МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

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

Студент гр. 9383 _____ Мосин К.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Разработать две программы с жадным и A* алгоритмами соответственно.

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

Выполнение работы.

1. Заполняется std::map с ключом-вершиной и значением, равным всем соседним вершинам.

2. Сортировка std::map по значению, если выполняется жадный алгоритм, или по приоритету, если выполняется A*.

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.

Улучшения

- Для жадного алгоритма сортировка std::map значений по ключу.
- Для A* использовалась приоритетная очередь.

Тестирование.

Результаты тестирования представлены в таблице 1 и таблице 2 для жадного и A* алгоритмов соответственно.

Табл. 1 - Результаты тестирования жадного алгоритма

Входные данные	Выходные данные
a e a b 4 a c 3 a d 2 a e 1	ae
a d a b 1 b a 1 a c 1 c a 1 b d 1 c d 1	abd

Табл. 2 - Результаты тестирования A*

Входные данные	Выходные данные
a ea b 4 a c 3 a d 2 a e 1	ae
a d a b 1 b a 1 a c 1 c a 1	acd

b d 1 c d 1	

Анализ алгоритма.

Вставка n элеменов в std::map занимает O(nlogn). Помимо взятия элемента производится поиск пути от данного элемента до конечного. Поиск в std::map n раз занимает O(nlogn). Итого: O(nlogn) + O(nlogn) = O(2nlogn).

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.

Вывод

В ходе выполнения лабораторной работы был разработан жадный алгоритм, а так же использован алгоритм A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: API.h

```
#pragma once

#include "iostream"
#include "string"
#include "map"
#include "unordered_map"
#include "algorithm"
#include "vector"
#include "queue"

bool input(std::map<char, std::vector<std::pair<char, float>>>& graph, char& start, char& end, std::istream& stream);
bool possible(std::map<char, std::vector<std::pair<char, float>>>& graph, std::string& current, char from, char link, char end);

namespace Greed {
   std::string way(std::map<char, std::vector<std::pair<char, float>>>& graph, char start, char end);
}

namespace Star {
   std::string way(std::map<char, std::vector<std::pair<char, float>>>& graph, char start, char end);
   int heuristic(char a, char b);
}
```

Название файла: API.cpp

```
#include "API.h"
```

```
bool input(std::map<char, std::vector<std::pair<char, float>>>& graph, char& start, char& end, std::istream& stream) {
    stream >> start >> end;
    if (start < 'a' || start > 'z' || end < 'a' || end > 'z') {
        return false;
    }

    float weight = 0;
    for (char node1 = '\0', node2 = '\0'; stream >> node1 >> node2;) {
        if (!(stream >> weight) || node1 < 'a' || node1 > 'z' || node2 < 'a' || node2 > 'z') {
            return false;
        }
        graph[node1].push_back(std::make_pair(node2, weight));
    }

    if (!graph.count(start)) {
        return false;
    }

    std::string temp;
    return possible(graph, temp, start, start, end);
}

bool possible(std::map<char, std::vector<std::pair<char, float>>>& graph, std::string& current, char from, char link, char end) {
    if (link == end) {
        return true;
    }

    for (auto it = graph[link].begin(); it != graph[link].end(); ++it) {
```

```
        if (it->first == end) {
            return true;
        }
        else if (graph.count(it->first) && it->first != from && std::find(current.begin(), current.end(), link) == std::end(current)) {
            return possible(graph, current, link, it->first, end);
        }
    }
    return false;
}

std::string Greed::way(std::map<char, std::vector<std::pair<char, float>>>& graph, char start, char end) {
    std::string result;
    result += start;

    while (start != end) {
        for (auto it = graph[start].begin(); it != graph[start].end(); ++it) {
            if (possible(graph, result, start, it->first, end)) {
                start = it->first;
                break;
            }
        }
        result += start;
    }

    return result;
}

std::string Star::way(std::map<char, std::vector<std::pair<char, float>>>& graph, char start, char end) {
```

```
    std::unordered_map<char, char> came_from;
    std::unordered_map<char, float> cost_so_far;

    std::priority_queue<std::pair<char, double>, std::vector<std::pair<char, double>>,
std::greater<std::pair<char, double>>> frontier;
    frontier.emplace(0, start);

    came_from[start] = start;
    cost_so_far[start] = 0;

    char current;
    while (!frontier.empty()) {
        current = frontier.top().second;
        frontier.pop();

        if (current == end) {
            break;
        }

        for (auto &next : graph[current]) {
            float new_cost = cost_so_far[current] + next.second;
            if (!cost_so_far.count(next.first) || new_cost < cost_so_far[next.first]) {
                cost_so_far[next.first] = new_cost;
                float priority = new_cost + heuristic(next.first, end);
                frontier.emplace(priority, next.first);
                came_from[next.first] = current;
            }
        }
    }

    std::vector<char> result;
```

```
    current = end;
    result.push_back(current);

    while (current != start) {
        current = came_from[current];
        result.push_back(current);
    }
    std::reverse(result.begin(), result.end());

    std::string path;
    for (int i = 0; i < result.size(); ++i) {
        path += result[i];
    }

    return path;
}

int Star::heuristic(char a, char b) {
    return std::abs(a - b);
}
```

