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. 9382 _____ Павлов Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы

Изучение, сравнение и реализация алгоритмов поиска пути во взвешенных графах.

Задание

Вар. 6. Реализация очереди с приоритетами, используемой в A*, через двоичную кучу.

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
```

Разработайте программу, которая решает задачу построения кратчайшего пути в *ориентированном* графе **методом A***. Каждая вершина в графе имеет

буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.
Пример входных данных

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
```

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

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

```
ade
```

Описание алгоритма

1. Жадный алгоритм

   1) Выбирается текущая вершина. Если у неё нет непросмотренных смежных вершин, вершины удаляются из стека, в котором впоследствии будет находиться найденный путь, до тех пор, пока не будет найдена вершина с непросмотренными смежными вершинами. Затем вершина заносится в стек.

   2) Просматривается список смежных вершин, из него выбирается одна, расстояние до которой от текущей вершины минимально. Эта вершина помечается как просмотренная, после чего становится текущей вершиной.

Цикл повторяется до тех пор, пока не найдена вершина, имя которой совпадает с именем целевой вершины.

2. Алгоритм «A*»

1) Из открытого списка выбирается вершина с наибольшим приоритетом (на первом шаге — начальная вершина).

2) У выбранной вершины просматривается список смежных ей вершин, каждая из которых заносится в открытый список, либо, если какая-нибудь из них уже есть в открытом списке, информация о ней обновляется в случае, если найден более короткий путь до неё, чем существовал до этого. При внесении изменений в открытый список для тех вершин, что находятся в нём, задаётся родительская вершина — та, из которой на данный момент путь в данную клетку наиболее короткий.

Цикл также повторяется до нахождения конечной вершины.

3) По родительским вершинам определяется путь из начальной вершины в конечную.

Сложность алгоритма

1. Жадный алгоритм

1) Сложность алгоритма по памяти — O(n), где n – количество входных данных (вершин), так как стек, запоминающий информацию о пройденных вершинах, линеен (список).

2) Сложность алгоритма по времени — O(n!/(n-2)!), поскольку в худшем случае любая вершина соединена со всеми остальными. Таким образом, сложность — O (n*(n-1)) = $O(n^2)$.

2. Алгоритм A*

1) Сложность этого алгоритма по памяти равна O(n) в лучшем случае, когда эвристическая функция допустима (не переоценивает точную эвристику), потому что постепенно оценка будет приводить к выбору вершин, находящихся всё ближе к конечной вершине. В худшем же случае алгоритм помнит экспоненциальное количество узлов (инцидентные вершины для каждой из вершин), и сложность становится

$O(a^n)$, где a – количество инцидентных рёбер каждой вершины в среднем (пути ветвятся).

2) Время выполнения алгоритма может быть полиномиальным, если ошибка эвристической функции растёт не быстрее, чем логарифм оптимальной эвристики. Сложность алгоритма по времени в таком случае составляет O(log h*(n)), где h* - оптимальная эвристика. В худшем случае время выполнения алгоритма растёт в зависимости от количества инцидентных вершине рёбер, что при наличии связей со всеми остальными вершинами даёт сложность $O(n^2)$.

Описание функций и СД

- **struct Vertex** {

```
std::string name;
Vertex* parent;
std::vector <std::pair<Vertex*, double>> adjacent;
double distance, func;
bool closed;

Vertex(std::string, double = 0.0);
Vertex(std::string, Vertex*, double, double = 0.0);

bool noAdjacent();
bool cmp(const Vertex&);
};
```

- структура, описывающая вершину; name – имя вершины, **parent** – указатель на родительскую клетку, **adjacent** – список смежных вершин, **distance** – расстояние от начальной клетки до текущей, **func** – эвристическая функция, **closed** – индикатор закрытой вершины

  - **bool noAdjacent()** - проверка на то, есть ли непросмотренные инцидентные вершины
  - **bool cmp(const Vertex& other)** - сравнение приоритетов вершин для A*; other – ссылка на вторую вершину

