

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. 9382

Рыжих Р.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Разработать жадный алгоритм и алгоритм A* для поиска пути в графе.

Задание.

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.

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 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 приведённых в примере входных данных ответом будет

```
abcde
```

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 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 приведённых в примере входных данных ответом будет

ade

Вариант 8

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

Описание алгоритма.

1. Заполняется std::map с ключом-вершиной и значением, равным всем соседним вершинам.
2. Сортировка std::map по значению перед началом работы алгоритм.

3. Для жадного алгоритма вынимается всегда первый элемент так как значения ключей `std::map` отсортированы по возрастанию. Для A^* учитывается вся стоимость пути, поэтому на каждой итерации вынимается по одному, начиная с заданной, постоянно обновляя стоимость передвижения.

Сложность.

Вставка n элементов в `std::map` занимает $O(n \log n)$. Помимо взятия элемента производится поиск пути от данного элемента до конечного. Поиск в `std::map` n раз занимает $O(n \log n)$. Итого, получается $O(2n \log n)$.

Временная сложность алгоритма A^* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x));$$

Где h^* - оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка $h(x)$ не должна расти быстрее, чем логарифм от оптимальной эвристики.

Описание функций и структур данных.

Создан класс `FindingPath`, который содержит в себе `std::map`, представляющий собой граф, начало и конец пути, количество вершин, а также следующие функции:

`vector<char> GreedyAlghoritm()` – жадный алгоритм, который выбирает самый короткий путь к смежной вершине и переходит по нему, если вершина не была посещена.

`vector<char> AStar()` – алгоритм A^* , который с помощью приоритетной очереди выбирает наикратчайший алгоритм поиска пути от начала, до конца графа.

`void Sort()` – функция, сортирующая смежные вершины по весу.

void Read() – функция считывания до символа ‘0’ (также, как принимает Stepik), которая заполняет граф.

Int Heuristic(char a, char b) – эвристическая функция.

Демонстрация работы.

Ввод	Вывод
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0 0	<p>Для вершины a есть следующие смежные вершины:</p> <p>b(3) c(1) g(8)</p> <p>Отсортированные вершины:</p> <p>c(1) b(3) g(8)</p> <p>Для вершины b есть следующие смежные вершины:</p> <p>d(2) e(3)</p> <p>Отсортированные вершины:</p> <p>d(2) e(3)</p> <p>Для вершины d есть следующие смежные вершины:</p> <p>e(4)</p> <p>Отсортированные вершины:</p> <p>e(4)</p> <p>Для вершины e есть следующие смежные вершины:</p> <p>a(1) f(2)</p> <p>Отсортированные вершины:</p> <p>a(1) f(2)</p> <p>Для вершины f есть следующие смежные вершины:</p> <p>g(1)</p> <p>Отсортированные вершины:</p>

	<p>g(1)</p> <p>Ответ: ag</p>
<p>a g</p> <p>a b 3.0</p> <p>a c 1.0</p> <p>b d 2.0</p> <p>b e 3.0</p> <p>d e 4.0</p> <p>e a 3.0</p> <p>e f 2.0</p> <p>a g 8.0</p> <p>f g 1.0</p> <p>0</p>	<p>Для вершины a есть следующие смежные вершины:</p> <p>b(3) c(1) g(8)</p> <p>Отсортированные вершины:</p> <p>c(1) b(3) g(8)</p> <p>Для вершины b есть следующие смежные вершины:</p> <p>d(2) e(3)</p> <p>Отсортированные вершины:</p> <p>d(2) e(3)</p> <p>Для вершины d есть следующие смежные вершины:</p> <p>e(4)</p> <p>Отсортированные вершины:</p> <p>e(4)</p> <p>Для вершины e есть следующие смежные вершины:</p> <p>a(3) f(2)</p> <p>Отсортированные вершины:</p>

	<p>f(2) a(3)</p> <p>Для вершины f есть следующие смежные вершины:</p> <p>g(1)</p> <p>Отсортированные вершины:</p> <p>g(1)</p> <p>Ответ:ag</p>
<p>a e</p> <p>a b 7.0</p> <p>a c 3.0</p> <p>b c 1.0</p> <p>c d 8.0</p> <p>b e 4.0</p>	<p>Для вершины a есть следующие смежные вершины:</p> <p>b(7) c(3)</p> <p>Отсортированные вершины:</p> <p>c(3) b(7)</p> <p>Для вершины b есть следующие смежные вершины:</p> <p>c(1) e(4)</p> <p>Отсортированные вершины:</p> <p>c(1) e(4)</p> <p>Для вершины c есть следующие смежные вершины:</p> <p>d(8)</p> <p>Отсортированные вершины:</p> <p>d(8)</p> <p>Ответ:abe</p>

Выводы.

