

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. 8382

Торосян Т.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с реализацией жадного алгоритма и алгоритма A*,
написать программу на языке программирования C++.

Вариант 3

Написать функцию, проверяющую эвристику на допустимость и
монотонность.

Жадный алгоритм.

Описание используемого класса.

Класс путей(class Edge):

Класс путей состоит из следующих свойств:

- char from – имя вершины откуда исходит ребро графа;
- char to – имя вершины куда входит ребро графа;
- double weight – вес ребра графа;

Класс пути содержит следующие методы:

- Edge(char from, char to, double weight) – конструктор, принимающий имя вершины начала, имя вершины конца и вес этого ребра.
- char getFrom() – метод, для получения поля nameFrom класса Edge.
- getTo() – метод, для получения поля nameOut класса Edge.
- double getWeight() – метод, для получения поля weightEdge класса Edge.

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",

"с"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес.

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

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 `func(std::vector<Edge>* vector, char curChar, char endChar, std::vector<char>* answer)`, реализующей жадный алгоритм. На этапе инициализации происходит проверка – не является ли подаваемая начальная вершина равной конечной.

Затем создается пустой временный вектор `ребер` (`temporaryVector`) и начинается проход по исходному вектору, в ходе которого происходит отбор подходящих путей, т.е. тех, у которых *nameFrom*(имя родителя) равняется текущей вершине, которую мы рассматриваем. Если это так, то записываем путь-ребро (в котором родитель – рассматриваемая вершина) в текущий временный вектор (`temporaryVector`).

Далее, для нахождения самого “дешевого” пути, вызывается функция сортировки вершин(sort), которая сортирует вершины по наименьшему весу ребер, после чего рекурсивно вызывается функция func() с тем условием, что теперь текущая вершина будет первой (минимальной по весу ребра) в векторе temporaryVector.

Описание main () :

Функция осуществляет ввод начальной и конечной вершин, списка ребер с весами, запуск алгоритма на введенных данных и вывод промежуточных результатов в консоль.

Описание дополнительных функций.

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

Сложность алгоритма по памяти.

Сложность для жадного алгоритма оценивается как $O(V+E)$, так как в исходном векторе хранится число всех ребер, а с каждой новой вершиной запускается рекурсия и создается 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 $O(E \log E)$.

Ал гор итм A*.

Описание используемых классов.

Класс вершин(class Vertex):

Класс вершин состоит из следующих свойств:

- char name – имя вершины;
- double EdgeToVertex – путь до текущей вершины;
- double heuristicF – эвристическая функция;

- `char nameFromT` – имя, откуда исходит вершина.
- `vector<char> coupled` – вектор для исходящих из вершины вершин.

Класс вершин содержит следующие методы:

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

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

В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

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.

Изначально создаются 2 вектора: с “открытыми” и “закрытыми” вершинами. (Изначально в вектор открытых вершин кладем начальную вершину). Первым делом из списка открытых вершин выбирается вершина, с наименьшим значением 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 потомкам текущей вершины. Если потомка нет в открытый список, он записывается в него. Затем обновляется информация о вершине-потомке: кратчайший путь до него, значение эвристической функции и название вершины-предка.

Цикл продолжается до тех пор, пока вектор открытых вершин не будет пуст. После этого, когда функция доходит до конечной вершины, вызывается функция для вывода ответа.

Описание дополнительных функций.

- `bool comp(Vertex a, Vertex b)` – функция принимает 2 переменные типа `Vertex`, используется для сортировки открытых вершин по значению эвристической функции.
- `void answer(std::vector<Vertex>& vectorVertices, char startVertex, char endVertex)` – функция принимает вектор вершин, а также начальную и искомую вершины. Данная функция используется для вывода ответа.
- `void changeInfo(std::vector<Vertex>& vectorVertices, std::vector<Vertex>& openVertexes, char a, char name, double temp_G, char endVertex)` – функция принимает вектор вершин, вектор открытых вершин, имя потомка вершины, которую рассматриваем, имя самой вершины, значение длины пути до него и имя конечной вершины, для вычисления эвристической оценки. Данная функция используется для обновления информации о вершине и ребенке

вершины.

