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: Жадный алгоритм и A*

Студент гр. 8303 _____ Пушпышев А.И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить и реализовать на языке программирования c++ жадный алгоритм поиска пути в графе и алгоритм A* поиска кратчайшего пути в графе между двумя заданными вершинами.

Задание.

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

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.

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.

Входные данные

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

Выходные данные

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

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

Соответствующие выходные данные

A*: ade

Жадный алгоритм: abcde

Описание алгоритма.

Жадный алгоритм:

Для удобства в начале работы жадного алгоритма поиска пути в ориентированном графе, список рёбер сортируется по не убыванию их весов. Алгоритм начинает поиск из заданной вершины. Текущая просматриваемая вершина добавляется в список просмотренных.

В отсортированном списке рёбер выбирается первое (сортировка гарантирует, что это будет минимальное), которое начинается в просматриваемой вершине, если эта вершина не просмотрена, то текущей вершиной становится та, в которой заканчивается это ребро, если она уже просмотрена, то выбирается следующее ребро. Если в какой-то момент из текущей вершины нет путей, то происходит откат на шаг назад, и в предыдущей вершине выбирается другое ребро, если это возможно.

Алгоритм заканчивает свою работу, когда текущей вершиной становится искомая, или когда были просмотрены все рёбра, которые начинаются из исходной вершины.

A:*

Поиск начинается из исходной вершины. В текущие возможные пути добавляются все рёбра из начальной вершины. Происходит выбор минимального пути, где учитывается эвристическая близость вершины к искомой (в нашем случае это близость в таблице ASCII), если в выбранном пути последняя вершина уже была просмотрена, то этот путь удаляется из открытого списка, и снова происходит выбор минимального пути.

Выбираются из всех рёбер графа те, которые начинаются из последней вершины в этом пути. Эта вершина добавляется к этому пути, и новый путь заносится в список возможных путей, с увеличением стоимости, равной переходу по этому ребру. Когда были выбраны все рёбра, которые начинаются из последней вершины в этом пути, то эта вершина добавляется в закрытый список, а сам путь удаляется из открытого списка путей. Дальше снова происходит выбор минимального пути.

Алгоритм заканчивает свою работу, когда достигается искомая вершина.

Сложность жадного алгоритма по операциям:

O (|V|*|E|)

Сложность жадного алгоритма по памяти:

O (|E|)

Сложность A* алгоритма по операциям:

O (|V|*log|V|) в случае оптимальной эвристики и O (|V|*|V|) в худшем случае

Сложность A* алгоритма по памяти:

O (|E|)

Так как алгоритм работает с графом, идейно проще оценивать сложность от количества вершин и ребер. |V| - количество вершин и |E| - количество ребер.

Описание функций и структур данных.

A*

1.

```
struct Edge//ребро графа
{
    char bgn;//начальная вершина
    char end;//конечная вершина
    double wt;//вес ребра
};
```

2.

```
struct Step//возможные пути
{
    string path;//путь
    double length;//длина пути
    char estuary;
};
```

3. `void input_graph()//ввод графа`

4. `bool is_visible(char value)//проверка доступа к вершине`

5. `void Search()//процесс выполнения A*`

Жадный алгоритм

1.

```
struct Edge//ребро графа
{
    char bgn;//начальная вершина
    char end;//конечная вершина
```

```
    double wt;//вес ребра
};
```

2.

```
struct Step//возможные пути
{
    string path;//путь
    double length;//длина пути
    char estuary;
};
```

3. `void input_graph()//ввод графа`

4. `bool is_visible(char value)//проверка доступа`

5. `void to_search()//инициализация жадного алгоритма`

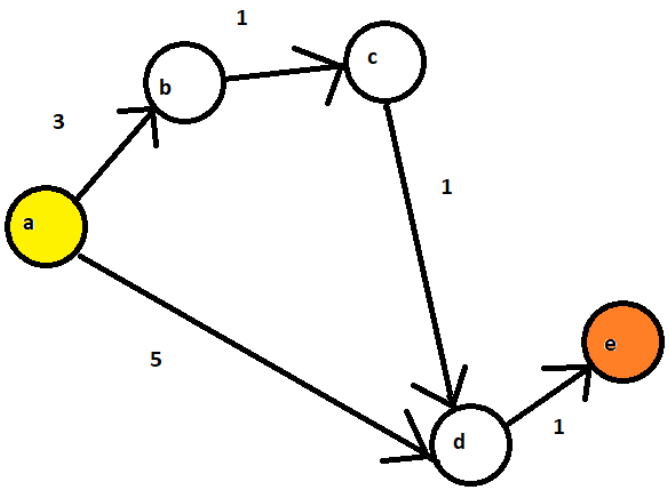
6. `bool Search(char value)//жадный алгоритм`

7. `void Print()//вывод результата`

Исследование и тестирование.

Входные данные	Результат	
	A*	Жадный алгоритм
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	abcde
a m a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 e f 3.0 j p 2.0 e p 1.0 p m 5.0	adepm	abcdepm
a b a b 3.0 m b 1.0 j b 1.0	ab	ab
a b a b 3.0	ab	ab

Графическое представление графа №1 из тестирования.



