

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

**ОТЧЕТ**  
**по практической работе №2**  
**по дисциплине «Построение и Анализ Алгоритмов»**  
**Тема: Алгоритмы на графах**

Студент гр. 8382

\_\_\_\_\_

Вербин К.М.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

### **Цель работы.**

Ознакомится с реализацией алгоритмов: жадный и A\*, написать программу на языке программирования C++.

### **Вариант 3**

Написать функцию, проверяющую эвристику на допустимость и монотонность.

### **Жадный алгоритм**

#### **Постановка задачи**

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

### **Описание алгоритма**

Решение поставленной задачи осуществляется с помощью рекурсивной функции `funk(std::vector* vector, char curChar, char endChar, std::vector* answer)`, которая и реализует жадный алгоритм. Вначале всегда происходит проверка – не является ли подаваемая начальная вершина равной конечной. Далее, пройдя проверку, создается пустой временный вектор путей(`temporaryVector`) и начинается проход по вектору исходному. В этом проходе происходит отбор нужных нам путей, т.е. тех, у которых `nameFrom`(имя родителя) равняется текущей вершине, которую мы рассматриваем. Если это так, то записываем путь (в котором родитель – рассматриваемая вершина) в текущий временный вектор(`temporaryVector`). Далее, так как нужен самый “дешевый” путь, то вызываем функцию сортировки вершин(`sort`), которая сортирует вершины по наименьшему весу ребер. И потом рекурсивно вызываем функцию `func()`, только теперь текущая вершина будет первая (минимальная по весу ребра) из вектора `temporaryVector`.

### **Описание используемого класса**

Класс путей(`class Path`): Класс путей состоит из следующих свойств:

- `char nameFrom` – имя вершины откуда исходит путь(т.е. ребро графа);
- `char nameOut` – имя вершины куда входит путь(т.е. ребро графа);
- `double weightPath` – вес ребра графа; Класс пути содержит следующие

методы:

- `Path(char nameFrom, char nameOut, double weightPath)` – конструктор, принимающий имя вершины начала, имя вершины конца и вес этого ребра.
- `char getNameFrom()` – метод, для получения поля `nameFrom` класса `Path`.
- `getNameOut()` – метод, для получения поля `nameOut` класса `Path`.

- `double getWeightPath()` – метод, для получения поля `weightPath` класса `Path`.

### **Описание `main ()` :**

В функции прописан ввод вершины начальной, вершины конечной, а также ввод ребер с их весами, вызовы функций и выходы промежуточных данных на консоль.

### **Описание дополнительных функций**

Функция `bool comp(Path a, Path b)`, принимает две переменные класса `Path`, и сравнивает их по весу.

### **Сложность алгоритма по памяти**

Сложность для жадного алгоритма оценивается как  $O(|V+E|)$ , так как в исходном векторе может максимум храниться число всех ребер, и с каждой новой вершиной мы ‘проваливаемся’ в рекурсию и создаем новый вектор.

### **Сложность алгоритма по времени**

Сложность для жадного алгоритма оценивается как  $O(|E \log E|)$ , так как в худшем случае будет совершаться обход по всем ребрам, а изначальная сортировка ребер по длине имеет сложность  $O(|E \log E|)$ .

### **Алгоритм $A^*$**

#### **Постановка задачи.**

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

### **Описание алгоритма**

Вначале создаются 2 вектора: с открытыми вершинами и закрытыми вершинами. (Изначально в вектор открытых вершин кладем начальную вершину). Первым делом из списка открытых вершин выбирается вершина, с наименьшей  $f(x)$  ( $f(x) = g(x) + h(x)$  где  $g(x)$  - стоимость пути от начальной вершины до вершины  $x$ , а  $h(x)$  – значение эвристической функции). (Выбор наименьшего значения осуществляется путем функции `sort()` и компаратора). Выбранная вершина удаляется из открытого списка и записывается в закрытый. Далее происходит проход по детям текущей вершины. Если ребенок не записан в открытый список, то записывается. Затем начинает обновляться информация о ребенке, а именно: кратчайший путь до него и значение эвристической функции и записывается информация откуда мы в него пришли (кто родитель). Цикл продолжается до тех пор, пока вектор открытых вершин будет не пуст. Далее, когда функция доходит до искомой конечной вершины, происходит вызов функции для вывода ответа.