- **class PriorQueue** {

```
private:
std::vector<Vertex*> heap;
```

```
void heapify(int i);
void stabilize();

public:
PriorQueue(Vertex* initial);
void push(Vertex* elem);
Vertex* top();
void pop();
};
```
- класс очереди с приоритетом; **heap** – куча (на массиве)

  - **void heapify(int i)** – перетасовка корня и двух его потомков; **i** – индекс корневого элемента
  - **void stabilize()** – упорядочивание элементов в куче
  - **void push(Vertex* elem)** – добавление элемента в очередь; **elem** – указатель на добавляемую вершину;
  - **Vertex* top()** – взятие приоритетного элемента из очереди
  - **void pop()** – удаление приоритетного элемента из очереди

- **class AStarPredicate** {

```
public:
bool operator() (Vertex* v1, Vertex* v2);
};
```
- класс компаратора очереди с приоритетом

  - **bool operator() (Vertex* v1, Vertex* v2)** – оператор сравнения двух вершин; **v1** – указатель на первую, **v2** – указатель на вторую

- **class Navigator** {

```
private:
std::string sourceName;
std::string destName;
std::vector <Vertex*>* vertices;

Navigator() : vertices(nullptr);
~Navigator();
void retrievePath(Vertex* dest, std::string& path);
void restoreGraph();
int searchIndex(std::vector<Vertex*>* arr, std::string name);
void getInput();
std::string greedy();
std::string aStar();
static void route();
};
```
- класс, используемый для хранения графа и работы с ним

- **void retrievePath(Vertex* dest, std::string& path)** – восстановление пути, начиная с конечной вершины; **dest** –конечная вершина, **path** – строка с записью пути
- **void restoreGraph()** – восстановление изначального состояния графа
- **int searchIndex(std::vector<Vertex*>* arr, std::string name)** – поиск индекса элемента в векторе; **arr** – рассматриваемый вектор; **name** – имя искомой вершины
- **void getInput()** – ввод исходных данных
- **std::string greedy()** – жадный алгоритм
- **std::string aStar()** – алгоритм A*
- **static void route()** – статический метод для работы пользователя с классом графа

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1

Входные данные	**Выходные данные**
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	Жадный: abdefg A*: ag
a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	Жадный: abcd A*: aed
a h a b 1 a c 2 b d 5 b g 10 b e 4 c e 2 c f 1 d g 2 e d 1 e g 7 f e 3 f h 8 g h 1	Жадный: abedgh A*: acedgh
a b a b 1	Жадный: ab A*: ab
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Жадный: abcde A*: ade

