**void МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Алгоритмы поиска пути в графах»**

|  |  |
| --- | --- |
| Студентка гр. 7381 | Машина Ю. Д. |
| Преподаватель | Фирсов М. А. |

Санкт-Петербург

2019

**Цель работы.**

Ознакомиться с задачей построения пути в ориентированном графе. Для решения данной задачи будут рассмотрены алгоритмы жадного поиска и модификация этого алгоритма – алгоритм A\*, использующий некую эвристическую функцию для более оптимального выбора вершины. Необходимо построить и проанализировать сложность указанных алгоритмов, а также решить задачу из пункта индивидуализации.

**Задание 1 – Жадный алгоритм.**

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.

**Входные данные.**

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их 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

**Выходные данные.**

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

**Пример выходных данных.**

abcde

**Описание алгоритма 1 – Алгоритм жадного поиска.**

0. Указываем в качестве текущей вершины стартовую ().

1. Если достигнут финиш, заканчиваем алгоритм.

2. Если у вершины нет соседей возвращаемся к её предку ().

3. Среди соседей выбираем вершину с минимальным расстоянием .

4. Указываем предком вершины вершину ().

5. Исключаем ребро из графа.

6. Обновляем текущую вершину (), переходим к п 1.

Найдём асимптотику работы данного алгоритма по времени.

Пусть – количество рёбер, а – количество вершин в графе .

Очевидно, в худшем случае, придётся пройти по всем рёбрам графа и просмотреть все вершины, если финишная вершина недостижима, то есть. На каждой итерации необходимо находить вершину-соседа с минимальной дистанцией до неё. Для этой цели подойдёт либо сортировка исходных данных за , которая позволит получать ближайшего соседа за , либо использовать очередь с приоритетом, которая вне зависимости от реализации даст на добавление и извлечение ребра оценку . В зависимости от выбранного подхода, итоговая оценка имеет асимптотику .

Для оценки затрат памяти на данных алгоритм необходимо учесть структуру данных, в которой будет храниться исходный граф и массивы, в которых будут храниться предки текущей вершины и факт посещённости. Размер массивов не зависит от числа рёбер в графе и требует памяти. Если хранить граф в виде списка смежности, то потребуется памяти для хранения рёбер графа. Используя для хранения графа очередь с приоритетом, реализованную на двоичной куче, дополнительных затрат по памяти относительно обычного способа хранения графа в списках смежности не возникает и составляется те же памяти.

Таким образом, затраты по памяти для жадного алгоритма получаются .

**Задание 2 – Алгоритм A\*.**

Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом А\*. 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.

**Входные данные.**

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их 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

**Выходные данные.**

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

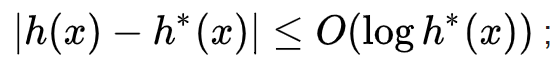
**Пример выходных данных.**

ade

**Описание алгоритма 2 – Алгоритм A\*.**

Граф хранится в виде списка смежности. В приоритетную очередь кладется вершина с которой начинается поиск. Пока очередь не окажется пуста или не наткнемся на искомую вершину, первая вершина в очереди заменяется ее потомками, после каждого пуша очередь сортируется по приоритету равному длине пути от старта до вершины + оценочное расстояние до цели.

Временна́я сложность алгоритма A\* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:



где h\* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

Но ещё бо́льшую проблему, чем временна́я сложность, представляют собой потребляемые алгоритмом ресурсы памяти. В худшем случае ему приходится помнить экспоненциальное количество узлов. Для борьбы с этим было предложено несколько вариаций алгоритма, таких как алгоритм A\* с итеративным углублением (iterative deeping A\*, IDA\*), A\* с ограничением памяти (memory-bounded A\*, MA\*), упрощённый MA\* (simplified MA\*, SMA\*) и рекурсивный поиск по первому наилучшему совпадению (recursive best-first search, RBFS).