### **Описание используемых классов**

Класс вершин(class Top): Класс вершин состоит из следующих свойств:

- char name – имя вершины;
- double pathToTop – путь до текущей вершины;
- double heuristicF – эвристическая функция;
- char nameFromT – имя, откуда исходит вершина.
- vector coupled – вектор для исходящих из вершины вершин. Класс

вершин содержит следующие методы:

- Top(char name) – конструктор, принимающий имя вершины, используется для создания самой первой вершины.
- Top() – конструктор, для создания вершин по умолчанию

### **Описание дополнительных функций**

- bool comp(Top a, Top b) – функция принимает 2 переменные типа Top, используется для сортировки открытых вершин по значению эвристической функции.

- Top a – 1 вершина
- Top b – 2 вершина

- void answer(std::vector& vectorTops, char startTop, char endTop) – функция принимает вектор вершин, а также начальную и искомую вершины. Данная функция используется для вывода ответа.

- std::vector& vectorTops – вектор вершин
- char startTop – начальная вершина
- char endTop – искомая вершина

- void changeInfo(std::vector& vectorTops, std::vector& openVertexes, char a, char name, double temp\_G, char endTop) – функция принимает вектор вершин, вектор открытых вершин, имя ребенка вершины, которую рассматриваем, имя самой вершины, значение длины пути до него и имя

конечной вершины, для вычисления эвристической оценки. Данная функция используется для смены информации о вершине и ребенке вершины.

- `std::vector& vectorTops` – вектор вершин
- `std::vector& openVertexes` – вектор открытых вершин
- `double temp_G` – имя ребенка вершины
- `char endTop` – имя самой вершины

• `int whatNumber(char a, std::vector& vectorTops)` – функция принимает имя вершины и вектор вершин. Данная функция ищет вершину в векторе и возвращает её индекс.

- `char a` – имя вершины
- `std::vector& vectorTops` – вектор вершин

• `bool check(std::vector& vectorPath, std::vector& vectorTops, char endTop, bool flagM, bool flagAd)` – функция принимает вектор пути, вектор вершин и имя искомой вершины. Данная функция сначала выполняет проверку эвристики на монотонность, по свойству монотонности: Эвристическая функция  $h(v)$  называется монотонной, если для любой вершины  $v1$  и её потомка  $v2$  разность  $h(v1)$  и  $h(v2)$  не превышает фактического веса ребра  $c(v1, v2)$  от  $v1$  до  $v2$ , а эвристическая оценка целевого состояния равна нулю. Далее, в случае, когда она не монотонна, функция выполняет проверку эвристики на допустимость по следующему свойству:

- `std::vector& vectorPath` – вектор пути
- `std::vector& vectorTops` – вектор вершин
- `char endTop` – имя искомой вершины
- `bool flagM` - флаг монотонности
- `bool flagAd` – флаг эвристики

Говорят, что эвристическая оценка  $h(v)$  допустима, если для любой вершины  $v$  значение  $h(v)$  меньше или равно весу кратчайшего пути от  $v$  до цели. Данная проверка происходит в случае, когда монотонность не

выполнилась, так как существует теорема: Любая монотонная эвристика допустима, однако обратное неверно.

### **Сложность алгоритма по памяти**

Сложность для алгоритма  $A^*$  оценивается как  $O(n+e)$  при более точной эвристической функции, так как программе приходится хранить граф целиком. В случае не точной эвристической функции придется просматривать каждое ребро. Тогда сложность может оказаться экспоненциальной ( $O(2^e)$ ).

