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 поиска пути в графах

Студент гр. 9382 _____ Дерюгин Д.А.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

Познакомиться с двумя алгоритмами поиска пути в графе: жадный алгоритм и алгоритм A*. Реализовать эти два алгоритма.

Задание.

Вариант 8. Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

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 неотрицательныйвес.

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

Описание алгоритмов.

Жадный алгоритм.

Начиная со стартовой вершины смотрятся все смежные вершины с текущей и выбирается та, путь до которой от текущей вершины наименьший. Выбранная вершина прибавляется к текущему пути. На следующем шаге смотрится уже вершина, которая была выбрана на текущем шаге. Если все

смежные вершины были просмотрены и путь до конечной из вершины из них не был найден, тогда алгоритм возвращается на одну вершину назад и продолжает поиск для других смежных с ней вершин. При попадании в конечную вершину алгоритм заканчивается.

Сложность:

Сложность алгоритма по времени: O(E*V), где E - количество ребер в графе, V - количество вершин в графе.

Сложность алгоритма по памяти: O(E + V). Так как граф хранится, как словарь, ключи которого вершины графа, а значения - список из ребер.

Алгоритм A*.

Начальная вершина помещается в открытую(приоритетную) очередь. На каждом шаге берется вершина с наименьшим приоритетом(из открытой очереди). Приоритет высчитывается по формулы: p(v) = g(v) + h(v), где g(v) - расстояние от начальной вершины до текущей, h(v) - эвристическая оценка расстояния от текущей вершины до конечной(близость символов в алфавите). Для выбранной вершины в цикле рассматриваются все смежные ей вершины. Если текущий путь до смежной вершины короче кратчайшего пути от начальной вершины до данной, тогда заменяется старый путь на текущий. После эта вершина помещается в очередь с приоритетом. Как только конечная вершина будит иметь наименьший приоритет в открытой очереди, алгоритм завершает свою работу.

Сложность:

Временная сложность алгоритма A* зависит от эвристики. В худшем придется проходить все возможные пути, если эвристическая функция угадывает правильный путь в последнюю очередь. В этом случае время работы будет расти экспотенциально по сравнению с длинной кратчайшего пути.

В лучшем случае, если эвристическая функция будет подбирать правильный путь с начала, сложность по времени будет O(E + V), где E -

количество ребер. V - количество вершин.

Сложность по памяти будет та же. В лучшем случае O(E + V), где E - количество ребер. V - количество вершин. В худшем из за того, что пути хранятся в очереди, память будет расти экспотенциально.

Описание функций и структур данных.

Для жадного алгоритма:

Class Graph - Класс, в котором хранятся данные о графе.

Поля класса:

graph - словарь, ключи которого вершины графа, а значения - список ребер, выходящих из данной вершины.

start - начальная вершина

end - конечная вершина

path - искомый путь из начала в конце

Методы класса:

def add_edge(self, source, dist, weight) - добавляет ребро из вершины source в вершину dist с весом weight в граф

def sort_edge(self) - сортирует смежные ребра для каждой вершины в порядке увеличения веса.

def draw_path(self) - реализует жадный алгоритм

def print_path(self) - выводит найденный путь

Для метода A*

def main() - начальная функция, из которой запускаются все остальные функции

def sort_edges(vertexes, end) - функция, которая сортирует смежные вершины по приоритету(учитываются расстояния до смежной вершины, а также эвристическое расстояние от смежной вершины до конечной)(индивидуализация)

vertexes - словарь вершин

end - конечная вершина

def h(vertex, end): - функция для расчета эвристического расстояния от вершины vertex до конечной вершины(end).

def min_f(opened, f) - функция для нахождения вершины из открытой очереди с наименьшим приоритетом.

Opened - открытая очередь

F - словарь значений эвристической функции для вершин

def a_star(start, end, vertexes) - функция, которая реализует алгоритм A*

start - начальная вершина

end - конечная вершина

vertexes - словарь вершин

Тестирование.

Тестирование индивидуализации.

№	Входные данные	Выходные данные
1	a g a b 100 a c 555 a e 21 a g 4	Before sorting a -> b = 100.0 a -> c = 555.0 a -> e = 21.0 a -> g = 4.0 After sorting a -> g = 4.0 a -> e = 21.0 a -> b = 100.0 a -> c = 555.0

2	a k a b 1 a m 6 a k 1	Before sorting a -> b = 1.0 a -> m = 6.0 a -> k = 1.0 After sorting a -> k = 1.0 a -> m = 6.0 a -> b = 1.0

Тестирование жадного алгоритма:

№	Входные данные	Выходные данные
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	--- Sorting edges --- Vertex a sorting Before sorting: a -> b = 3.0 a -> d = 5.0 After Sorting: a -> b = 3.0 a -> d = 5.0 Vertex b sorting Before sorting: b -> c = 1.0 After Sorting: b -> c = 1.0 Vertex c sorting Before sorting: c -> d = 1.0 After Sorting: c -> d = 1.0 Vertex d sorting Before sorting: d -> e = 1.0 After Sorting: d -> e = 1.0 For vertex 'a' there is edges: b with weight: 3.0 d with weight: 5.0 Go to 'b'

