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. 8383 _____ Бабенко Н.С.

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.

Входными параметрами для нахождения кратчайшего пути являются:

- **граф** (набор вершин и ребер(**пути**));
- **начальная вершина**;
- **конечная вершина**.

В стек помещается название **начальной вершины** и она же присваивается переменной, используемой для хранения значения текущего значения.

Далее в цикле с постусловием (пока стек не будет пуст) по вершинам **графа** происходит последовательная выборка элементов. Условием досрочной остановки является момент, когда текущей вершиной становится **конечная вершина.**

Для переменной производятся проверка наличия доступных **путей**:

При отсутствии **путей**, вершина помечается как посещенная, она удаляется из стека, новой текущей вершиной становится верхний элемент стека,

Если доступные пути есть то среди них выбирается самый дешевый, он кладется на стек и делается текущей вершиной.

Сложность алгоритма.

По скорости.

Граф имеет **n** вершин, **m** ребер.

Жадный алгоритм является модификацией поиска в глубину, только он переходит не в первую попавшуюся вершина, а для каждой вершины просматривает все ребра и выбирает из них минимальную. В худшем случае придется обойти весь граф.

Сложность получается O(n * m+ n)

По памяти.

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

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

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. Выходные значение граф записанный в map<char, vector<pair<char, double >>>

2. void greedySearch()

Функция, которая реализует жадный поиск. Выходное значение stack<char> way на котором хранится весь путь от начальной вершины до цели.

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

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

Описание алгоритма A*.

Входными параметрами для нахождения кратчайшего пути являются:

- **граф** (набор вершин и ребер(**пути**));

- **начальная вершина**;
- **конечная вершина**;
- **количество элементов извлекаемых из очереди с приоритетом за раз.(N)**

В очередь с приоритетом помещается название начальной вершины, ее приоритет (сумма реального пути и предполагаемого), и название родительской вершины(в данном случае пустое значение).

Далее по циклу производиться выборка N элементов очереди в убывания приоритета. Они помещаются в массив.

Для каждого элемента массива в цикле рассматриваются все смежные вершины, перед помещением в очередь они проверяются на:

- Не отмечена ли смежная вершина как уже посещенная, если отмечена, то вершина игнорируется.
- Есть ли вершины нет в очереди, то добавляем в очередь название анализируемой вершины, приоритет (сумма реального пути и эвристической оценки), родительскую вершину.
- Если вершина уже есть в очереди, то сравниваем уже посчитанный реальный путь, с вновь рассчитанным путем. Если уже рассчитанное значение больше нового, то меняем его и меняем родительскую вершину на текущую, если меньше, то ничего не делаем

После того как все смежные вершины были рассмотрены, текущая вершина помечается как просмотренная.

Условие окончания выборки является пустая очередь. Если с очереди будет снята конечная вершина, цикл закончится заранее.

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. Выходнон значение - граф записанный в map<char, vector<pair<char, double >>>

2. void aStar()

Функция, которая реализует жадный поиск. Возвращает map<char, pair<char, double>> в котором хранится длинна пути до вершины и родительская вершина

3. void printWay(char a)

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

Сложность алгоритма.

По скорости.

Граф имеет **n** вершин, **m** ребер.

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 лучшем случае, когда эвристика **допустима** (для любой вершины и ее потомка разность эвристической функции не превышает фактического веса ребра) и **монотонна** (для любой вершины эвристическая оценка меньше или равно минимальному пути до цели), то сложность получается O(n+m), так как на каждом шаге мы будем приближаться к цели.

В худшем случае, когда эвристика нам не помогает(эвристическая функция подобрана плохо) придется просмотреть все пути. В таком случае алгоритм просто превратиться в алгоритм Дейкстры и сложность возрастет до O(n^2).

Лучший случай: O(n+m),

Худший случай: O(n^2)

По памяти.

В лучшем случае O (n+m)

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(b^m), где b— среднее число ветвлений.

