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. 8303 _____ Гришин К. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы

Изучить «Жадный алгоритм» и алгоритм A* для поиска пути в графе, а также имплементировать их на языке C++.

Задание, вариант 8

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.

Example input	output
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	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.

Example input	output
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade

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

*Код программы, а также Makefile располагаются в приложениях

Описание работы программы

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных предоставляется строка в которой перечислены все вершины, по которым необходимо пройти от начальной вершины до конечной.

Описание алгоритмов

Жадный алгоритм

Пусть вершина, из которой нужно искать путь — *src*; вершина, в которую нужно попасть — *dst*.

1. Инициализируется стек (*stack*); таблица посещения, в которой отмечается каждая вершина и была ли она уже посещена (*visited*); таблица связности, в которой каждой вершине соответствует вершина из которой в нее пришли (*prev*), изначально инициализорвана нулями, т. е. неопределенными вершинами

2. В *stack* заносится вершина *src*.

3. Пока в *stack* есть элементы:

    a) *u* ← элемент с верхушки *stack*

    b) если *u* == *dst*, восстановить путь по *prev*

    c) *stack.pop*

    d) если *u* посещена, пропустить ее

    e) отметить *u* как посещенную (*visited[u] = 1*)

    f) *neighbours* ← смежные с *u* вершины, упорядоченные по убыванию

    g) для всех не посещенных *n* (*visited[n] == 0*) в *neighbours*

        добавить *n* в стек

        в таблицу связности для *n* записать *u* (*prev[n] = u*)

4. Если *stack* пуст, то путь не найден

Алгоритм A*

Пусть вершина, из которой нужно искать путь — *src*; вершина, в которую нужно попасть — *dst*.

1. *hueristic(char)* — лямбда выражение, которое возвращает разность между кодами символов в таблице ASCII; *closed* — контейнер содержащий посещенные вершины; *open* — контейнер содержащий вершины, которые можно осмотреть на данной итерации; *prev* — таблица связности, в которой отмечается откуда была посещена вершина (изначально инициализируется нулями); *g_val* — таблица, отображающая значение пути пройденного от src дл вершины; *f_val* — таблица содержащая сумму соответствующих *g_val* и *hueristic*. (g_val и f_val изначально инициализируются бесконечностями)

2. Для каждой вершины в графе отсортировать смежные по возрастанию эвристической функции

4. g_val для src = 0, f_val для src = эвристической функции от src

3. Поместить src в open

4. Пока в open есть элементы:

a) u ← первый встретившийся наименьший элемент в open

b) удалить u из open

c) поместить u в closed

d) если u == dst, восстановить путь по prev

e) neighbours ← смежные с u вершины

a) для всех n в neighbours:

если n в closed, пропустить ее

alt_g ← расстояние от n до src; alt_f ← alt_g + hueristic(n)

если alt_g ≥ g_val для n, то пропустить

изменить g_val для n на alt_g (g_val[n] = alt_g)

изменить f_val для n на alt_f (f_val[n] = alt_f)

в таблицу связности для n добавить u (prev[n] = u)

добавить n в open, если его там нет

5. если open пуст, то путь не найден

Описание классов и их методов

Класс Vertex: содержит в себе поле *char name*, хранящее в себе имя элемента и контейнер *vector<Edge*>* хранящий в себе инцидентные к вершине ребра.

Методы класса:

Деструктор — удаляет все ребра из памяти

Конструктор(char n) — записывает в поле name значение n

*void addNeighbour(Vertex *other, float w)* — создает новое ребро с размером *w*, которое ведет от *this* к *other*

Класс Edge: содержит в себе поле *float w*, хранящее в себе размер ребра, а также два указателя на вершины: *Vertex *from* — вершина откуда ведет ребро и *Vertex *to* — вершина куда ведет ребро.

