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. 7383 _____ Тян Е.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

СОДЕРЖАНИЕ

1. ЦЕЛЬ РАБОТЫ

Цель работы: исследовать и реализовать задачу построения кратчайшего пути в ориентированном графе помощью метода A*.

Формулировка задачи: необходимо разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A* до любой из представленных вершин. 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.

Входные данные: в первой строчке через пробел указываются начальная и две конечные вершины. Далее в каждой строке указываются ребра графа и их вес.

2. РЕАЛИЗАЦИЯ ЗАДАЧИ

В данной работе используются главная функция (int main()) и структуры данных(struct node, class list), содержащие методы void add_edge(char v1, char v2, double weight), double evristic(int i, double way), void path(char first, char last1, char last2, priority_queue <node> & queue1, double way, vector <char>& str_way)), а также перегрузка оператора bool operator < (const node& a, const node & b).

Параметры, хранящиеся в структуре данных struct node:

- key – приоритет нахождения в очереди;
- dif – разница между ASCII символов;
- str – путь в символом виде.

Параметры, содержащиеся в структуре данных class list:

- edge – массив, хранящий информацию о ребрах.

Параметры, передаваемые в метод void add_edge(char v1, char v2, double weight) структуры данных class list:

- v1 – вершина, из которой исходит ребро;
- v2 – вершина, в которую входит ребро;
- weight – вес ребра.

Параметры, передаваемые в метод double evristic(int i, double way) структуры данных class list:

- i – индекс вершины в графе;
- way – длина пути, содержащая только веса ребер.

Параметры, передаваемые в метод void path(char first, char last1, char last2, priority_queue <node> & queue1, double way, vector <char>& str_way) структуры данных class list:

- first – начальная вершина;
- last1 – первая конечная вершина;
- last2 – вторая конечная вершина;
- queue1 – очередь с приоритетом;
- way – длина пути, содержащая только веса ребер;

• str_way – путь.

Параметры передаваемые в bool operator < (const node& a, const node & b):

• a – элемент struct node;

• b – элемент struct node.

В функции main() считываются начальная вершина и две конченые вершины, для которых нужно будет найти путь. Далее в цикле начинается считывание ребер графа и построение массива, хранящего ребра графа. Записывается значение начальной вершины, откуда будем начинать путь. Вызывается метод класса list path(char first, char last1, char last2, priority_queue <node> & queue1, double way, vector <char>& str_way), который идет по графу, используя эвристическую функцию, и образует очередь с приоритетами. Под эвристической функцией понимается функция вычисляющая сумму разницы между конечной вершиной пути и вершиной, в которую ведет ребро, по которому мы идем, и длины пройденного пути. Используя свойства очереди с приоритетами строиться кратчайший путь для вершины, которая на данном шаге находится ближе. В конце работы программа выводит кратчайший путь до любой из вершин, если таковой имеется.

Рассмотрим пример работы программы на графе приведенном на рис. 1.

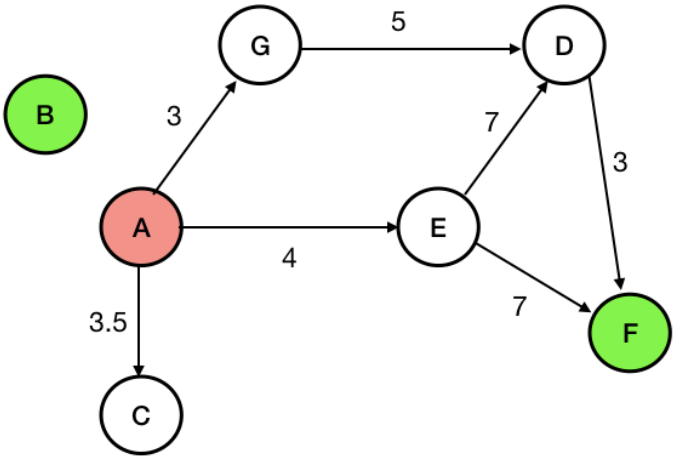

Рисунок 1 — граф, описывающий пример работы программы

Программа считывает значения, хранящиеся в файле «test.txt». На

первом шаге поиска кратчайшего пути сначала в очередь с приоритетом будет помещена вершина G и потом вершина E, т.к. приоритет для пути в вершину G = 4, а для E = 5. В очередь будут класться сначала вершины с меньшим приоритетом. Далее будет выниматься вершина G как в обычной очереди сначала вынимаются элементы, лежащие в начале. Дальнейшие шаги приведены на рис. 2, 3

