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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритмы на графах

Студент гр. 9383 Поплавский И.

Преподаватель Фирсов М.А.

Санкт-Петербург

2021

Цель работы

Изучить принцип работы алгоритмов на графах на примерах жадного алгоритма и A*. Решить с их помощью задачи на языке программирования Python.

Основные теоретические положения

Жадный алгоритм – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Поиск A* – алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной). Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость». Эта функция – сумма двух других: функции стоимости достижения рассматриваемой вершины из начальной, и функции эвристической оценки расстояния от рассматриваемой вершины к конечной.

Постановка задачи

Жадный алгоритм

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

Алгоритм A*

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

Вар. 7. Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Реализация задачи

Описание алгоритма

Была написана программа, которая реализует поиск пути в ориентированном взвешенном графе с помощью жадного алгоритма.

Программа работает следующим образом. При вводе данных для каждой пары вершин вызывается функция, которая инициализирует граф в виде списка смежности.

После инициализации графа вызывается рекурсивная функция, которая реализует жадный поиск. В начале функции выполняется проверка на конечную вершину. В случае, если конечная вершина была достигнута, происходит выход из рекурсивной функции. Иначе вызывается функция для поиска соседа с наименьшим весом ребра, после чего рекурсивная функция вызывается заново с новой стартовой вершиной. Поиск пути с помощью жадного алгоритма не гарантирует нахождения кратчайшего пути в графе.

Сложность по времени — O(|V|+|E|), т. к. нужно просмотреть все ребра и найти ребро минимального веса. Сложность по памяти — линейная от числа вершин и ребер $O(|V|^2)$, т. к. необходимо хранение всех вершин и ребер.

Была написана программа, которая реализует поиск кратчайшего пути в графе с помощью алгоритма A*.

Программа работает следующим образом. При вводе данных для каждой пары вершин вызывается функция, которая инициализирует граф в виде списка смежности. Так же в функции для создания графа вычисляется эвристическое значение, которое обозначает близость символов, обозначающих вершины графа, в таблице ASCII.

После создания графа вызывается функция, реализующая алгоритм A*. В теле этой функции выполняется проверка на достижение конечной вершины. Если конечная вершина еще не достигнута, тогда алгоритм вычисляет приоритет достижения соседних вершин.

Далее в функции Astar выбирается самая приоритетная вершина, и поиск выполняется заново с текущей вершины. В конце функции, когда путь был найден, он записывается в список, который хранит путь.

Сложность по памяти — линейная от количества вершин и ребер O(|V|+|E|), т. к. необходимо хранить путь.

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

Описание функций и структур данных

Класс Queue – является очередью пар с приоритетом. Включает в себя стандартные методы очереди, но при добавлении новой пары очередь сортируется.

Класс Graph представляется в виде словаря вершин с ребрами, которые его связывают.

Методы класса Graph:

- Add_edge(head, leave, value) – функция добавления вершины с ребром, принимает на вход head – вершина из которой будет проведено ребро, leave – вершина, к которой проведут ребро и value – вес ребра. Ничего не возвращает
- print_graph() – функция вывода списка смежности для вершин переданного графа graph.
- a_star(start, end) – основная функция поиска пути в графе. В алгоритме используется очередь с приоритетом. Если верхний элемент очереди равен итоговой вершине, то работа алгоритма окончена и возвращаем

priority_sort(tree, end) – функция выполняющая предобработку графа: для каждой вершины сортирует список смежных вершин по приоритету.

Тестирование (Алгоритм A*)

Входные данные	Выходные данные
a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	abef
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 5.5	ade

Тестирование (Жадный алгоритм)

Входные данные	Выходные данные
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde
a b a c 1	acb

a d 2 c f 3 f g 4 g c 1 g b 5 c b 1	
a b a c 1 a d 2 c f 3 f g 4 g c 1 g b 5 g d 1 d b 1	acfgdb

Также было написано юнит тестирование, в котором проверяется корректная работоспособность очереди, добавление ребер графа и тестирование всей программы на вход-выход.

