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. 8382 _____ Кобенко В.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Научиться использовать жадный алгоритм и алгоритм A* поиска кратчайшего пути на графе путём разработки программ.

Задание.

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

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.

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.

Алгоритм принимает начальную и конечную вершины на графе. Текущая вершина (для нулевой итерации это начальная) записывается как посещённая, и все непосещённые вершины, которые из неё можно достичь (соседи), помещаются в массив. Если массив оказался пустым, то извлекаем из массива посещения вершин (выполняет роль стека) последнюю и она становится текущей. Цикл повторяется снова. Если массив соседей оказался непустым, то среди соседей ищется минимальный путь до одного из них. Тогда текущая вершина кладётся в массив посещений и новой текущей вершиной становится найденная соседняя, а в массив пар переходов <куда, откуда> кладётся новая пара, выполняется следующая итерация для новой текущей вершины.

Условием выхода является совпадение текущей вершины на некоторой итерации с вершиной, которую нужно достичь.

В файл выводятся промежуточные значения в виде введённого графа и начальной, конечной вершин, номера итерации, текущей вершины, её соседей и расстояний до непосещённых из них. В конце выводится итоговый кратчайший путь.

Сложность алгоритма по операциям:

В худшем случае алгоритм обойдёт все рёбра графа. После прохода через очередное ребро будет осуществляться поиск минимального пути из достигнутой вершины, который будет использовать O(e*n) операций, где e – количество рёбер в графе, n – количество вершин, поскольку для вершины перебираются все исходящие пути (e) и для каждой достижимой вершины проверяется, была ли она посещена (n операций). Если пути до соседних непосещённых вершин нашлись, то среди этих путей ищется минимальное

ребро (n операций). Тогда общая сложность составит O (e*(n*e + n)) = O(n*e*e + n*e).

Сложность алгоритма по памяти:

O (n + e), где n – количество вершин в графе, e – количество рёбер.

Описание функций и структур данных.

1.

```
class Graph{
        map <Uchar, vector <pair <Uchar, Udouble>>> graph;
```

Является структурой данных, используемой для представления графа.

Graph – контейнер типа map, содержащий весь граф. Uchar, Udouble – типы переменных определённых через typedef. Может быть как char, так и int. В переменной graph каждой вершине ставится в соответствие пара или несколько пар вида <имя вершины, путь до неё>.

2.

```
int minPathfunc(vector <pair <Uchar, Udouble> >& Vpaths)
```

Функция предназначена для поиска индекса вершины с минимальным до неё путём в массиве соседей Vpaths. Vpaths – вектор пар непосещённых соседей и расстояний до них. Функция возвращает индекс минимальной вершины.

3.

```
void resFunc(vector <pair <Uchar, Uchar> >& Vpeeks, Uchar begin,
Uchar end)
```

Функция предназначена для печати полученной последовательности путей. Vpeeks – вектор пар вершин <куда, откуда>, заполненный в ходе передвижения по вершинам в основном алгоритме, begin – начальная вершина (старт), end – конечная вершина (финиш).

4.

```
void path_from_S_to_P(vector <pair <Uchar, Udouble> >& Vpaths)
```

Функция выводит промежуточные данные, а именно вершины, которые можно достичь из текущей, и расстояния до них. Vpaths – вектор непосещённых соседей вершины.

5.

```
void algGr(Uchar begin, Uchar end)
```

Метод класса Graph, реализующий алгоритм жадного поиска. Begin – начальная вершина на графе, end – конечная вершина.

6.

```
void init()
```

Метод класса Graph, позволяющий ввести данные из командной строки терминала.

Тестирование.

1.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ gr.cpp && ./a.out
a e a b 3 b c 1 c d 1 a d 3 d e 1
a e
a b 3
b c 1
c d 1
a d 3
d e 1
!
----------
Текущая вершина: a
Пути из начальной вершины:в (b) за (3), в (d) за (3),
----------
Текущая вершина: b
Пути из начальной вершины:в (c) за (1),
----------
Текущая вершина: c
Пути из начальной вершины:в (d) за (1),
----------
Текущая вершина: d
Пути из начальной вершины:в (e) за (1),
----------
Текущая вершина: e
abcde
vlad@vlad-GL62M-7RDX:~/PAA$
```

2.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ gr.cpp && ./a.out
a e a b 2 b c 2 c d 2 a d 1 d e 2
a e
a b 2
b c 2
c d 2
a d 1
d e 2
!
----------
Текущая вершина: a
Пути из начальной вершины:в (b) за (2), в (d) за (1),
----------
Текущая вершина: d
Пути из начальной вершины:в (e) за (2),
----------
Текущая вершина: e
ade
vlad@vlad-GL62M-7RDX:~/PAA$
```