### **Сложность алгоритма по времени**

Сложность алгоритма по операциям зависит от эвристики  $|h(x)-h^*(x)| \leq O(\log h^*(x))$ . В цикле выполняется  $e$  итераций, где  $e$  – количество рёбер в графе. На каждой итерации в массиве кратчайших расстояний выполняется поиск вершины с кратчайшим до неё путём с учётом эвристической функции ( $n$  операций, где  $n$  – количество вершин графа). Далее идёт цикл по всем соседям вершины со всеми прямыми путями до них ( $e$ ). Проверка на непосещённость вершины из текущей занимает  $n$  операций, проверка на содержание этой вершины в массиве кратчайших расстояний тоже занимает  $n$  операций. Если вершина содержится в массиве кратчайших расстояний, то на поиск вершины в этом массиве и замену величины кратчайшего пути до неё (если найден лучше) тратится  $(n + n)$  операций. Все вставки и извлечения из массивов константны по операциям. Получаем  $O(e(n + e(n + n + n + n))) = O(e(n + 4 * e * n)) = O(e * n + 4 * n * e * e)$  в лучшем случае, где  $e$  – количество рёбер в графе,  $n$  – количество вершин.  $O(2^e)$  в худшем случае,  $e$  – количество рёбер.

### **Тестирование**

Input	Output
<b>Жадный Алгоритм</b>	
a e a b 3.0	abcde



b c 1.0 c d 1.0 a d 5.0 d e 1.0	
a h a b 3.0 a c 1.0 a d 2.0 b e 5.0 c f 2.0 d g 6.0 e h 1.0 f h 3.0 g h 1.0	acfh
a g a b 1.0 b c 1.0 a d 3.0 d e 1.0 d f 2.0 f g 4.0	adfg
a e a b 2.0 b c 1.0 a d 3.0	Error!
<b>Алгоритм А*</b>	
Input	Output
a e a b 3.0 b c 1.0	ade

c d 1.0 a d 5.0 d e 1.0	
a h a b 3.0 a c 1.0 a d 2.0 b e 5.0 c f 2.0 d g 6.0 e h 1.0 f h 3.0 g h 1.0	acfh
a g a b 1.0 b c 1.0 a d 3.0 d e 1.0 d f 2.0 f g 4.0	adfg
a e a b 2.0 b c 1.0 a d 3.0	Error!

### **Вывод**

В результате работы была написана полностью рабочая программа, выполняющая поставленную задачу.

## ПРИЛОЖЕНИЕ

### КОД ПРОГРАММЫ

#### Жадный алгоритм

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>    // std::setw

using namespace std;

class Path {
private:
    char nameFrom;
    char nameOut;
    double weightPath;

public:
    Path(char nameFrom, char nameOut, double weightPath)
        : nameFrom(nameFrom), nameOut(nameOut), weightPath(weightPath) {}

    char getNameFrom() const {
        return nameFrom;
    }

    char getNameOut() const {
        return nameOut;
    }

    double getWeightPath() const {
        return weightPath;
    }
};

bool comp(Path a, Path b) { //comparator
    return a.getWeightPath() < b.getWeightPath();
}

bool func(std::vector<Path>* vector, char curChar, char endChar,
std::vector<char>* answer, int depth) {
    depth++;

    std::cout << setw(depth + 1) << ' ' << "Processing the vertex:  " << curChar <<
std::endl;
```

```

    if (curChar == endChar) { //exit from recursion
        std::cout << setw(depth + 1) << ' ' << "Reached the desired peak  " <<
endChar << ". The function returns TRUE and shuts down." << std::endl;
        return true;
    }

    std::cout << setw(depth + 1) << ' ' << "Search for paths leading from this
vertex." << std::endl;
    std::vector<Path> temporaryVector;
    temporaryVector.reserve(0);
    for (Path path : *vector) { // all vertexes in the vector will be passed
        if (path.getNameFrom() == curChar) { //selects all paths from the desired
vertex
            std::cout << setw(depth + 1) << ' ' << "Since the vertex  " <<
path.getNameOut() << "  comes from the current vertex, writing this path to the
vector." << std::endl;
            temporaryVector.emplace_back(path); //written to a vector
        }
    }

