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 9

Студентка гр. 8303 _____ Потураева М.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение алгоритмов поиска пути в графе и вывод его графического представления.

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

Вывод графического представления графа.

Описание жадного алгоритма.

В начале работы жадного алгоритма поиска пути в ориентированном графе, список рёбер сортируется по не убыванию их весов. Алгоритм начинает поиск из заданной вершины. Текущая просматриваемая вершина добавляется в список просмотренных. В отсортированном списке рёбер выбирается первое (сортировка гарантирует, что оно будет минимальное), которое начинается в просматриваемой вершине, если эта вершина не просмотрена, то текущей вершиной становится та, в которой заканчивается это ребро, если она уже просмотрена, то выбирается следующее ребро. Если в какой-то момент из текущей вершины нет путей, то происходит откат на шаг назад, и в предыдущей вершине выбирается другое ребро, если это возможно. Алгоритм заканчивает свою работу, когда текущей вершиной становится искомая, или, когда были просмотрены все ребра, которые начинаются из исходной вершины.

Худший случай для данного алгоритма, когда переход будет выполняться к вершине, выходящие ребра которой хранятся в памяти максимально далеко от взятого ребра. Тогда сложность алгоритма составит o(n*m), где n — количество ребер, а m — количество вершин.

На протяжении всей программы для хранения графа используется вектор ребер, количество проходов по нему в худшем случае будет равно количеству рёбер в графе. Тогда сложность по памяти составляет o(m), где m-количество ребер.

Описание A* алгоритма.

Поиск начинается из исходной вершины. В текущие возможные пути добавляются все ребра из начальной вершины. Происходит выбор минимального пути, где учитывается эвристическая близость вершины к искомой (близость в таблице ASCII), если в выбранном пути последняя вершина уже была просмотрена, то этот путь удаляется из списка, и снова происходит выбор минимального пути. Выбираются из всех рёбер графа те, которые начинаются из последней вершины в этом пути. Эта вершина добавляется к этому пути, и новый путь заносится в список возможных путей, с увеличением веса, равному переходу по этому ребру. Когда были выбраны все ребра, которые начинаются из последней вершины в этом пути, то эта вершина добавляется в список просмотренных, а сам путь удаляется из списка возможных путей. Дальше снова происходит выбор минимального пути. Алгоритм заканчивает свою работу, когда достигается искомая вершина.

Временная 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: $|h(x) - h^*(x)| \leq O(\log h^*(x))$, 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.

В худшем случае, когда эвристика не помогает (эвристическая функция подобрана плохо) придется просмотреть все пути. В таком случае алгоритм просто превратиться в алгоритм Дейкстры и сложность станет O(n^2), где n-количество вершин.

Сложность по памяти O(m), где m — количество ребер, алгоритм хранит вектор ребер.

Описание структур данных.

Класс edge:

char str-начало ребра
char end-конец ребра
double heft-вес

Используется для хранения информации о рёбрах.

vector <edge> graph- используется для хранения графа.

vector <char> result_way-используется для хранения промежуточного решения, а в итоге правильного пути.

vector <char> viewingtop- используется для хранения просмотренных вершин.

Описание функций

1) bool isViewing(char value)– проверка просмотренная данная вершина или еще нет.
Функция возвращает true или false, в зависимости от того, просмотрена ли вершина.
Value-рассматриваемая вершина

2) void initSearch()-начинает поиск пути.

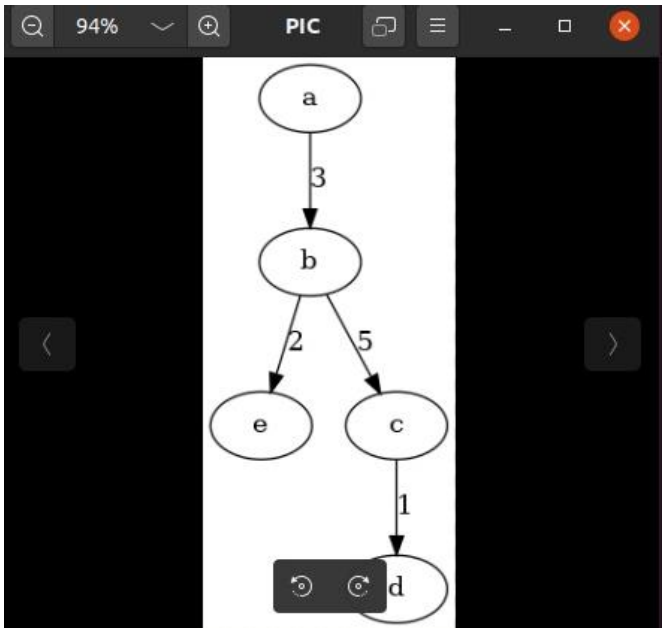
3) bool Search(char value)- рекурсивная функция поиска кратчайшего пути в графе. Функция возвращает true или false, в зависимости от того, найден путь или нет.
Value-рассматриваемая вершина

4) void Print()- печать результата на экран.

