

**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

\_\_\_\_\_

Лосев М.Л.

Преподаватель

\_\_\_\_\_

Жангиров Т.Р.

Санкт-Петербург

2018

## **Постановка задачи.**

**Цель работы:** получение знаний и опыта в использовании алгоритмов построения пути в ориентированном графе: жадного алгоритма и алгоритма A\*.

Формулировка 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 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.
- 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

**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.

## **Реализация**

Был использован следующие классы:

Класс Edge:

Поля:

```
friend class Graph;  
type first;  
type last;  
double dist;
```

Методы:

```

friend const bool operator < (const Edge &v1, const Edge &v2);
friend const bool operator <= (const Edge &v1, const Edge &v2);
friend const bool operator > (const Edge &v1, const Edge &v2);
friend const bool operator >= (const Edge &v1, const Edge &v2);
friend const bool operator != (const Edge &v1, const Edge &v2);
friend const bool operator == (const Edge &v1, const Edge &v2);
– операторы сравнения, сравнивают по длине ребра.

```

Класс Vertex:

Поля:

```

type value;
multiset <Edge> adj_list;

```

Методы:

```

Vertex(type value) – конструктор
Vertex() - конструктор
void add_adjacent_ver(Edge e) – добавляет инцидентное ребро
void print_value() const – выводит значение вершины
Vertex& operator = (const Vertex &v) – оператор присваивания
friend const bool operator < (const Vertex &v1, const Vertex &v2);
friend const bool operator <= (const Vertex &v1, const Vertex &v2);
friend const bool operator > (const Vertex &v1, const Vertex &v2);
friend const bool operator >= (const Vertex &v1, const Vertex &v2);
friend const bool operator != (const Vertex &v1, const Vertex &v2);
friend const bool operator == (const Vertex &v1, const Vertex &v2);
– операторы сравнения, сравнивают по значению вершины.

```

Класс Way:

Поля:

```

string path;
double length;
type last;

```

Методы:

```

Way() – конструктор
Way(const Way &other) – конструктор
Way(type start) – конструктор
int cost() const – возвращает стоимость перехода в последнюю вершину
void print() const – выводит путь

friend const bool operator < (const Way &v1, const Way &v2);
friend const bool operator <= (const Way &v1, const Way &v2);
friend const bool operator > (const Vertex &v1, const Way &v2);
friend const bool operator >= (const Way &v1, const Way &v2);
friend const bool operator != (const Way &v1, const Way &v2);
friend const bool operator == (const Way &v1, const Way &v2);
– операторы сравнения, сравнивают по длине пути

```

Класс Graph

Поля:

Vertex start; – начальная вершина

Vertex finish; – конечная вершина

Vertex current; – текущая вершина

bool final; – тэг того, что путь найден

set <type> ver\_list – список величин вершин

vector <Vertex> vertexes – список вершин

Way ans; – путь, который надо посторить

Методы:

Graph() – конструктор

void input\_start() – ввод начальной и конечной вершин

void input\_edges() – ввод ребер

void greedy() – жадный плгоритм

void a\_star() – алгоритм A\*

## Исследование

Пусть  $|E|$  - множество ребер графа,  $|V|$  – множество его вершин. Алгоритм в худшем случае пройдет каждую вершину. При прохождении каждой вершины в очередь будут добавляться пути через эту вершину в смежные с ней из начальной. Добавление в очередь требует в худшем случае столько сравнений, сколько в очереди элементов. Но элементов в очереди не может быть больше, чем  $|V|$ , если не хранить несколько путей в одну и ту же вершину, что бессмысленно (для оптимальности достаточно рассматривать лишь более короткий из них). Получается  $O(|V|*|V|)$ . Но, с другой стороны, очевидно, что на самом деле эта величина должна зависеть не только от  $|V|$ , но и от  $|E|$ . Некоторые исследователи полагают, что сложность алгоритма  $A^*$  составляет  $O(|V|*|V|*\log(|E|))$ .

