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 поиска пути в графах
Вариант 2

Студент гр. 8303 _____ Данилов А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучение алгоритмов поиска пути в графе.

Задание 1.

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

Задание 2.

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

Индивидуализация.

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

Описание жадного алгоритма.

В программе используется жадный алгоритм поиска пути. Он заключается в выборе ребра с минимальной длиной на каждом шаге, в надежде, что итоговое решение будет оптимальным.

На каждом шаге рассматривается ребро с минимальным весом, выходящее из текущей вершины. Найдя минимальный путь, переходим к следующему ребру. Если из текущей вершины нет пути, то помечаем вершину как посещенную (и более не используем) и откатываемся назад.

Сложность алгоритма по операциям.

В худшем случае алгоритм обойдет все ребра графа. Тогда сложность будет равна O(N * E), где N — количество вершин, E — количество ребер.

Так как для хранения ответа используется вектор, то в худшем случае придется запомнить все вершины графа. Значит сложность алгоритма будет составлять O(N), где N – количество вершин.

Описание алгоритма A*

A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению $f(x) = g(x) + h(x)$. Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Сложность алгоритма по операциям зависит от эвристики

$|h(x)-h^*(x)| \leq O(\log h^*(x))$, где h^* - оптимальная эвристика, т.е. точная оценка расстояния из вершины в x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

В лучшем случае, когда эвристическая функция оптимальна, сложность будет равна O(N+E), где N – количество вершин, E – количество ребер. Максимальная длина пути – N и на каждом шаге требуется пройти по всем ребрам, выходящим из этой вершины, чтобы найти путь с наименьшим приоритетом.

В худшем случае, когда вершины расположены случайным образом и эвристическая функция дает результат, не соотносящийся с реальным расстоянием до цели, будут рассмотрены все пути. Отсюда сложность по времени $O(2^E)$, где E – количество ребер в графе.

Сложность по памяти O(N), где N — количество вершин, т. к. алгоритм хранит вектор из элементов с приоритетом (т.е. очередь с приоритетом), массив открытых и закрытых вершин и в худшем случае в каждый их них будет добавлено N элементов, где N — количество вершин.

Описание структур данных.

```
struct Node
{
   float distance = 0;
   bool isVizited = false;
   //A*
   float gX = 0;  // len
   float hX = 0;  // heuristic
   char prevNode = '\0';
};
```

Структура для описания вершины. Поле distance задает расстояние между текущей вершиной и предыдущей. Поле visited задает флаг для проверки посещения текущей вершины. Так же для алгоритма a* введены поля gX, hX, prevTop означающие дистанцию от старта до текущей вершины, значение эвристической функции для данной вершины и вершина, из которой пришли.

```
struct PriorityOne
{
   char next = '\0';
   char prev = '\0';
   float gX = 0;  // len
   bool operator==(const PriorityOne &tmp) const
   {
      return tmp.next == next && tmp.prev == prev && tmp.gX == gX;
   }
};
```

Структура для описания очереди. Поля next, gX, prev задают значения следующей вершины, расстояния до нее и предыдущей вершины.

Оператор сравнения нужен для корректной работы алгоритма aStar.

```
class Graph
{
   std::vector<std::vector<Node>> matrix;
   std::string result;
```

public:
…

}

класс, описывающий граф и методы для работы с ним. Сам граф задается матрицей с элементами типа Node. result — контейнер, использующийся для хрпнения пути

Описание методов класса Graph.

- Graph() - конструктор класса.

- bool greedyOne(char start, char end) — метод, описывающий жадный алгоритм.
  start — стартовая вершина.
  end — конечная вершина.
  Возвращает true, если путь существует, false в противном случае.

- bool aStar(char start, char end) — метод, описывающий алгоритм A*.
  start — стартовая вершина.
  end — конечная вершина.
  Возвращает true, если путь существует, false в противном случае.

