Выходные данные**
7 6 a f a b 1 a c 2 b c 1 c d 2 b e 3	Current node is a Updating path to b through a Now path to b is: ab Updating path to c through a Now path to c is: ac Current node is b Updating path to e through b

d f 4 e f 1 a 6 b 4 c 4 d 1 e 2 f 0	Now path to e is: abe Current node is c Updating path to d through c Now path to d is: acd Current node is d Updating path to f through d Now path to f is: acdf Current node is e Updating path to f through e Now path to f is: abef Current node is f Answer is: abef
11 9 a i a b 3 a c 2 b e 2 b f 1 c d 4 c h 2 e g 5 f h 1 d i 17 h g 1 g i 7 a 11 b 9 c 6 d 3 e 7 f 6.5 g 4 h 6 i 0	Current node is a Updating path to b through a Now path to b is: ab Updating path to c through a Now path to c is: ac Current node is c Updating path to d through c Now path to d is: acd Updating path to h through c Now path to h is: ach Current node is d Updating path to i through d Now path to i is: acdi Current node is h Updating path to g through h Now path to g is: achg Current node is g Updating path to i through g Now path to i is: achgi Current node is b Updating path to e through b Now path to e is: abe Updating path to f through b Now path to f is: abf Current node is f Current node is e Current node is i Answer is: achgi

Таблица 2 – Тестирование жадного алгоритма.

Входные данные	**Выходные данные**
7 6 a f a b 1 a c 2 b c 1 c d 2 b e 3 d f 4 e f 1 a 6 b 4 c 4 d 1 e 2 f 0	abcdf
11 9 a i a b 3 a c 2 b e 2 b f 1 c d 4 c h 2 e g 5 f h 1 d i 17 h g 1 g i 7 a 11 b 9 c 6 d 3 e 7 f 6.5 g 4 h 6 i 0	achgi

Оценка сложности алгоритма.

Для жадного алгоритма имеем по времени выполнения: сортировка ребер |E|log|E|, просмотр каждой вершины и каждого ребра |V|+|E|. Итого O(|V| + |E|log|E|). По памяти O(|V|+|E|).

Для алгоритма A* оценка по памяти O(|V|+|E|). Оценка по времени зависит от эвристической функции и используемой структуры для хранения очереди, списка посещенных вершин и т.п. Вставка и получение минимума используются для каждой вершины по одному разу. Получаем |V|log|V| для очереди с приоритетами с логарифмическими операциями. Так же, просматривается каждое ребро. Получается O(|E|+|V|log|V|). Хорошая эвристическая функция способна уменьшить время выполнения в константное число раз, что не влияет на асимптотику, но может значительно повлиять на реальное время работы в прикладной задаче. Плохая эвристическая функция может иметь отрицательное значение или скачки, что снижает эффективность выбора текущей вершины и нахождения оптимального пути в целом. При этом в любом случае будет просмотрена каждая вершина один раз, но не будут отсечены заведомо неперспективные ребра, что приблизит время выполнения к наихудшей оценке.

Вывод.

В результате выполнения работы была разработана программа для нахождения минимального пути во взвешенном графе с помощью алгоритма A* и жадного алгоритма. Была проанализирована асимптотика данных алгоритмов, а также их корректность.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ.

```
import queue
from sys import stdin
import operator

DEBUG = True

inf = float("inf")
"""def heuristic(currentNode, to):
    return abs(ord(currentNode)-ord(to))"""

def reconstructPath(path, currentNode):
    """ восстановление пути """
    answer = [currentNode]
    while path.get(currentNode):
        answer.append(path[currentNode])
        currentNode = path[currentNode]
    return "".join(reversed(answer))

pQueue = queue.PriorityQueue() #очередь с приоритетами
edg, vert = map(int, input().strip().split()) # количество ребер графа и вершин
с эвристиками
from_, to_ = input().strip().split() # две вершины откуда и куда строить путь
graph = dict(); vertex = set()
for _ in range(edg):
    """ чтение графа """
    u, v, c = input().strip().split()
    cost = float(c);
    graph[u] = graph.get(u, []) + [(cost, v)]
    vertex.add(u); vertex.add(v)

heuristic = dict()
for _ in range(vert):
    """ чтение эвристик """
    u, h = input().strip().split()
    curNodeHeuristic = float(h)
    if curNodeHeuristic < 0.:
        raise Exception(f"Incorrect heuristic value for node {u}")
    heuristic[u] = curNodeHeuristic

path = dict(); visited = dict()

""" начало жадного алгоритма
for k, v in graph.items():
    graph[k] = sorted(v, key=operator.itemgetter(1))
def dfs(cur, to):
    visited[cur] = True
    if cur == to:
```

```
        print(cur)
        exit()
    if not graph.get(cur, False):
        return
    print(cur, end="")

    for v, c in graph[cur]:
        if not visited.get(v, False):
            dfs(v, to)
конец жадного алгоритма """

distance = {key: inf for key in vertex} # изначально все расстояния равны оо
distance[from_] = 0
pQueue.put((0, from_)) # добавляем стартовую вершину
while not pQueue.empty():
    _, currentNode = pQueue.get()
    if visited.get(currentNode, False):
        continue # уже обработанная вершина
    visited[currentNode] = True
    if DEBUG:
        print(f"Current node is {currentNode}")
    if currentNode == to_:
        break # нашли путь
    if not graph.get(currentNode, False):
        continue # из узла нет ребер
    for node in graph[currentNode]:
        cost, next_ = node
        tentative_score = distance.get(currentNode, inf) + cost # путь до
следующей вершины
        heuristic_score = tentative_score + heuristic.get(next_, inf)# оценка с
использованием эвристики #heuristic(next_, to_)
        if tentative_score < distance.get(next_, inf):
            path[next_] = currentNode
            pQueue.put((heuristic_score, next_))
            if DEBUG:
                print(f"Updating path to {next_} through {currentNode}")
                print(f"Current distance to {next_} is {distance[next_]}")
                print(f"New distance to {next_} is {distance[currentNode]} +
{cost}")
                print(f"Now path to {next } is: ", end="")
                print(reconstructPath(path, next_))
            distance[next_] = tentative_score

""" восстанавливаем путь """
print("Answer is: ", end="")
print(reconstructPath(path, to_))
```