Методы класса:

Деструктор — удаляет себя из списка ребер у вершины *from*

Конструктор(Vertex from, Vertex *to, float w)* — инициализирует соответствующие переданным параметрам поля, добавляется в контейнер с ребрами у вершины *from*

Класс Graph: хранит в себе контейнер *vector<Vertex*>* хранящий в себе все вершины графа.

Методы класса:

Деструктор — удаляет все вершины из своего контейнера

Конструктор(bool logger) — открывает файл для логов, на вход получает значение флага *logger* = 1 — записывать логи в файл.

*vector<char> greedySearch(Vertex *src, Vertex *dst)* — возвращает путь от *src* до *dst*, найденный с помощью жадного алгоритма

*vector<char> astarSearch(Vertex *src, Vertex *dst)* — возвращает кратчайший путь от *src* до *dst*, найденный с помощью алгоритма *A**

int addVertex(char name) — создает вершину с названием *name*, если такой еще нет и добавляет в свой контейнер и возвращает 0, иначе возвращает 1

int addVertex(char name, char from, float w) — создает вершину *name*, в которую ведет ребро размером *w* из *from*. Если *name* уже существует, то возвращает 1; если *from* указана как 0 или равна *name*, то возвращается 3; если *from* не существует, то возвращается 2; иначе возвращается 0.

int delVertex(char name) — удаляет вершину с именем *name* и возвращает 0, если вершина не найдена, то возвращает 1.

int connectVertices(char from_n, char to_n) — создает ребро от *from_n* к *to_n*. Если *from_n* или *to_n* не существует, создает их. Причем, если обе вершины существуют, возвращает 0; если не было только *from_n*, то возвращает 1; если только *to_n*, то возвращает 2; если не существовало обеих вершин, то возвращает 3.

vector<char> search(char src_n, char dst_n, int type) — если вершины *src_n* и *dst_n* существуют, то запускает поиск, в зависимости от типа (0 — greedy, 1 — Dijkstra, 2 — A*) и возвращает путь. Если какая-то из вершин не существует, то возвращается пустой вектор.

Сложность алгоритмов

«Жадный» алгоритм представляет из себя поиск в глубину с выбором ближайшей вершины, т. к. граф хранится в виде списка связности и является направленным, по каждый вершине и каждому ребру алгоритм проходится один раз, следовательно ***сложность по операциям O(\|V\|+\|E\|)***. Помимо памяти, которая занята уже хранящимся графом, алгоритму требуется выделить стек и хеш-таблицы, которые не могут иметь размер больше, чем количество вершин в графе, следовательно ***сложность по памяти O(\|V\|)***.

A* алгоритм в худшем случае увеличивает количество исследуемых вершин экспоненциально, по сравнения с *длиной оптимального пути (d)*, то есть сложность в ***худшем случае равна*** $O(b^d)$, где *d — коэффициент ветвления*

(среднее число наследников каждой вершины). При этом 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, это значит, что для получения лучшей скорости работы алгоритма, ошибка эвристики не должна расти быстрее, чем логарифм от оптимальной эвристики. Во время работы алгоритму нужно два массива и три хеш-таблицы размером не больше |V|, следовательно ***сложность по памяти O(|V|)***.

Хранение решения

Граф представлен в виде списка связности вершин (vector<Vertex*> vertices), где каждая вершина хранит указатели на ребра, ведущие к другим вершинам.

Использованные оптимизации

Для алгоритма A* для каждой вершины, список ее инцидентных ребер сортируется по возрастанию эвристической функции.