**Индивидуализация.**

Вариант 3с. Списки смежности. Написать функцию, проверяющую эвристику на допустимость и монотонность.

**Описание функций и структур данных.**

Класс Graph используется для инкапсуляции методов и данных, необходимых для работы с графом.

Graph();

Конструктор класса, инициализирует пустые списки рёбер.

void add\_edge(size\_t to, size\_t from, double weight);

Метод добавляет в графе ребро с весом . Метод ничего не возвращает.

double cost\_edge(size\_t a, size\_t b), где a и b - названия вершин  
возвращает вес ребра ab

std::Graph<std::vector<std::pair<double, size\_t>>, N> graph;

Этот член класса представляет из себя списки смежности.

**Тестирование.**

Для проведения тестирования был написан чекер, который используя make собирает проект и запускает полученную программу со всеми тестовыми данными из папки «tests/» в папке проекта. Результаты тестирования представлены в табл. 1. В качестве результата первого теста представлен ответ вместе с сопровождающими сообщениями, для остальных – только ответ.

Таблица 1 – Результаты тестирования алгоритма A\*.

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | a e  a b 3.0  b c 1.0  c d 1.0  a d 5.0  d e 1.0 | Path: ade  Admissible and not monotone |
| 2 | a f  a b 3.0  b c 2.0  c e 1.0  c d 1.0  a d 3.0  d e 1.0  e f 4.0 | Path: adef  Admissible and not monotone |
| 3 | a f  a b 5  a c 5  b d 2  c e 7  d f 10  e g 4 | Path: abdf  Admissible and not monotone |
| 4 | a a  a a 5.0  a a 3.0  a a 1.0 | Path: a  Admissible and monotone |
| 5 | a e  a b 1  b c 2  d c 3  c e 4 | Path: ace  Admissible and not monotone |

Таблица 2 – Результаты тестирования жадного алгоритма.

|  |  |  |
| --- | --- | --- |
| № | Входные данные | Выходные данные |
| 1 | a f  a b 5  a c 5  b d 1  c e 7  d f 10  e g 4 | abdf |
| 2 | a e  a b 3.0  b c 1.0  c d 1.0  a d 5.0  d e 1.0 | abcde |
| 3 | a a  a a 5.0  a a 3.0  a a 1.0 | a |

**Выводы.**

В ходе выполнения лабораторной работы реализованы алгоритмы поиска пути в графе такие, как жадный, A\* и A\* с модификацией для поиска пути сразу в две финишные вершины. Все алгоритмы имеют одинаковую асимптотику работы и одинаковые затраты по памяти . Алгоритм A\* позволяет найти кратчайший путь быстрее и лучше при условии использования хорошей эвристической функции. Эвристическая функция обладает двумя свойствами: допустимость (эвристическая оценка должна быть меньше, либо равна весу кратчайшего пути от до цели , т. е. и монотонность (разность эвристических оценок для вершины и её потомка не превышает фактического веса ребра между этими вершинами, т. е. ).

**ПРИЛОЖЕНИЕ А**

**КОД ЖАДНОГО АЛГОРИТМА ПОИСКА**

#include <iostream>

#include <vector>

#include <algorithm>

class Graph {

public:

Graph(size\_t n);

void add\_edge(size\_t from, size\_t to, double weight);

std::vector <size\_t> pathfinder(size\_t from, size\_t to);

private:

std::vector < std::vector < std::pair < double, size\_t > >> graph;

};

Graph::Graph(size\_t n) : graph(n)

{

}

void Graph::add\_edge(size\_t from, size\_t to, double weight) {

graph[from].push\_back(std::make\_pair(weight, to));

// graph[to].push\_back(std::make\_pair(weight, from));

}

std::vector<size\_t> Graph::pathfinder(size\_t from, size\_t to) {

for (size\_t i = 0; i < graph.size(); i++) {

std::sort(graph[i].begin(), graph[i].end());