		Vertex 'b was writed to the path. Current path: ab For vertex 'b' there is edges: c with weight: 1.0 Go to 'c' Vertex 'c was writed to the path. Current path: abc For vertex 'c' there is edges: d with weight: 1.0 Go to 'd' Vertex 'd was writed to the path. Current path: abcd For vertex 'd' there is edges: e with weight: 1.0 Go to 'e' Next vertex 'e' equals to end vertex. Path was found Result: abcde
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 1.0 e f 2.0 a g 8.0 f g 1.0	abdeag

3	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
4	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	acdf
5	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	abdeag

Тестирование A*

№	Входные данные	Выходные данные
1	a e a b 3.0 b c 1.0 c d 1.0	--- Sorting edges--- Sorting vertex "a" Before sorting a -> b = 3.0 a -> d = 5.0 After sorting a -> b = 3.0

	a d 5.0 d e 1.0	a -> d = 5.0 Vertex "e" has not edges Sorting vertex "b" Before sorting b -> c = 1.0 After sorting b -> c = 1.0 Sorting vertex "c" Before sorting c -> d = 1.0 After sorting c -> d = 1.0 Sorting vertex "d" Before sorting d -> e = 1.0 After sorting d -> e = 1.0 Opened queue: a Remove vertex a from opened queue Looking path a -> b = 3.0 Path(g) to vertex b equals 3.0 Add vertex b with g = 3.0 to weight of paths g(x) Heuristic of vertex b is 6.0 Add vertex b to opened queue Looking path a -> d = 5.0 Path(g) to vertex d equals 5.0 Add vertex d with g = 5.0 to weight of paths g(x) Heuristic of vertex d is 6.0 Add vertex d to

		opened queue Opened queue: b d Remove vertex b from opened queue Looking path b -> c = 1.0 Path(g) to vertex c equals 4.0 Add vertex c with g = 4.0 to weight of paths g(x) Heuristic of vertex c is 6.0 Add vertex c to opened queue Opened queue: d c Remove vertex d from opened queue Looking path d -> e = 1.0 Path(g) to vertex e equals 6.0 Add vertex e with g = 6.0 to weight of paths g(x) Heuristic of vertex e is 6.0 Add vertex e to opened queue Opened queue: c e Remove vertex c from opened queue Looking path c -> d = 1.0 Path(g) to vertex d equals 5.0 Vertex d was already visited from other

		vertex Opened queue: e Path was found ade
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 1.0 e f 2.0 a g 8.0 f g 1.0	ag
3	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
4	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
5	a g a b 3.0 a c 1.0	ag

	b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0	

Выводы.

В результате выполнения данной лабораторной работы были изучены, а также реализованы алгоритмы поиска пути в графе: жадный алгоритм, алгоритм A*.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл lab2_1.py

```
class Graph:
    graph = {}

    def __init__(self, start_v, end_v):
        self.start = start_v  # start vertex
        self.end = end_v  # end vertex
        self.path = start_v  # path

    def add_edge(self, source, dist, weight):
        # if source vertex hasn't in graph than create this vertex otherwise add
new edge to this vertex
        if source in self.graph.keys():
            self.graph[source].append([dist, float(weight)])
        else:
            self.graph[source] = [[dist, float(weight)]]

    def sort_edge(self):

        print('--- Sorting edges ---')
        for vertex in self.graph:
            print(f'Vertex {vertex} sorting')
            print('Before sorting:')
            for item in self.graph[vertex]:
                print(f'{vertex} -> {item[0]} = {item[1]}')
            self.graph[vertex] = sorted(self.graph[vertex], key=lambda k: k[1])  #
sorting edges by weight
            print('After Sorting:')
            for item in self.graph[vertex]:
                print(f'{vertex} -> {item[0]} = {item[1]}')
        print('\n')

    def print_path(self):
        print('Result: ', self.path)

    def draw_path(self):

        while self.path[-1] != self.end:
            start_vertex = self.graph[self.start]
            print(f"For vertex '{self.start}' there is edges:")
            for vertex in start_vertex:
                print(f"{vertex[0]} with weight: {vertex[1]}")
            i = -1
            for edges in start_vertex:

                i += 1
                if i == len(start_vertex) - 1 and len(edges) == 3:
                    self.path = self.path[0: -1]
                    self.start = self.path[-1]
                    print(f" Path changed. Current path: {self.path}")
                    break
                if len(edges) == 3:
                    continue

                print(f"Go to '{edges[0]}'")
                self.graph[self.start][i].append(1)  # add 1 as this vertex was
visited
                if edges[0] == self.end:  # if current vertex equal end vertex
then path was found. break logo
```

```
                    print(f"Next vertex '{edges[0]}' equals to end vertex. Path
was found")
                    self.path += edges[0]
                    break
                if edges[0] not in self.graph.keys():  # if current vertex haven't
edges then break algo
                    print(f"Vertex '{edges[0]}' haven't edges. Algorithm was
returned to the previous step")
                    break
                # add new vertex to path
                self.path += edges[0]
                self.start = edges[0]
                print(f"Vertex '{edges[0]} was writed to the path. Current path:
{self.path}")
                print('\n')
                break


start, end = input().split(' ')

graph = Graph(start, end)  # create graph

while True:
    try:
        edge = input()
        if len(edge) == 0:
            break
    except EOFError:  # end of line error exception
        break
    edge = edge.split(' ')
    graph.add_edge(edge[0], edge[1], edge[2])  # add edges to graph

# sorting edge
graph.sort_edge()

# found path
graph.draw_path()

# print path
graph.print_path()
```