Тестирование

Greedy Algorithm

input	output	log of first input
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	Both vertices from 'a' and to 'b' are created Path from 'a' to 'b' is built Vertex to 'c' created Path from 'b' to 'c' is built Vertex to 'd' created Path from 'c' to 'd' is built Node a and node dare exist Path from 'a' to 'd' is built Vertex to 'e' created Path from 'd' to 'e' is built
a e a b 1.0 a d 3.0 b c 2.0 c a 1.0 c e 1.0 c d 2.0 d e 4.0	abce	

a g a b 1.0 a d 4.0 a g 4.0 b c 1.0 c d 1.0 d g 1.0	abcdg	Initialized search from 'a' to 'e' Start greedy search: Initialize greedy algorithm Vertex from top of stack is 'a' Vertex unvisited neighbours: 'd' 'b' Vertex from top of stack is 'b' Vertex unvisited neighbours: 'c' Vertex from top of stack is 'c' Vertex unvisited neighbours: 'd' Vertex from top of stack is 'd' Vertex unvisited neighbours: 'e' Vertex from top of stack is 'e' Path found: abcde
a c a b 1.0 a z 1.0 b c 1.0 z c 1.0	azc	
a c a b 3.0 b c 1.0 c a 1.0	abc	
a d a b 1.0 b a 1.0 b c 1.0 c b 1.0 d c 1.0		

A* Algorithm

input	output	log of first input
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Both vertices from 'a' and to 'b' are created Path from 'a' to 'b' is built Vertex to 'c' created Path from 'b' to 'c' is built Vertex to 'd' created Path from 'c' to 'd' is built Node a and node dare exist Path from 'a' to 'd' is built Vertex to 'e' created Path from 'd' to 'e' is built
a e a b 1.0 a d 3.0 b c 2.0 c a 1.0 c e 1.0 c d 2.0 d e 4.0	abce	
a g a b 1.0 a d 4.0 a g 4.0 b c 1.0 c d 1.0 d g 1.0	ag	Initialized search from 'a' to 'e' Start A* search: Initialize A* algorithm Vertex 'a' with least f score = 4 Unvisited neighbours are: 'd' with new parameters: prev = 'a', dist = 5, f score = 6; earlier was: prev = 'nothing', dist = inf, f score = inf 'b' with new parameters: prev = 'a', dist = 3, f score = 6; earlier was: prev = 'nothing', dist = inf, f score = inf Vertex 'b' with least f score = 6 Unvisited neighbours are:
a c a b 1.0 a z 1.0 b c 1.0 z c 1.0	abc	

a c a b 3.0 b c 1.0 c a 1.0	abc	'c' with new parameters: prev = 'b', dist = 4, f score = 6; earlier was: prev = 'nothing', dist = inf, f score = inf Vertex 'c' with least f score = 6 Unvisited neighbours are: 'd' Vertex 'd' with least f score = 6 Unvisited neighbours are: 'e' with new parameters: prev = 'd', dist = 6, f score = 6; earlier was: prev = 'nothing', dist = inf, f score = inf Vertex 'e' with least f score = 6 path found: ade
a d a b 1.0 b a 1.0 b c 1.0 c b 1.0 d c 1.0		

Вывод

В ходе выполнения лабораторной работы были изучены «Жадный» алгоритм и алгоритм A* для поиска пути в графе. «Жадный» алгоритм не гарантирует нахождения кратчайшего пути, в свою очередь A* является модифицированной версией алгоритма Дийкстры и находит кратчайший путь от одной вершины до другой опираясь на эвристическую функцию предугаданного расстояния. Оба алгоритма имплементированы на языке C++, все промежуточные действия записываются в файл log, расположенный рядом с программой.

ПРИЛОЖЕНИЕ А

Файл edge.h

```
#ifndef EDGE_H
#define EDGE_H


struct Vertex;

struct Edge{
    friend struct Vertex;

    float w;
    class Vertex* from;
    class Vertex* to;

    ~Edge();

private:
    Edge(Vertex *from, Vertex *to, float w);
};

#endif
```

Файл edge.cpp

```
#include "edge.h"
#include "vertex.h"


Edge::~Edge()
{
    for(auto it = from->neighbours.begin();
        it < from->neighbours.end(); it++)
    {
        if(*it == this){
            from->neighbours.erase(it);
            break;
        }
    }
}



Edge::Edge(Vertex *from, Vertex *to, float w):
    from(from), to(to), w(w)
{
    from->neighbours.push_back(this);
}
```