3.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ gr.cpp && ./a.out
a e a b 1 a c 2 a d 3 d e 2 c e 1 !
a e
a b 1
a c 2
a d 3
d e 2
c e 1
----------
Текущая вершина: a
Пути из начальной вершины:в (b) за (1), в (c) за (2), в (d) за (3),
----------
Текущая вершина: b
----------
Текущая вершина: a
Пути из начальной вершины:в (c) за (2), в (d) за (3),
----------
Текущая вершина: c
Пути из начальной вершины:в (e) за (1),
----------
Текущая вершина: e
ace
vlad@vlad-GL62M-7RDX:~/PAA$
```

Задание.

A*.

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

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.

Алгоритм принимает начальную и конечную вершины на графе. Перед циклом в массив пар вершин и расстояний до них из начальной вершины помещается пара <начало, 0>.

Запускается цикл. Текущей вершиной становится вершина из массива расстояний, сумма эвристической функции которой и расстояния до неё минимальна. Вершина отмечается посещённой. Далее идёт цикл перебора всех соседей текущей вершины. Рассматриваются только непосещённые вершины. Если найденного соседа нет в массиве расстояний, т.е. путь до этой вершины найден впервые, то она добавляется в этот массив и в массив кратчайших путей пар вершин <куда, откуда>. Если найденный сосед есть в массиве расстояний до вершин, то проверяется: является ли текущий путь до соседа более оптимальным, если так, то в массиве расстояний перезаписывают расстояние до соседа и в массиве путей второй член пары <куда, откуда> заменяется на текущую вершину, если же текущий найденный путь не лучше уже найденного, перебор соседей продолжается.

Когда перебор всех вершин, которые можно достичь из текущей, закончен, из массива расстояний удаляется пара с текущей вершиной, поскольку были получены производные от неё пути. Также на этом этапе выводятся промежуточные данные. Цикл снова повторяется: среди массива расстояний снова находят оптимальную вершину (приоритет которой минимален), которая становится текущей, её отмечают как посещённую и начинается перебор всех соседей этой вершины.

Условием выхода является совпадение текущей вершины на некоторой итерации с вершиной, которую нужно достичь или выполнением всего числа итераций (количество вершин в графе).

В файл выводятся промежуточные значения в виде введённого графа и начальной, конечной вершин, номера итерации, текущей вершины, её соседей и

лучших расстояний до непосещённых вершин. В конце выводится итоговый кратчайший путь.

Сложность алгоритма по операциям зависит от эвристики

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

В цикле выполняется e итераций, где e – количество рёбер в графе.

На каждой итерации в массиве кратчайших расстояний выполняется поиск вершины с кратчайшим до неё путём с учётом эвристической функции (n операций, где n – количество вершин графа).

Далее идёт цикл по всем соседям вершины со всеми прямыми путями до них (e). Проверка на непосещённость вершины из текущей занимает n операций, проверка на содержание этой вершины в массиве кратчайших расстояний тоже занимает n операций. Если вершина содержится в массиве кратчайших расстояний, то на поиск вершины в этом массиве и замену величины кратчайшего пути до неё (если найден лучше) тратится (n + n) операций. Все вставки и извлечения из массивов константны по операциям. Получаем O (e (n + e (n + n + n + n))) = O (e (n + 4*e*n)) = O (e*n + 4*n*e*e) в лучшем случае, где e – количество рёбер в графе, n – количество вершин.

O (2^e) в худшем случае, e – количество рёбер.

Сложность алгоритма по памяти:

O (n + e) в лучшем случае, n - количество вершин, e – количество ребер

O (2^e) в худшем случае, e – количество ребер.

Описание функций и структур данных.

1.

```
class Graph{
        map <Uint, vector<pair <Uint, Udouble> > > graph;
```

Является структурой данных, используемой для представления графа.

Graph – контейнер типа map, содержащий весь граф. Uint, Udouble – типы переменных определённых через typedef. Может быть как char, так и int. В переменной graph каждой вершине ставится в соответствие пара или несколько пар вида <имя вершины, путь до неё>.

2.

