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 _____ Хохлов Г.О.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить жадный алгоритм и алгоритм A*. Написать программу, реализующую эти алгоритмы поиска кратчайшего пути в графе.

Вариант 1. В A* вершины именуются целыми числами (в т. ч. отрицательными).

Жадный алгоритм

Задание.

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

Описание алгоритма.

В начале работы алгоритму на вход подается начальная и конечная вершины графа. Текущая вершина записывается как посещённая, и все не посещённые вершины, соседние для текущей, помещаются в вектор вершин соседей. Если вектор соседей оказался пустым, то извлекаем из вектора (играет роль стека) посещений вершин вершину и объявляем ее текущей.

Цикл повторяется снова. Если вектор соседей непустой, то среди соседей ищется вершина с минимальным ребром до нее, текущая вершина отмечается в контейнере посещенных вершин и новой текущей вершиной становится найденная соседняя, а в контейнер пар переходов кладётся новая пара. Итерация повторяется уже для новой вершины.

Выход из цикла осуществляется, когда текущая вершина совпадает с вершиной, которую необходимо достичь.

В консоль выводится результат работы алгоритма и промежуточные результаты, такие как текущие вершины, соседи текущей вершины, расстояния до соседних вершин и осуществляемый переход.

Сложность алгоритма по операциям: O (N+E), N – количество вершин, E – количество ребер

Сложность алгоритма по памяти: O (N+E), N – количество вершин, E – количество ребер

Описание функций и структур данных.

std::map<char, std::vector<std::pair<char, double>>> graph;

Структура данных, используемая для хранения графа. Представляет собой ассоциативный контейнер хранения пар вершин графа. Для каждой вершины хранится ее вектор пар (соседних достижимых вершин) и расстояния до них.

std::map<char, std::vector<std::pair<char, double>>>::iterator graph_iterator;

Итератор для графа graph.

void fillingGraph()

Функция заполнения графа данными. В цикле считываются пары вершин графа и расстояния между ними, далее происходит заполнение ассоциативного контейнера graph этими данными.

void runSearch(char start, char finish)

Функция, обеспечивающая алгоритм жадного поиска пути в графе от вершины start до вершины finish.

int minimal_element(std::vector<std::pair<char, double>> neighbours)

Функция нахождения минимального элемента в векторе пар neighbours. Возвращаемым значением является индекс минимального значения в этом векторе пар.

Тестирование.

Входные данные:

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

Результат работы программы:

Current vertex: a

Next neighbors of vertex a and path to their:
    b --> 3
    d --> 5

Go to the b

Current vertex: b

Next neighbors of vertex b and path to their:
    c --> 1

Go to the c

Current vertex: c

Next neighbors of vertex c and path to their:
    d --> 1

Go to the d

Current vertex: d

Next neighbors of vertex d and path to their:
    e --> 1

Go to the e

The algorithm has finished work
Result of algorithm: a b c d e

Входные данные:

a e
a b 3.0
b c 1.0
c d 1.0
a d 4.0
d e 1.0

Результат работы программы:

```
---------------------------------
Current vertex: a

Next neighbors of vertex a and path to their:
      b --> 3
      d --> 2

Go to the d
---------------------------------
Current vertex: d

Next neighbors of vertex d and path to their:
      e --> 1

Go to the e
--------------------------------
The algorithm has finished work
Result of algorithm: a d e
```

Входные данные:

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

Результат работы программы:

```
---------------------------------
Current vertex: a

Next neighbors of vertex a and path to their:
      b --> 3
```

d --> 4

Go to the b

Current vertex: b

Next neighbors of vertex b and path to their:
d --> 2
c --> 3

Go to the f

Current vertex: f

Next neighbors of vertex f and path to their:
c --> 3

Go to the c

Current vertex: c

Next neighbors of vertex c and path to their:
d --> 1

Go to the d

Current vertex: d

Next neighbors of vertex d and path to their:
e --> 1

Go to the e

The algorithm has finished work
Result of algorithm: a b f c d e

A*.

Задание.

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. Вершины именуются целыми числами (в т. ч. отрицательными).

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.