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

Имя файла: alg_lab2.cpp

```
// Метка CUSTOM_QUEUE определяет, используется ли
// std::priority_queue или пользовательский тип PriorQueue
// OUTPUT включает вывод промежуточной информации
// AUTO_INPUT убирает лишний вывод при задействовании скрипта

#define AUTO_INPUT
#define CUSTOM_QUEUE
//#define OUTPUT

#include <iostream>
#include "vertex.h"

#ifdef CUSTOM_QUEUE
#include "prior_queue.h"
#else
#include <queue>
#endif

// Компаратор приоритетов вершин для std::prioirity_queue
#ifndef CUSTOM_QUEUE
class AStarPredicate {
public:
	bool operator() (Vertex* v1, Vertex* v2)
	{
		return v2->cmp(*v1);
	}
};
#endif

// Класс, содержащий список веришн графа, имена вершин старта и финиша,
// а также необходимые для работы с графом операции
class Navigator {
private:
	std::string sourceName;
	std::string destName;
	std::vector <Vertex*>* vertices;

	Navigator() : vertices(nullptr) {
		getInput();
	}

	~Navigator() {
		for (auto v : *vertices) {
			delete v;
		}
		if (vertices != nullptr) {
			delete vertices;
		}
	}

	// Вычисление пути от начальной вершины до конечной
	void retrievePath(Vertex* dest, std::string& path) {
#ifdef OUTPUT
		std::cout << "\nВосстановление пути от начальной вершины до конечной\nКонечная
вершина: " << dest->name << "\n";
#endif
		path = dest->name + path;
		// Просмотр вершин, начиная с конечной и заканчивая начальной
```

```
		while (dest->parent != nullptr) {
			dest = dest->parent;
#ifdef OUTPUT
			std::cout << "Предыдущая вершина: " << dest->name << "\n";
#endif
			path = dest->name + path;
		}
	}

	void restoreGraph() {
		for (int j = 0; j < vertices->size(); j++) {
			auto i = vertices->at(j);
			i->distance = 0.0;
			i->func = 0.0;
			i->closed = false;
			i->parent = nullptr;
		}
	}

	// Поиск индекса вершины с указанным именем
	int searchIndex(std::vector<Vertex*>* arr, std::string name) {
		if (arr->empty()) {
			return -1;
		}
		std::vector<Vertex*>::iterator v;
		int i = arr->size() - 1;
		for (v = arr->end() - 1; v >= arr->begin(); v--) {
			if ((*v)->name == name) {
				return i;
			}
			i--;
			if (v == arr->begin()) {
				break;
			}
		}
		return i;
	}

	// Получение исходных данных
	void getInput() {
		// Список вершин графа - новый объект
		vertices = new std::vector<Vertex*>(0);
		// Переменные для записи информации о рёбрах
		std::string main, adj;
		double weight;
		// Добавление начальной и конечной вершин
#ifdef OUTPUT
		std::cout << "Для остановки ввода вершин нажать Ctrl+Z, затем Enter\n";
		std::cout << "\nПожалуйста, введите исходную и конечную вершины ( <вершина1>
<вершина2>): ";
#endif
		std::cin >> sourceName >> destName;
#ifdef OUTPUT
		system("cls");
#endif
		vertices->push_back(new Vertex(sourceName));
#ifdef OUTPUT
		std::cout << "Вершина " << vertices->back()->name << " добавлена в граф\n";
#endif
		vertices->push_back(new Vertex(destName));
#ifdef OUTPUT
		std::cout << "Вершина " << vertices->back()->name << " добавлена в граф\n";
#endif

		while (true) {
```

```
#ifdef OUTPUT
                        std::cout << "\nПожалуйста, введите ребро ( <вершина1> <вершина2>
<расстояние> ): ";
#endif
                        std::cin >> main >> adj >> weight;
#ifdef OUTPUT
                        system("cls");
#endif
                        if (!std::cin) {
                                break;
                        }

                        int m = searchIndex(vertices, main);
                        int i = searchIndex(vertices, adj);
                        // Если не существует второй указанной вершины, создать её и поместить
в список
                        if (i < 0) {
                                i = vertices->size();
                                vertices->push_back(new Vertex(adj));
#ifdef OUTPUT
                                std::cout << "Вершина " << vertices->back()->name << " добавлена
в граф\n";
#endif

                        }
                        // Если не существует первой указанной вершины, создать её и поместить
в список,
                        // указав смежную (вторую) вершину и вес инцидентного им ребра
                        if (m < 0) {
                                vertices->push_back(new Vertex(main, vertices->at(i), weight));
#ifdef OUTPUT
                                std::cout << "Вершина " << vertices->back()->name << " добавлена
в граф\n";
#endif

                        }
                        // Если она уже есть в списке, только указать смежную вершину
                        else {
                                vertices->at(m)->adjacent.push_back({ vertices->at(i),
weight });
                        }
#ifdef OUTPUT
                        std::cout << "Ребро " << main << "--" << adj << " с весом " << weight
<< " добавлено в граф\n";
#endif
                }
        }

        // Жадный алгоритм
        std::string greedy() {
                std::vector <Vertex*> stack;

                Vertex* cur = vertices->at(0);
                std::string res = "";

                // Главный цикл поиска пути
                do {
                        int index = 0;
                        // Возврат до первой вершины, у которой остались непосещённые смежные
                        while (cur->noAdjacent()) {
                                cur = stack.back();
                                stack.pop_back();
                        }
                        stack.push_back(cur);
#ifdef OUTPUT