- void addEdge(char start, char end, float distance) — метод для добавления ребра в граф,
  start — имя вершины, из которой выходит ребро.
  end — имя вершины, в которую входит ребро.
  distance — расстояние между вершинами.

- bool isInPlenty(std::string& plenty, char node) const— метод для проверки наличия рассматриваемой вершины в заданном множестве,
  plenty — множество вершин.
  node — искомая вершина.
  Возвращает true, если это так, false в противном случае.

- bool isAnyOne() const — метод для проверки существования непройденных ребер, служит флагом завершения в жадном алгоритме,

- void clear() - метод для очистки стартовых состояний, необходим при последовательной работе двух алгоритмов,

- void setStepicHeuristics(char node) — метод, задающий эвристику для степика
  node — конечная вершина,

- void setOwnHeuristics() - метод для задания собственной эвристики в качетстве входных данных,

- void setGX(char next, char prev, float distance) — метод для задания расстояния пути от старта до текущей вершины,
  next — индекс вершины в матрице графа, в которую входит ребро.
  prev — символьное название вершины, из которой выходит ребро.
  distance — расстояние от старта, до следующей вершины.

- void restorePath(char start, char end) — метод для восстановления пути при работе с алгоритмом A*,
  start — стартовая вершина.
  end— конечная вершина.

- void updateClosedNodes (std::string& open, std::string& closed, char node) — метод для обновления множеств вершин.
  open — множество вершин, требующих обработки.
  closed — множество обработанных вершин.
  node — вершина, перемещаемая из множества open в closed.

- void printDistance (std::vector<PriorityOne>& que) const — метод для вывода текущего состояния очереди (расстояний между вершинами).

- void printResult() const — метод для вывода пути.

Тестирование

1. Жадный алгоритм

1.1

```
b e
a b 1.0
a c 2.0
b d 7.0
b e 8.0
a g 2.0
b g 6.0
c e 4.0
d e 4.0
g e 1.0
.
Greedy algoritm
Start node is b
g is added to result
Current result is: bg
e is added to result
Current result is: bge
Shortest way is: bge
```

1.2

```
a g
a b 3.0
a c 1.0
b d 2.0
b e 3.0
d e 4.0
e a 3.0
e f 2.0
a g 8.0
f g 1.0
c m 1.0
m n 1.0
.
```

```
Greedy algoritm
Start node is a
c is added to result
Current result is: ac
m is added to result
Current result is: acm
n is added to result
Current result is: acmn
There is no way from:  n  to: g
Current result is: acm
There is no way from:  m  to: g
Current result is: ac
There is no way from:  c  to: g
Current result is: a
b is added to result
Current result is: ab
d is added to result
Current result is: abd
u is added to result
Current result is: abdu
There is no way from:  u  to: g
Current result is: abd
e is added to result
Current result is: abde
u is added to result
Current result is: abdeu
There is no way from:  u  to: g
Current result is: abde
f is added to result
Current result is: abdef
u is added to result
Current result is: abdefu
There is no way from:  u  to: g
Current result is: abdef
g is added to result
Current result is: abdefg
Shortest way is: abdefg
```

1.3

```
a d
a b 1.0
b c 1.0
c a 1.0
a d 8.0
.
Greedy algoritm
Start node is a
b is added to result
Current result is: ab
c is added to result
Current result is: abc
There is no way from:  c  to: d
Current result is: ab
There is no way from:  b  to: d
Current result is: a
d is added to result
Current result is: ad
Shortest way is: ad
```