Выводы.

В ходе выполнения данной лабораторной работы были изучены и реализованы два алгоритма. На основе анализа результатов тестирования алгоритмов, мы явным образом показали главное их отличие. При этом можно сказать, что алгоритм A* имеет схожий образ по сравнению с алгоритмом Дейкстры, пройденном в рамках курса Дискретной математики. Отличие состоит в том, что сложность эвристики у алгоритма Дейкстры равна 0, при этом он обходит больше “территории”, по сравнению с A*, что делает его медленнее. Жадный алгоритм не входит в сравнение, потому что он в принципе не гарантирует кратчайший путь.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

•

```
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <cfloat>
#include <algorithm>

using namespace std;


struct Edge//ребро графа
{
    char bgn;//начальная вершина
    char end;//конечная вершина
    double wt;//вес ребра
};

struct Step//возможные пути
{
    string path;//путь
    double length;//длина пути
    char estuary;//конец пути
};

class A_star_graph
{
private:
    vector <Edge> graph;//список смежности
    vector <Step> result;//преобразовываемый (открытый) список конечных путей
    vector <char> current;//закрытый список вершин, содержит текущий путь

    char source;
    char estuary;

public:
    A_star_graph(){
    };
    void input_graph()
    {
        cin >> source >> estuary;
        char temp;
        while(cin >> temp)
        {
            Edge element;
            element.bgn = temp;
            cin >> element.end;
            cin >> element.wt;
            graph.push_back(element);
        }
        string buf = "";
        buf += source;
        for(auto & i : graph)
        {
            if(i.bgn == source)
            {
                buf += i.end;
```

```
                result.push_back({buf, i.wt});
                result.back().estuary = estuary;
                buf.resize(1);//запись всех ребер, которые исходят из начальной позиции
            }
        }
        current.push_back(source);
    }

    size_t min_elem() //возвращает индекс минимального элемента из непросмотренных
    {
        double min;
        min = DBL_MAX;
        size_t temp = -1;
        for(size_t i(0); i < result.size(); i++)
        {
            if(result.at(i).length + abs(estuary - result.at(i).path.back()) < min)
            {
                if(is_visible(result.at(i).path.back()))
                {
                    result.erase(result.begin() + i);
                }
                else
                {
                    min = result.at(i).length + abs(estuary -
result.at(i).path.back());
                    temp = i;
                }
            }
        }
        return temp;
    }

    bool is_visible(char value)//проверка доступа к вершине
    {
        for(char i : current) {
            if (i == value) {
                return true;
            }
        }
        return false;
    }

    void Search()
    {
        sort(result.begin(), result.end(), [](const Step & a, const Step & b) -> bool
        {
            return a.length + a.estuary - a.path.back() > b.length + b.estuary -
b.path.back();
        });
        while(true)
        {
            size_t min = min_elem();
            if(min == -1){
                cout << "Wrong graph";
                break;
            }
            if(result.at(min).path.back() == estuary)
            {
                cout << result.at(min).path;
                return;
            }
            for(auto & i : graph)
```

```
            {
                if(i.bgn == result.at(min).path.back())
                {
                    string buf = result.at(min).path;
                    buf += i.end;
                    //cout << buf << endl;
                    result.push_back({buf, i.wt + result.at(min).length});
                }
            }
            current.push_back(result.at(min).path.back());
            result.erase(result.begin() + min);
        }
    }
};


int main()
{
    A_star_graph element;
    element.input_graph();
    element.Search();
    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


struct Edge
{
    char bgn;
    char end;
    double wt;
};


class Greedy_graph
{
private:
    vector <Edge> graph;
    vector <char> result;
    vector <char> current;
    char source;
    char estuary;

public:
    Greedy_graph()
    {

    }
    void input_graph(){
        cin >> source >> estuary;
        char temp;
        while(cin >> temp)
        {
            Edge element;
            element.bgn = temp;
            if(!(cin >> element.end))
                break;
            if(!(cin >> element.wt))
                break;
            graph.push_back(element);
        }
        sort(graph.begin(), graph.end(), [](Edge first, Edge second)
        {
            return first.wt < second.wt;
        });
    }

    bool is_visible(char value)
    {
        for(char i : current)
            if(i == value)
                return true;
        return false;
    }


    void to_search()
    {
        if(source != estuary)
            Search(source);
    }
```

```
    bool Search(char value)
    {
        if(value == estuary)
        {
            result.push_back(value);
            return true;
        }
        current.push_back(value);
        for(auto & i : graph)
        {
            if(value == i.bgn)
            {
                if(is_visible(i.end))
                    continue;
                result.push_back(i.bgn);
                bool flag = Search(i.end);
                if(flag)
                   return true;
                result.pop_back();
            }
        }
        return false;
    }

    void Print()
    {
        for(char i : result)
            cout << i;
    }
};

int main()
{
    Greedy_graph element;
    element.input_graph();
    element.to_search();
    element.Print();
    return 0;
```