```
int heuristicFunction(Uint cur, Uint end)
```

Эвристическая функция. Возвращает величину близости чисел. cur – текущая вершина графа, end – конечная.

3.

```
int minPathfunc(vector <pair <Uint, Udouble> >& Vpaths, Uint end)
```

Функция предназначена для поиска индекса вершины с оптимальным по расстоянию и значению эвристической функции до неё путём в векторе соседей Vpaths. Vpaths – вектор пар непосещённых соседей, end – конечная вершина графа. Функция возвращает индекс минимальной вершины.

4.

```
int getN(vector <pair <Uint, Udouble> >& Vpaths, Uint neighbor)
```

Функция находит индекс вершины в массиве расстояний. Vpaths – вектор пар вершин и кратчайших до них путей, neighbor – имя соседней вершины, которую нужно в этом массиве найти. Функция возвращает индекс соседа.

5.

```
void delP(vector <pair <Uint, Udouble> > &Vpaths, pair <Uint, Udouble> cur)
```

Функция удаляет пару <вершина, расстояние до неё> из массива кратчайших расстояний. Vpaths – вектор пар вершин с кратчайшим до них

расстоянием из стартовой вершины, cur – текущая пара, подлежащая удалению, поскольку найдены производные от неё пути.

6.

```
int neighborfunc(Uint neighbor, vector <pair <Uint, Udouble> >& Vpaths)
```

Функция определяет, содержится ли сосед в массиве кратчайших расстояний. neighbor – имя соседней вершины для поиска в массиве, Vpaths - вектор пар вершин с кратчайшим до них расстоянием из стартовой вершины. Возвращает 1, если сосед содержится в массиве, иначе – 0.

7.

```
void resFunc(vector <pair <Uint, Uint> >& Vpeeks, Uint begin, Uint end)
```

Функция предназначена для печати полученной последовательности путей. Vpeeks – вектор пар вершин <куда, откуда>, заполненный в ходе передвижения по вершинам в основном алгоритме, begin – начальная вершина (старт), end – конечная вершина (финиш).

8.

```
void path_from_S_to_P(vector <pair <Uint, Udouble> >& Vpaths)
```

Функция выводит промежуточные данные, а именно достигнутые вершины и кратчайшие до них пути. Vpaths – вектор пар вершин с кратчайшим до них расстоянием из стартовой вершины

9.

```
void algA(Uint begin, Uint end)
```

Метод класса Graph, реализующий алгоритм A*. Begin – начальная вершина на графе, end – конечная вершина.

10. `void init()`

Метод класса Graph, позволяющий ввести данные из командной строки терминала.

Тестирование.

1.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ a_star.cpp && ./a.out
1 5
1 5
1 -1 2 -1 3 2 3 -5 1 -5 5 6 1 -5 10 !
1 -1 2
-1 3 2
3 -5 1
-5 5 6
1 -5 10
----------
Текущая вершина: 1
Пути из начальной вершины: в (-1) за (2), в (-5) за (10),
----------
Текущая вершина: -1
Пути из начальной вершины: в (-5) за (10), в (3) за (4),
----------
Текущая вершина: 3
Пути из начальной вершины: в (-5) за (5),
----------
Текущая вершина: -5
Пути из начальной вершины: в (5) за (11),
----------
Текущая вершина: 5
1-13-55
vlad@vlad-GL62M-7RDX:~/PAA$
```

2.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ a_star.cpp && ./a.out
1 5 1 -1 2 -1 3 2 3 -5 1 -5 5 6 1 -5 1 !
1 5
1 -1 2
-1 3 2
3 -5 1
-5 5 6
1 -5 1
----------
Текущая вершина: 1
Пути из начальной вершины: в (-1) за (2), в (-5) за (1),
----------
Текущая вершина: -1
Пути из начальной вершины: в (-5) за (1), в (3) за (4),
----------
Текущая вершина: 3
Пути из начальной вершины: в (-5) за (1),
----------
Текущая вершина: -5
Пути из начальной вершины: в (5) за (7),
----------
Текущая вершина: 5
1-55
vlad@vlad-GL62M-7RDX:~/PAA$
```

3.