ПРИЛОЖЕНИЕ В

Файл vertex.h

```
#ifndef VERTEX_H
#define VERTEX_H

#include <vector>

struct Edge;

struct Vertex{
    friend class Graph;

    char name;
    std::vector<Edge*> neighbours;

    ~Vertex();

    void addNeighbour(Vertex *other, float w);

private:
    Vertex(char name);
};

#endif
```

Файл vertex.cpp

```
#include "vertex.h"
#include "edge.h"

Vertex::~Vertex()
{
    for(auto edge: neighbours){
        delete edge;
    }
}


Vertex::Vertex(char name):
    name(name)
{}


void Vertex::addNeighbour(Vertex *other, float w)
{
    new Edge(this, other, w);
}
```

ПРИЛОЖЕНИЕ С

Файл main.cpp

```
#include <iostream>
#include <ostream>
#include <fstream>

#include <string>
#include <limits>
#include <algorithm>
#include <vector>
#include <map>
#include <stack>

#include "edge.h"
#include "vertex.h"


#define INF std::numeric_limits<float>::infinity()

#define GREEDY 0
#define DIJKSTRA 1
#define ASTAR 2

#define TYPE     ASTAR // type there type of search

class Graph{
    std::vector<Vertex*> vertices;
    std::ofstream log;  // log file descriptor
    bool logger = 0;    // is loggin on.


    std::vector<char> dijkstraSearch(const Vertex *src, const Vertex *dst){
        if(logger)
            log << "Initialize Dijkstra algorithm" << std::endl;
        // copy all vertices to another container, call it graph
        std::vector<Vertex*> graph = vertices;
        std::vector<char> path;

        // create containers of algorithm
        std::map<char, float> dist;
        std::map<char, char> prev;

        // initialize algorithm containers
        for(auto v: graph){
            dist[v->name] = INF;
            prev[v->name] = 0;
        }
        dist[src->name] = 0;

        while(!graph.empty()){
            // find vertex with min distance
            Vertex *u = graph.back();
            float d = dist[u->name];

            for(auto v: graph){
                if(dist[v->name] < d){
                    d = dist[v->name];
                    u = v;
                }
            }

            if(logger)
                log << "Vertex \'" << u->name << "\' with least distance to src = " << d << std::endl;

            // delete u from vertex set
            for(auto it = graph.begin(); it < graph.end(); it++)
```

```
            if(*it == u){
                graph.erase(it);
                break;
            }

        // restore path and return if dst target spotted
        if(u == dst){
            char v_n = u->name;
            if(prev[v_n] == 0)
                path.push_back(v_n);
            else
                while(v_n != 0){
                    path.insert(path.begin(), v_n);
                    v_n = prev[v_n];
                }
            if(logger){
                log << "path found: ";
                for(auto c: path)
                    log << c;
                log << std::endl << std::endl << std::endl;
            }

            return path;
        }

        // check neighbours
        if(logger){
            if(u->neighbours.empty())
                log << "Vertex doesn't have neighbours" << std::endl;
            else
                log << "Unvisited neighbours are:" << std::endl;
        }
        for(auto edge: u->neighbours){
            auto n = edge->to;

            // if neighbours is not in graph, it's already checked
            if(std::find(graph.begin(), graph.end(), n) == graph.end())
                continue;

            if(logger)
                log << "\'" << n->name << "\'";

            float alt = edge->w + dist[u->name];
            if(alt < dist[n->name]){
                if(logger){
                    log << " with new parameters: ";
                    log << "prev = \'" << u->name << "\', ";
                    log << "dist = " << alt << "; ";
                    log << "earlier was: prev = \'";
                    if(prev[n->name] == 0) log << "nothing";
                    else log << prev[n->name];
                    log << "\', dist = " << dist[n->name];
                }