}

std::vector<bool> already\_visited(graph.size());

std::vector<size\_t> parent(graph.size());

size\_t from\_backup = from;

while (from != to) {

already\_visited[from] = true;

//std::cout<<(char)(from+'a')<<" already visited"<<std::endl;

bool go\_back\_required = true;

for (size\_t i = 0; i < graph[from].size() && go\_back\_required; i++) {

size\_t vertex = graph[from][i].second;

if (!already\_visited[vertex]) {

parent[vertex] = from;

from = vertex;

go\_back\_required = false;

}

}

if (go\_back\_required) {

from = parent[from];

}

}

std::vector<size\_t> path;

from = from\_backup;

while (from != to) {

path.push\_back(to);

to = parent[to];

}

path.push\_back(from);

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

return path;

}

int main()

{

char from, to;

std::cin >> from >> to;

Graph graph(26);

while (!std::cin.eof()) {

char from, to;

double weight;

std::cin >> from >> to >> weight;

graph.add\_edge(from - 'a', to - 'a', weight);

}

// std::cout << "[" << from1 << "] [" << to1 << "]" << std::endl;

std::vector<size\_t> path = graph.pathfinder(from - 'a', to - 'a');

for (size\_t i = 0; i < path.size(); i++) {

std::cout << char(path[i] + 'a');

}

return 0;

}**ПРИЛОЖЕНИЕ Б**

**КОД АЛГОРИТМА ПОИСКА A\* (STEPIK)**

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <functional> //std::greater-компаратор для приоритетной очереди

#include <cmath> //std::abs

class Graph {

public:

Graph(size\_t n);

void add\_edge(size\_t from, size\_t to, double weight);

std::vector <size\_t> pathfinder(size\_t from, size\_t to);

double cost\_edge(size\_t, size\_t);

private:

std::vector < std::vector < std::pair < double, size\_t > >> graph;

};

Graph::Graph(size\_t n) : graph(n)

{

}

/\*double Graph::cost\_edge(size\_t a, size\_t b) {

for (size\_t i = 0; i < graph[a].size(); i++) {

auto sec = graph[a][i];

if (sec.second == b)

return sec.first;

}

} \*/

double Graph::cost\_edge(size\_t a, size\_t b) {

for (auto& i : graph[a])

if (i.second == b) return i.first;

}

void Graph::add\_edge(size\_t from, size\_t to, double weight) {

graph[from].push\_back(std::make\_pair(weight, to));

}

double heuristic(size\_t a, size\_t b) { //расстояние между символами в алфавите

return std::abs(double(a) - double(b));

}

std::vector<size\_t> Graph::pathfinder(size\_t from, size\_t to) {

typedef std::pair<double, size\_t> qel; // queue element

std::priority\_queue<qel, std::vector<qel>, std::greater<qel>> frontier; //элемент очереди, контейнер, компаратор чтоб вершина с наименьшим приоритетом была наверху в очереди

frontier.push(std::make\_pair(0, from));

std::vector<bool> already\_visited(graph.size());

std::vector<size\_t> parent(graph.size());

std::vector<double> cost\_to\_start(graph.size()); //А\* - оценка приоритета вершины: distance on a square grid return abs(a.x - b.x) + abs(a.y - b.y) (ф-я heuristic)

parent[from] = from;

already\_visited[from] = true;

cost\_to\_start[from] = 0;

//---начало алгоритма-------

while (!frontier.empty()) {

auto current = frontier.top(); //вытащенная вершина заменяется ее потомками, а очередь сортируется

frontier.pop();

if (current.second == to) //"ранний выход"

break;

for (auto& next : graph[current.second]) {

double new\_cost = cost\_to\_start[current.second] + next.first; //расстояние до старта для исследуемого потомка, чем оно меньше тем выгоднее нам достать вершину из очереди

if (!already\_visited[next.second] || new\_cost < cost\_to\_start[next.second]) {

double priority = new\_cost + heuristic(next.second, to); //приоритет для очереди

frontier.push(std::make\_pair(priority, next.second));

already\_visited[next.second] = true;

parent[next.second] = current.second;

cost\_to\_start[next.second] = new\_cost;