- `int whatNumber(char a, std::vector<Vertex>& vectorVertices)` – функция принимает имя вершины и вектор вершин. Данная функция ищет вершину в векторе и возвращает её индекс.
- `bool check(std::vector<Edge>& vectorEdge, std::vector<Vertex>& vectorVertices, char endVertex, 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: говорят, что 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 выполнялась, так как существует теорема: *Любая монотонная эвристика допустима, однако обратное неверно.*

Сложность алгоритма по памяти.

Сложность для алгоритма A^* оценивается как $O(|V+E|)$, так как программе приходится хранить граф целиком.

Сложность алгоритма по времени.

Временная сложность алгоритма A^* зависит от эвристики. В данном случае, $N \cdot \log(N)$ - сложность сортировки, где N длина очереди в данный момент тогда ответ $O(1 + N \cdot \log(N)(V+E))$, упрощая - $O(N \cdot \log(N)(V+E))$.

Тестирование жадного алгоритма.

Входные данные:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

```
"C:\Program Files\CodeBlocks\Projects\123\bin\Debug\123.exe"
c d 1.0
a d 5.0
d e 1.0 !!!
Starting Mr. Greedy...
Current vertex: a
Lookin for edges coming from this vertex.
Since the vertex b comes from the current vertex, writing this edge to the vector.
Since the vertex d comes from the current vertex, writing this edge to the vector.
Sorting vertices to find minimum weight.
Current vertex: b
Lookin for edges coming from this vertex.
Since the vertex c comes from the current vertex, writing this edge to the vector.
Sorting vertices to find minimum weight.
Current vertex: c
Lookin for edges coming from this vertex.
Since the vertex d comes from the current vertex, writing this edge to the vector.
Sorting vertices to find minimum weight.
Current vertex: d
Lookin for edges coming from this vertex.
Since the vertex e comes from the current vertex, writing this edge to the vector.
Sorting vertices to find minimum weight.
Current vertex: e
Reached the final vertex - e.
Writing a vertex e into result vector
Writing a vertex d into result vector
Writing a vertex c into result vector
Writing a vertex b into result vector
The Greedy is going down.
The initial vertex is added to the result.
Reversing the result vector.
Answer:
abcde
Process returned 0 (0x0) execution time : 3.852 s
Press any key to continue.
```

Входные данные:

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

Ответ: acfh

Входные данные:

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

Ответ: adfg

Тестирование алгоритма A*.