                dist[n->name] = alt;
                prev[n->name] = u->name;
            }
            log << std::endl;
        }
    }
    log << "There is no path" << std::endl << std::endl;

    return path;
}


std::vector<char> greedySearch(Vertex *src, Vertex *dst){
    if(logger)
        log << "Initialize greedy algorithm" << std::endl;
```

```
// create algorithm containers
std::stack<Vertex*> stack;
std::map<char, bool> visited;
std::map<char, char> prev;

std::vector<char> path;

// intialize algorithm containers
for(auto v: vertices){
    visited[v->name] = 0;
    prev[v->name] = 0;
}
stack.push(src);

bool path_found = 0;
while(!stack.empty()){
    // get vertex from top of stack
    Vertex *u = stack.top();
    if(logger)
        log << "Vertex from top of stack is \'" << u->name << "\'" << std::endl;

    if(u == dst){
        path_found = 1;
        break;
    }
    stack.pop();

    // if vertex is already visited, skip it
    if(visited[u->name]){
        log << "Vertex is already visited" << std::endl;
        continue;
    }
    visited[u->name] = 1;

    // sort all neighbours descending
    std::sort(u->neighbours.begin(), u->neighbours.end(),
              [](const Edge* e1, const Edge* e2){
                  return e2->w < e1->w;
              });

    // add all non-visited neighbours on stack
    // neighbour with shortest edge will be on top of stack
    if(logger){
        if(u->neighbours.empty()){
            log << "Vertex doesn't have neighbours" << std::endl;
            continue;
        }
        else
            log << "Vertex unvisited neighbours: " << std::endl;
    }
    for(auto edge: u->neighbours){
        if(!visited[edge->to->name]){
            stack.push(edge->to);
            if(logger)
                log << "\'" << edge->to->name << "\' ";

            // set u as previous vertex of neighbour
            prev[edge->to->name] = u->name;
        }
    }
    log << std::endl;
}

if(!path_found){
    if(logger)
        log << "There's no path" << std::endl << std::endl;
    return path;
}
```

```
    char d_n = dst->name;
    // restore the way by map of coherency (prev)
    if(prev[d_n] != 0)
        while(d_n){
            path.insert(path.begin(), d_n);
            d_n = prev[d_n];
        }
    else{
        path.push_back(d_n);
    }

    if(logger){
        log << "Path found: ";
        for(auto c: path)
            log << c;
        log << std::endl << std::endl;
    }

    return path;
}


std::vector<char> astarSearch(Vertex *src, Vertex *dst){
    log << "Initialize A* algorithm" << std::endl;
    // lambda heuristic int this case
    auto heuristic = [dst](char c){
        return float(std::abs(dst->name - c));
    };

    // cmp to sort neighbours
    auto edgeCmp = [&heuristic](const Edge * a, const Edge * b){
        return heuristic(a->to->name) < heuristic(b->to->name);
    };

    // path container
    std::vector<char> path;

    // containers of algorithm
    std::vector<Vertex*> closed, open = {src};
    std::map<char, char> prev;
    std::map<char, float> g_val, f_val;

    // initialize algorithm containers
    for(auto v: vertices){
        // sort neighbours
        std::sort(v->neighbours.begin(), v->neighbours.end(), edgeCmp);
        prev[v->name] = 0;
        g_val[v->name] = INF;
        f_val[v->name] = INF;
    }
    g_val[src->name] = 0;
    f_val[src->name] = heuristic(src->name);

    while(!open.empty()){
        // find vertex with min distance (call it u)
        Vertex *u = open.back();
        float f = f_val[u->name];

        for(auto v: open){
            if(f_val[v->name] < f){
                f = f_val[v->name];
                u = v;
            }
        }

        if(logger)
            log << "Vertex \'" << u->name << "\' with least f score = " << f << std::endl;

        // delete u from vertices set
```

```
for(auto it = open.begin(); it < open.end(); it++){
    if(*it == u){
        open.erase(it);
        break;
    }
}
// add vertex to closed list
closed.push_back(u);