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
ach
ac
aci
Greedy search: acie
```

```
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
```

```
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
Greedy search: abcde
```

```
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 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
```

Тестирование A*.

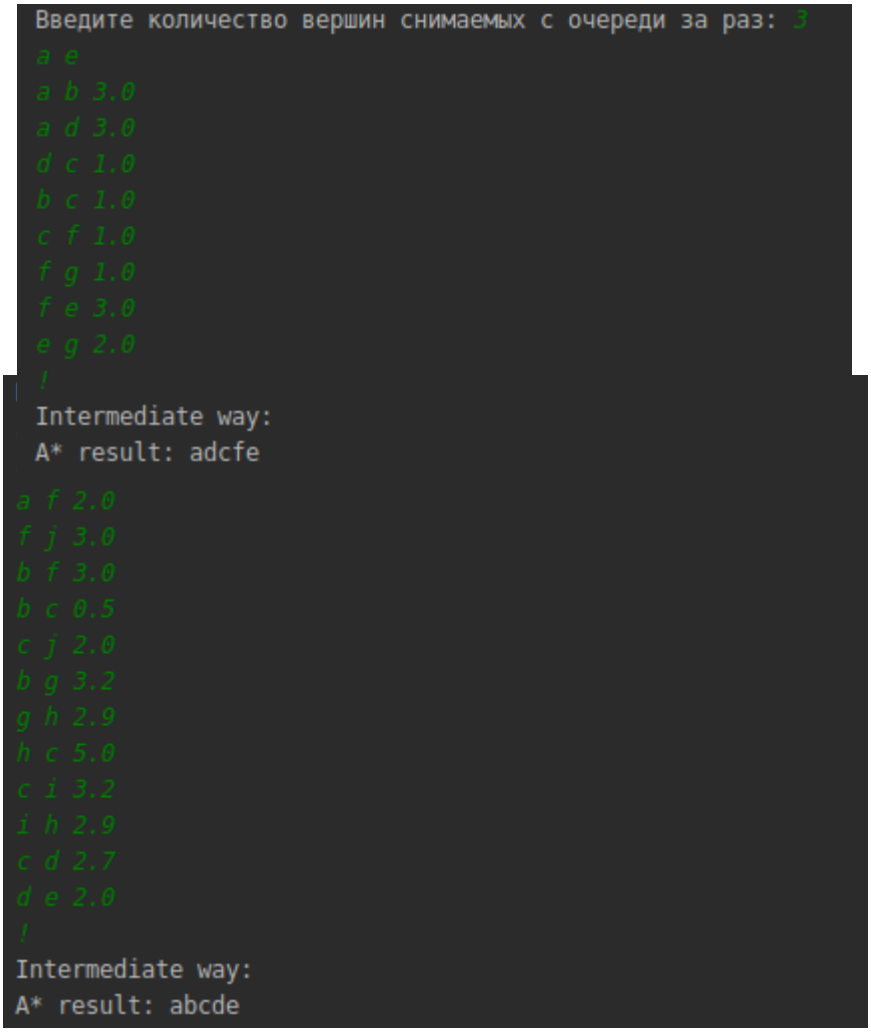
```
Введите количество вершин снимаемых с очереди за раз: 2
a e
a b 1.0
a d 4.0
d c 1.0
b c 1.0
c f 1.0
f g 1.0
f e 3.0
e g 2.0
!
Intermediate way:
A* result: abcfe
```

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

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: 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: 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
```



Вывод.

В ходе выполнения лабораторной работы были изучены алгоритмы поиска пути в графе путем написания программ, реализующих жадные алгоритм поиска и A*.

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; // здесь хранится значение по ключу мы получаем доступ ко всем вершинам, в которые можем попасть из вершины - ключа
map<char, bool> visited; // список всех уже посещенных вершин


char from, to; // начальная и конечная вершина


void readGraph() // функция, которая считывает граф и помечает все вершины, как не посещенные
{
    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) // рекурсивная функция, которая раскручивает стек, для получения пути от начально вершины в конечную
{
    if (result.empty())
        return;

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

void greedySearch() // функция, которая реализует жадный поиск
{

    stack<char> way; // стек на котором будет хранится путь до текущей вершины
    stack<char> intermediateDataOutput; // стек для промежуточных данных

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

    //cout << "Intermediate way: \n";
    do // цикл, который работает пока на верху стека не окахется конечная вершина или не будет обойден весь граф
    {
        //intermediateDataOutput = way;
        //print(intermediateDataOutput);
        //cout << "\n";
        bool anyWay = false; // есть ли из текущей вершины, пути в другие еще не просмотренные вершины

        char nextPeak;
        double minDistance;
```

```
        if (graph[currPeak].empty()) // проверка на то, есть ли  пути вообще, если путей нет вершина помечается как посещенная
        {
            visited[currPeak] = true;

