

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Жадный алгоритм и A^*

Студент гр. 8382

Щеглов А.С.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Написать программу для поиска наименьшего пути в графе между двумя заданными вершинами, используя жадный алгоритм и A^* .

Задание.

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.

Входные данные:

Начальная и конечная вершины, ребра с их весами

Выходные данные:

Минимальный путь из начальной вершины в конечную, написанный слитно

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

```
ade
```

Вариант дополнительного задания.

Вар. 2. В A^* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Описание алгоритма

Жадный алгоритм:

На каждом шаге выбирается последняя посещенная вершина, выбирается

соседняя не посещённая вершина с минимальным весом ребра. Процесс повторяется до тех пор, пока не будет обработана конечная вершина или вершины закончатся. Такой алгоритм не гарантирует нахождение оптимального решения при его существовании.

Алгоритм A*:

Данный алгоритм во многом схож с алгоритмом Дейкстры. На каждом шаге проверяем, меньше ли расстояние до соседа через текущую вершину, и обновляем наименьший путь до соседней вершины. Текущая вершина на каждой итерации должна иметь минимальную дистанцию от начальной вершины, т.е. должна поддерживаться очередь с приоритетами с возможностью извлечения минимума. Отличие алгоритма A* от алгоритма Дейкстры заключается в том, что к приоритету вершины прибавляется значение некоторой эвристической функции от этой вершины до целевой. Эвристическая функция должна быть монотонна и допустима, иначе алгоритм A* будет асимптотически хуже алгоритма Дейкстры. Правильно подобранная эвристическая функция в ряде прикладных задач позволяет значительно ускорить поиск пути, уменьшив фронт вершин поиска. Алгоритм A* гарантирует нахождение оптимального решения, если оно существует, при условии, что эвристическая функция допустима, монотонна и определена в тех же единицах измерений, что и веса ребер. A* отличается от жадного алгоритма тем, что учитывает уже построенный путь и некоторую топологическую оценку нахождения целевой вершины.

Особенности реализации алгоритма.

Для реализации жадного алгоритма ребра записываются в вектор и в цикле, пока не программа не придет в конечную вершину ищем наименьшее ребро, исключая из списка уже пройденные.

Для реализации алгоритма A* создана структура, описывающая каждую вершину. Открытый и закрытый список вершин выражен через вектор

структур, описывающий точки. Программа идет, постепенно заполняя вектор закрытых вершин, пока не дойдет до последней точки, опираясь на функцию $f(v)=g(v)+h(v)$ выбирает следующую вершину, по наименьшему значению f , выстраивая и запоминая (через указатель на пред вершину) путь.

Описание функций и методов.

- `void foo()`— функция, непосредственно алгоритм A^* , начинает со начальной вершины, создает закрытый и открытый список и составляет путь, по которому в дальнейшем будет построен вектор минимального пути.
- `struct Node` — структура описывающая вершину. Имеет имя - `s`, вес - `g`, эвристическое приближение - `h`, указатель на предыдущую вершину - `*prev`, а также вектор ребер исходящих из этой вершины — `neig` и функцию `f()`, считающую сумму `g` и `h`.

Тестирование.

Для A^* :

```
5
a e
a 4
b 3
c 2
d 1
e 0
a b 3
b c 1
c d 1
a d 5
d e 1
and
ade
```

```
9
a i
a 11
b 9
c 6
d 3
e 7
f 6.5
g 4
h 6
i 0
a b 3
a c 2
b e 2
b f 1
c d 4
c h 2
e g 5
f h 1
d i 17
h g 1
g i 7
and
achgi
```

Для Жадного алгоритма:

```
a e
a b 3
a c 1
c d 1
a d 5
d e 1
and
acde
```

```
a e
a b 3
b c 1
c d 1
a d 5
d e 1
and
abcde
```

Сложность алгоритма

Для жадного алгоритма имеем по времени выполнения: сортировка ребер $|E|\log|E|$, просмотр каждой вершины и каждого ребра $|V|+|E|$. Итого $O(|V| + |E|\log|E|)$. По памяти $O(|V|+|E|)$.