В начале работы алгоритму на вход подается начальная и конечная вершины графа. Перед началом работы в вектор пар вершин-расстояний помещается пара <начальная вершина, 0>.

Далее запускается основной цикл алгоритма. Из вектора расстояний выбирается вершина, путь до которой от начальной вершины плюс значение эвристической функции минимально. Вершина отмечается посещённой в

ассоциативном контейнере значений и идёт цикл перебора всех соседей текущей вершины, которые еще не были посещены. Если соседа-вершины нет в векторе расстояний (путь до вершины найден впервые), то она добавляется в вектор дистанций и в ассоциативный контейнер путей пар вершин.

В случае, когда сосед есть в векторе расстояний до вершин, то осуществляется проверка на то, является ли текущий путь до этого соседа короче. В этом случае в вектор расстояний перезаписывается расстояние до соседа и в контейнер путей также перезаписывается новая пара. В противном случае перебор соседей продолжается.

Когда осуществлен перебор всех соседних вершин для текущей вершины, из вектора расстояний удаляется пара с текущей вершиной.

Цикл повторяется до тех пор, пока текущая вершина не окажется финишной. Среди вектора расстояний снова находят оптимальную вершину, объявляют ее текущей, отмечают как посещённую и начинается перебор всех соседних вершин.

Результатом работы алгоритма является кратчайший путь от начальной до конечной вершины, который выводится на консоль. Также выводятся промежуточные значения, такие как текущая вершина, ее соседи, путь до ее соседей, значение эвристической функции для них и оптимальные пути до вершин.

Сложность алгоритма по операциям зависит от эвристики

$|h(x)-h^*(x)| \leq O(\log h^*(x))$

в лучшем случае - O (N+E), N – количество вершин, E – количество ребер

в худшем случае - O (2^E), E – количество ребер

Сложность алгоритма по памяти: в лучшем случае - O (N+E), N - количество вершин, E – количество ребер

в худшем случае - O (2^E), E – количество ребер

Описание функций и структур данных.

std::map<char, std::vector<std::pair<char, double>>> graph;

Структура данных, используемая для хранения графа. Представляет собой ассоциативный контейнер хранения пар вершин графа. Для каждой вершины хранится ее вектор пар (соседних достижимых вершин) и расстояния до них.

std::map<char, std::vector<std::pair<char, double>>>::iterator graph_iterator;

Итератор для графа graph.

void fillingGraph()

Функция заполнения графа данными. В цикле считываются пары вершин графа и расстояния между ними, далее происходит заполнение ассоциативного контейнера graph этими данными.

void runSearch(char start, char finish)

Функция, обеспечивающая алгоритм A* поиска кратчайшего пути в графе от вершины start до вершины finish.

void removeEl(std::vector<std::pair<int, double>>& distance, std::pair<int, double> el)

Функция удаления элемента из вектора пар вершина-расстояние distance, el – элемент пара вершина-расстояние для удаления из distance.

int findEl(std::vector<std::pair<int, double>>& distance, int vertex)

Функция поиска элемента в векторе пар вершина-расстояние distance. Vertex – вершина, для которой осуществляется поиск. Возвращаемым значением функции является индекс элемента поиска в векторе.

std::pair<int, double> minimal_element(std::vector<std::pair<int, double>> distance, int finish)

Функция нахождения вершины с оптимальным по расстоянию и значению эвристической функции до неё путём в векторе пар вершина-расстояние distance. Finish - финишная вершина поиска пути. Возвращаемым значением является минимальная пара вершина-расстояние.

int heuristic(int first, int second)

Эвристическая функция, возвращающая величину близости двух чисел. First, second – вершины, для которых высчитывается функция.

Тестирование.