Рисунок 2 — представление очереди после первого шага

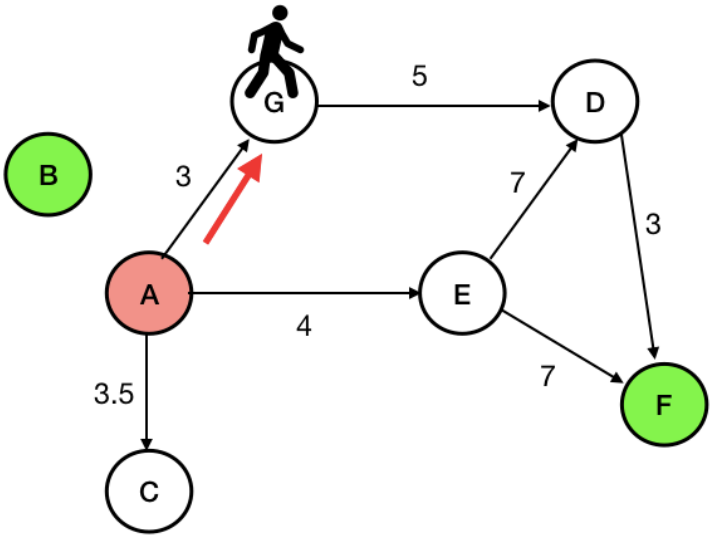

Рисунок 3 — действие программы после извлечения первого элемента из очереди

Рисунок 4 — представление очереди после второго шага

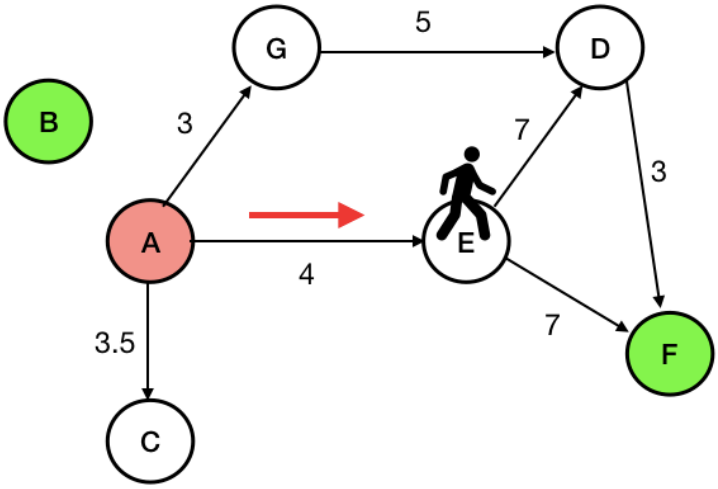

Рисунок 5 — действие программы после извлечения второго элемента из очереди

Рисунок 6 — представление очереди, после третьего шага

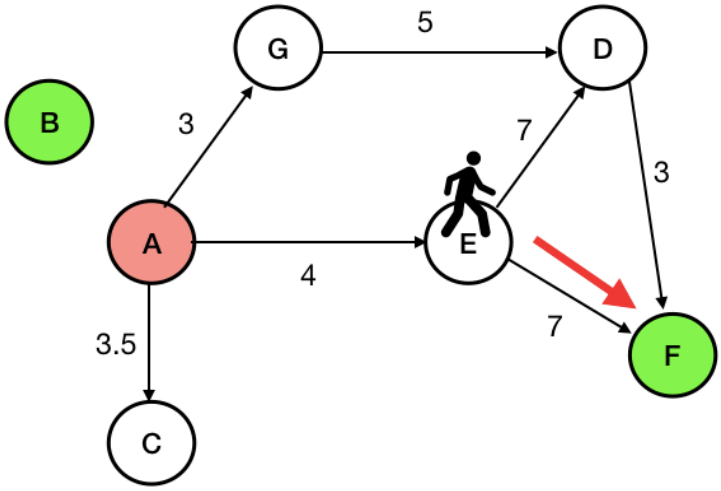

Рисунок 7 — представление конечного шага

Результатом работы программы будет строка «aef». Несмотря на то, что разница между символами B и A была меньше, чем между A и F, программа е искала путь между A и B, т.к. его не существует.