Входные данные:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

```
"C:\Program Files\CodeBlocks\Projects\123\bin\Debug\123.exe"
Input:
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0 !!!
Starting A*:
Adding a vertex to the open vertexes vector:  a
Open vertices sort:
Current vertex:  a
Moving a to the vector of closed vertexes.
Evaluating the shortest path to the b
It's about 3
b is not in the open vtcs vector, putting it into...
Calling the information update function.
vertex data update:
vertex b is set. Its ancestor:  a
Path to the b is 3
Heuristic estimation for a vertex b is 6
Evaluating the shortest path to the d
It's about 5
d is not in the open vtcs vector, putting it into...
Calling the information update function.
vertex data update:
vertex d is set. Its ancestor:  a
Path to the d is 5
Heuristic estimation for a vertex d is 6
Open vertices sort:
Current vertex:  b
Moving b to the vector of closed vertexes.
Evaluating the shortest path to the c
It's about 4
c is not in the open vtcs vector, putting it into...
Calling the information update function.
vertex data update:
vertex c is set. Its ancestor:  b
Path to the c is 4
Heuristic estimation for a vertex c is 6
Open vertices sort:
Current vertex:  d
Moving d to the vector of closed vertexes.
Evaluating the shortest path to the e
It's about 6
```

```
"C:\Program Files\CodeBlocks\Projects\123\bin\Debug\123.exe"
Vertex data update:
Vertex d is set. Its ancestor:  a
Path to the d is 5
Heuristic estimation for a vertex d is 6
Open vertices sort:
Current vertex:  b
Moving b to the vector of closed vertexes.
Evaluating the shortest path to the c
It's about 4
c is not in the open vtcs vector, putting it into...
Calling the information update function.
vertex data update:
vertex c is set. Its ancestor:  b
Path to the c is 4
Heuristic estimation for a vertex c is 6
Open vertices sort:
Current vertex:  d
Moving d to the vector of closed vertexes.
Evaluating the shortest path to the e
It's about 6
e is not in the open vtcs vector, putting it into...
Calling the information update function.
vertex data update:
vertex e is set. Its ancestor:  d
Path to the e is 6
Heuristic estimation for a vertex e is 6
Open vertices sort:
Current vertex:  c
Moving c to the vector of closed vertexes.
Open vertices sort:
Current vertex:  e
The current vertex is equal to the one you are looking for, so we call the answer function.
The answer function starts.
writing down the last vertex.e
writing down the vertex.d
writing down the vertex.a
Reversing result...
Answer:
ade
Starting the monotony and validness function check.
The heuristic is monotonous and valid!

Process returned 0 (0x0)   execution time : 10.742 s
Press any key to continue.
```

Входные данные:

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

Ответ: acfh

Входные данные:

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

Ответ: adfg

Входные данные:

a e
a b 2.0
b c 1.0
a d 3.0

Ответ: Ошибка! (возникла т. к. невозможно достичь конечной вершины)

Вывод.

В результате работы были реализованы A* и жадный алгоритмы, выполняющие поставленную задачу. Реализован анализ эвристической функции на монотонность и допустимость.

Приложение.

Файл Greedy.cpp(жадный алгоритм):

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

using namespace std;

class Edge {
private:
    char from;
    char to;
    double weight;

public:
    Edge(char from, char to, double weight)
        : from(from), to(to), weight(weight) {}

    char getFrom() const {
        return from;
    }

    char getTo() const {
        return to;
    }

    double getweight() const {
        return weight;
    }
};

bool comp(Edge a, Edge b) {
    return a.getweight() < b.getweight();
}

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

```

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

if (curChar == endChar) { //exit from recursion
    std::cout << setw(depth + 1) << ' ' << "Reached the final vertex - " << endChar << "." << std::endl;
    return true;
}

std::cout << setw(depth + 1) << ' ' << "Lookin for edges coming from this vertex." << std::endl;
std::vector<Edge> temporaryVector;
temporaryVector.reserve(0);
for (Edge Edge : *vector) { // all vertexes in the vector will be passed
    if (Edge.getFrom() == curChar) { //selects all Edges from the desired vertex
        std::cout << setw(depth + 1) << ' ' << "Since the vertex  " << Edge.getTo() << "  comes from the current vertex,
writing this edge to the vector." << std::endl;
        temporaryVector.emplace_back(Edge); //written to a vector
    }
}

//since we need the cheapest way we will cohabit

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

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

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

return false;
}

int main() {

```

```
setlocale(LC_ALL, "rus");

int depth = 0;
int flag = 1;

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

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

char startChar;
char endChar;
std::cout << "Input: ";

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

char start, end;
double weight;

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

std::cout << "Starting Mr. Greedy..." << std::endl;

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

if (flag)
{
    std::cout << "The Greedy is going down." << std::endl;
    std::cout << "The initial vertex is added to the result." << std::endl;
    answer.emplace_back(startChar);
    std::cout << "Reversing the result vector." << std::endl;
```

```

std::reverse(answer.begin(), answer.end());

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

return 0;
}

```

Файл Astar.cpp(алгоритм A*):