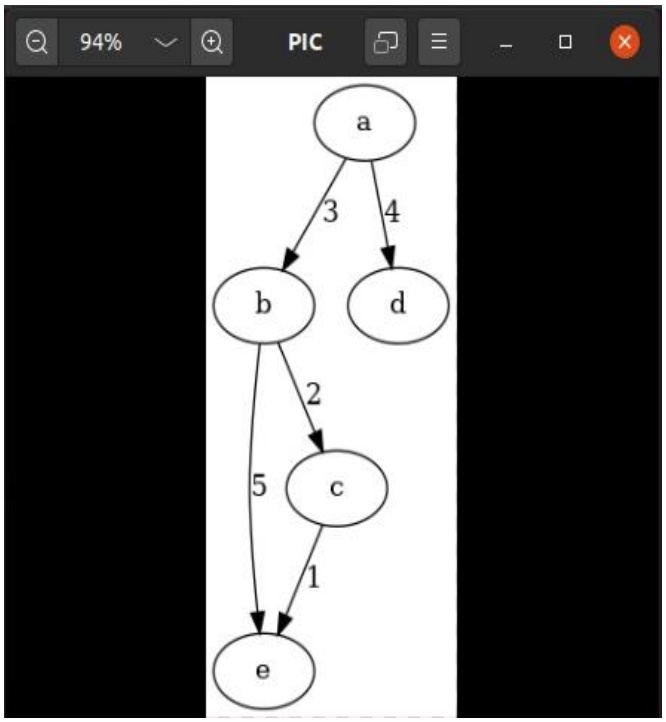
5) bool cmp(edge first, edge second)- компаратор для сравнения вершин по имени. Функция возвращает true , если номер первой вершины в таблице символов меньше.

6) size_t MinSearch()- поиск минимального элемента из не просмотренных ребер.
Функция возвращает индекс минимального элемента в result.

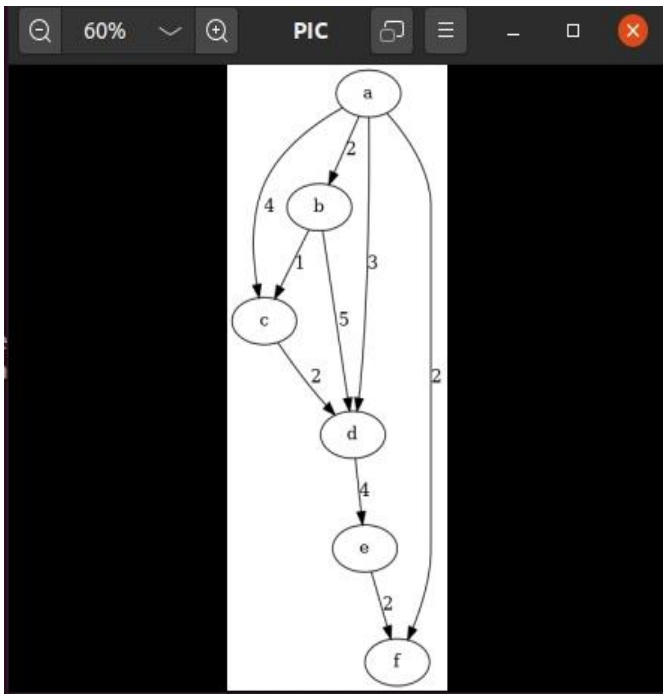
ТЕСТИРОВАНИЕ.




```
a d
a b 3.0
b e 2.0
b c 5.0
c d 1.0
```




```
a e
a b 3.0
a d 4.0
b e 5.0
b c 2.0
c e 1.0
```




```
a f
a b 2.0
a d 3.0
a c 4.0
b c 1.0
c d 2.0
d e 4.0
e f 2.0
b d 5.0
a f 2.0
```

Вывод графического представления графа

Для визуализации графа в процессе выполнения программы создается текстовый файл с расширением «.dot». DOT — это язык для описания графов. С помощью функций для работы со строками он заполняется информацией о графе. Строка формируется по правилам синтаксиса dot. В графическом виде описанный таким образом граф представляется с помощью команды терминала с флагами формата результата и имени создаваемого файла.Dot либо устанавливается на устройство, либо используется онлайн-конвертор. Например,http://www.webgraphviz. Для вывода изображения была написана программа, выполняющая через функцию system() необходимые терминальные команды. Так же создан Makefile для компиляции и запуска проекта с уже установленным dot, в результате которого открывается PNG-файл.