Выводы.

В результате работы была написана полностью рабочая программа, решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны.

ПРИЛОЖЕНИЕ А (АЛГОРИТМ А*)

```
from sys import stdin
from collections import namedtuple
pair = namedtuple('pair', ['first', 'second'])

class Queue:
    def __init__(self):
        self.__data = []

    def __compare(self, a, b):
        if a.second == b.second:
            return a.first < b.first
        else:
            return a.second > b.second

    def top(self):
        return self.__data[-1]

    def push(self, el):
        self.__data.append(el)
        self.__sort()

    def __sort(self):
        for i in range(len(self.__data) - 1):
            for j in range(len(self.__data) - i - 1):
                if not self.__compare(self.__data[j], self.__data[j + 1]):
                    self.__data[j], self.__data[j + 1] = self.__data[j + 1], self.__data[j]

    def pop(self):
        self.__data.pop()

    def empty(self):
        return len(self.__data) == 0


class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, head, leave, value):
        if head not in self.graph:
```

```
            self.graph[head] = {}
        self.graph[head][leave] = value

    def print_graph(self):
        print(self.graph)

    def a_star(self, start, end):
        shortPath = {}
        queue = Queue()
        queue.push(pair(start, 0))
        vector = [start]
        shortPath[start] = (vector, 0)
        while not queue.empty():
            if queue.top().first == end:
                return shortPath[end][0]
            temp = queue.top()
            print("Верхний элемент очереди равен {}".format(queue.top()))
            print("Текущая вершина {}".format(temp[0]))
            queue.pop()
            if temp.first in self.graph:
                for i in list(self.graph[temp.first].keys()):
                    currentPathLength = shortPath[temp.first][1] +
self.graph[temp.first][i]
                    if i not in shortPath or shortPath[i][1] > currentPathLength:
                        path = []
                        for j in shortPath[temp.first][0]:
                            path.append(j)
                        path.append(i)
                        shortPath[i] = (path, currentPathLength)
                        evristic = abs(ord(end) - ord(i))
                        queue.push(pair(i, evristic + shortPath[i][1]))
        return shortPath[end][0]


def priority_sort(tree, end):
    for node in tree.graph.items():
        temp_list = sorted(list(node[1]), key=lambda x: abs(ord(end) - ord(x[0])))
        edges = dict()
        for x in temp_list:
            edges[x] = node[1][x]
        tree.graph[node[0]] = edges
    return tree

if __name__ == '__main__':
```

```
    data = []
    for line in stdin:
        data.append(line.split())

    tree = Graph()
    for i in range(len(data)):
        if i > 0:
            tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
    tree = priority_sort(tree, data[-1][1])

    ans = tree.a_star(data[0][0], data[0][1])
    for i in ans:
        print(i, end=")


data = []
for line in stdin:
    data.append(line.split())

tree = Graph()
for i in range(len(data)):
    if i > 0:
        tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
tree = priority_sort(tree, data[-1][1])

ans = tree.a_star(data[0][0], data[0][1])
for i in ans:
    print(i, end=")
```