Входные данные:

-1 3

-1 0 3.0

0 1 1.0

1 2 1.0

-1 2 5.0

2 3 1.0

Результат работы программы:

Current vertex: -1

Find new vertex 0 with path length 3

Value of heuristic function: 3

Find new vertex 2 with path length 5

Value of heuristic function: 1

Optimal length path to vertex:

    0 --> 3

    2 --> 5

Current vertex: 0

Find new vertex 1 with path length 4

Value of heuristic function: 2

Optimal length path to vertex:

    2 --> 5

    1 --> 4

Current vertex: 2

Find new vertex 3 with path length 6

Value of heuristic function: 0

Optimal length path to vertex:

    1 --> 4

    3 --> 6

Current vertex: 1

Find new vertex 2 with path length 6

Value of heuristic function: 1

Optimal length path to vertex:

3 --> 6

The algorithm has finished work

Result of algorithm: -1 2 3

Входные данные:

-1 3

-1 0 3.0

0 1 1.0

1 2 1.0

-1 2 7.0

2 3 1.0

Результат работы программы:

Current vertex: -1

Find new vertex 0 with path length 3

Value of heuristic function: 3

Find new vertex 2 with path length 7

Value of heuristic function: 1

Optimal length path to vertex:

0 --> 3

2 --> 7

Current vertex: 0

Find new vertex 1 with path length 4

Value of heuristic function: 2

Optimal length path to vertex:

2 --> 7

1 --> 4

Current vertex: 1

Find new vertex 2 with path length 4

Value of heuristic function: 1

Optimal length path to vertex:

2 --> 5

Current vertex: 2

Find new vertex 3 with path length 6

Value of heuristic function: 0

Optimal length path to vertex:

3 --> 6

The algorithm has finished work

Result of algorithm: -1 0 1 2 3

Входные данные:

-1 3

-1 0 3.0

-1 4 1.0

4 2 1.0

0 1 1.0

1 2 1.0

-1 2 7.0

2 3 1.0

Результат работы программы:

Current vertex: -1

Find new vertex 0 with path length 3

Value of heuristic function: 3

Find new vertex 4 with path length 1

Value of heuristic function: 1

Find new vertex 2 with path length 7

Value of heuristic function: 1

Optimal length path to vertex:

    0 --> 3

    4 --> 1

2 --> 7

Current vertex: 4

Find new vertex 2 with path length 2

Value of heuristic function: 1

Optimal length path to vertex:

0 --> 3

2 --> 7

2 --> 2

Current vertex: 2

Find new vertex 3 with path length 3

Value of heuristic function: 0

Optimal length path to vertex:

0 --> 3

2 --> 2

3 --> 3

Current vertex: 2

Find new vertex 3 with path length 3

Value of heuristic function: 0

Optimal length path to vertex:

0 --> 3

3 --> 3

3 --> 3

The algorithm has finished work

Result of algorithm: -1 4 2 3

Выводы.