Вывод.

В ходе лабораторной работы была написана программа, реализующая жадный алгоритм и алгоритм A*, а также написана программа для визуализации графа. Таким образом, были изучены алгоритмы поиска кратчайшего пути в графе и способы визуализации графов.

ПРИЛОЖЕНИЯ А. ИСХОДНЫЙ КОД

Visual.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <istream>
#include <string.h>
#include <fstream>

using namespace std;

int main(){
ifstream path;
path.open("path");
char *mydir = new char[200];
path » mydir;
char *command = new char[500];
strcat(command, "dot -Tpng -oPIC ");
strcat(mydir, "/Source/graph.dot");
strcat(command, mydir);
system(command);
system("xdg-open PIC");
delete [] mydir;
delete [] command;
return 0;
}
```

Makefile_a

```
all:show

show:run
	pwd>path
	./start
	./show

run:Source/a_star.cpp
	g++ ./Source/a_star.cpp -o start

clean:
	rm -rf start show path
```

Makefile_g

```
all:show

show:run
	pwd>path
	./start
	./show
run:Source/greedy.cpp
	g++ ./Source/greedy.cpp -o start
clean:
	rm -rf start show path
```

a_star.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <string.h>
#include <cmath>
#include <fstream>
#include <cstdlib>
using namespace std;

struct edge     //ребро графа
{
	char str; //начало
	char end; //конец
	double heft;//вес
};

struct step     //возможные пути
{
	string path; //путь
	double length;//длина
};

class Graph //граф
{
private:
	vector <edge> graph; //изначальный граф
	vector <step> result; //конечный путь
	vector <char> viewingtop;
```

```
	char source; //начало графа
	char finish; //конец графа

public:
	Graph() //инициализация графа
	{
		ofstream out;
		char* cur = new char[200];
		ifstream dir;
		dir.open("path");
		dir >> cur;
		strcat(cur, "/Source/graph.dot");
		out.open(cur);
		out.open(cur, ofstream::app);
		out.clear();
		out << "digraph MyGraph {\n";
		string tmp;
		getline(cin, tmp);
		source = tmp[0];
		finish = tmp[2];
		while (getline(cin, tmp) && tmp != "")
		{
			edge elem;
			elem.str = tmp[0];
			elem.end = tmp[2];
			elem.heft = stod(tmp.substr(4));
			graph.push_back(elem); //добавление ребра
		}
		for (auto i : graph) {
			out << "    " << i.str << " -> " << i.end << " [label=" << i.heft <<
"];\n";
		}
		out << "}\n";
		system("g++ ./Source/visual.cpp -o show");
		delete[] cur;
		string buf = ""; //для ребра
		buf += source;
		for (size_t i(0); i < graph.size(); i++)
		{
			if (graph.at(i).str == source)
			{
				buf += graph.at(i).end;
				result.push_back({ buf, graph.at(i).heft });//добавляем все
ребра начинающиеся с source
```

```
                        buf.resize(1); //возвращаем размер 1
                }
        }
        viewingtop.push_back(source); //добавляем первую вершмну как просмотренную
    }

    size_t MinSearch() //возвращает индекс минимального элемента из непросмотренных
    {
        double min = 10000;//недостижимый минимум
        size_t tmp;//индекс мин элемента
        for (size_t i(0); i < result.size(); i++)//поиск мин элемента среди вершин result
        {
            if (result.at(i).length + abs(finish - result.at(i).path.back()) < min)//сравниваем значение с минимумом
            {
                if (isViewing(result.at(i).path.back()))//проверяем просмотрена вершина или нет
                {
                    result.erase(result.begin() + i); //если уже просматривали конец ребра то удаляем этот элемент из result
                }
                else
                {
                    min = result.at(i).length + abs(finish - result.at(i).path.back());//устанавливаем новый минимум
                    tmp = i;//индекс элемента
                }
            }
        }
        return tmp;
    }

    bool isViewing(char val)//проверка на просомтренную вершину
    {
        for (size_t i = 0; i < viewingtop.size(); i++)
            if (viewingtop.at(i) == val)
                return true;
        return false;
    }

    void Search() //поиск мин пути
```

```
	{
		while (true)
		{
			size_t min = MinSearch();//запускаем поиск мин элемента
			if (result.at(min).path.back() == finish) //если нашли путь до конечной вершины
			{
				cout << "Минимальный путь:";
				cout << result.at(min).path<<endl;//выводим результат
				return;
			}
			for (size_t i(0); i < graph.size(); i++)//проходимся по элементам графа
			{
				if (graph.at(i).str == result.at(min).path.back())//если начало ребра равно концу мин ребра
				{
					string buf = result.at(min).path;//помещаем в buf мин элемент
					buf += graph.at(i).end;//доавляем конец текущего ребра
					result.push_back({ buf, graph.at(i).heft + result.at(min).length });//помещаем это в result
				}
			}
			viewingtop.push_back(result.at(min).path.back());//добавлем конечную вершину в просмотренные
			result.erase(result.begin() + min);//удаляем мин из result
		}
	}
};

int main()
{
	Graph elem;
	elem.Search();
	system("g++ ./Source/visual.cpp -o show");
	return 0;
}
```