ПРИЛОЖЕНИЕ В (ЖАДНЫЙ АЛГОРИТМ)

```
from sys import stdin
import numpy as np

class Graph:
    def __init__(self):
        self.graph = {}
        self.node_in_graph = []

    def add_edge(self, head, leave, value):
        if head not in self.graph:
            self.graph[head] = {}
        self.graph[head][leave] = value

 def print_graph(self):
        print(self.graph)

    def preparing(self, start, end):
        done = []
        check = False
        ans = []

        while not check:
            ans, done = self.greedy(start, end, done)
            if ans[-1] == end:
                check = True
        return ans

    def greedy(self, start, end, done):
        key = start
        ans = []
        while key in self.graph and any(self.graph[key]):
            ans.append(key)
            min = np.Inf
            next = None
            for i in self.graph[key]:
                if min > self.graph[key][i] and i not in done:
                    if i in self.graph:
                        next = i
                        min = self.graph[key][i]
```

```
        elif i == end:
            next = i
            min = self.graph[key][i]
    key = next
```

```
            done.append(key)
            if key == end:
                ans.append(key)
                return ans, done
        return ans, done

    def get_graph(self):
        return self.graph


if __name__ == "__main__":
    a_lst = []
    for line in stdin:
        a_lst.append(line.split())
    tree = Graph()
    for i in range(len(a_lst)):
        if i > 0:
            tree.add_edge(a_lst[i][0], a_lst[i][1], float(a_lst[i][2]))
    ans = tree.preparing(a_lst[0][0], a_lst[0][1])
    for i in ans:
        print(i, end=")
```

ПРИЛОЖЕНИЕ С (ТЕСТЫ ДЛЯ A*)

```
import unittest
from Astar import *
class TestCaseAstar(unittest.TestCase):
	def test1(self):
		queue = Queue()
		queue.push((1, 3))
		self.assertEqual(queue.top(), (1, 3))

	def test2(self):
		queue = Queue()
		queue.push((4, 2))
		queue.push((4, 6))
		queue.push((6, 2))
		self.assertEqual(queue.empty(), 0)

	def test3(self):
		tree = Graph()
		tree.add_edge('a', 'z', 5.2)
		tree.add_edge('a', 'c', 52.4)
		priority_sort(tree, 'f')
		self.assertEqual(tree.graph['a'], {'c': 52.4 ,'z': 5.2})

	def test4(self):
		data = [['a', 'e'],
				['a', 'b', 3.0],
				['b', 'c', 1.0],
				['c', 'd', 1.0],
				['a', 'd', 5.0],
				['d', 'e', 1.0]]
		tree = Graph()
		for i in range(len(data)):
		  if i > 0:
		    tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
		tree = priority_sort(tree, data[-1][1])
		ans = tree.a_star(data[0][0], data[0][1])
		self.assertEqual(''.join(ans), 'ade')
```

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

ПРИЛОЖЕНИЕ Д (ТЕСТЫ ДЛЯ ЖАДНОГО АЛГОРИТМА)

```
import unittest
from greed import *

class TestCaseGreed(unittest.TestCase):
        def test1(self):
                tree = Graph()
                tree.add_edge('a', 'b', 3.7)
                tree.add_edge('g', 'b', 2.5)
                self.assertEqual(tree.graph['a']['b'], 3.7)

        def test2(self):
                data = [['a', 'e'],
                                ['a', 'b', 3.0],
                                ['b', 'c', 1.0],
                                ['c', 'd', 1.0],
                                ['a', 'd', 5.0],
                                ['d', 'e', 1.0]]
                tree = Graph()
                for i in range(len(data)):
                   if i > 0:
                      tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
                ans = tree.preparing(data[0][0], data[0][1])
                self.assertEqual(''.join(ans), "abcde")


        def test3(self):
                data = [['a', 'e'],
                                ['a', 'b', 4.2],
                                ['b', 'd', 3.0],
                                ['a', 'b', 3.4],
                                ['d', 'e', 1.0]]
                tree = Graph()
                for i in range(len(data)):
                   if i > 0:
                      tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
                ans = tree.preparing(data[0][0], data[0][1])
                self.assertEqual(''.join(ans), "abde")
```

```
def test4(self):
        data = [['a', 'f'],
                        ['a', 'b', 3.0],
```

```
                    ['b', 'd', 4.0],
                    ['c', 'f', 6.3],
                    ['a', 'c', 4.0]]
            tree = Graph()
            for i in range(len(data)):
                if i > 0:
                    tree.add_edge(data[i][0], data[i][1], float(data[i][2]))
            ans = tree.preparing(data[0][0], data[0][1])
            self.assertEqual(''.join(ans), "acf")

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