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 _____ Сенюшкин Е.В.

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.

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

abcde

Алгоритм A*.

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.

Вариант 7.

"Мультипоточный" A*: на каждом шаге из очереди с приоритетами извлекается n вершин (или все вершины, если в очереди меньше n вершин). n задаётся пользователем.

Описание жадного алгоритма.

Алгоритм принимает на вход начальную и конечную вершину и граф. Для начала на стек кладется начальная вершина и она же делается текущей. Для текущей вершины проверяется есть ли из нее пути, если путей нет, то вершина помечается как посещенная, удаляется со стека и текущей вершиной становится самая верхняя вершина на стеке, если пути есть то среди них выбирается самый дешевый, он кладется на стек и делается текущей вершиной. Условием остановки является момент, когда текущей вершиной становится конечная вершина.

Сложность алгоритма.

По скорости.

В худшем случае алгоритм обойдет весь граф и для каждой вершины будет искать минимальный путь.

Сложность получается O(n * m + n), где n — кол-во вершин, а m — кол-во ребер.

По памяти.

В памяти хранится только граф, полностью просмотренные вершины и уже пройденный путь. Граф хранится в списке смежности O(n+m), просмотренные вершины и путь не могут занимать больше чем количество вершин O(2n).

Сложность по памяти получается O(3n+m) = O(n+m),

где n — кол-во вершин, а m — кол-во ребер.

Описание функций и структур данных.

Структуры данных.

1. map<char, vector<pair<char, double>>> graph

Структура данных для хранения графа. graph контейнер типа map, ключ название вершины, значением является vector, в котором хранятся все вершины, с которыми связана вершина — ключ.

2. map<char, bool> visited

Структура данных для запоминания вершин, в которых все пути уже были просмотрены. viseted контейнер типа map, ключ название вершины, значение есть ли в вершине еще не просмотренные пути.

3. stack<char> way

Стек на котором хранятся вершины из которых состоит текущий путь.

Функции.

1. void readGraph()

Функция, которая считывает граф.

2. void greedySearch()

Функция, которая реализует жадный поиск.

3. void print(stack<char>& result)

Функция, которая после достижение искомой вершины выводит путь до нее.

Описание алгоритма A*.

Алгоритм принимает на вход граф, стартовую и конечную вершину. Стартовая клетка добавляется в очередь с приоритетом. Следующие шаги повторяются пока из очереди с приоритетом не будет снята конечная вершина или очередь не будет пуста. Из очереди с приоритетом снимается вершина, она помечается как посещенная и делается текущей. Для каждой вершины в которую можно попасть из текущей выполняем следующие:

1) Проверяем не отмечена ли она как уже посещенная, если отмечена, то игнорируем.

2) Есть ли клетка в очереди, если нет, то добавляем ее туда, при этом рассчитываем для нее реальный путь и эвристическую оценку и запоминаем для нее родительскую вершину.

3) Если вершина уже есть в очереди, но еще не была посещена, то сравниваем ее значение реального пути, со значением таким, что если бы к ней пришли через текущую вершину. Если сохраненное значение больше нового, то меняем его и меняем родительскую вершину на текущую, если меньше, то ничего не делаем.

Описание функций и структур данных.

Структуры данных.

1. map<char, vector<pair<char, double >>> graph

Структура данных для хранения графа. graph контейнер типа map, ключ название вершины, значением является vector, в котором хранятся все вершины, с которыми связана вершина — ключ.

2. map<char, bool> closeList

Структура данных для хранения уже посещенных вершин. closeList контейнер типа map, ключ название вершины, значение была ли уже посещена вершина.

3. map<char, pair<char, double>> realWay

Структура данных для хранения минимального известного пути до вершины и родительской вершины(вершина из которой мы попали в эту вершину). realWay контейнер типа map ключ название вершины, значение — пара из названия родительской клетки и минимального известный путь до клетки.

4. struct Cell{

```
    char name;
    char parent;
    double rough;
}
```

Структура для хранения вершин и их приоритета в очереди с приоритетом.

5. struct Cmp структура в которой перегружен operator(), для сортировки вершин в очереди с приоритетом.

6. priority_queue <Cell, vector<Cell>, Cmp> openList

Структура для хранения вершин открытых для посещения. openList контейнер типа priority_queue, Cell тип элемента для хранения в очереди, vector<Cell> тип контейнера используемый для реализации очереди, Cmp компаратор с помощью, которого происходит сортировка элементов.

Функции.

1. void readGraph()

Функция, которая считывает граф.

2. void aStar()

Функция, которая реализует жадный поиск.

3. void printWay(char a)

Функция, которая после достижение искомой вершины выводит путь до нее.

Сложность алгоритма.

По скорости.

