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: Алгоритмы на графах

Студент гр. 7304 _____ Пэтайчук Н.Г.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2019

Цель работы

Исследование алгоритмов поиска путей в графе на примере жадного алгоритма и алгоритма A*.

Постановка задачи

Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма и алгоритма A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Входные данные

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

Выходные данные

Строка, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Ход работы

1. Объявление структуры ребра графа, которая включает в себя имя вершины, куда можно попасть по ребру, и вес ребра, а также объявление следующих имён:
   a) *Vertex* - имя вершины;
   b) *Vertex_List* - список вершин;
   c) *Graph* - словарь, где в качестве ключей выступают имена вершин, в качестве значений - список рёбер, куда можно попасть из вершины-ключа;
   d) *graph_iterator* - итератор по графу;
2. Написание функции, реализующей работу поиска пути с помощью жадного алгоритма. Суть алгоритма заключается в следующем: на каждом шаге алгоритма берём последнюю посещённую вершину и переходим по наименьшему ребру из этой вершины в следующую и так до тех пор, пока не придём в нужную вершину;
3. Написание функции, использующей алгоритм A* для поиска наименьшего пути из начальной вершины в конечную. На каждом шаге алгоритма мы выбираем ближайшую к старту непросмотренную вершину (по умолчанию расстояние в вершине-старте равно 0, в остальных - бесконечности) и помечаем её как просмотренную, затем идёт пересчёт путей до смежных вершин (если вершина не просмотрена и путь до текущей вершины + вес ребра меньше, чем нынешний путь до смежной вершины, то мы заменяем значение расстояния до данной смежной вершины на длину пути до текущей вершины + вес ребра). При этом используется некоторая

эвристическая функция, которая позволяет среди равных по длине путей выбрать следующим тот, что по идее ближе к конечной вершине (в нашем случае в качестве данной эвристической функции выступает функция расстояния между символами в ASCII-таблице);

4. Написание головной функции, где происходит ввод начальной и конечной вершин, а так же самого графа, где надо будет проложить наикратчайший путь, и вывод результата работы алгоритмов, то есть строки, содержащие вершины, которые входят в построенный алгоритмом путь;

Весь исходный код программы представлен в Приложении 1.

Тестирование программы

1) Тест 1: Простой граф

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
end
A* algorithm:
ade
Greedy algorithm:
abcde
```

2) Тест 2: Граф посложнее

```
a m
a b 1.0
a d 2.0
b c 1.0
c e 1.0
d e 1.0
e f 1.0
e h 2.0
f g 3.0
g i 1.0
h i 1.0
i j 1.0
i l 2.0
j k 2.0
k m 2.0
l m 2.0
end
A* algorithm:
adehilm
Greedy algorithm:
abcefgijkm
```

Вывод

В ходе лабораторной работы были изучены некоторое алгоритмы на графах, а именно жадный алгоритм и алгоритм A*, целью которых является нахождение наикратчайшего пути от одной вершины до другой. Кроме того, данные алгоритмы были реализованы на языке программирования C++ и протестированы. Результаты показали, что алгоритм A* находит более оптимальное решение в плане количества вершин, которые надо пройти, чем жадный алгоритм (а в некоторых случаях, путь, полученный жадным алгоритмом, не всегда является минимальным), что демонстрирует следующий факт: выбор на каждом шаге алгоритма наиболее оптимального решения не гарантирует получения оптимального решения в итоге работы алгоритма.

Приложение 1: Исходный код программы

- grath_algorithms.h

```
#pragma once

#include <string>
#include <algorithm>
#include <vector>
#include <map>

using namespace std;

typedef char Vertex;
struct Edge
{
    Vertex where;
    double length;

    Edge(Vertex end, double length) : where(end), length(length) {}
};

typedef vector<Vertex> Vertex_List;
typedef map<Vertex, vector<Edge>> Graph;
typedef Graph::iterator graph_iterator;

string findWay_Greedy(Graph &field, Vertex start, Vertex end);

string findWay_AStar(Graph &field, Vertex start, Vertex end);
```

- grath_algorithms.cpp

```
#include "graph_algorithms.h"

string findWay_AStar(Graph &field, Vertex start, Vertex end)
{
     map<Vertex, bool> visited_vertexes;
     map<Vertex, Vertex> way_to_end;
     map<Vertex, double> destination_from_start = {{start, 0}};
     string answer = "";

     for (graph_iterator it = field.begin(); it != field.end();
it++)
         visited_vertexes[it->first] = false;
     for (unsigned int i = 0; i < field.size(); i++)
     {
         map<Vertex, double>::iterator now_element =
min_element(destination_from_start.begin(),
destination_from_start.end(),

[end](const pair<Vertex, double> &one, const pair<Vertex, double>
&another)
```

```
        {
            return (one.second + (end - one.first)) <
(another.second + (end - another.first));
        });
        if (now_element->first == end)
            break;
        visited_vertexes[now_element->first] = true;
        for (Edge next_way : field[now_element->first])
        {
            if (visited_vertexes[next_way.where] == false)
            {
                map<Vertex, double>::iterator way_to_vertex =
destination_from_start.find(next_way.where);
                if ((way_to_vertex ==
destination_from_start.end()) || (now_element->second +
next_way.length < way_to_vertex->second))
                {
                    destination_from_start[next_way.where] =
now_element->second + next_way.length;
                    way_to_end[now_element->first] =
next_way.where;
                }
            }
        }
        destination_from_start.erase(now_element);
    }
    for (Vertex now_vertex = start; answer[answer.size()- 1] !=
end; now_vertex = way_to_end.find(now_vertex)->second)
        answer += now_vertex;
    return answer;
}

string findWay_Greedy(Graph &field, Vertex start, Vertex end)
{
    map<Vertex, bool> visited_vertexes;
    Vertex_List way_to_end = {start};
    string answer = "";

    for (graph_iterator it = field.begin(); it != field.end();
it++)
        visited_vertexes[it->first] = false;
    while (way_to_end[way_to_end.size() - 1] != end)
    {
        vector<Edge> possible_ways;

        visited_vertexes[way_to_end[way_to_end.size() - 1]] =
true;
        for (Edge possible_way :
field[way_to_end[way_to_end.size() - 1]])
            if (visited_vertexes[possible_way.where] == false)
                possible_ways.push_back(possible_way);
        if (possible_ways.empty())
        {
```

```
                way_to_end.pop_back();
                continue;
            }
            Edge next_way = *min_element(possible_ways.begin(),
possible_ways.end(),
                                         [](const Edge &first,
const Edge &second)
            {
                return first.length < second.length;
            });
            way_to_end.push_back(next_way.where);
        }
        for (unsigned int i = 0; i < way_to_end.size(); i++)
            answer += way_to_end[i];
        return answer;
}
```

- main.cpp

```
#include <iostream>
#include "graph_algorithms.h"

int main()
{
    Graph field;
    Vertex start, finish, start_vertex, finish_vertex;
    double length;

    cin >> start >> finish;
    while (cin >> start_vertex >> finish_vertex >> length)
        field[start_vertex].push_back(Edge(finish_vertex, length));


    cout << "A* algorithm:" << endl;
    cout << findWay_AStar(field, start, finish) << endl;
    cout << "Greedy algorithm:" << endl;
    cout << findWay_Greedy(field, start, finish) << endl;

    return 0;
}
```