В ходе выполнения лабораторной работы были получены навыки по использованию алгоритмов поиска кратчайшего пути в графах. Написаны программы, реализующие жадный алгоритм поиска и алгоритм A*.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД
ЖАДНЫЙ АЛГОРИТМ ПОИСКА

```
#include <iostream>
#include <vector>
#include <map>
#include <fstream>

std::ifstream in;

class Graph{                                    //Класс для работы с
графом
private:
    std::map<char, std::vector<std::pair<char, double>>> graph;
//Хранение пар вершин с расстоянием между ними
    std::map<char, std::vector<std::pair<char, double>>>::iterator
graph_iterator;    //Итератор вершин графа

public:

    void fillingGraph() {                       //Заполнение графа
        while (true) {
```

```
            char vertex1;                   //Пара вершин
            char vertex2;                   //и расстояние
            double weight;                  //между ними
            if (in >> vertex1 && in >> vertex2 >> weight){
                graph[vertex1].push_back(std::pair<char,
double>(vertex2, weight));
                continue;
            }
            break;
        }
    }

    void runSearch(char start, char finish){
//Функция жадного алгоритма
        std::map<char,bool> visitedVertexs;
//Посещенные вершины
        std::map<char,char> way;
//Контейнер путей
        std::vector<std::pair<char, double>> stackPath;
//Стэк для хранения вершин

        std::pair<char, double> current = std::pair<char,
double>(start, 0);

        for(graph_iterator = graph.begin();
graph_iterator!=graph.end(); graph_iterator++)
            visitedVertexs[graph_iterator->first]=false;

        while(true){

            if(current.first == finish){
//Пока текущая вершина не будет конечной выполняется цикл
                std::cout << "--------------------------------
"<<std::endl;
                std::cout << "The algorithm has finished work";
                break;
            }
```

```
            std::cout << "-----------------------------------" <<
std::endl;
            std::cout << "Current vertex: " << current.first <<
std::endl<<std::endl;
            std::vector<std::pair<char, double>> neighbours;
//Вектор соседей текущей вершины
            visitedVertexs[current.first]=true;

            for(std::pair<char,double> neighbor :
graph[current.first]) {
                if (!visitedVertexs[neighbor.first])
                    neighbours.push_back(neighbor);
//Заполнение вектора соседних вершин
            }

            std::cout << "Next neighbors of vertex " << current.first
<< " and path to their" << ":" << std::endl;
            if (neighbours.size() == 0)
                std::cout << "Empty"<<std::endl;

            for (int i=0; i<neighbours.size(); i++){
                std::cout <<"\t"<< neighbours[i].first << " --> " <<
neighbours[i].second << std::endl;
            }
            std::cout << std::endl;

            if (neighbours.size() == 0){
//Если зашли в тупик, то возвращаемся в предыдущую вершину
                current = stackPath[stackPath.size()-1];
                std::cout << "Back to the: " << current.first <<
std::endl;
                stackPath.pop_back();
            }

            else {
                stackPath.push_back(current);
//Иначе переходим в новую вершину, старую кладем на стэк
                current = neighbours[minimal_element(neighbours)];
                std::cout << "Go to the " << current.first <<
```

```
std::endl;
                way[current.first] = stackPath[stackPath.size()-
1].first;
            }

        }


        std::vector<char> path;       //Вектор для хранения пути
вершин

        char vertex = finish;               //Идем с конца наперед,
восстанавливая путь, пока не придем в начало
        while (vertex != start ){
            path.push_back(vertex);
            vertex = way.find(vertex)->second;

        }
        path.push_back(vertex);


        std::cout<< std::endl <<"Result of algorithm: ";
        for (int i=path.size()-1; i>=0; i--)
            std::cout << path[i] << " ";                //Печать пути
        std::cout << std::endl;
    }

private:

    int minimal_element(std::vector<std::pair<char, double>>
neighbours){  //Функция нахождения минимального элемента в векторе
соседей
        double min_value = neighbours[0].second;
        int min = 0;
        for (int i = 1; i<neighbours.size(); i++){
            if (neighbours[i].second < min_value) {         //Если
находится элемент меньше минимально, то он становится минимальным
                min_value = neighbours[i].second;
                min = i;
```

```
            }
        }
        return min;                                    //Возвращается
индекс минимального элемента
    }

};

int main() {
    in.open("test.txt");
    char start, finish;
    in >> start;
    in >> finish;
    Graph graph;                                       //Создание
графа
    graph.fillingGraph();                              //Заполнение
графа
    graph.runSearch(start,finish);                     //Поиск
кратчайшего пути
    return 0;
}
```