### **Тестирование.**

Тестирование проводилось в ОС Ubuntu 18.04 компилятором GCC. Кроме того, программа прошла тесты на Stepic. Результаты тестирования показали, что программа работает корректно.

**Вывод.**

При выполнении работы были изучены и реализованы жадный алгоритм и алгоритм  $A^*$ . Была исследована сложность алгоритма  $A^*$ .

## ПРИЛОЖЕНИЕ А

### Код программы

```
#include <iostream>
#include <list>
#include <string>
#include <vector>
#include <set>
#include <queue>
#include <utility>
#include <algorithm>

#include <cstdio>
#include <cctype>

using namespace std;

typedef char type;

class Edge
{
public:
    friend class Graph;
    type first;
    type last;
    double dist;

    Edge(char first, char last, double dist) : first(first), last(last),
dist(dist) {
    }

    friend std::ostream& operator<<(std::ostream& out, const Edge & obj);

    // need folowing operators to use multiset of type Edge
    // (in fact only < operator needed by std::less which is used by default
by set, multiset and priority_queue)
    friend const bool operator < (const Edge &v1, const Edge &v2);
    friend const bool operator <= (const Edge &v1, const Edge &v2);
    friend const bool operator > (const Edge &v1, const Edge &v2);
    friend const bool operator >= (const Edge &v1, const Edge &v2);
    friend const bool operator != (const Edge &v1, const Edge &v2);
    friend const bool operator == (const Edge &v1, const Edge &v2);
};

std::ostream& operator<<(std::ostream& out, const Edge & obj)
{
    out << obj.first << " " << obj.last << " " << obj.dist << endl;
    return out;
}

const bool operator < (const Edge &v1, const Edge &v2)
{
    return v1.dist < v2.dist;
}
const bool operator <= (const Edge &v1, const Edge &v2)
```



```

{
    return v1.dist <= v2.dist;
}
const bool operator > (const Edge &v1, const Edge &v2)
{
    return v1.dist > v2.dist;
}
const bool operator >= (const Edge &v1, const Edge &v2)
{
    return v1.dist >= v2.dist;
}
const bool operator != (const Edge &v1, const Edge &v2)
{
    return v1.dist != v2.dist;
}
const bool operator == (const Edge &v1, const Edge &v2)
{
    return v1.dist == v2.dist;
}

class Vertex
{
public:
    friend class Graph;
    type value;
    multiset <Edge> adj_list; // adjacency list; using multiset because it's
always sorted so theres no need to search for minimal element

    Vertex(type value) : value(value) {
    }

    Vertex() {
    }

    void add_adjacent_ver(Edge e) {
        adj_list.insert(e);
    }

    void print_value() const {
        cout << value;
    }

    Vertex& operator = (const Vertex &v)
    {
        this->value = v.value;
        this->adj_list = multiset <Edge> (v.adj_list);

        return *this;
    }

    // need folowing operators to use multiset of type Edge
    // (in fact only < operator needed by std::less which is used by default
by set, multiset and priority_queue)
    friend const bool operator < (const Vertex &v1, const Vertex &v2);
    friend const bool operator <= (const Vertex &v1, const Vertex &v2);

```

```

        friend const bool operator > (const Vertex &v1, const Vertex &v2);
        friend const bool operator >= (const Vertex &v1, const Vertex &v2);
        friend const bool operator != (const Vertex &v1, const Vertex &v2);
        friend const bool operator == (const Vertex &v1, const Vertex &v2);
};

const bool operator < (const Vertex &v1, const Vertex &v2)
{
    return v1.value < v2.value;
}
const bool operator <= (const Vertex &v1, const Vertex &v2)
{
    return v1.value <= v2.value;
}
const bool operator > (const Vertex &v1, const Vertex &v2)
{
    return v1.value > v2.value;
}
const bool operator >= (const Vertex &v1, const Vertex &v2)
{
    return v1.value >= v2.value;
}
const bool operator != (const Vertex &v1, const Vertex &v2)
{
    return v1.value != v2.value;
}
const bool operator == (const Vertex &v1, const Vertex &v2)
{
    return v1.value == v2.value;
}

struct Way
{
public:
    string path;
    double length;
    type last;

    Way() {
        path = "";
        length = 0.0;
        last = ' ';
    }
    Way(const Way &other) {
        path = other.path;
        length = other.length;
        last = other.last;
    }

    Way(type start) {
        length = 0.0;
        last = start;
        path = "";
        path += last;
    }
}