Файл lab2.py

```
import math


def sort_edges(vertexes, end):
    print('--- Sorting edges---')

    for vertex in vertexes:
        if not vertexes[vertex]:
            print(f'Vertex "{vertex}" has not edges')
            continue
        print(f'Sorting vertex "{vertex}"')
        print('Before sorting')
        for edges in vertexes[vertex]:
            print(f"{vertex} -> {edges[0]} = {edges[1]}")
            # sorting edges
        vertexes[vertex] = sorted(vertexes[vertex], key=lambda item:
abs(ord(item[0]) - ord(end)) + item[1])
        print('After sorting')
        for edges in vertexes[vertex]:
            print(f"{vertex} -> {edges[0]} = {edges[1]}")
```

```
def h(vertex, end):
    # found Heuristic estimate of vertex to end vertex
    return abs(ord(vertex) - ord(end))


def print_anti_priorities(opened, f):
    print("List of anti Priorities:")

    temp = sorted(opened, key=lambda vertex: f[vertex])
    print(temp)


def min_f(opened, f, g):
    # found values of Heuristic function
    minimum = [f[opened[0]], g[opened[0]][0], opened[0]]
    for vertex in opened:
        if f[vertex] == minimum[0] and minimum[0] - minimum[1] > f[vertex] -
g[vertex][0]:
            minimum = [f[vertex], g[vertex][0], vertex]
        elif f[vertex] < minimum[0]:
            minimum = [f[vertex], g[vertex][0], vertex]
    return minimum[2]


def a_star(start, end, vertexes):
    fromed = {}  # found paths
    closed = []  # closed queue
    opened = [start]  # opened queue
    # costs of path from start vertex to current vertex
    g = {
        start: [0, None]
    }
    # values of Heuristic function
    f = {
        start: g[start][0] + h(start, end)
    }

    while opened:
        print_anti_priorities(opened, f)
        print(f'Opened queue: ')
        for item in opened:
            print(item, end=' ')
        print('\n')
        # found minimum value of Heuristic function between vertexes in opened
queue
        current = min_f(opened, f, g)
        if current == end:
            print(f'Path was found')
            break
        # remove from opened queue and add to closed queue
        opened.remove(current)
        closed.append(current)
        print(f'Remove vertex {current} from opened queue')
        # if vertex hasn't edges
        if current not in vertexes:
            print(f'Vertex "{current}" has not edges')
            continue
        # iterating over neighbors of current vertex
        for neighbor in vertexes[current]:
            print(f'Looking path {current} -> {neighbor[0]} = {neighbor[1]}')
            # found costs of price to neighbor
            temp_g = g[current][0] + neighbor[1]
            print(f'Path(g) to vertex {neighbor[0]} equals {temp_g}')
```

```
            if neighbor[0] in closed and temp_g >= g[neighbor[0]][0]:
                print(f'Vertex {neighbor[0]} was already visited from other
vertex')
                continue
            # add new vertex to g(x) and f(x) functions
            if neighbor[0] not in closed or temp_g < g[neighbor[0]][0]:
                fromed[neighbor[0]] = current
                g[neighbor[0]] = [temp_g, current]
                print(f'Add vertex {neighbor[0]} with g = {temp_g} to weight of
paths g(x) ')
                f[neighbor[0]] = g[neighbor[0]][0] + h(neighbor[0], end)
                print(f'Heuristic of vertex {neighbor[0]} is {h(neighbor[0],
end)}')
            # add vertex to opened queue
            if neighbor[0] not in opened:
                print(f"Add vertex {neighbor[0]} to opened queue")
                opened.append(neighbor[0])

    vertex = fromed[end]
    path = end + vertex
    # print result
    while vertex != start:
        vertex = fromed[vertex]
        path += vertex
    print(path[::-1])


def main():
    start, end = input().split(' ')

    # dict of vertexes
    vertexes = {
        start: [],
        end: []
    }

    while True:
        try:
            edge = input()
            if len(edge) == 0:
                break
        except EOFError:  # end of line error exception
            break
        edge = edge.split(' ')
        if edge[0] in vertexes:
            vertexes[edge[0]].append([edge[1], float(edge[2])])
        else:
            vertexes[edge[0]] = [[edge[1], float(edge[2])]]
    sort_edges(vertexes, end)

    a_star(start, end, vertexes)


main()
```