2. Алгоритм A*

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
.
ASTAR algoritm
Enter heuristic for top (a) : 10
Enter heuristic for top (b) : 10
Enter heuristic for top (c) : 10
Enter heuristic for top (d) : 1
Enter heuristic for top (e) : 2
all nodes connected with (a) are visited
(a) was added to closed nodes
new iteration
In queue:
distance from (a) to (b) = 3
distance from (a) to (d) = 5
computing priority
f(x) of current min (a)→(b) = 13
current min f(b) = g(b) + h(b)  = 3 + 10 = 13
-------------------------------------
f(x) of current element of queue (a)→(b) = 13
current element f(b) = g(b) + h(b) = 3 + 10 = 13
-------------------------------------
computing priority
f(x) of current min (a)→(b) = 13
current min f(b) = g(b) + h(b)  = 3 + 10 = 13
-------------------------------------
f(x) of current element of queue (a)→(b) = 13
current element f(d) = g(d) + h(d) = 5 + 1 = 6
-------------------------------------
```

```
min value in queue: from (a) to (d) cost g(x) = 5
set g(e) = 6add way from (d) to (e) cost g(x) = 6
new iteration
In queue:
distance from (a) to (b) = 3
distance from (d) to (e) = 6
computing priority
f(x) of current min (a)→(b) = 13
current min f(b) = g(b) + h(b)  = 3 + 10 = 13
-------------------------------------
f(x) of current element of queue (a)→(b) = 13
current element f(b) = g(b) + h(b) = 3 + 10 = 13
-------------------------------------
computing priority
f(x) of current min (a)→(b) = 13
current min f(b) = g(b) + h(b)  = 3 + 10 = 13
-------------------------------------
f(x) of current element of queue (a)→(b) = 13
current element f(e) = g(e) + h(e) = 6 + 2 = 8
-------------------------------------
min value in queue: from (d) to (e) cost g(x) = 6
 set g(e) = 6
Shortest way is: ade
```

Вывод.

В ходе выполнения лабораторной работы были изучены алгоритмы поиска пути в графе путем написания программ, реализующих жадный алгоритм и A*.

Приложения А. Исходный код

```
#include <iostream>
#include <vector>
#include <string>
#include <cfloat>
#include <cmath>
#include <algorithm>

#define PRINT
#define GREEDY
#define ASTAR

constexpr size_t matrixSize = 26;


//constexpr size_t matrixSize = 26;

struct Node
{
    float distance = 0;
    bool isVizited = false;
    //A*
    float gX = 0;  // len
    float hX = 0;  // heuristic
    char prevNode = '\0';
};

struct PriorityOne
{
    char next = '\0';
    char prev = '\0';
    float gX = 0;  // len
    bool operator==(const PriorityOne &tmp) const
    {
        return tmp.next == next && tmp.prev == prev && tmp.gX == gX;
    }
};

// граф
class Graph
{
    std::vector<std::vector<Node>> matrix;
    std::string result;
public:
    //init
    Graph()
    {
        matrix.resize(matrixSize);
        for (auto i = 0; i < matrixSize; ++i)
            matrix[i].resize(matrixSize);
    }
    //--------------------------------------------------------------------------
--------
    // жадный алгоритм true == есть решение
    bool greedyOne(char start, char end)
    {
        result.push_back(start);
        if(start == end)
            return true;  // if a->a
        #ifdef PRINT
        std::cout << "Start node is " << start << std::endl;
        #endif
```

```
        auto minNode = start;

        while (isAnyOne())
        {
            auto minDistance = FLT_MAX;
            auto tmpNode = '-';

            // поиск ребра наименьшего размера из данной вершины
            for (auto i = 0; i < matrixSize ; i++) {
                // проверка на то, была ли вершина посещена и не находится ли
она в решении
                if (matrix[minNode - 'a'][i].isVizited || isInPlenty(result,
i+'a'))
                    continue;
                // нашли путь оптимальнее прежнего
                if (matrix[minNode - 'a'][i].distance < minDistance &&
                    matrix[minNode - 'a'][i].distance != 0)
                {
                    minDistance = matrix[minNode - 'a'][i].distance;
                    tmpNode = i; // вершина с мин путем из текущей
                }
            }
            // если нашли мин ребро,
            // то добавляем его в решение

            if (minDistance!=FLT_MAX && tmpNode!= '-') {
                matrix[minNode - 'a'][tmpNode].isVizited = true;
                result.push_back(tmpNode+'a');