                        std::cout << "\nПереход к вершине " << cur->name << "\n";
#endif
```

```
                        // Поиск кратчайшего из путей до смежных вершин
                        double shortest = 0.0;
                        for (int i = 0; i < cur->adjacent.size(); i++) {
                                if (!cur->adjacent.at(i).first->closed) {
                                        shortest = cur->adjacent.at(i).second;
                                        index = i;
                                        break;
                                }
                        }
                        for (int i = index + 1; i < cur->adjacent.size(); i++) {
                                if ( (!cur->adjacent.at(i).first->closed) && (cur-
>adjacent.at(i).second < shortest ||
                                        (cur->adjacent.at(i).second == shortest && (cur-
>adjacent.at(i).first->name < cur->adjacent.at(index).first->name))) {
                                        shortest = cur->adjacent.at(i).second;
                                        index = i;
                                }
                        }
#ifdef OUTPUT
                        std::cout << "\n\tБлижайшая вершина: " << cur-
>adjacent.at(index).first->name << ", длина пути: " << shortest << "\n";
#endif
                        // Переход к следующей вершине
                        cur->adjacent.at(index).first->closed = true;
                        cur = cur->adjacent.at(index).first;
                } while (cur->name != destName);
#ifdef OUTPUT
                std::cout << "\nПуть до конечной вершины обнаружен, завершение поиска\n";
#endif
                stack.push_back(cur);

                // Запись пути в результирующую строку
                for (auto& i : stack) {
                        res = res + i->name;
                }

                return res;
        }

        // Алгоритм A*
        std::string aStar() {
        // Инициализация очереди и строки с указанием пути
#ifdef CUSTOM_QUEUE
                PriorQueue openList = PriorQueue(vertices->at(0));
#else
                std::priority_queue <Vertex*, std::vector<Vertex*>, AStarPredicate> openList;
                openList.push(vertices->at(0));
#endif

                std::string res = "";
                Vertex* cur;
                // Главный цикл поиска пути
                do {
                        // Извлечение самого приоритетного элемента, установление метки
закрытой вершины (уже просмотрено)
                        cur = openList.top();
                        openList.pop();
#ifdef OUTPUT
                        std::cout << "\nИз открытого списка извлечена вершина " << cur->name <<
"\n";
#endif
                        // Просмотр списка инцидентных вершин
                        for (auto& v : cur->adjacent) {
#ifdef OUTPUT
```

```
                        std::cout << "\n\tРассматривается смежная вершина " << v.first->name << "\n";
#endif
                        const double f = cur->distance + v.second + destName[0] - v.first->name[0];
#ifdef OUTPUT
                        std::cout << "\tПриоритет данной смежной вершины равен " << f << "\n";
#endif
                        // Если оценка выгодности пути меньше имеющейся для данной вершины, обновить путь
                        if (f < v.first->func) {
#ifdef OUTPUT
                                std::cout << "\tТекущий приоритет выше существующего (" << v.first->func << " > " << f << "), обновление пути\n";
#endif
                                v.first->distance = f - (destName[0] - v.first->name[0]);
                                v.first->func = f;
                                v.first->parent = cur;
                        }
                        // Если вершины ещё нет в открытом списке, добавить её в него
                        else if (v.first->func == 0) {
#ifdef OUTPUT
                                std::cout << "\tВершины ещё нет в открытом списке, добавлена в него  и записана в путь\n";
#endif
                                v.first->distance = f - (destName[0] - v.first->name[0]);
                                v.first->func = f;
                                v.first->parent = cur;
                                openList.push(v.first);
                        }
                        else {
#ifdef OUTPUT
                                std::cout << "\tТекущий путь менее выгоден, чем уже существующий, и не рассматривается\n";
#endif
                        }
                }
        } while (cur->name != destName);
#ifdef OUTPUT
        std::cout << "\nПуть до конечной вершины обнаружен, завершение поиска\n";
#endif
        // Запись пути в результирующую строку
        retrievePath(vertices->at(1), res);
        return res;
    }
public:
    static void route() {
        Navigator* n = new Navigator();
        std::cout << "Жадный Путь от начальной вершины до конечной: " << n->greedy() << "\n";
        n->restoreGraph();
        std::cout << "A* Путь от начальной вершины до конечной: " << n->aStar() << "\n";
        delete n;
    }
};

int main() {
#ifdef AUTO_INPUT
    Navigator::route();
#else
    int command = 0;
    setlocale(LC_ALL, "Ru");
    std::cout << "Проложить маршрут? 0 - нет, 1 - да\n";
```

```
	std::cin >> command;
	switch (command) {
	case 0:
		std::cout << "Завершение работы\n";
		break;
	case 1:
		Navigator::route();
		break;
	default:
		std::cout << "Неизвестная команда, выход из программы\n";
	}
#endif
	return 0;
}
```