Название файла: main.cpp

```
#include "API.h"

int main(int argc, char *argv[]) {
    std::cout << "INPUT:" << std::endl;
    char start, end;
    std::map<char, std::vector<std::pair<char, float>>> graph;
    if (input(graph, start, end, std::cin)) {
        std::string convert(argv[1]);
        if (convert == "Greed") {
```

```
            for (auto it = graph.begin(); it != graph.end(); ++it) {
                std::sort(it->second.begin(), it->second.end(), [](std::pair<char, float> &a, std::pair<char, float> &b) -> bool {return a.second < b.second;});
            }
            std::cout << Greed::way(graph, start, end) << std::endl;
        }
        else if (convert == "Star"){
            for (auto it = graph.begin(); it != graph.end(); ++it) {
                std::sort(it->second.begin(), it->second.end(), [end](std::pair<char, float> &a, std::pair<char, float> &b) -> bool {return Star::heuristic(a.first, end) < Star::heuristic(b.first, end);});
            }
            std::cout << Star::way(graph, start, end) << std::endl;
        }
    }

    return 0;
}
```

ПРИЛОЖЕНИЕ В
ТЕСТЫ

Название файла: test.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "API.h"

TEST_CASE("Input syntax") {
    char start, end;
    std::map<char, std::vector<std::pair<char, float>>> graph;

    SECTION("Check start and finish input syntax") {
        std::istringstream stream("a 3");
        REQUIRE(input(graph, start, end, stream) == false);
        graph.clear();
    }

    SECTION("Check vertices and weight input syntax") {
        std::istringstream stream("a e\n a 2 b");
        REQUIRE(input(graph, start, end, stream) == false);
        graph.clear();
    }

    SECTION("Check for a start vertex") {
        std::istringstream stream("a c\n b c 1");
        REQUIRE(input(graph, start, end, stream) == false);
        graph.clear();
    }

    SECTION("Check for a finish vertex") {
```

```
        std::istringstream stream("a c\n a b 1");
        REQUIRE(input(graph, start, end, stream) == false);
        graph.clear();
    }

    SECTION("Valid input syntax") {
        std::istringstream stream("a e\n a b 1\nb e 1");
        REQUIRE(input(graph, start, end, stream) == true);
        graph.clear();
    }
}

TEST_CASE("algorithm") {
    /*
    a d
    a b 1
    a c 2
    b d 3
    c d 1
    */
    std::map<char, std::vector<std::pair<char, float>>> graph = {{'a',{{'b', 1},{'c', 2}}},{'b',{{'d',3}}},{'c',{{'d',1}}}};
    SECTION("Greed") {
        REQUIRE(Greed::way(graph, 'a', 'd') == "abd");
    }

    SECTION("Star") {
        REQUIRE(Star::way(graph, 'a', 'd') == "acd");
    }
}
```