Для алгоритма A^* оценка по памяти $O(|V|+|E|)$. Оценка по времени зависит от эвристической функции и используемой структуры для хранения очереди, списка посещенных вершин и т.п. Вставка и получение минимума используются для каждой вершины по одному разу. Получаем $|V|\log|V|$ для очереди с приоритетами с логарифмическими операциями. Так же, просматривается каждое ребро. Получается $O(|E|+|V|\log|V|)$. Хорошая эвристическая функция способна уменьшить время выполнения в константное число раз, что не влияет на асимптотику, но может значительно повлиять на реальное время работы в прикладной задаче. Плохая эвристическая функция может иметь отрицательное значение или скачки, что снижает эффективность выбора текущей вершины и нахождения оптимального пути в целом. При этом в любом случае будет просмотрена каждая вершина один раз, но не будут отсечены заведомо неперспективные ребра, что приблизит время выполнения к наихудшей оценке

Выводы.

В результате выполнения работы была разработана программа для нахождения минимального пути во взвешенном графе с помощью алгоритма A^* и жадного алгоритма. Была проанализирована асимптотика данных алгоритмов, а также их корректность.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <fstream>
#include <map>
#include <set>

using std::vector;
using std::cout;
using std::cin;
using std::endl;
using std::ifstream;
using std::map;
using std::set;

struct A {
    char one;
    char two;
    double three;
    A(char a, char b, double c) {
        one = a;
        two = b;
        three = c;
    }
};

struct Node {
    static Node* beg;
    static Node* end;
    char name;
    double g;
    double h;
    Node* prev;
    vector<std::pair<Node*, double> > neig;

    Node(char c) {
        name = c;
        g = 999999; //
        prev = nullptr;
    }

    double f() {
        return g+h;
    }
};

Node* Node::beg;
Node* Node::end;

void foo() {
    vector<Node*> q;
    vector<Node*> u;
    q.push_back(Node::beg);
    Node::beg->g = 0;
    while (q.size() != 0) {
```

```

int ind = 0;
for (int i = 1; i < q.size(); i++)
    if (q[i]->f() < q[ind]->f())
        ind = i;
Node* curr = q[ind];
if (curr == Node::end)
    return;

q.erase(q.begin() + ind);
u.push_back(curr);
for (int i = 0; i < curr->neig.size(); i++) {
    double score = curr->g + curr->neig[i].second;
    Node* v = curr->neig[i].first;
    int j = 0;
    for (; j < u.size(); j++)
        if (u[j] == v)
            break;
    if (j < u.size() && score >= v->g)
        continue;

    v->prev = curr;
    v->g = score;
    j = 0;
    for (; j < q.size(); j++)
        if (q[j] == v)
            break;
    if (j >= q.size())
        q.push_back(v);
}

}
}

int main() {
    int n;
    char a, b;
    double c;
    map<char, Node*> nodes;
    Node* beg;
    Node* end;

    cin >> n;
    cin >> a >> b;
    if (a == b) {
        cout << a;
        return 0;
    }
    beg = new Node(a);
    end = new Node(b);
    nodes[a] = beg;
    nodes[b] = end;
    Node::beg = beg;
    Node::end = end;

    for (int i = 0; i < n; i++) {
        cin >> a >> c;
        if (nodes.find(a) == nodes.end()) {
            nodes[a] = new Node(a);
        }
    }
}

```

```

    nodes[a]->h = c;
}

while(cin >> a >> b >> c) {
    if (nodes.find(a) == nodes.end()) {
        nodes[a] = new Node(a);
    }
    if (nodes.find(b) == nodes.end()) {
        nodes[b] = new Node(b);
    }
    nodes[a]->neig.push_back(std::pair<Node*, double>(nodes[b], c));
}

foo();
vector<Node*> vec;
Node* node = Node::end;
while(node != nullptr) {
    vec.push_back(node);
    node = node->prev;
}
for (int i = vec.size() - 1; i >= 0; i--)
    cout << vec[i]->name;
}

```