МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

"ЛЭТИ" ИМ. В.И.УЛЬЯНОВА(ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск пути в ориентированном графе.

Студент гр. 1304 _____ Мусаев А.И.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы

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

Задание

1. 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.

2. 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.

Выполнение работы

Задание 1

В этом задании алгоритм не представляет из себя ничего сложного, сначала происходит считывание графа, он хранится с помощью словаря, где ключ - это имя узла, значение по ключу - массив кортежей, где кортеж - ребро, которое хранится как число и буква, где число - вес ребра, буква - имя узла, до которого идет данное ребро. Кроме того, все ребра по ключам отсортированы, сначала идут ребра с наименьшим весом.

Для такого графа (рис. 1)

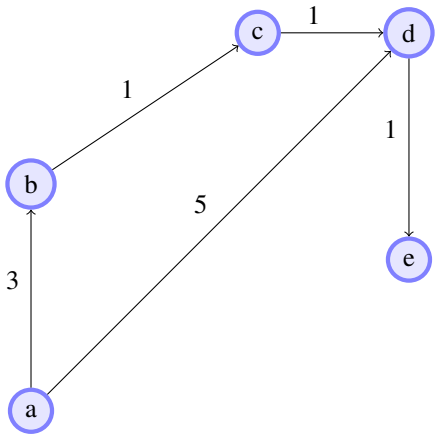

Рисунок 1 – Пример графа для демонстрации словаря

Будет такой словарь:

`{'a': [(3.0, 'b'), (5.0, 'd')], 'b': [(1.0, 'c')], 'c': [(1.0, 'd')], 'd': [(1.0, 'e')]}`

После считывания и сортировки словаря происходит поиск, начиная со стартовой вершины. Находится минимальное ребро нынешней вершины, проверяется, что у вершины, с которой инцидентно это ребро есть вершины, в которые мы можем пойти. Переходим в найденную вершину и делаем то же самое из нее, пока не дойдем до вершины, путь до которой нужно дойти. Мы запоминаем путь, который проходим и выводим его.

На рис. 2 синим цветом представлен путь для рис. 1.

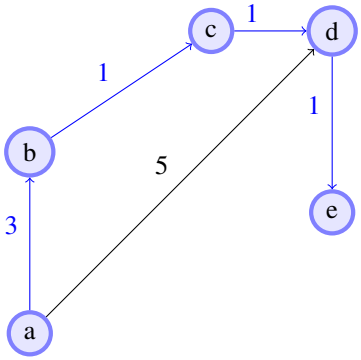

Рисунок 2 – Путь обхода для рис. 1

Код программы представлен в Приложении А.

Задание 2

Во втором задании мы так же сначала считываем граф, но теперь храним каждый узел через класс. В классе хранится стоимость достижения вершины, вершина, из которой мы попали в храняющуюся, имя вершины и соседи данной вершины с помощью массива, в котором хранятся кортежи, в кортеже первый - элемент вершина, инцидентная данному ребру, а второй элемент - вес этого ребра.

Мы заводим массив $reachable$ - это вершины, до которых мы можем добраться на данный момент, так же заводим массив $explord$ - это вершины, которые мы уже проверили.

На каждом шаге поиска мы выбираем один из узлов, который мы знаем, как достичь, и смотрим, до каких новых узлов можем добраться из него. Если мы определим, как достичь конечного (целевого) узла, то задача решена! В противном случае мы продолжаем поиск.

Мы выбираем один из узлов, до которого знаем, как добраться, и который пока не исследован:

```
node = choose_node(reachable, end_node)
```

Выбор происходит так: мы смотрим, до какой вершины из $reachable$ дешевле всего добраться и идём в неё. Стоимость перемещения считаем так: $cost\_start\_to\_node$ + $cost\_node\_to\_goal$, где $cost\_start\_to\_node$ - расстояние, которое мы уже прошли до данной вершины, а $cost\_node\_to\_goal$ - значение эвристической функции, которая определяет примерную стоимость достижения финальной вершины из нынешней.

Когда мы выбрали вершину, мы смотрим ее соседей, добавляем их в $reachable$, если нужно. Кроме того, мы переопределяем стоимость достижения соседей. Если из ныненшей вершины добраться дешевле, чем уже найденная стоимость, мы переопре-

деляем стоимость и меняем путь к этой вершине, выставляя предыдущей вершину, которую мы взяли из *reachable*:

```
for elem in new_reachable:
    adjacent = elem[1]
    dist = elem[0]
    if adjacent not in reachable:
        reachable.append(adjacent)


    if node.cost + dist < adjacent.cost:
        adjacent.previous = node
        adjacent.cost = node.cost + dist
```

После нахождения нужной вершины происходит вывод на экран, это происходит с помощью восстановления для каждой вершины ее предыдущей верщины:

```
def build_path(to_node):
    path = ''
    while to_node != None:
        path += str(to_node)
        to_node = to_node.previous
    return path[::-1]
```

На рис.3 представлен путь обхода графа с рис.1 с помощью A*.