Сложность алгоритма зависит от эвристики. В лучшем случае, когда эвристика функции идеальна и на каждом шаге выбирается верный путь сложность получается O(n+m), так как максимальная длинна пути до цели может быть n и на каждом шаге нужно добавить в очередь все ребра, которые выходят из текущей вершины. В худшем случае придется просмотреть все пути в таком случае сложность по скорости будет O(a^m), где a — среднее число ветвлений, а n — кол-во вершин.

По памяти.

В лучшем случае O (n+m), n — кол-во вершин, m — кол-во ребер.

В абсолютно худшем случае каждый шаг будет неправильным для каждой новой снятой вершины придется проверить все смежные вершины и если каждый путь в них будет короче чем уже посчитанный их придется добавить в

очередь, тогда сложность будет расти как экспонента. Сложность по памяти будет O(a^m), где a — среднее число ветвлений, а n — кол-во вершин.

Тестирование жадного алгоритма.

```
a e
a b 3.0
a c 5.0
b d 1.0
b f 3.5
b g 1.2
d j 1.0
d k 2.0
f l 12.0
g m 4.0
c h 3.0
c i 3.3
i e 4.0
!
Intermediate way:
a
ab
abd
abdj
abd
abdk
abd
ab
abg
abgm
abg
ab
abf
abfl
abf
ab
a
ac
```

```
a e
a b 0.0
a d 5.0
b c 1.0
c f 1.0
f g 1.0
d e 3.0
e g 2.0
!
Intermediate way:
a
ab
abc
abcf
abcfg
abcf
abc
ab
a
ad
Greedy search: ade
```

```
a e
a b 5.0
a f 2.0
f j 3.0
b f 3.0
b c 0.5
c j 2.0
b g 3.2
g h 2.9
h c 5.0
c i 3.2
i h 2.9
c d 2.7
d e 2.0
!
Intermediate way:
a
af
afj
af
a
ab
abc
abcd
```

```
a e
a b 1.0
a d 4.0
d c 1.0
b c 1.0
c f 1.0
f g 1.0
f e 3.0
e g 2.0
!
Intermediate way:
a
ab
abc
abcf
abcfg
abcf
Greedy search: abcfe
```

```
a e
a b 3.0
a d 3.0
d c 1.0
b c 1.0
c f 1.0
f g 1.0
f e 3.0
e g 2.0
!
Intermediate way:
a
ab
abc
abcf
abcfg
abcf
Greedy search: abcfe
```

```
c d
c j 3.3
c b 4.0
c i 3.4
j o 10.0
j b 4.1
i f 2.7
i k 10.4
k l 1.0
k f 4.3
b f 3.0
o b 3.1
b e 3.0
o e 4.3
e h 3.2
e f 3.2
h f 4.5
h d 8.5
f d 1.4
l d 3.2
!
Intermediate way:
c
cj
cjb
cjbf
Greedy search: cjbfd
```

Тестирование A*.

```
Введите количество вершин снимаемых с очереди за раз: 4
a e
a b 3.0
a c 5.0
b d 1.0
b f 3.5
b g 1.2
d j 1.0
d k 2.0
f l 12.0
g m 4.0
c h 3.0
c i 3.3
i e 4.0
!
Intermediate way:
A* result: acie
```

```
Введите количество вершин снимаемых с очереди за раз: 3

a e
a b 0.0
a d 5.0
b c 1.0
c f 1.0
f g 1.0
d e 3.0
e g 2.0
!
Intermediate way:
A* result: ade
```

```
Введите количество вершин снимаемых с очереди за раз: 1
a e
a b 5.0
a f 2.0
f j 3.0
b f 3.0
b c 0.5
c j 2.0
b g 3.2
g h 2.9
h c 5.0
c i 3.2
i h 2.9
c d 2.7
d e 2.0
!
Intermediate way:
A* result: abcde
```

```
Введите количество вершин снимаемых с очереди за раз: 2
a e
a b 1.0
a d 4.0
d c 1.0
b c 1.0
c f 1.0
f g 1.0
f e 3.0
e g 2.0
!
Intermediate way:
A* result: abcfe
```

```
Введите количество вершин снимаемых с очереди за раз: 3
c d
c j 3.3
c b 4.0
c i 3.4
j o 10.0
j b 4.1
i f 2.7
i k 10.4
k l 1.0
k f 4.3
b f 3.0
o b 3.1
b e 3.0
o e 4.3
e h 3.2
e f 3.2
h f 4.5
h d 8.5
f d 1.4
l d 3.2
!
Intermediate way:
A* result: cifd
```

```
Введите количество вершин снимаемых с очереди за раз: 3
a e
a b 3.0
a d 3.0
d c 1.0
b c 1.0
c f 1.0
f g 1.0
f e 3.0
e g 2.0
!
Intermediate way:
A* result: adcfe
```

Приложение А.

Исходный код программы.