    //since we need the cheapest way we will cohabit

    std::cout << setw(depth + 1) << ' ' << "Sorting of vertices of minimum weight."
<< std::endl;

    std::sort(temporaryVector.begin(), temporaryVector.end(), comp);

    for (Path path : temporaryVector) { //going through all the vertexes
        if (func(vector, path.getNameOut(), endChar, answer, depth)) { //new variable
            depth--;
            std::cout << setw(depth + 1) << ' ' << "Writing a vertex  " <<
path.getNameOut() << "  in the response vector" << std::endl;
            answer->emplace_back(path.getNameOut());
            return true;
        }
    }

    return false;
}

int main() {
    setlocale(LC_ALL, "rus");

```

```

int depth = 0;
int flag = 1;

std::vector<Path> vector;
vector.reserve(0);

std::vector<char> answer;
answer.reserve(0);

char startChar;
char endChar;
std::cout << "Please, enter the starting vertex and ending vertex, as well as the
edges of the graph with its weight: " << std::endl;

std::cin >> startChar;
std::cin >> endChar;

char start, end;
double weight;

while (std::cin >> start >> end >> weight) {
    vector.emplace_back(Path(start, end, weight));
}

std::cout << "The greedy algorithm function is started" << std::endl;

if (!func(&vector, startChar, endChar, &answer, depth))
{
    std::cout << "Error!" << std::endl;
    flag = 0;
}

if (flag)
{
    std::cout << "The greedy algorithm function shuts down" << std::endl;
    std::cout << "The initial vertex is added to the response vector." << std::endl;
    answer.emplace_back(startChar);
    std::cout << "Reversing the response vector." << std::endl;
    std::reverse(answer.begin(), answer.end());

    std::cout << "Answer:  " << std::endl;
    for (char sym : answer) {
        std::cout << sym;
    }
}

```

```

    }
}

```

```

    return 0;
}

```

### Алгоритм A\*

```
include <iostream>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <set>
```

```
class Top { //vertex class
```

```
public:
```

```
    char name;
```

```
    double pathToTop; //path to the current vertex - g
```

```
    double heuristicF; //heuristic function - f
```

```
    char nameFromT;
```

```
    std::vector<char> coupled; //vector for vertexes originating from a vertex
```

```
    Top(char name) //constructor1 - required to fill in the initial vertex
```

```
    : name(name) {
```

```
        heuristicF = 0;
```

```
        pathToTop = -1; //will be used for unprocessed vertexes instead of the infinity
```

```
sign
```

```
        nameFromT = '-';
```

```
    }
```

```
    Top() { //constructor2
```

```
        name = '!'; //
```

```
        heuristicF = 0;
```

```
        pathToTop = -1; //
```

```
        nameFromT = '-';
```

```
    }
```

```
};
```

```
class Path { //path class. It stores only the path: from where to where and how much
the path weighs
```

```
public:
```

```
char nameFromP;  
char nameOutP;  
double weightPath;
```

```
Path(char nameFromP, char nameOutP, double weightPath)  
    : nameFromP(nameFromP), nameOutP(nameOutP), weightPath(weightPath)  
{}
```

```
char getNameFromP() const {  
    return nameFromP;  
}
```

```
char getNameOutP() const {  
    return nameOutP;  
}
```

```
double getWeightPath() const {  
    return weightPath;  
}  
};
```

```
bool check(std::vector<Path>& vectorPath, std::vector<Top>& vectorTops, char  
endTop, bool flagM, bool flagAd)  
{  
    std::cout << "\nThe monotony and tolerance check function is started " <<  
std::endl;  
    if (abs(endTop - endTop) != 0) {  
        std::cout << "\nThe heuristic estimate of the target state is not zero!" <<  
std::endl;  
        flagM = false;  
  
    }  
  
    for (unsigned int i = 0; i < vectorPath.size(); i++) {  
  
        if ((abs(endTop - vectorPath[i].nameFromP) - abs(endTop -  
vectorPath[i].nameOutP)) > vectorPath[i].weightPath) {  
            std::cout << "\nThe monotony is broken." << std::endl;  
            flagM = false;  
  
        }  
    }  
}
```

```

    }
    //checking for validity
    if (!flagM)
    {
        for (unsigned int i = 0; i < vectorTops.size(); i++) {

            if ((abs(endTop - vectorTops[i].name) > (vectorTops[vectorTops.size() -
1].pathToTop - vectorTops[i].pathToTop)))
            {
                std::cout << "\nTolerance is violated." << std::endl;
                flagAd = false;
            }