                #ifdef PRINT
                std::cout << char(tmpNode+'a') << " is added to result" <<
std::endl;
                std::cout << "Current result is: ";
                printResult();
                #endif

                minNode = tmpNode + 'a';
            }
                //мин ребро не найдено -> тупиковая вершина
            else
            {
                if (result.empty())
                    break; // -> false
                //throw std::length_error("There is no way");
                auto lonelyOne = result[result.length()-1];
                result.pop_back();

                #ifdef PRINT
                std::cout << "There is no way from:  " << lonelyOne <<"  to: "
<< end<<  std::endl;
                std::cout << "Current result is: ";
                printResult();
                #endif

                // обнуляем все пути ведущие в эту вершину
                for (auto i = 0; i < matrixSize; i++)
                    matrix[i][lonelyOne ].isVizited = true;

                if (result.empty())
                    break; // -> false

                minNode = result[result.length()-1];
            }
```

```
            if (result[result.length()-1] == end)
                return true;
        }
        return false;
    }
    //-------------------------------------------------------------------------
----------
    // эвристический алгоритм true == есть решение

    bool aStar(char start, char end)
    {
        if(start == end)
        {
            result.push_back(start);
            return true;
        }

        std::vector<PriorityOne> que;
        std::string closed, open;
        char currentNode;
        open.push_back(start);

        // добавляем все вершины, доступные из начальной
        for (char nextNode = 0; nextNode < matrixSize; nextNode++)
        {
            currentNode = start - 'a';
            // есть путь
            if (matrix[currentNode][nextNode].distance != 0)
            {
                float tmpGX = matrix[currentNode][nextNode].distance +
                              matrix[currentNode][nextNode].gX;
                PriorityOne buf;
                buf.next = nextNode + 'a';
                buf.prev = start;
                buf.gX = tmpGX;

                matrix[currentNode][nextNode].isVizited = true;
                setGX(nextNode, start, tmpGX);
                que.push_back(buf);
            }
        }
        updateClosedNodes(open, closed, start);
        // пока есть хоть одно необработанное ребро
        while (!que.empty())
        {
            #ifdef PRINT
            std::cout << "new iteration\n";
            printDistance(que);
            #endif
            PriorityOne minNode = que[0];
            auto  iterPosition = que.begin();

            // нахождение минимального элемента в очереди и его извлечение
            for (auto iter : que)
            {
                float fXMin = minNode.gX + matrix[minNode.prev - 'a']
[minNode.next - 'a'].hX;
                float fXIter = iter.gX + matrix[iter.prev - 'a'][iter.next -
'a'].hX;

                #ifdef PRINT
                std::cout << "computing priority \n";
                printf("f(x) of current min (%c)->(%c) = %g\n",
```

```
                    minNode.prev, minNode.next, fXMin);
                printf("current min f(%c) = g(%c) + h(%c)  = %g + %g = %g\n",
                    minNode.next, minNode.next, minNode.next, minNode.gX,
                    matrix[minNode.prev - 'a'][minNode.next - 'a'].hX,
fXMin);
                std::cout << "-------------------------------------\n";

                printf("f(x) of current element of queue (%c)->(%c) = %g\n",
minNode.prev, minNode.next, fXMin);
                printf("current element f(%c) = g(%c) + h(%c) = %g + %g = %g\n",
                     iter.next, iter.next,iter.next, iter.gX,
                     matrix[iter.prev - 'a'][iter.next - 'a'].hX, fXIter);
                std::cout << "-------------------------------------\n";
                #endif