// restore path and return if dst spotted
if(u == dst){
    if(prev[u->name] == 0)
        path.push_back(u->name);
    else{
        char u_n = u->name;
        while(u_n){
            path.insert(path.begin(), u_n);
            u_n = prev[u_n];
        }
    }

    if(logger){
        log << "path found: ";
        for(auto c: path)
            log << c;
        log << std::endl << std::endl << std::endl;
    }

    return path;
}

// check neighbours
if(logger){
    if(u->neighbours.empty())
        log << "Vertex doesn't have neighbours" << std::endl;
    else
        log << "Unvisited neighbours are:" << std::endl;
}
for(auto edge: u->neighbours){
    auto n = edge->to;
    // if neighbour in closed list, skip it
    if(std::find(closed.begin(), closed.end(), n) != closed.end())
        continue;

    if(logger)
        log << "\'" << n->name << "\'";

    // find g and f of this neighbour
    float alt_g = g_val[u->name] + edge->w;
    float alt_f = alt_g + heuristic(n->name);

    // change neighbours characteristics
    if(alt_g < g_val[n->name]){

        if(logger){
            log << " with new parameters: ";
            log << "prev = \'" << u->name << "\', ";
            log << "dist = " << alt_g << ", ";
            log << "f score = " << alt_f << "; ";
            log << "earlier was: prev = \'";
            if(prev[n->name] == 0) log << "nothing";
            else log << prev[n->name];
            log << "\', dist = " << g_val[n->name] << ", ";
            log << "f score = " << f_val[n->name];
        }

        // add neighbour to open list if it doesn't in
        if(std::find(open.begin(), open.end(), n) == open.end())
            open.push_back(n);
```

```
                    g_val[n->name] = alt_g;
                    f_val[n->name] = alt_f;
                    prev[n->name] = u->name;
                }
                log << std::endl;
            }
        }
        log << "There is no path" << std::endl << std::endl;
        // returns empty path if there's no path
        return path;
    }



public:
    ~Graph()
    {
        for(auto vertex: vertices){
            delete vertex;
        }
        log.close();
    }

    Graph(bool logger = 0){
        this->logger = logger;
        log.open("log", std::ios::out | std::ios::trunc);
    }

    void logData(bool logger){
        this->logger = logger;
    }

    // add isolated vertex
    int addVertex(char name)
    {
        for(auto vertex: vertices){
            if(vertex->name == name)
                return 1; // Node allready exists
        }

        auto vertex = new Vertex(name);
        vertices.push_back(vertex);
        return 0;
    }


    // add vertex 'name' connected with 'from' by path with length 'w'
    // from -> name
    int addVertex(char name, char from, float w)
    {
        for(auto vertex: vertices){
            if(vertex->name == name)
                return 1; // Node allready exists
        }

        Vertex *neighbour = nullptr;

        if(from != 0 && from != name){
            int isIn = 0;
            for(auto vertex: vertices){
                if(vertex->name == from){
                    neighbour = vertex;
                    isIn = 1;
                    break;
                }
            }
            if(!isIn){
                return 2; // Neighbour doesn't exist;
```

```
        }
    }
    else return 3;

    auto vertex = new Vertex(name);
    neighbour->addNeighbour(vertex, w);
    vertices.push_back(vertex);
    return 0;
}


int delVertex(char name)
{
    for(auto it = vertices.begin();
        it < vertices.end(); it++)
    {
        if((*it)->name == name){
            delete *it;
            vertices.erase(it);
            return 0;
        }
    }

    return 1; // Vertex doesn't exist
}


// connect two vertices
// if any of the vertices doesn't exist, creates it and returns nonzero
int connectVertices(char from_n, char to_n, float w)
{
    int ret = 0;
    Vertex *from = nullptr;
    Vertex *to = nullptr;

    for(auto vertex: vertices){