        }
    }
    if (flagM)
    {
        std::cout << "\nThe heuristic is monotonous and valid!" <<
std::endl; //monotonna i dopustima
        return true;
    }
    else if (!flagM && flagAd)
    {
        std::cout << "\nThe heuristic is valid!" << std::endl; //dopustima
        return true;
    }
    else
    {
        std::cout << "\nThe heuristic is not monotonous and is not allowed!" <<
std::endl;
        return false;
    }
}

```

```

int whatNumber(char a, std::vector<Top>& vectorTops) {

    for (unsigned int i = 0; i < vectorTops.size(); i++) {
        if (vectorTops[i].name == a) {

```



```

        return i;
    }
}
return -1;
}

bool comp(Top a, Top b) { //comparator, used for sorting in an open list
    return a.heuristicF < b.heuristicF;
}

void answer(std::vector<Top>& vectorTops, char startTop, char endTop)
{
    std::cout << "\nThe response output function starts. \n" << std::endl;
    std::vector<Top> answer;
    answer.reserve(0);
    Top temp = vectorTops[whatNumber(endTop, vectorTops)];
    std::cout << "\tWriting the last vertex to the response vector " << endTop <<
std::endl;
    answer.emplace_back(temp);
    while (temp.name != startTop) {
        temp = vectorTops[whatNumber(temp.nameFromT, vectorTops)];
        std::cout << "\tWriting a vertex to the response vector " << temp.name <<
std::endl;
        answer.emplace_back(temp);
    }
    std::cout << "\nReversing the response vector " << std::endl;
    std::reverse(answer.begin(), answer.end()); //since it was filled in the reverse
order, we do reverse
    std::cout << "\nAnswer: " << std::endl;
    for (Top ans : answer) {
        std::cout << ans.name;
    }
    std::cout << std::endl;
}

void changeInfo(std::vector<Top>& vectorTops, std::vector<Top>&
openVertexes, char a, char name, double temp_G, char endTop)
{
    std::cout << "\nUpdate information: \n" << std::endl;

    vectorTops[whatNumber(a, vectorTops)].nameFromT = name;

    vectorTops[whatNumber(a, vectorTops)].pathToTop = temp_G;
    openVertexes[whatNumber(a, openVertexes)].nameFromT = name;
    openVertexes[whatNumber(a, openVertexes)].pathToTop = temp_G;
}

```

```

    openVertexes[whatNumber(a, openVertexes)].heuristicF = temp_G +
abs(endTop - a);
    std::cout << "\tVertex " << a << " set that comes from " << name <<
std::endl;
    std::cout << "\tPath to the vertex " << a << " = " << temp_G << std::endl;
    std::cout << "\tF(" << a << ") (Heuristic estimation + cost) = " << temp_G +
abs(endTop - a) << std::endl;
    std::cout << "\nThe end of the update information. \n" << std::endl;
}

```

```

bool A(std::vector<Path>& vectorPath, std::vector<Top>& vectorTops, char
startTop, char endTop) {

```

```

    Top temp;
    double temp_G;
    std::vector<Top> closedVertexes;
    closedVertexes.reserve(0);
    std::vector<Top> openVertexes;
    openVertexes.reserve(0);

```

```

    std::cout << "\tAdding a vertex to the vector of open vertexes " <<
vectorTops[0].name << std::endl;

```

```

    openVertexes.emplace_back(vectorTops[0]);

```

```

while (!openVertexes.empty()) {
    Top min = openVertexes[0];
    std::cout << "\nSort the open tops of the \n" << std::endl;
    std::sort(openVertexes.begin(), openVertexes.end(), comp);
    temp = openVertexes[0]; //minimum f from openVertexes
    std::cout << "\tCurrent vertex: " << temp.name << std::endl;

```

```

    if (temp.name == endTop) {
        std::cout << "\nThe current vertex is equal to the one you are looking for,
so we call the response output function.\n" << std::endl;
        answer(vectorTops, startTop, endTop);
        return true;
    }

```