В ходе выполнения лабораторной работы был разработан эадный алгоритм, а также разработан алгоритм A*

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <algorithm>
using namespace std;

class FindingPath {
public:
    FindingPath() = default;
    vector<char> GreedyAlgorithm();
    vector<char> AStar();
    void Sort();
    void Read();
    int Heuristic(char a, char b);

private:
    map<char, vector<pair<char, double>>> graph;
    map<char, bool> visited;
    char start;
    char end;
    int number;
};

struct Sorting { //функция сортировки для приоритетной очереди
    bool operator() (pair<char, double> a, pair<char, double> b) {
        //если стоимость двух вершин равна, то возвращается меньшая из них в алфавитном
        //порядке, если стоимость разная, то большая из них
        if (a.second == b.second)
            return (a.first < b.first);
        else
            return (a.second > b.second);
    }
};

int FindingPath::Heuristic(char a, char b) {
    return abs(a-b);
}

vector<char> FindingPath::AStar() { //A*
    map<char, pair<vector<char>, double>> ShortPathes; //текущие кратчайшие пути
    vector<char> vertex;
    priority_queue < pair<char, double>, vector<pair<char, double>>, Sorting> Priority-
    tyQueue; //очередь в алгоритме

    PriorityQueue.push({ start, 0 });
    vertex.push_back(start);
    ShortPathes[start].first = vertex;

    while (!PriorityQueue.empty()) { //пока очередь не пуста
        if (PriorityQueue.top().first == end) { //если найдена конечная вершина
            return ShortPathes[end].first; //то заканчивается поиск
        }

        auto TmpVertex = PriorityQueue.top(); //достается приоритетная вершина из оче-
        PriorityQueue.pop();
    }
}
```



```

        for (auto& i : graph[TmpVertex.first]) { //рассматриваются все вершины, которые
соединены с текущей вершиной
            double CurLength = ShortPathes[TmpVertex.first].second + i.second;
            if (ShortPathes[i.first].second == 0 || ShortPathes[i.first].second > Cur-
Length) { //если пути нет или найденный путь короче
                vector<char> path = ShortPathes[TmpVertex.first].first; //добавляется в
путь родительской вершины текущая вершина с кратчайшим путем
                path.push_back(i.first);
                ShortPathes[i.first] = { path, CurLength }; //обновление пути и расстоя-
ния
                int heur = Heuristic(end, i.first);
                //cout << i.first << ' ' << heuristic[i.first] << '\n';
                PriorityQueue.push({ i.first, heur + ShortPathes[i.first].second });
            } //записывается в очередь текущая вершина
        }

    }

    return ShortPathes[end].first;
}

```

```

void FindingPath::Sort() {
    for (auto it = graph.begin(); it != graph.end(); ++it) {
        cout << "Для вершины " << it->first << " есть следующие смежные вершины:\n";
        for (int i = 0; i < it->second.size(); i++) {
            cout << it->second[i].first << '(' << it->second[i].second << ')' << ' ';
        }
        cout << endl;
        std::sort(it->second.begin(), it->second.end(), [](pair<char, double>& a,
pair<char, double>& b) -> bool {return a.second < b.second; });

        cout << "Отсортированные вершины:\n";
        for (int j = 0; j < it->second.size(); j++) {
            cout << it->second[j].first << '(' << it->second[j].second << ')' << ' ';
        }
        cout << endl;
    }
}

```

```

void FindingPath::Read() {
    char start, end;
    cin >> start >> end;
    this->start = start;
    this->end = end;
    int count = 0;

    while (cin >> start) {
        if (start == '0') //символ остановки ввода данных
            break;
        double weight;
        cin >> end >> weight;
        graph[start].push_back({ end, weight });
        visited[start] = false;
        visited[end] = false;
        count++;
    }
    this->number = count;
}

```

```

vector<char> FindingPath::GreedyAlgorithm() {
    double min;
    vector<char> result;
    result.reserve(this->number);
    result.push_back(this->start);

    char CurVertex = this->start;

    while (CurVertex != this->end) {
        char NextVertex;
        min = 100;
        bool found = false;

        for (auto& i : this->graph[CurVertex]) {
            if (!visited[i.first] && i.second < min) {
                min = i.second;
                NextVertex = i.first;
                found = true;
            }
        }
        visited[CurVertex] = true;

        if (!found) {
            if (!result.empty()) {
                result.pop_back();
                CurVertex = result.back();
            }
            continue;
        }
        CurVertex = NextVertex;
        result.push_back(CurVertex);
    }

    return result;
}

int main() {
    setlocale(LC_ALL, "Russian");
    FindingPath answer;
    answer.Read();
    answer.Sort();
    vector<char> out = answer.AStar();
    cout << "Ответ:";
    for (auto& i : out) {
        cout << i;
    }
    return 0;
}

```