                if (fXMin > fXIter || (fXMin == fXIter && minNode.next <
iter.next))
                {
                    minNode = iter;
                    iterPosition = std::find(que.begin(), que.end(), iter);
                }
            }
            #ifdef PRINT
            printf("min value in queue: from (%c) to (%c) cost g(x) = %g\n",
                    minNode.prev, minNode.next, minNode.gX);
            #endif
            // если дошли до конечной вершины, восстанавливаем путь
            if (minNode.next == end)
            {
                #ifdef PRINT
                std::cout << "set g(" << minNode.next << ") = " << minNode.gX <<
std::endl;
                #endif
                setGX(minNode.next - 'a', minNode.prev, minNode.gX);
                restorePath(start,end);
                return true;
            } else
            {
            // не дошли до конца
                que.erase(iterPosition);
                updateClosedNodes(open, closed, minNode.prev);
                //смотрим всех сосеедй этой вершины
                for (char i = 0; i < matrixSize; i++)
                //по аналогии со стартом
                {
                    //есть ребро
                    if (matrix[minNode.next - 'a'][i].distance != 0)
                    {
                        float tmpGX = minNode.gX + matrix[minNode.next - 'a']
[i].distance;
                        PriorityOne buf;
                        buf.next = i + 'a';
                        buf.prev = minNode.next;
                        buf.gX = tmpGX;
                        //если элемент ведет в обработанную вершину ->
игнорируем
                        if (isInPlenty(closed, i + 'a'))
                            continue;
                        // если эту вершину еще не трогали -> трогаем и помещаем
в open
                        if(!isInPlenty(open, i + 'a'))
                        {
                            #ifdef PRINT
```

```
                            std::cout << "set g(" << buf.next << ") = " <<
tmpGX;
                            #endif
                            setGX(buf.next - 'a', buf.prev, tmpGX);
                            open += buf.next;
                        } else  // если уже проходили - смотрим, можем ли
уменьшить путь
                        {
                            if (matrix[i][0].gX > tmpGX)
                            {
                                #ifdef PRINT
                                std::cout << "update g(x) for node: " << 'a'+i
<<
                                std::endl << "old = " << matrix[i][0].gX << ",
new = " <<
                                tmpGX << std::endl;
                                #endif
                                // обновляем, если меньше
                                setGX(i,minNode.next, tmpGX);
                            }
                        }
                        #ifdef PRINT
                        printf("add way from (%c) to (%c) cost g(x) = %g\n",
                                buf.prev, buf.next, buf.gX);
                        #endif
                        que.push_back(buf);
                    }
                }
            }
        }
        return false;
    }

    //-------------------------------------------------------------------------
---------
    // добавление ребра
    void addEdge (char start, char end, float distance)
    {
        if(start - 'a' >= matrixSize || end - 'a' >= matrixSize || matrix[start
- 'a'][end - 'a'].distance != 0)
            throw std::range_error("Node should be from a to z") ;
        matrix[start - 'a'][end - 'a'].distance = distance;
    }

    //принадлежит ли вершина множеству
    bool isInPlenty (std::string& plenty, char node) const
    {
        for (auto &iter : plenty)
            if (iter == node)
                return true;
        return false;
    }
    // есть ли непросмотренное ребро
    bool isAnyOne() const
    {
        for (auto i = 0; i < matrixSize; i++)
            for (auto j = 0; j < matrixSize; j++)
                if (!matrix[i][j].isVizited)
                    return true;
        return false;
    }

    // необходим при работе обоих алгоритмов
```

```
    void clear()
    {
        for(auto i = 0; i < matrixSize; i++ )
            for(auto j = 0; j < matrixSize; j++ )
                matrix[i][j].isVizited = false;
        result.clear();
    }
    // необходимо для корректного прохода тестов на степике
    void setStepicHeuristics(char node)
    {
        for (auto i = 0; i < matrixSize; i++ )
            for (auto j = 0; j < matrixSize; j++ )
                matrix[i][j].hX = std::fabs(static_cast<float>(node - (j +
'a')));
    }