3. ТЕСТИРОВАНИЕ

Программа была собрана в компиляторе G++ в OS Linux Ubuntu 12.04. Программа может быть скомпилирована с помощью команды:

g++ <имя файла>.cpp -std=c++11

Тестовые случаи представлены в Приложении А.

Исходя из тестовых случаев можно увидеть, что тестовые случаи не выявили некорректной работы программы, что говорит о том, что по результатам тестирования было показано, что поставленная задача была выполнена.

4. ИССЛЕДОВАНИЕ

Компиляция была произведена в компиляторе в g++ Version 4.2.1. В других ОС и компиляторах тестирование не проводилось.

На каждом шаге работы программы просматриваются всевозможные пути из искомой вершины. В худшем случае могут быть просмотрены все пути данного графа. Тогда сложность зависит от количества ребер и количества вершин графа. В таком случае временную сложность алгоритма можно свести к показательной. Аналогичной будет сложность по памяти, т.к. в худшем случае придется хранить в очереди приоритетов всевозможные пути.

5. ВЫВОД

В ходе выполнения лабораторной работы была решена задача нахождения кратчайшего пути в графе методом A* на языке C++, и исследован алгоритм A*. Полученный алгоритм имеет сложность показательную как по времени, так и по памяти. Так же была изучена структура данных очередь с приоритетом.

Была написана программа строящая граф в виде списка смежности, очередь с приоритетом, и вычисляющая кратчайший путь от заданной вершины до конечной, если такой существует.

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

№	Ввод	Вывод
1	a x f a b 3 b d 5 a e 4 a c 3.5 e f 7 e d 7 d f 3	aef
2	a k e a b 3.1 a c 4 a d 2.9 b e 3.052 e g 8.6 f g 6 e h 7 c e 6 d e 3 e f 1 e h 7 h k 3 g k 1.4 f i 1.2 i k 1.1 d f 1.099 f l 2.5 i h 6 h l 0.1	adfl
3	e a a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp:

```
#include <iostream>
#include <cmath>
#include <queue>

using namespace std;

typedef struct node{
   double key;
   double dif;
   vector <char> str;
}node;

bool operator < (const node& a,const node& b){
   return a.key > b.key;
}

class list{
private:
   vector <tuple <char, char, double> > edge;
public:
   void add_edge(char v1, char v2, double weight){
      edge.push_back(make_tuple(v1, v2, weight));
   }
   double evristic(int i,double way){
      return get<2>(edge[i])+way;
   }
   void path(char first, char last1, char last2,
                                priority_queue <node> &queue1, double way,
                                                                vector <char>& str_way){
      vector<char> str1;
      while(true){
         for(int i = 0; i < edge.size(); i++)
            if(get<0>(edge[i]) == first){
               node elem;
               elem.dif=min( fabs(last1-get<1>(edge[i])),
                                                        fabs(last2-get<1>(edge[i])));
               elem.key=evristic(i,way)+elem.dif;
               for(auto j:str1)
                  elem.str.push_back(j);
               elem.str.push_back(get<1>(edge[i]));
               queue1.push(elem);
            }
```

```
            if(queue1.empty())
                break;
            if(!queue1.empty()){
                node popped;
                popped=queue1.top();
                queue1.pop();
                first=popped.str[popped.str.size()-1];
                str1=popped.str;
                way=popped.key-popped.dif;
            }
            if(str1[str1.size()-1] == last1 ||
                                                    str1[str1.size()-1] == last2){
                for(auto j:str1)
                    str_way.push_back(j);
                break;
            }
        }
    }
};

int main(){
    char first,last1,last2,v1;
    cin>>first;
    cin>>last1;
    cin>>last2;
    list list1;
    while(cin>>v1){
        char v2;
        cin>>v2;
        double weight;
        cin>>weight;
        if(isalpha(v1) && isalpha(v2))
            list1.add_edge(v1, v2, weight);
    }
    vector <char> str;
    str.push_back(first);
    priority_queue <node> queue1;
    double way=0;
    list1.path(first, last1, last2, queue1, way, str);
    if(str[str.size()-1] == last1 || str[str.size()-1] == last2)
        for(auto j:str)
            cout<<j;
    return 0;
}
```