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

class Vertex { //vertex class

public:

    char name;
    double pathToVertex; //path to the current vertex - g
    double heuristicF; //heuristic function - f
    char nameFromT;
    std::vector<char> coupled; //vector for vertexes originating from a vertex

    Vertex(char name) //constructor1 - required to fill in the initial vertex
    : name(name) {
        heuristicF = 0;
        pathToVertex = -1; //will be used for unprocessed vertexes instead of the infinity sign
        nameFromT = '-';
    }

    Vertex() { //constructor2
        name = '!'; //
        heuristicF = 0;
        pathToVertex = -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<Vertex>& vectorVertices, char endVertex, bool flagM, bool  
flagAd)
```

```
{  
    std::cout << "Starting the monotony and validness function check." << std::endl;  
    if (abs(endVertex - endVertex) != 0) {  
        std::cout << "The heuristic estimate of the target state is not zero!" << std::endl;  
        flagM = false;  
    }  
}
```



```

for (unsigned int i = 0; i < vectorPath.size(); i++) {

    if ((abs(endVertex - vectorPath[i].nameFromP) - abs(endVertex - vectorPath[i].nameOutP)) >
vectorPath[i].weightPath) {

        std::cout << "The monotony isn't as good." << std::endl;

        flagM = false;

    }

}

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

        if ((abs(endVertex - vectorVertices[i].name) > (vectorVertices[vectorVertices.size() - 1].pathToVertex -
vectorVertices[i].pathToVertex)))
        {
            std::cout << "It isn't valid." << std::endl;

            flagAd = false;

        }

    }

}

if (flagM)
{
    std::cout << "The heuristic is monotonous and valid!" << std::endl;

    return true;

}

else if (!flagM && flagAd)
{
    std::cout << "The heuristic is valid!" << std::endl;

    return true;

}

else
{
    std::cout << "The heuristic is neither monotonous nor valid!" << std::endl;

    return false;

}

```

```
}
```

```
int whatNumber(char a, std::vector<Vertex>& vectorVertices) {
```

```
    for (unsigned int i = 0; i < vectorVertices.size(); i++) {
```

```
        if (vectorVertices[i].name == a) {
```

```
            return i;
```

```
        }
```

```
    }
```

```
    return -1;
```

```
}
```

```
bool comp(Vertex a, Vertex b) { //comparator, used for sorting in an open list
```

```
    return a.heuristicF < b.heuristicF;
```

```
}
```

```
void answer(std::vector<Vertex>& vectorVertices, char startVertex, char endVertex)
```

```
{
```

```
    std::cout << "The answer function starts." << std::endl;
```

```
    std::vector<Vertex> answer;
```

```
    answer.reserve(0);
```

```
    Vertex temp = vectorVertices[whatNumber(endVertex, vectorVertices)];
```

```
    std::cout << "Writing down the last vertex." << endVertex << std::endl;
```

```
    answer.emplace_back(temp);
```

```
    while (temp.name != startVertex) {
```

```
        temp = vectorVertices[whatNumber(temp.nameFromT, vectorVertices)];
```

```
        std::cout << "Writing down the vertex." << temp.name << std::endl;
```

```
        answer.emplace_back(temp);
```

```
    }
```

```
    std::cout << "Reversing result..." << std::endl;
```

```
    std::reverse(answer.begin(), answer.end()); //since it was filled in the reverse order, we do reverse
```

```
    std::cout << "Answer: " << std::endl;
```

```
    for (Vertex ans : answer) {
```

```
        std::cout << ans.name;
```

```
    }
```

```

    std::cout << std::endl;
}

void changeInfo(std::vector<Vertex>& vectorVertices, std::vector<Vertex>& openVertexes, char a, char name, double
temp_G, char endVertex )
{
    std::cout << "Vertex data update: " << std::endl;