}

}

}

//----конец--------

std::vector<size\_t> path;

while (from != to) {

path.push\_back(to);

to = parent[to];

}

path.push\_back(from);

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

return path;

}

int main()

{

char from, to;

std::cin >> from >> to;

Graph graph(26);

while (!std::cin.eof()) {

char from, to;

double weight;

std::cin >> from >> to >> weight;

graph.add\_edge(from - 'a', to - 'a', weight);

}

// std::cout << "[" << from1 << "] [" << to1 << "]" << std::endl;

std::vector<size\_t> path = graph.pathfinder(from - 'a', to - 'a');

for (size\_t i = 0; i < path.size(); i++) {

std::cout << char(path[i] + 'a');

}

return 0;

}

**ПРИЛОЖЕНИЕ А**

**КОД АЛГОРИТМА ПОИСКА A\* С ПРОВЕРКОЙ ДОПУСТИМОСТИ И МОНОТОННОСТИ (ИНДИВИДУАЛИЗАЦИЯ)**

#include <assert.h>

#include <algorithm> // std::copy

#include <cstddef> // size\_t

#include "pch.h"

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <functional> //std::greater-компаратор для приоритетной очереди

#include <cmath> //std::abs

class Graph {

public:

Graph(size\_t n);

void add\_edge(size\_t from, size\_t to, double weight);

std::vector <size\_t> pathfinder(size\_t from, size\_t to);

double cost\_edge(size\_t, size\_t);

size\_t path\_size(std::vector<size\_t>& path, size\_t i);

void admissibility\_and\_monotonicity(std::vector<size\_t>& path);

private:

std::vector < std::vector < std::pair < double, size\_t > >> graph;

};

Graph::Graph(size\_t n) : graph(n)

{

}

/\*double Graph::cost\_edge(size\_t a, size\_t b) {

for (size\_t i = 0; i < graph[a].size(); i++) {

auto sec = graph[a][i];

if (sec.second == b)

return sec.first;

}

} \*/

double Graph::cost\_edge(size\_t a, size\_t b) {

for (auto& i : graph[a])

if (i.second == b) return i.first;

}

void Graph::add\_edge(size\_t from, size\_t to, double weight) {

graph[from].push\_back(std::make\_pair(weight, to));

}

double heuristic(size\_t a, size\_t b) { //расстояние между символами в алфавите

return std::abs(double(a) - double(b));

}

std::vector<size\_t> Graph::pathfinder(size\_t from, size\_t to) {

typedef std::pair<double, size\_t> qel; // queue element

std::priority\_queue<qel, std::vector<qel>, std::greater<qel>> frontier; //элемент очереди, контейнер, компаратор чтоб вершина с наименьшим приоритетом была наверху в очереди

frontier.push(std::make\_pair(0, from));

std::vector<bool> already\_visited(graph.size());

std::vector<size\_t> parent(graph.size());

std::vector<double> cost\_to\_start(graph.size()); //А\* - оценка приоритета вершины: distance on a square grid return abs(a.x - b.x) + abs(a.y - b.y) (ф-я heuristic)

parent[from] = from;

already\_visited[from] = true;

cost\_to\_start[from] = 0;

//---начало алгоритма-------

while (!frontier.empty()) {

auto current = frontier.top(); //вытащенная вершина заменяется ее потомками, а очередь сортируется

frontier.pop();

if (current.second == to) //"ранний выход"

break;

for (auto& next : graph[current.second]) {

double new\_cost = cost\_to\_start[current.second] + next.first; //расстояние до старта для исследуемого потомка, чем оно меньше тем выгоднее нам достать вершину из очереди

if (!already\_visited[next.second] || new\_cost < cost\_to\_start[next.second]) {

double priority = new\_cost + heuristic(next.second, to); //приоритет для очереди

frontier.push(std::make\_pair(priority, next.second));

already\_visited[next.second] = true;

parent[next.second] = current.second;

cost\_to\_start[next.second] = new\_cost;