greedy.cpp

```
#include <iostream>
#include <vector>
```

```
#include <algorithm>
#include <fstream>
#include <string>
#include <string.h>


using namespace std;

struct edge//ребро
{
	char str;//начало
	char end;//конец
	double heft;//вес
};

bool cmp(edge first, edge second)//сравнение элементов
{
	return first.heft < second.heft;
}

class Graph//граф
{
private:
	vector <edge> graph;//граф
	vector <char> result;//конечный путь
	vector <char> viewingtop;//просмотренные вершины
	char source;//начальная вершина
	char finish;//конечная

public:
	Graph()//инициализация графа
	{
		ofstream out;
		char* cur = new char[200];
		ifstream dir;
		dir.open("path");
		dir >> cur;
		strcat(cur, "/Source/graph.dot");
		out.open(cur);
		out.open(cur, ofstream::app);
		out.clear();
		out << "digraph MyGraph {\n";
		string tmp;
		getline(cin, tmp);
```

```
		source = tmp[0];
		finish = tmp[2];
		while (getline(cin, tmp) && tmp != "")
		{
			edge elem;
			elem.str = tmp[0];
			elem.end = tmp[2];
			elem.heft = stod(tmp.substr(4));
			graph.push_back(elem); //добавление ребра
		}
		sort(graph.begin(), graph.end(), cmp);//сортируем элементы
		for (auto i : graph) {
			out << "    " << i.str << " -> " << i.end << " [label=" << i.heft <<
"];\n";
		}
		out << "}\n";
		system("g++ ./Source/visual.cpp -o show");
		delete[] cur;
	}

	bool isViewing(char value)//проверка на просомтренную вершину
	{
		for (size_t i = 0; i < viewingtop.size(); i++)
			if (viewingtop.at(i) == value)
				return true;
		return false;
	}

	void initSearch()//начало поиска
	{
		if (source != finish)
			Search(source);
	}

	bool Search(char value)//поиск мин пути
	{
		if (value == finish)//если вершина равна конечной
		{
			result.push_back(value);//добавляем в result
			return true;
		}
		viewingtop.push_back(value);//если нет то добавляем в список про-
сомтренных вершин
		for (size_t i(0); i < graph.size(); i++)//проходим по графу
```

```
		{
			if (value == graph.at(i).str)//если веришна равна начальной вершине ребра
			{
				if (isViewing(graph.at(i).end))//если конец ребра просмотрен то переходим к следующей итерации цикла
					continue;
				result.push_back(graph.at(i).str); //иначе помещаем начало ребра в result
				bool flag = Search(graph.at(i).end); //запускаем поиск по конечной вершине
				if (flag)//если нашли нужный путь
					return true;
				result.pop_back();//иначе удаляем рассматриваемую вершину из result
			}
		}
		return false; //если путь не найден
	}

void Print()//печать результата
	{
	cout << "Минимальный путь:";
	for (size_t i(0); i < result.size(); i++)
		cout << result.at(i);
	cout << endl;
	}
};

int main()
{
	Graph element;
	element.initSearch();
	element.Print();
	system("g++ ./Source/visual.cpp -o show");
	return 0;
}
```