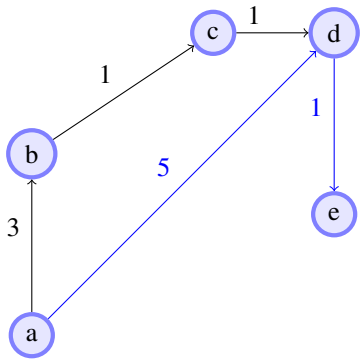

Рисунок 3 – Путь обхода для рис. 1 с помощью A*

На первом шаге мы вбираем вершину d, потому что расстояние до нее + значение эвристической функции меньше, чем это значение для b. Эвристическая функция в нашем случае - близость символов, обозначающих вершины графа, в таблице ASCII. 5 + (d - e) < 3 + (b - e).

Когда программы к заданию 2 представлен в приложении B.

Вывод

Разработаны программы для обхода ориентированного графа, одна работает с помощью жадного алгоритма, другая - алгоритма A*. Тесты на платформе Stepik были пройдены успешно. В первом алгоритме был реализован словарь для хранения графа и жадный алгоритм, который обходит граф. При написании второй программы был реализован класс узел для удобства работы, кроме того. В алгоритме A* была использована эвристика, предложенная на Stepik, а именно - отдаленность символов в таблице ASCII. кроме того, была реализована функция, которая осуществляет оптимальный выбор вершины, в которую мы пойдем.

Приложение А

```
#include <iostream>
start, finish = input().split()
graph = {}
input_string = input().split()
while input_string:
    node1, node2, dist = input_string[0], input_string[1], float(input_string[2])
    if node1 not in graph.keys():
        graph[node1] = [(dist, node2)]
    else:
        graph[node1].append((dist, node2))
    try:
        input_string = input().split()
    except:
        break
for key in graph.keys():
    graph[key].sort()
cur = way = start
while cur != finish:
    for elem in graph[cur]:
        if elem[1] == finish or elem[1] in graph.keys() and elem[1] not in way:
            cur = elem[1]
            way += cur
            break
print(way)
```

Приложение В

```
from math import inf


class Node:
    def __init__(self, name, adjacent, previous=None):
        self.cost = inf
        self.previous = previous
        self.name = name
        self.adjacent = adjacent

    def add_adj(self, adj, dist):
        self.adjacent += [(adj, dist)]

    def __str__(self):
        return self.name


def Heuristic(currentNode, finish):
    return abs(ord(finish.name) - ord(currentNode.name))


def build_path(to_node):
    path = ''
    while to_node != None:
        path += str(to_node)
        to_node = to_node.previous
    return path[::-1]


def choose_node(reachable, finish):
    min_cost = inf
    best_node = None

    for node in reachable:
        cost_start_to_node = node.cost
        cost_node_to_goal = Heuristic(node, finish)
        total_cost = cost_start_to_node + cost_node_to_goal
```

```
        if min_cost > total_cost:
            min_cost = total_cost
            best_node = node

    return best_node


def A_star(start_node, end_node):
    reachable = [start_node]
    explored = []
    while reachable:
        node = choose_node(reachable, end_node)
        if node == end_node:
            return build_path(end_node)
        reachable.remove(node)
        explored.append(node)
        new_reachable = node.adjacent
        for elem in new_reachable:
            adjacent = elem[1]
            dist = elem[0]
            if adjacent not in reachable:
                reachable.append(adjacent)

            if node.cost + dist < adjacent.cost:
                adjacent.previous = node
                adjacent.cost = node.cost + dist
    return None


start, finish = input().split()
graph = {}
input_string = input().split()
while input_string:
    node1, node2, dist = input_string[0], input_string[1], float(
        input_string[2])
    if node2 not in graph.keys():
        graph[node2] = Node(node2, [])
```

```
    if node1 not in graph.keys():
        graph[node1] = Node(node1, [(dist, graph[node2])])
    else:
        graph[node1].add_adj(dist, graph[node2])
    try:
        input_string = input().split()
    except:
        break
graph[start].cost = 0
print(A_star(graph[start], graph[finish]))
```