}

}

}

//----конец--------

std::vector<size\_t> path;

while (from != to) {

path.push\_back(to);

to = parent[to];

}

path.push\_back(from);

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

return path;

}

void Graph::admissibility\_and\_monotonicity(std::vector<size\_t>& path) {

bool monotonicity = true;

for (size\_t i = 0; i < path.size(); i++) {

for (size\_t j = i; j < path.size(); j++) {

if (j == i)continue;

//size\_t size = Graph::path\_size(path, i);

if (heuristic(path[i], path.back()) - heuristic(path[j], path.back()) < cost\_edge(path[i], path[j])) {

monotonicity = false;

break;

}

}

}

bool admissibility = true;

size\_t sum\_path = 0;

for (size\_t i = 0; i < path.size() - 1; i++) {

sum\_path += cost\_edge(path[i], path[i + 1]);

}

size\_t j = 0;

if (!monotonicity) {

for (size\_t i = 0; i < path.size(); i++) {

if (heuristic(path[i], path.back()) > sum\_path) {

admissibility = false;

break;

}

if (i) {

sum\_path -= cost\_edge(path[j], path[i]);

j++;

}

}

}

if (admissibility)

std::cout << "\nAdmissible and ";

else

std::cout << "\nNot admissible and ";

if (monotonicity)

std::cout << "monotonous\n";

else

std::cout << "not monotonous\n";

}

int main()

{

char from, to;

std::cin >> from >> to;

Graph graph(26);

while (!std::cin.eof()) {

char from, to;

double weight;

std::cin >> from >> to >> weight;

graph.add\_edge(from - 'a', to - 'a', weight);

}

//std::cout << "[" << from1 << "] [" << to1 << "]" << std::endl;

std::vector<size\_t> path = graph.pathfinder(from - 'a', to - 'a');

std::cout << "Path: ";

for (size\_t i = 0; i < path.size(); i++) {

std::cout << char(path[i] + 'a');

}

graph.admissibility\_and\_monotonicity(path);

return 0;

}

**ПРИЛОЖЕНИЕ Г**

**КОД АЛГОРИТМА ПОИСКА A\* С ПРОВЕРКОЙ ДОПУСТИМОСТИ И МОНОТОННОСТИ (ИНДИВИДУАЛИЗАЦИЯ) С ПРОМЕЖУТОЧНЫМ ВЫВОДОМ**

#include <assert.h>

#include <algorithm> // std::copy

#include <cstddef> // size\_t

#include "pch.h"

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <functional> //std::greater-компаратор для приоритетной очереди

#include <cmath> //std::abs

class Graph {

public:

Graph(size\_t n);

void add\_edge(size\_t from, size\_t to, double weight);

std::vector <size\_t> pathfinder(size\_t from, size\_t to);

double cost\_edge(size\_t, size\_t);

size\_t path\_size(std::vector<size\_t>& path, size\_t i);

void admissibility\_and\_monotonicity(std::vector<size\_t>& path);

private:

std::vector < std::vector < std::pair < double, size\_t > >> graph;

};

Graph::Graph(size\_t n) : graph(n)

{

}

double Graph::cost\_edge(size\_t a, size\_t b) {

for (auto& i : graph[a])

if (i.second == b) return i.first;

}

void Graph::add\_edge(size\_t from, size\_t to, double weight) {

graph[from].push\_back(std::make\_pair(weight, to));

}

double heuristic(size\_t a, size\_t b) { //расстояние между символами в алфавите

return std::abs(double(a) - double(b));