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

        for (int i = 0; i < graph[currPeak].size(); i++) //проверка на, то есть ли еще не посещенные вершины
        {
            if (!visited[graph[currPeak][i].first])
            {
                anyWay = true;
                nextPeak = graph[currPeak][i].first;
                minDistance = graph[currPeak][i].second;
                break;
            }
        }

        if (!anyWay) // если все вершины уже просмотренные, то вершина помечается как посещенная
        {
            visited[currPeak] = true;

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

        for (int i = 0; i < graph[currPeak].size(); i++) // поиск самого маленького ребра
        {
            if (!visited[graph[currPeak][i].first] && minDistance > graph[currPeak][i].second)
            {
                nextPeak = graph[currPeak][i].first;
                minDistance = graph[currPeak][i].second;
            }
        }

        way.push(nextPeak); //переходим в вершину путь до которой был самый короткий
        currPeak = way.top();

    }while (currPeak != to);
```

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

}


int main() {

    readGraph();

    greedySearch();

    return 0;
}
```

Приложение Б.

Исходный код программы.

A*.

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


using namespace std;

struct Cell{ // структура для хранения названия вершины, ее родителя и пути до нее
    char name;
    char parent;

    double rough;
};

struct Cmp{ // компоратор для очереди с приорететом
    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; // граф
map<char, bool> closeList; // уже просмотреннный вершины
map<char, pair<char, double> > realWay; // кратчайщие пути до вершин
char from, to; // начальная и конечная вершина
int n; // количество вершин снимаемых за раз



void readGraph(){ // функция, которая считывает граф
    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){ // функция которая востанавливает путь от конечной вершины до начальной,
    if (a == from)     // так как в кратчайщих путях мы так же храним из какой вершины мы в нее
попали, то мы можем проследить весь путь от конца до началаа е
    {
        cout << a;
        return;
    }
    printWay(realWay[a].first);
    cout << a;
}



void aStar()
{
```

```
    vector <Cell> cells; // массив, в котором будут хранится n снятых вершин
    priority_queue <Cell, vector<Cell>, Cmp> openList; // открытый список, куда кладудтся все вершина, котоыре нужно рассмотреть, в верху очереди находится вершины с самым маленьким приоерететом

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

    //cout << "Intermediate way:\n";

    while(!openList.empty()){ // цикл работает пока очередь не опустеет или с нее не будет снята конечная вершина
        /*
        for(auto& it : realWay)
        {
            cout << "minWay[" << it.first << "]: ";
            printWay(it.first);
            cout << ' ';
        }

        cout << '\n';
         */
        if (openList.top().name == to) // если была снята конечная вершина, то алгоритм останавливается
        {
            cout << "A* result: ";
            printWay(to);
            cout << endl;
            return;
        }

        for (int i = 0 ; i < n && !openList.empty(); i++){ // вершины снимаются пока мы не снимим n вершин или очередь не опустеет
            Cell tmp = openList.top();                  // или если была встреченна конечная вершина, и она была не первая в очереди

            if (tmp.name == to) continue;

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

        for(int i = 0; i < cells.size(); i++) { // рассмотрение всех снятых верщин

            Cell currCell = cells[i];
```

```
            closeList[currCell.name] = true;

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

                if (closeList[newCell.first]) // если вершина уже была рассмотренна, то мы ее не
рассматриваем
                    continue;

                if (realWay[newCell.first].second == 0 || realWay[newCell.first].second >
realWay[currCell.name].second + newCell.second) // если вершина еще не была рассмотренна лии она
все еще находится в открытом списке
                {                                                                                    // то мы
проверяем короче ли новый найденный путь или нет, если да
                    realWay[newCell.first].second = realWay[currCell.name].second + newCell.second;
// то мы добавляем в очередь эту вершину, и запоминаем новый кратчайщий путь до нее,
                    realWay[newCell.first].first = currCell.name;
// если не была рассмотренна, то просто добавляем
                    openList.push({newCell.first, currCell.name, realWay[newCell.first].second + double(to -
newCell.first)});
                }


            }
        }
        cells.clear();
    }
}


int main(){
    readGraph();

    aStar();

    return 0;
}
```