```

    std::cout << "\tAdding a vertex " << openVertexes[0].name << " in the
vector of closed vertexes. And delete it from the vector of open vertexes." <<
std::endl;

```

```

    closedVertexes.emplace_back(temp); //adding the processed vertex
    openVertexes.erase(openVertexes.begin()); //deleting the processed vertex

```

```

    for (unsigned int i = 0; i < temp.coupled.size(); i++) { //for each neighbor

```

```

        if (whatNumber(temp.coupled[i], closedVertexes) != -1) { //if the neighbor
is in closedVertexes (already processed)
            continue;
        }
        int j = 0;
        while (true) {
            if (vectorPath[j].nameFromP == temp.name && vectorPath[j].nameOutP
== temp.coupled[i]) {
                std::cout << "\tCounting the value of the shortest path to the vertex  "
<< vectorPath[j].nameOutP << std::endl;
                temp_G = vectorPath[j].weightPath + temp.pathToTop;
                std::cout << "\tShortest path to the top = " << temp_G << std::endl;
                break;
            }
            j++;
        }

        if (whatNumber(temp.coupled[i], openVertexes) == -1) { //if the neighbor
is not in openVertexes
            std::cout << "\tSince the neighboring vertex  " << temp.coupled[i] << "
if it is not in the vector of open vertexes, then we add it to it." << std::endl;
            openVertexes.emplace_back(vectorTops[whatNumber(temp.coupled[i],
vectorTops)]); //adding a neighbor
            std::cout << "\nCalling the information update function." << std::endl;
            changeInfo(vectorTops, openVertexes, temp.coupled[i], temp.name,
temp_G, endTop);
        }
        else {
            if (temp_G < openVertexes[whatNumber(temp.coupled[i],
openVertexes)].pathToTop) {
                std::cout << "\nSince a shorter path was found(" << temp_G << ") up
to top  " << temp.coupled[i] << ". Update information." << std::endl;
                changeInfo(vectorTops, openVertexes, temp.coupled[i], temp.name,
temp_G, endTop);
            }
        }

    }
}
return false;
}

```

```

int main() {

```

```

setlocale(LC_ALL, "Russian");
bool flag = true;
bool flagM = true;
bool flagAd = true;
std::vector<Path> vectorPath;//vector paths
vectorPath.reserve(0);
std::vector<Top> vectorTops;//vector tops
vectorTops.reserve(0);

char startTop;
char endTop;
std::cout << "Please enter the starting vertex and ending vertex, as well as the
edges of the graph with its weight: " << std::endl;

std::cin >> startTop;
std::cin >> endTop;

char start, end;
double weight;

while (std::cin >> start >> end >> weight) {
    vectorPath.emplace_back(Path(start, end, weight));
}

std::set<char> set;//

set.insert(startTop);//inserting the first vertex

vectorTops.emplace_back(Top(startTop));//creating the initial vertex and putting
it in the vector

int number;

for (Path path : vectorPath) {going through the path vector
    char from = path.getNameFromP();//
    char out = path.getNameOutP();

    if (set.find(from) == set.end()) {checks that there is no from in the set
        set.insert(from);
        vectorTops.emplace_back(Top(from));
    }
    if (set.find(out) == set.end()) {
        set.insert(out);
    }
}

```

```

        vectorTops.emplace_back(Top(out));

    }
}
//the path vector is full, but the neighbor vector is not =>
//performing a pass through the path vector again
for (Path path : vectorPath) { //going through the path vector
    char from = path.getNameFromP(); //
    char out = path.getNameOutP();

    if (set.find(from) != set.end()) { //checks that the set has from
        number = whatNumber(from, vectorTops);
        vectorTops[number].coupled.emplace_back(out); //adding a vertex neighbor
    }

}

vectorTops[0].pathToTop = 0;
vectorTops[0].heuristicF = abs(endTop - startTop);
std::cout << "\nLaunching the algorithm function A*!\n" << std::endl;

if (!A(vectorPath, vectorTops, startTop, endTop)) {
    flag = false;
    std::cout << "\nERROR! ERROR! ERROR!" << std::endl;
}

if (flag)
{
    check(vectorPath, vectorTops, endTop, flagM, flagAd);
}
}

```