        if(vertex->name == from_n)
            from = vertex;
        if(vertex->name == to_n)
            to = vertex;
        if(from != nullptr && to != nullptr) break;
    }

    if(from == nullptr){
        from = new Vertex(from_n);
        vertices.push_back(from);
        ret = 1;
    }
    if(to == nullptr){
        to = new Vertex(to_n);
        vertices.push_back(to);
        if(ret) ret = 3;
        else ret = 2;
    }

    if(logger){
        switch(ret){
        case 0:
            log << "Node " << from_n << " and "
                << "node " << to_n << "are exist" << std::endl;
            break;

        case 1:
            log << "Vertex from \'" << from_n << "\' created" << std::endl;
            break;

        case 2:
            log << "Vertex to \'" << to_n << "\' created" << std::endl;
            break;
```

```
        case 3:
            log << "Both vertices from \'" << from_n << "\' and to \'"
                << to_n << "\' are created" << std::endl;

        default: break;
        }
        log << "Path from \'" << from_n << "\' to \'" << to_n << "\' is built" << std::endl << std::endl;
    }


    from->addNeighbour(to, w);
    return ret;
}


// gets two nodes
// find path from src to dst
// type [0 - greedy; 1 - dijkstra; 2 - A*; others - empty path]
std::vector<char> search(char src_n, char dst_n, int type = 0){
    log << "Initialized search from \'" << src_n << "\' to \'"
        << dst_n << "\'" <<std::endl;

    Vertex *src = nullptr;
    Vertex *dst = nullptr;

    std::vector<char> path;

    for(auto vertex: vertices){
        if(vertex->name == src_n)
            src = vertex;
        if(vertex->name == dst_n)
            dst = vertex;
        if(src != nullptr && dst != nullptr) break;
    }

    if(src == nullptr || dst == nullptr){
        if(logger){
            if(src == nullptr)
                log << "Vertex \'" << src_n << "\' doesn't exist" << std::endl;
            if(dst == nullptr)
                log << "Vertex \'" << dst_n << "\' doesn't exist" << std::endl;
            log << "Search wasn't started" << std::endl;
        }
        return path;
    }

    switch(type){
    case 0:
        if (logger)
            log << "Start greedy search:" << std::endl;
        path = greedySearch(src, dst);
        break;

    case 1:
        if (logger)
            log << "Start Dijkstra search:" << std::endl;
        path = dijkstraSearch(src, dst);
        break;

    case 2:
        if (logger)
            log << "Start A* search:" << std::endl;
        path = astarSearch(src, dst);
        break;

    default: break;
    }
```

```
        return path;
    }



    // prints all pairs of connected vertices
    // also prints if vertex is isolated
    void print(std::ostream& os){
        for(auto vertex: vertices){
            if(vertex->neighbours.size() == 0)
                os << vertex->name << " isolated" << std::endl;
            else{
                for(auto edge: vertex->neighbours){
                    os << edge->from->name << " " << edge->to->name << " " << edge->w << std::endl;
                }
            }
        }
    }
};



int main(){
    auto graph = new Graph;
    graph->logData(1);

    // get src and dst of path
    char src, dst;
    std::cin >> src >> dst;

    // get graph (two points and length between)
    char from, to;
    float len;
    while(std::cin >> from && std::cin >> to && std::cin >> len){
        graph->connectVertices(from, to, len);
    }

    // get path between src and dst
    auto path = graph->search(src, dst, TYPE);

    // print path
    for(auto c: path){
        std::cout << c;
    }
    std::cout << std::endl;

    return 0;
}
```

ПРИЛОЖЕНИЕ D

Makefile

```
CC = g++
CFLAGS = -g -std=c++17


all: main

main: main.o edge.o vertex.o
	$(CC) $(CFLAGS) main.o edge.o vertex.o -o main

main.o: main.cpp
	$(CC) $(CFLAGS) -c main.cpp

edge.o: edge.h edge.cpp
	$(CC) $(CFLAGS) -c edge.cpp

vertex.o: vertex.h vertex.cpp
	$(CC) $(CFLAGS) -c vertex.cpp

clean:
	rm -rf *.o main
```