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. 8304 _____ Масалыкин Д.Р.
Преподаватель Размочаева Н.В.

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.

Формулировка задания.

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

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.

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.

Ход выполнения работы.

Жадный алгоритм:

Для удобства в начале работы жадного алгоритма поиска пути в ориентированном графе, список рёбер сортируется по неубыванию 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 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 список просмотренных, а сам путь удаляется из списка возможных путей. Дальше снова происходит выбор минимального пути. Алгоритм заканчивает свою работу, когда достигается искомая вершина.

Сложность жадного алгоритма по операциям:

O (N^2)

Сложность жадного алгоритма по памяти:

O(V|+|E|)

Сложность A* алгоритма по операциям:

O(log h*(N))

Сложность A* алгоритма по памяти:

O (|V|+|E|)

Описание функций и структур данных. A*

1.

```
struct Rib//ребро графа
{
    char begin;//начальная вершина
char end;//конечная вершина
     double weight;//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 Rib//ребро графа
{
    char begin;//начальная вершина
char end;//конечная вершина
double weight;//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`

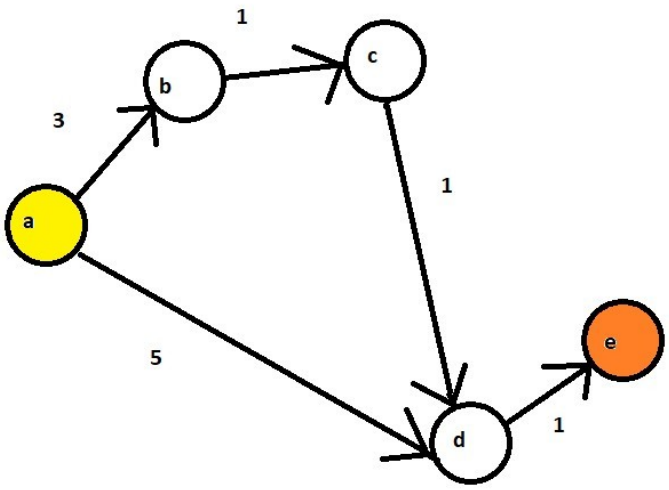
7. `void Print()//вывод результата`

Результат работы программы.

Входные данные	Результаты	
	A*	Жадный
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. Первый – жадный алгоритм поиска пути в ориентированном графе. Этот алгоритм выбирает наименьший путь на каждом шаге – в этом заключается жадность, алгоритм достаточно прост, но за это платит своей надёжностью, так как он не гарантирует, что найденный путь будет минимальным возможным. Второй – алгоритм поиска минимального пути в ориентированном графе A*, который является модификацией алгоритма Дейкстры. Модификация состоит в том, что A* находит минимальные пути не до каждой вершины в графе, а для заданной. В ходе его работы при выборе пути учитывается не только вес ребра, но и эвристическая близость вершины к искомой. A* гарантирует, что найденный путь будет минимальным возможным.

Приложение А.

Исходный код.

Жадный алгоритм.

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

using namespace std;


struct Rib
{
   char begin;
   char end;
   double weight;
};


class Greedy_graph
{
private:
   vector <Rib> graph;
   vector <char> res;
   vector <char> curr;
   char source;
   char dst;

public:
   Greedy_graph()
   {

   }
   void input_graph(){
      cin >> source >> dst;
      char temp;
      while(cin >> temp)
      {
         Rib element;
         element.begin = temp;
         if(!(cin >> element.end))
            break;
         if(!(cin >> element.weight))
            break;
         graph.push_back(element);
      }
      sort(graph.begin(), graph.end(), [](Rib first, Rib second)
      {
         return first.weight < second.weight;
      });
   }
```

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


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

    bool Search(char value)
    {
        if(value == dst)
        {
            res.push_back(value);
            return true;
        }
        curr.push_back(value);
        for(auto & i : graph)
        {
            if(value == i.begin)
            {
                if(is_visible(i.end))
                    continue;
                res.push_back(i.begin);
                bool flag = Search(i.end);
                if(flag)
                    return true;
                res.pop_back();
            }
        }
        return false;
    }

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

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

}

Приложение В.

Исходный код.

A*.

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

using namespace std;


struct Rib//ребро графа
{
   char begin;//начальная вершина
   char end;//конечная вершина
   double weight;//вес ребра
};

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

class A_star_graph
{
private:
   vector <Rib> graph;//список смежности
   vector <Step> res;//преоброзовываемый (открытый) список путей
   vector <char> curr;//закрытый список вершин, содержит текущий
путь
   char source;
   char estuary;

public:
   A_star_graph(){
   };
   void input_graph()
   {
      cin >> source >> estuary;
      char tmp;
      while(cin >> tmp)
```

```
        {
            Rib elem;
            elem.begin = tmp;
            cin >> elem.end;
            cin >> elem.weight;
            graph.push_back(elem);
        }
        string buf = "";
        buf += source;
        for(auto & i : graph)
        {
            if(i.begin == source)
            {
                buf += i.end;
                res.push_back({buf, i.weight});
                res.back().estuary = estuary;
                buf.resize(1);//запись всех ребер, которые исходят из
начальной позиции
            }
        }
        curr.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 < res.size(); i++)
        {
            if(res.at(i).length + abs(estuary - res.at(i).path.back()) < min)
            {
                if(is_visible(res.at(i).path.back()))
                {
                    res.erase(res.begin() + i);
                }
                else
                {
                    min = res.at(i).length + abs(estuary - res.at(i).path.back());
                    temp = i;
                }
            }
        }
        return temp;
    }
```

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

    void Search()
    {
        sort(res.begin(), res.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(res.at(min).path.back() == estuary)
            {
                cout << res.at(min).path;
                return;
            }
            for(auto & i : graph)
            {
                if(i.begin == res.at(min).path.back())
                {
                    string buf = res.at(min).path;
                    buf += i.end;
                    //cout << buf << endl;
                    res.push_back({buf, i.weight + res.at(min).length});
                }
            }
            curr.push_back(res.at(min).path.back());
            res.erase(res.begin() + min);
        }
    }
};
```

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