```
vlad@vlad-GL62M-7RDX:~/PAA$ g++ a_star.cpp && ./a.out
-1 1 -1 5 3 -1 2 1 5 1 2 2 1 4 !
-1 1
-1 5 3
-1 2 1
5 1 2
2 1 4
----------
Текущая вершина: -1
Пути из начальной вершины: в (5) за (3), в (2) за (1),
----------
Текущая вершина: 2
Пути из начальной вершины: в (5) за (3), в (1) за (5),
----------
Текущая вершина: 1
-121
vlad@vlad-GL62M-7RDX:~/PAA$
```

Выводы.

Были получены умения по использованию алгоритмов поиска кратчайшего пути в графах. Написаны программы, реализующие жадный алгоритм поиска, принимающий локальные оптимальные решения, и алгоритм A*, принимающий глобально оптимальные решения благодаря эвристической функции.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

ЖАДНЫЙ АЛГОРИТМ ПОИСКА

```
#INCLUDE <IOSTREAM>
#include <vector>
#include <map>
#include <utility>

typedef char Uchar;
typedef long double Udouble;

using namespace std;

int minPathfunc(vector <pair <Uchar, Udouble> >& Vpaths){
pair <Uchar, Udouble> min = Vpaths[0];
int index = 0;
for (int i = 1; i < Vpaths.size(); i++){
if (Vpaths[i].second < min.second){
min = Vpaths[i];
index = i;
}
}
return index;
}

void resFunc(vector <pair <Uchar, Uchar> >& Vpeeks, Uchar begin, Uchar end){

vector <Uchar> path;
Uchar tmp = end;
while(tmp != begin){
for (pair <Uchar, Uchar> curPath : Vpeeks){
if (tmp == curPath.first){
path.insert(path.begin(), tmp);
tmp = curPath.second;
break;
}
}
}
path.insert(path.begin(), tmp);
for (Uchar elem : path)
cout << elem;
cout << endl;
}

void path_from_S_to_P(vector <pair <Uchar, Udouble> >& Vpaths){
cout << "Пути из начальной вершины:";
for (pair <Uchar, Udouble> pair : Vpaths){
cout << "в (" << pair.first << ") за (" << pair.second << "), ";
}
cout << endl;
```

```
}

class Graph{
map <Uchar, vector <pair <Uchar, Udouble>>> graph;
public:
Graph(){
}
void algGr(Uchar begin, Uchar end) {
string Upath;
vector <pair <Uchar, Udouble> > Vpath_f;
pair <Uchar, Udouble> cur = pair <Uchar, Udouble> (begin, 0);
vector <pair <Uchar, Uchar> > Vpeeks;
int i = 0;

while (true) {
cout << "---------- " << endl;
cout << "Текущая вершина: " << cur.first << endl;
if (cur.first == end) {
break;
}
Upath.push_back(cur.first);
vector <pair <Uchar, Udouble> > Vpaths;
for (pair <Uchar, Udouble> neighbor : graph[cur.first]) {
if (Upath.find(neighbor.first) == string::npos) {
Vpaths.push_back(neighbor);
}
}
if (Vpaths.size() != 0) {
Vpath_f.push_back(cur);
pair <Uchar, Udouble> prevCur = cur;
cur = Vpaths[minPathfunc(Vpaths)];
Vpeeks.push_back(pair <Uchar, Uchar> (cur.first, prevCur.first));
}
else {
cur = Vpath_f[Vpath_f.size() - 1];
Vpath_f.pop_back();
continue;
}
path_from_S_to_P(Vpaths);
}
resFunc(Vpeeks, begin, end);
}
void init(){
Uchar sp, ep;
Udouble length;
while(cin >> sp && sp != '!' && cin >> ep && cin >> length){
graph[sp].push_back (pair <Uchar, Udouble> (ep, length));
cout << sp << " " << ep << " " << length << endl;
}
}
};
```

```
int main() {
Uchar sp, ep;
cin >> sp >> ep;
cout << sp << " " << ep << endl;
Graph graph;
graph.init();
graph.algGr(sp, ep);
return 0;
}
```

АЛГОРИТМ A*