Имя файла: vertex.cpp

```
#include "vertex.h"

Vertex::Vertex(std::string s, double dist) : name(s), distance(dist), func(0.0),
parent(nullptr), closed(false) {}

Vertex::Vertex(std::string s, Vertex* inc, double w, double dist) : name(s),
distance(dist), func(0.0), parent(nullptr), closed(false) {
	adjacent.push_back({ inc, w });
}

bool Vertex::noAdjacent(){
	for (auto& v : adjacent) {
		if (!v.first->closed) return false;
	}
	return true;
}

bool Vertex::cmp(const Vertex& other) {
	if (this->func == other.func) {
		return this->name > other.name;
	}
	return (this->func) < (other.func);
}
```

Имя файла: prior_queue.cpp

```
#include "prior_queue.h"

void PriorQueue::heapify(int i) {
	int least = i;
	int l = 2 * i + 1;
	int r = 2 * i + 2;
	if (l < heap.size() && heap.at(l)->cmp(*heap.at(least))) {
		least = l;
	}
	if (r < heap.size() && heap.at(r)->cmp(*heap.at(least))) {
		least = r;
	}
	if (least != i) {
		std::swap(heap.at(i), heap.at(least));
		heapify(least);
	}
}
```

```
void PriorQueue::stabilize() {
	for (int i = heap.size() / 2 + 1; i >= 0; i--) {
		heapify(i);
	}
}

PriorQueue::PriorQueue(Vertex* initial) {
	heap.push_back(initial);
}

void PriorQueue::push(Vertex* elem) {
	heap.push_back(elem);
}

Vertex* PriorQueue::top() {
	if (heap.empty()) return nullptr;
	stabilize();
	return heap.at(0);
}

void PriorQueue::pop() {
	if (heap.empty()) return;
	std::swap(heap.at(0), heap.at(heap.size() - 1));
	heap.resize(heap.size() - 1);
}
```

Имя файла: vertex.h

```
#pragma once
#include <vector>
#include <string>

struct Vertex {
	std::string name;
	Vertex* parent;
	std::vector <std::pair<Vertex*, double>> adjacent;
	double distance, func;
	bool closed;

	Vertex(std::string, double = 0.0);
	Vertex(std::string, Vertex*, double, double = 0.0);

	bool noAdjacent();
	bool cmp(const Vertex&);	// Компаратор приоритетов вершин для
PriorQueue
};
```

Имя файла: prior_queue.h

```
#pragma once
#include <vector>
#include "vertex.h"

class PriorQueue {
private:
	std::vector<Vertex*> heap;

	void heapify(int i);// Перетасовка узла и его потомков
	void stabilize(); // Пирамидизация
public:
	PriorQueue(Vertex* initial);
```

```
        void push(Vertex* elem);
        Vertex* top();
        void pop();
};
```