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. 8382 Гордиенко А.М.

Преподаватель Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать жадный алгоритм поиска кратчайшего пути в графе и алгоритм A* в графе между двумя заданными вершинами.

Постановка задачи.

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 посещенная вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешевым из последней посещенной вершины. Каждая вершина в графе имеет буквенное значение (a, b, c, ...), каждое ребро имеет неотрицательный вес.

Пример входных данный:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Для приведенных в примере входных данных ответом будет abcde.

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.

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

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются начальная и конечная вершины.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Для приведенных выше входных данных ответом будет ade.

Индивидуальное задание.

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

Описание алгоритма.

Жадный алгоритм.

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

В отсортированном списке ребер выбирается первое, которое начинается в просматриваемой вершине, если эта вершина не просмотрена, то текущей вершиной становится та, к которой ведет это ребро. Если вершина 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 заданная искомая, или когда были просмотрены все ребра, которые начинаются из исходной вершины.

*A**.

Поиск начинается из исходной вершины. В текущие возможные пути добавляются все ребра 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.

Из всех ребер выбирается 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 ребра, 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 работу, когда достигнута искомая коненая вершина.

Оценка сложности.

Сложность Жадного алгоритма.

По памяти.

Так как в памяти хранится только список ребер, то сложность составляет O(|E|).

По времени.

Сложность составляет O(|V| * |E|).

Сложность A*.

По памяти.

Так же, как и в жадном алгоритме, так как структура хранения такая же O(|E|).

По времени.

Без оптимизации сложность будет квадратичная O(2^|V|). С оптимальной эвристикой O(|E| * log2(|E|)).

Описание функций и структур.

Общие.

Структура-ребро графа. Имеет поля начало ребра from, конец ребра to, и вес(цену) ребра - weight.

```
struct Edge {
	char from;
	char to;
	double weight;
};
```

Структура-путь графа. Имеет поле самого пути - path, длину (суммарный вес ребер) этого пути - len, и наименование конечной вершины пути end.

```
struct Step {
	string path;
	double len;
	char end;
};
```

Функции void readGraph() для считывания графа из потока.

Функция void Search() выполняет соответствующий поиск пути в графе.

A*.

Имеет функции is_visible(char value) ищет ребро к вершине с именем value.

Жадный алгоритм.

Имеет функции void solve(), которая запускает поиск решение.

Тестирование.

Входные данные.

a g

a b 1

b c 1

c f 2

e b 1

a d 1.49

d e 1.39

e g 2

Результат поиска жадного алгоритма.

abcfg

Результат поиска алгоритма A*.

adeg

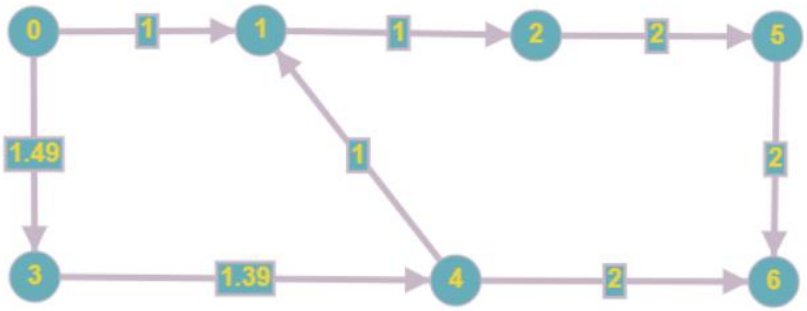

Рисунок 1. Графическое представление графа из тестирования.

Выводы.