Жадный алгоритм.

```
#include <iostream>

#include <map>
#include <vector>
#include <stack>

using namespace std;


map<char, vector<pair<char, double>>> Graph;
map<char, bool> Visited;


char from, to;


void readGraph()
{
    char start, end;
    double distance;

    std::cin >> from >> to;

    while (cin >> start)
    {
        if (start == '!')
            break;
        cin >> end >> distance;

        Graph[start].push_back(make_pair(end,distance));
        Graph[end];
```

```
        Visited[start] = false;
        Visited[end] = false;
    }
}


void print(stack<char>& result)
{
    if (result.empty())
        return;

    char tmp = result.top();
    result.pop();
    print(result);
    cout << tmp;
}

void greedySearch()
{

    stack<char> way;
    stack<char> intermediateDataOutput;

    way.push(from);
    char currPeak = way.top();

    cout << "Intermediate way: \n";
    do
    {
        intermediateDataOutput = way;
        print(intermediateDataOutput);
        cout << "\n";
        bool anyWay = false;
        char nextPeak;
        double minDistance;


        if (Graph[currPeak].empty())
        {
            Visited[currPeak] = true;

            way.pop();
            currPeak = way.top();
```

```
            continue;
        }

        for (int i = 0; i < Graph[currPeak].size(); i++)
        {
            if (!Visited[Graph[currPeak][i].first])
            {
                anyWay = true;
                nextPeak = Graph[currPeak][i].first;
                minDistance = Graph[currPeak][i].second;
                break;
            }
        }

        if (!anyWay)
        {
            Visited[currPeak] = true;

            way.pop();
            currPeak = way.top();
            continue;
        }

        for (int i = 0; i < Graph[currPeak].size(); i++)
        {
            if (!Visited[Graph[currPeak][i].first] && minDistance >
Graph[currPeak][i].second)
            {
                nextPeak = Graph[currPeak][i].first;
                minDistance = Graph[currPeak][i].second;
            }
        }

        way.push(nextPeak);
        currPeak = way.top();

    }while (currPeak != to);

    cout << "Greedy search: ";
    print(way);

}
```

```
int main() {
    readGraph();
    greedySearch();
    return 0;
}
```

A*.

```
#include <iostream>
#include <map>
#include <utility>
#include <vector>
#include <queue>


using namespace std;

struct Cell{
    char name;
    char parent;

    double rough;
};

struct Cmp{
    bool operator()(const Cell& a, const Cell& b)
    {
        if (a.rough == b.rough)
        {
            return a.name < b.name;
        }
        return a.rough > b.rough;
    }

};

map<char, vector<pair<char, double >>> graph;
map<char, bool> closeList;
map<char, pair<char, double>> realWay;
char from, to;
int n;
```

```
void readGraph(){
    char start, finish;
    double way;

    cout << "Введите количество вершин снимаемых с очереди за раз: ";
    cin >> n;

    std::cin >> from >> to;

    while (cin >> start)
    {
        if (start == '!')
            break;
        cin >> finish >> way;
        graph[start].push_back(make_pair(finish, way));
    }
}

void printWay(char a){
    if (a == from)
    {
        cout << a;
        return;
    }
    printWay(realWay[a].first);
    cout << a;
}


void aStar()
{
    vector <Cell> cells;
    priority_queue <Cell, vector<Cell>, Cmp> openList;

    openList.push(Cell{from, '\0', 0 + double(to - from)});

    cout << "Intermediate way:\n";

    while(!openList.empty()){
        /*
        for(auto& it : realWay)
        {
            cout << "mW[" << it.first << "]: ";
```

```
            printWay(it.first);
            cout << ' ';
        }

        cout << '\n';
         */
        if (openList.top().name == to)
        {
            cout << "A* result: ";
            printWay(to);
            return;
        }

        for (int i = 0 ; i < n && !openList.empty(); i++){
            Cell tmp = openList.top();

            if (tmp.name == to) continue;

            cells.push_back(tmp);
            openList.pop();
        }

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

            Cell currCell = cells[i];
            closeList[currCell.name] = true;

            for (int j = 0; j < graph[currCell.name].size(); j++) {
                pair<char, double> newCell = graph[currCell.name][j];


                if (closeList[newCell.first])
                    continue;

                if (realWay[newCell.first].second == 0 ||
realWay[newCell.first].second > realWay[currCell.name].second + newCell.second)
                {
                    realWay[newCell.first].second =
realWay[currCell.name].second + newCell.second;
                    realWay[newCell.first].first = currCell.name;
                }
```

```
                openList.push(Cell{newCell.first, currCell.name,
realWay[newCell.first].second + double(to - newCell.first)});
            }
        }
        cells.clear();
    }
}


int main(){
    readGraph();

    aStar();

    return 0;
}
```