```
#INCLUDE <IOSTREAM>
#include <vector>
#include <map>
#include <utility>

typedef long int Uint;
//typedef char Uint;
typedef long double Udouble;

using namespace std;

int heuristicFunction(Uint cur, Uint end){
return abs(cur - end);
}

int minPathfunc(vector <pair <Uint, Udouble> >& Vpaths, Uint end){
pair <Uint, Udouble> min = Vpaths[0];
int index = 0;
for (int i = 1; i < Vpaths.size(); i++){
if (Vpaths[i].second + heuristicFunction(Vpaths[i].first, end) < min.second +
heuristicFunction(min.first, end)){
min = Vpaths[i];
index = i;
}
}
return index;
}

int getN(vector <pair <Uint, Udouble> >& Vpaths, Uint neighbor) {
for (int i = 0; i < Vpaths.size(); i++) {
if (Vpaths[i].first == neighbor)
return i;
}
return Vpaths.size()-1;
}

void delP(vector <pair <Uint, Udouble> > &Vpaths, pair <Uint, Udouble> cur) {
for (int i = 0; i < Vpaths.size(); i++) {
if (Vpaths[i].first == cur.first) {
for (i; i < Vpaths.size() - 1; i++) {
Vpaths[i] = Vpaths[i + 1];
}
Vpaths.pop_back();
return;
}
}
}

int neighborfunc(Uint neighbor, vector <pair <Uint, Udouble> >& Vpaths){
```

```
for (int i = 0; i < Vpaths.size(); i++) {
if (Vpaths[i].first == neighbor) {
return 1;
}
}
return 0;
}

void resFunc(vector <pair <Uint, Uint> >& Vpeeks, Uint begin, Uint end){

vector <Uint> path;
Uint tmp = end;
while(tmp != begin){
for (pair <Uint, Uint> curPath : Vpeeks){
if (tmp == curPath.first){
path.insert(path.begin(), tmp);
tmp = curPath.second;
break;
}
}
}
path.insert(path.begin(), tmp);
for (Uint elem : path)
cout << elem;
cout << endl;
}

void path_from_S_to_P(vector <pair <Uint, Udouble> >& Vpaths){
cout << "Пути из начальной вершины: ";
for (pair <Uint, Udouble> pair : Vpaths){
cout << "в (" << pair.first << ") за (" << pair.second << "), ";
}
cout << endl;
}

class Graph{
map <Uint, vector<pair <Uint, Udouble> > > graph;

public:
Graph(){
}
void algA(Uint begin, Uint end) {
string Upath; //Путь пользователя
vector <pair <Uint, Udouble> > Vpaths; //Вектор путей
vector <pair <Uint, Uint> > Vpeeks; //Вектор вершин
Vpaths.push_back(pair <Uint, Udouble> (begin, 0));
for (int i = 0; i <= graph.size(); i++) {
cout << "---------- " << endl;
pair <Uint, Udouble> cur = (Vpaths[minPathfunc(Vpaths, end)]);
cout << "Текущая вершина: " << cur.first << endl;
if (cur.first == end) {
```

```
break;
}
Upath.push_back(cur.first);

for (pair <Uint, Udouble> neighbor : graph[cur.first]) {
if (Upath.find(neighbor.first) == string::npos) {
if(!neighborfunc(neighbor.first, Vpaths)) {
Vpaths.push_back(pair <Uint, Udouble> (neighbor.first, cur.second + neighbor.second));
Vpeeks.push_back(pair <Uint, Uint> (neighbor.first, cur.first));
}
else {
pair <Uint, Udouble> *neighborInVpaths = &(Vpaths[getN(Vpaths, neighbor.first)]);
if (cur.second + neighbor.second < neighborInVpaths->second) {
for (int i = 0; i < Vpeeks.size(); i++) {
if (Vpeeks[i].first == neighbor.first)
Vpeeks[i].second = cur.first;
}
neighborInVpaths->second = cur.second + neighbor.second;
}
}
}
}
delP(Vpaths, cur);
path_from_S_to_P(Vpaths);
}
resFunc(Vpeeks, begin, end);
}
void init(){
Uint sp, ep;
Udouble length;
while(cin >> sp /*&& sp != '!'*/ && cin >> ep && cin >> length){
if (length <= 0)
break;
graph[sp].push_back (pair <Uint, Udouble> (ep, length));
cout << sp << " " << ep << " " << length << endl;
}
}
};

int main() {
Uint sp, ep;
cin >> sp >> ep;
cout << sp << " " << ep << endl;
Graph graph;
graph.init();
graph.algA(sp, ep);
return 0;
}
```