    // видоизменение за счет варианта; сам вводишь эвристику для каждой вершины
    void setOwnHeuristics()
    {
        constexpr double inaccuracy = 0.00000000001;
        bool flag;
        for (auto i = 0; i < matrixSize; i++)
        {
            flag = false;
            for (int j = 0; j < matrixSize ; j++)
            {
                if (matrix[i][j].distance != 0 && matrix[i][j].distance >
inaccuracy )
                {
                    flag = true;
                    break;
                }
            }
            if (!flag)
            {
                for (int j = 0; j < matrixSize ; j++)
                {
                    if (matrix[j][i].distance != 0 && matrix[j][i].distance >
inaccuracy)
                    {
                        flag = true;
                        break;
                    }
                }
            }
            if (!flag)
                continue;
            float val = -1;
            while (val < 0)
            {
                std::cout << "Enter heuristic for top (" << char(i + 'a') <<") :
";
                std::cin >> val;
            }
            for (auto j = 0; j < matrixSize; j++)
                matrix[j][i].hX = val;
        }
    }
    // установление расстояния от старта до текущей вершины
    void setGX(char next, char prev, float  distance)
    {
        for (char i = 0; i < matrixSize; i++){
            matrix[next][i].prevNode = prev;
```

```
            matrix[next][i].gX = distance;
        }
    }

    // восстановлениенайденного пути
    void restorePath(char start, char  end)
    {
        std::string reversedResult;
        auto prevNode = end;
        while (true)
        {
            if (prevNode == start || prevNode == '\0')
            {
                reversedResult += start;
                break;
            }
            reversedResult += prevNode;
            prevNode = matrix[prevNode - 'a'][0].prevNode;
        }
        for (auto i = reversedResult.rbegin(); i != reversedResult.rend(); i++)
            result += *i;
    }
    // добавление вершины в множесво закрытых (обработанных) вершин
    void updateClosedNodes (std::string& open, std::string& closed, char node)
    {
        for (char & iter : open)
            if (iter == node)
            {
                #ifdef PRINT
                std::cout << "all nodes connected with (" << iter << ") are
visited\n";
                std::cout << "(" << iter << ") was added to closed nodes" <<
std::endl;
                #endif

                closed.push_back(iter);
                auto tmpIter = std::find(open.begin(), open.end(), iter);
                open.erase(tmpIter);
                break;
            }
    }

    // выввод расстояний между вершинами
    void printDistance (std::vector<PriorityOne>& que) const
    {
        std::cout << "In queue:\n";
        for (auto item : que)
            std::cout << "distance from (" << item.prev << ") to (" << item.next
<< ") = "
            << item.gX << std::endl;
    }
    // вывод результата
    void printResult () const
    {
        std::cout << result << std::endl;
    }

};

int main() {
    char startNode, endNode, start, end;
    float distance;
    Graph graph;
```

```
    std::cin >> start >> end;
    std::cin >> startNode;
    while (startNode != '.')
    {
        std::cin >> endNode >> distance;
        try
        {
            graph.addEdge(startNode, endNode, distance);
        }
        catch (std::range_error &e)
        {
            std::cout <<"\n ERRROR :"<< e.what() << std::endl;
            break;
        }
        std::cin >> startNode;
    }
    #ifdef GREEDY

    #ifdef PRINT
        std::cout << "Greedy algoritm\n";
    #endif

    if (graph.greedyOne(start, end))
    {
        #ifdef PRINT
        std::cout << "Shortest way is: ";
        #endif

        graph.printResult();
    }
    else
        std::cout << "there is no way from: " << start << "to: " << end <<
std::endl;
    #endif

    #ifdef ASTAR

    #ifdef PRINT
    std::cout << "ASTAR algoritm\n";
    #endif
    graph.clear();
//    graph.setStepicHeuristics(end);
    graph.setOwnHeuristics();
    if (graph.aStar(start, end))
    {
        #ifdef PRINT
        std::cout << "Shortest way is: ";
        #endif

        graph.printResult();
    } else
        std::cout << "there is no way from: " << start << "to: " << end <<
std::endl;
    #endif

    return 0;
}
```