АЛГОРИТМ A*

```
#include <iostream>
#include <vector>
#include <map>
#include <fstream>
#include <algorithm>

std::ifstream in;

class Graph{
//Класс для работы с графом
private:
    std::map<int, std::vector<std::pair<int, double>>> graph;
//Хранение пар вершин с расстоянием между ними
    std::map<int, std::vector<std::pair<int,double>>>::iterator
graph_iterator;             //Итератор вершин графа

public:

    void fillingGraph() {                           //Заполнение графа
        while (true) {
            int vertex1;                            //Пара вершин
            int vertex2;                            //и расстояние
            double weight;                          //между ними
            if (in >> vertex1 && in >> vertex2 >> weight){
                graph[vertex1].push_back(std::pair<int,
double>(vertex2, weight));
                continue;
            }
```

```
            break;
        }
    }



    void runSearch(int start, int finish){
//Функция поиска кратчайшего пути
        std::vector<std::pair<int, double>> distance;
//Контейнер дистанций между вершинами
        std::map<int,bool> visitedVertexs;
//Посещенные вершины
        std::map<int,int> way;
//Контейнер путей

        for(graph_iterator = graph.begin();
graph_iterator!=graph.end(); graph_iterator++)
            visitedVertexs[graph_iterator->first]=false;

        distance.emplace_back(std::pair<int, double>(start, 0));

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

            std::pair<int, double> current = minimal_element(distance,
finish);      //Нахождение ближайшей вершины

            if(current.first == finish){
//Если текущая вершина финишная, то выходим из цикла
                std::cout << "--------------------------------
"<<std::endl;
                std::cout << "The algorithm has finished work";
                break;
            }


            std::cout << "-----------------------------------" <<
std::endl;
            std::cout << "Current vertex: " << current.first <<
std::endl<<std::endl;

            visitedVertexs[current.first]=true;

            for(std::pair<int,double> neighbor : graph[current.first])
{            //Цикл для всех соседних вершин, относительно текущей
                if (!visitedVertexs[neighbor.first]) {
                    std::pair<int, double> *distanceToNeighbor =
&distance[findEl(distance, neighbor.first)];  //Находим вершину в
векторе дистанций
                    if (distanceToNeighbor->first ==
distance[distance.size() - 1].first) {                          //Если
еще не посещалась вершина
                        distance.push_back(std::pair<int,
double>(neighbor.first, current.second + neighbor.second));
                        way[neighbor.first] = current.first;
```

```
//Добавляем вершину в путь

                    }
                    else if (current.second + neighbor.second <
distanceToNeighbor->second) {                    //Если путь до уже
найденной вершины короче из текущей вершины
                        way[neighbor.first] = current.first;
                        distanceToNeighbor->second = current.second +
neighbor.second;                                 //Перестраиваем путь
                    }

                }
                std::cout << "Find new vertex  " << neighbor.first <<
"  with path length " << distance[distance.size()-1].second <<
std::endl;
                std::cout << "Value of heuristic function: " <<
heuristic(neighbor.first, finish) << std::endl << std::endl;


            }
                removeEl(distance, current);

                std::cout << "Optimal length path to vertex:" <<
std::endl;
                for (int i = 0; i<distance.size(); i++){
                    std::cout << "\t" << distance[i].first << " --> "
<< distance[i].second << std::endl;
                }

        }

            std::vector<int> path;                    //Вектор пути

            int vertex = finish;                      //Обратным путем
из финиша до конца восстанавливаем путь
            while (vertex != start ){
                path.push_back(vertex);
                vertex = way.find(vertex)->second;
            }
            path.push_back(vertex);


            std::cout<< std::endl <<"Result of algorithm: ";
            for (int i=path.size()-1; i>=0; i--)
                std::cout << path[i] << " ";
//Выводим найденный путь
    }

private:
    void removeEl(std::vector<std::pair<int, double>>& distance,
std::pair<int, double> el){    //Функция удаления элемента из вектора
дистанций
        for (int i=0; i<distance.size(); i++){
            if (distance[i].first == el.first){
```

```
                for (i; i<distance.size() - 1; i++)
                    distance[i] = distance[i+1];
                distance.pop_back();
                return;
            }
        }
    }

    int findEl(std::vector<std::pair<int, double>>& distance, int
vertex){      //Функция нахождения элемента в векторе дистанций
        for (int i=0; i<distance.size(); i++){
            if (distance[i].first == vertex)
                return i;
//Возвращается индекс элемента в векторе
        }
        return distance.size()-1;
    }

    std::pair<int, double> minimal_element(std::vector<std::pair<int,
double>> distance, int finish){  //Функция нахождения минимального
элемента в векторе
        double min_value = distance[0].second +
heuristic(distance[0].first, finish);
        std::pair<int, double> min = distance[0];
        for (int i = 1; i<distance.size(); i++){
            if (distance[i].second + heuristic(distance[i].first,
finish) < min_value) {
                min_value = distance[i].second +
heuristic(distance[i].first, finish);
                min = distance[i];
            }
        }
        return min;
//Возвращается индекс минимального элемента
    }


    int heuristic(int first, int second){
//Эвристическая функция
        return abs(first - second);
    }


};

int main() {
    in.open("test.txt");
    int start, finish;
    in >> start;
    in >> finish;
    Graph graph;                                    //Создание графа
    graph.fillingGraph();                           //Заполнение графа
    graph.runSearch(start,finish);                  //Поиск
кратчайшего пути
```

```
    return 0;
}
```