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

    vectorVertices[whatNumber(a, vectorVertices)].pathToVertex = temp_G;
    openVertexes[whatNumber(a, openVertexes)].nameFromT = name;
    openVertexes[whatNumber(a, openVertexes)].pathToVertex = temp_G;
    openVertexes[whatNumber(a, openVertexes)].heuristicF = temp_G + abs(endVertex - a);
    std::cout << "Vertex " << a << " is set. Its ancestor: " << name << std::endl;
    std::cout << "Path to the " << a << " is " << temp_G << std::endl;
    std::cout << "Heuristic estimation for a vertex " << a << " is " << temp_G + abs(endVertex - a) << std::endl;
}

bool A(std::vector<Path>& vectorPath, std::vector<Vertex>& vectorVertices, char startVertex, char endVertex) {

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

    std::cout << "Adding a vertex to the open vertexes vector: " << vectorVertices[0].name << std::endl;

    openVertexes.emplace_back(vectorVertices[0]);

    while (!openVertexes.empty()) {
        Vertex min = openVertexes[0];
        std::cout << "Open vertices sort: " << std::endl;
        std::sort(openVertexes.begin(), openVertexes.end(), comp);
        temp = openVertexes[0]; //minimum f from openVertexes
        std::cout << "Current vertex: " << temp.name << std::endl;

        if (temp.name == endVertex) {

```

```

        std::cout << "The current vertex is equal to the one you are looking for, so we call the answer function." <<
std::endl;

        answer(vectorVertices, startVertex, endVertex);

        return true;
    }

    std::cout << "Moving " << openVertexes[0].name << " to the vector of closed 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 << "Evaluating the shortest path to the " << vectorPath[j].nameOutP << std::endl;
                temp_G = vectorPath[j].weightPath + temp.pathToVertex;
                std::cout << "It's about " << temp_G << std::endl;
                break;
            }
            j++;
        }

        if (whatNumber(temp.coupled[i], openVertexes) == -1) { //if the neighbor is not in openVertexes
            std::cout << temp.coupled[i] << " is not in the open vtcs vector, putting it into..." << std::endl;
            openVertexes.emplace_back(vectorVertices[whatNumber(temp.coupled[i], vectorVertices)]); //adding a
neighbor
            std::cout << "Calling the information update function." << std::endl;
            changeInfo(vectorVertices, openVertexes, temp.coupled[i], temp.name, temp_G, endVertex);
        }
        else {
            if (temp_G < openVertexes[whatNumber(temp.coupled[i], openVertexes)].pathToVertex) {
                std::cout << "A shorter path up to " << temp.coupled[i] << " was found (" << temp_G << "). Update..."<<
std::endl;
                changeInfo(vectorVertices, openVertexes, temp.coupled[i], temp.name, temp_G, endVertex);
            }
        }
    }
}

```

```

    }
    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<Vertex> vectorVertices;//vector Vertices
    vectorVertices.reserve(0);

    char startVertex;
    char endVertex;
    std::cout << "Input: " << std::endl;

    std::cin >> startVertex;
    std::cin >> endVertex;

    char start, end;
    double weight;

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

    std::set<char> set;

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

    vectorVertices.emplace_back(Vertex(startVertex));//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);
    vectorVertices.emplace_back(Vertex(from));
}
if (set.find(out) == set.end()) {
    set.insert(out);
    vectorVertices.emplace_back(Vertex(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, vectorVertices);
        vectorVertices[number].coupled.emplace_back(out); //adding a vertex neighbor
    }

}

vectorVertices[0].pathToVertex = 0;
vectorVertices[0].heuristicF = abs(endVertex - startVertex);
std::cout << "Starting A*!" << std::endl;

if (!A(vectorPath, vectorVertices, startVertex, endVertex)) {
    flag = false;
    std::cout << "Unexpected error!" << std::endl;
}

if (flag)
{
    check(vectorPath, vectorVertices, endVertex, flagM, flagAd);
}

```