В ходе работы были изучены и реализованы два алгоритма. На основе тестирования можно сделать выводы об исполнении алгоритмов. Видно, что жадный алгоритм быстро находит решение, которое не всегда является верным. В то время как алгоритм A* уступает по времени, но находит точно верное решение, если оно есть.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ ЖАДНЫЙ АЛГОРИТМ ПОИСКА

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge
{
    char from;
    char to;
    double weight;
};

class Greedy_graph
{
private:
    vector <Edge> edges;
    vector <char> result;
    vector <char> cur;
    char source;
    char estuary;
public:
    Greedy_graph() {}
    void readGraph(){
        cin >> source >> estuary;
        char tmp;
        while(cin >> tmp)
        {
            Edge Gr;
            Gr.from = tmp;
            if(!(cin >> Gr.to))
```

```
            break;
        if(!(cin >> Gr.weight))
            break;
        edges.push_back(Gr);
    }
    sort(edges.begin(), edges.end(), [](Edge first, Edge second)
    {
        return first.weight < second.weight;
    });
}
bool is_visible(char value) {
    for(char i : cur)
        if(i == value)
            return true;
    return false;
}
void solve() {
    if(source != estuary)
        Search(source);
}
bool Search(char value) {
    if(value == estuary) {
        result.push_back(value);
        return true;
    }
    cur.push_back(value);
    for(auto & i : edges) {
        if(value == i.from) {
            if(is_visible(i.to))
                continue;
            result.push_back(i.from);
```

```
                bool flag = Search(i.to);
                if(flag)
                   return true;
                result.pop_back();
            }
        }
        return false;
    }


    void Print() {
        for(char i : result)
            cout << i;
    }
};
int main() {
    Greedy_graph Gr;
    Gr.readGraph();
    Gr.solve();
    Gr.Print();
    return 0;
}
```

АЛГОРИТМ A*

```
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <cfloat>
#include <algorithm>
using namespace std;
struct Edge { // ребро графа
    char from; // начальная вершина
    char to; // конечная вершина
    double weight; // вес ребра
};
struct Step { // возможные пути
    string path; // путь
    double len; // длина пути
    char end; // конец пути
};
class A_star_graph
{
private:
    vector <Edge> edges;//список смежности
    vector <Step> result;//преоброзовываемый (открытый) список путей
    vector <char> cur;//закрытый список вершин, содержит текущий
путь
    char begin;
    char end;
public:
    A_star_graph(){
    };
    void input_graph()
    {
```

```
        cin >> begin >> end;
        char temp;
        while(cin >> temp)
        {
            Edge element;
            element.from = temp;
            cin >> element.to;
            cin >> element.weight;
            edges.push_back(element);
        }
        string buf = "";
        buf += begin;
        for(auto & i : edges)
        {
            if(i.from == begin)
            {
                buf += i.to;
                result.push_back({buf, i.weight});
                result.back().end = end;
                buf.resize(1);//запись всех ребер, которые исходят
из начальной позиции
            }
        }
        cur.push_back(begin);
    }
    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).len + abs(end -
result.at(i).path.back()) < min)
            {
                if(is_visible(result.at(i).path.back()))
                {
                    result.erase(result.begin() + i);
                }
                else
                {
                    min = result.at(i).len + abs(end -
result.at(i).path.back());
                    temp = i;
                }
            }
        }
        return temp;
    }
    bool is_visible(char value)//проверка доступа к вершине
    {
        for(char i : cur) {
            if (i == value) {
                return true;
            }
        }
        return false;
    }
    void Search()
    {
        sort(result.begin(), result.end(), [](const Step & a, const
Step & b) -> bool
        {
```

```
            return a.len + a.end - a.path.back() > b.len + b.end -
b.path.back();
        });
        while(true)
        {
            size_t min = min_elem();
            if(min == -1){
                cout << "Wrong edges";
                break;
            }
            if(result.at(min).path.back() == end)
            {
                cout << result.at(min).path;
                return;
            }
            for(auto & i : edges)
            {
                if(i.from == result.at(min).path.back())
                {
                    string buf = result.at(min).path;
                    buf += i.to;
                    //cout << buf << endl;
                    result.push_back({buf, i.weight +
result.at(min).len});
                }
            }
            cur.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;
}
```