```

```

    int coast() const {
        // finish - last1 < finish - last2 <=> -last1 < -last2
        return -last;
    }

    void print() const {
        cout << path << endl;
    }

    // need following operators to use multiset of type Edge
    // (in fact only < operator needed by std::less which is used by default
    by set, multiset and priority_queue)
    friend const bool operator < (const Way &v1, const Way &v2);
    friend const bool operator <= (const Way &v1, const Way &v2);
    friend const bool operator > (const Vertex &v1, const Way &v2);
    friend const bool operator >= (const Way &v1, const Way &v2);
    friend const bool operator != (const Way &v1, const Way &v2);
    friend const bool operator == (const Way &v1, const Way &v2);
};

const bool operator < (const Way &v1, const Way &v2) {
    return v1.length < v2.length;
}
const bool operator <= (const Way &v1, const Way &v2) {
    return v1.length <= v2.length;
}
const bool operator > (const Way &v1, const Way &v2) {
    return v1.length > v2.length;
}
const bool operator >= (const Way &v1, const Way &v2) {
    return v1.length >= v2.length;
}
const bool operator != (const Way &v1, const Way &v2) {
    return v1.length != v2.length;
}
const bool operator == (const Way &v1, const Way &v2) {
    return v1.length == v2.length;
}

struct heuristic
{
    bool operator()( const Way &lhs, const Way &rhs ) const {
        // length is the distance from ways last node to the goal node
        return lhs.length + lhs.coast() == rhs.length + rhs.coast() ?
        rhs.last < lhs.last : lhs.length + lhs.coast() > rhs.length + rhs.coast();
    }
}; // needed (instead of std::less) to use priority_queue of type Way

class Graph
{
public:
    Vertex start;          // not a real vertex, only value, no adjacency
    list

```

```

Vertex finish;          // not a real vertex, only value, no adjacency
list
Vertex current;

bool final;

set <type> ver_list; // using set because there's always no equal
elements in set
vector <Vertex> vertexes; // can't use set to store vertexes because sets
elements must be const

Way ans;

Graph() : final (false) {
}

void input_start() { // reads first input string (start and finist
vertexes); sets start and finish
    type tmp_start, tmp_finish;
    cin >> tmp_start;
    cin >> tmp_finish;

    ver_list.insert(tmp_start);
    vertexes.push_back(Vertex(tmp_start));
    start = *find(vertexes.begin(), vertexes.end(), Vertex(tmp_start));

    current = start;

    ans.last = start.value;

    ver_list.insert(tmp_finish);

    vertexes.push_back(Vertex(tmp_finish));
    finish = *find(vertexes.begin(), vertexes.end(),
Vertex(tmp_finish));
}

bool is_there_ver(type v) {
    return ver_list.find(v) == ver_list.end() && *ver_list.end() != v;
}

void input_edges() {
    type tmp_first;
    type tmp_last;
    double tmp_dist;
    while (cin >> tmp_first && isalpha(tmp_first)) {
        cin >> tmp_last >> tmp_dist;
        if (!isalpha(tmp_first) || !isalpha(tmp_last) || cin.fail())
            break; // bad input case

        if (is_there_ver(tmp_first)) { //
if there is no vertex named tmp_first
            ver_list.insert(tmp_first); //
add vertex named tmp_first to vertexes list
            vertexes.push_back(Vertex(tmp_first)); // add
new vertex to vertexes