}

std::vector<size\_t> Graph::pathfinder(size\_t from, size\_t to) {

typedef std::pair<double, size\_t> qel; // queue element

std::priority\_queue<qel, std::vector<qel>, std::greater<qel>> frontier; //элемент очереди, контейнер, компаратор чтоб вершина с наименьшим приоритетом была наверху в очереди

frontier.push(std::make\_pair(0, from));

std::vector<bool> already\_visited(graph.size());

std::vector<size\_t> parent(graph.size());

std::vector<double> cost\_to\_start(graph.size()); //А\* - оценка приоритета вершины: distance on a square grid return abs(a.x - b.x) + abs(a.y - b.y) (ф-я heuristic)

parent[from] = from;

already\_visited[from] = true;

cost\_to\_start[from] = 0;

//---начало алгоритма-------

while (!frontier.empty()) {

auto current = frontier.top(); //вытащенная вершина заменяется ее потомками, а очередь сортируется

std::cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\nOn top of queue: " << char(current.second + 'a') <<"\n";

frontier.pop();

if (current.second == to) //"ранний выход"

break;

std::cout << "\nIts neighbors:\n";

for (auto& next : graph[current.second]) {

std::cout << "\nVertex: " << char( next.second+'a')<<"\n";

double new\_cost = cost\_to\_start[current.second] + next.first; //расстояние до старта для исследуемого потомка, чем оно меньше тем выгоднее нам достать вершину из очереди

if (!already\_visited[next.second] || new\_cost < cost\_to\_start[next.second]) {

std::cout << "\nHasn't been visited or has lower path-to-start length\n" ;

double priority = new\_cost + heuristic(next.second, to); //приоритет для очереди

frontier.push(std::make\_pair(priority, next.second));

already\_visited[next.second] = true;

parent[next.second] = current.second;

cost\_to\_start[next.second] = new\_cost;

}

}

}

//----конец--------

std::vector<size\_t> path;

while (from != to) {

path.push\_back(to);

to = parent[to];

}

path.push\_back(from);

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

return path;

}

void Graph::admissibility\_and\_monotonicity(std::vector<size\_t>& path) {

bool monotonicity = true;

for (size\_t i = 0; i < path.size(); i++) {

for (size\_t j = i; j < path.size(); j++) {

if (j == i)continue;

//size\_t size = Graph::path\_size(path, i);

if (heuristic(path[i], path.back()) - heuristic(path[j], path.back()) < cost\_edge(path[i], path[j])) {

monotonicity = false;

break;

}

}

}

bool admissibility = true;

size\_t sum\_path = 0;

for (size\_t i = 0; i < path.size() - 1; i++) {

sum\_path += cost\_edge(path[i], path[i + 1]);

}

size\_t j = 0;

if (!monotonicity) {

for (size\_t i = 0; i < path.size(); i++) {

if (heuristic(path[i], path.back()) > sum\_path) {

admissibility = false;

break;

}

if (i) {

sum\_path -= cost\_edge(path[j], path[i]);

j++;

}

}

}

if (admissibility)

std::cout << "\nAdmissible and ";

else

std::cout << "\nNot admissible and ";

if (monotonicity)

std::cout << "monotone\n"; // разность эвристических оценок для вершины v\_1 и её потомка v\_2 не превышает фактического веса ребра между этими вершинами

else

std::cout << "not monotone\n";

}

int main()

{

char from, to;

std::cin >> from >> to;

Graph graph(26);

while (!std::cin.eof()) {

char from, to;

double weight;

std::cin >> from >> to >> weight;

graph.add\_edge(from - 'a', to - 'a', weight);

}

std::vector<size\_t> path = graph.pathfinder(from - 'a', to - 'a');

std::cout << "Path: ";

for (size\_t i = 0; i < path.size(); i++) {

std::cout << char(path[i] + 'a');

}

graph.admissibility\_and\_monotonicity(path);

return 0;

}