```

```

    }

    if (is_there_ver(tmp_last)) {
if there is no vertex named tmp_last
        ver_list.insert(tmp_last);
add vertex named tmp_last to vertexes list
        vertexes.push_back(Vertex(tmp_last));
new vertex to vertexes
    }

    find(vertexes.begin(), vertexes.end(), Vertex(tmp_first))-
>add_adjacent_ver(Edge(tmp_first, tmp_last, tmp_dist));
    // added second vertex to first vertex's adjacency list
    // two vertexes is equal if their values are.
}

}

void output_graph() {
    cout << "start: " << start.value << endl << "finish: " <<
finish.value << endl;
    for_each(vertexes.begin(), vertexes.end(), [](const Vertex &v) {
        cout << "Vertex name: " << v.value << endl << "Vertexes edges:
" << endl;
        for_each(v.adj_list.begin(), v.adj_list.end(), [](const Edge
&e) {
            cout << "(" << e.first << " --> " << e.last << ", " <<
e.dist << ") " << endl;
        } );
    } );
}

void greedy() {
    if (final) // if the path is already found
        return;

    if (current == finish) {
        final = true;
        ans.path += current.value;
        return;
    } // if current vertex is the last one

    if (!vertexes.empty()) { // if there are some vertexes in graph
        auto cur = find(vertexes.begin(), vertexes.end(), current); //
find current vertex in list
        if (cur->adj_list.empty()){
            return;
        } // if there are no edges in current vertexes adjacency list

        auto min_edge = cur->adj_list.begin(); // set is always
sorted; min_edge is now the nearest vertex to current
        double min_dist = min_edge->dist;

        while (!final) { // check all the vertexes of current's
adjacency list (in case the nearest doesn't lead to finish)
            move(*find(vertexes.begin(), vertexes.end(),
Vertex(min_edge->last)));

```

```

        greedy();

        if (final) {
            ans.length += min_dist;
            ans.path = min_edge->first + ans.path;
            return;
        } // if the path is already found

        if (min_edge == cur->adj_list.end())
            break; // if this is the last vertex in
currents adjacency list
            ++min_edge; // next vertex in list
            min_dist = min_edge->dist;
        }

    }

}

void a_star() { // folowing code is essentially translation of pseudocode
from wikipedia
    set <Vertex> closed;
        // var closed := the empty set
    priority_queue <Way, vector<Way>, heuristic> open;
        // var open := make_queue(f); f is heuristic
    open.push(Way(start.value));
        // enqueue(open, path(start))

    while (!open.empty()) {
        // while open is not empty
        Way p = open.top();
            // var p := remove_first(open)
        open.pop();
        Vertex x = *find(vertexes.begin(), vertexes.end(),
Vertex(p.last)); // var x := the last node of p
        cout << "found vertex: " << x.value << endl;
        if (x == finish && final == false) {
            // if x = goal
            final = true;
            ans = p;
            //return;
                // return p (???)
        }
        if (x == finish && final == true) {
            if (p.length < ans.length) {
                ans = p;
            }
        }

        closed.insert(x);
            // add(closed, x)

        for_each(x.adj_list.begin(), x.adj_list.end(),
// foreach y in successors(x)
[this, p, &open] (const Edge &e) {
            Vertex y = *find(vertexes.begin(), vertexes.end(),
Vertex(e.last));

```

```

        Way tmp_way(p);
        tmp_way.last = y.value;
        tmp_way.path += y.value;
        tmp_way.length += e.dist;
        open.push(tmp_way);
        // enqueue(open, add_to_path(p, y))
    } );
}

void move(Vertex vertex) {
    current = vertex;
}

};

int main()
{
    Graph g;

    g.input_start();
    g.input_edges();
    //g.output_graph();
    //g.greedy();
    g.a_star();
    if (g.final) {
        cout << "a_star: ";
        g.ans.print();
    }

    g.final = false;
    g.ans = Way();
    g.greedy();
    if (g.final) {
        cout << "greedy: ";
        g.ans.print();
    }
    return 0;
}

```