МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студентка гр. 9382 _____ Сорочина М.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить принцип действия жадного алгоритма и алгоритма A*. А также на основе данных алгоритмов реализовать программы, выполняющие поиск пути в ориентированном графе.

Задание.

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

Вариант.

Вар. 4. Модификация A* с двумя финишами (требуется найти путь до любого из двух).

Описание функций и структур данных.

Задание 1. Жадный алгоритм:

1)
```
struct Edge
{
char v1;
char v2;
double len;
Edge(char a, char b, double c) : v1(a), v2(b), len(c) {}
};
```

Структура для хранения ребер графа. `v1` - имя вершины, из которой выходит ребро, `v2` - имя вершины, в которую входит ребро, `len` - длина ребра.

2) `void input(std::vector<Edge> &edges)`

Функция для записи введенных данных. `edges` - вектор, хранящий все ребра графа.

3) `vectPair adjacentVer(char ver, std::vector<Edge> edges)`

(vectPair название для std::vector<std::pair<char, double>>). Функция поиска смежных с передаваемой вершин. ver - имя вершины, для которой нужно найти смежные, edges - вектор, хранящий все ребра графа. Возвращает вектор пар, где каждая пара состоит из имени смежной вершины и длины ребра до этой вершины.

4)void findWay(char from, char to, std::vector<Edge> &edges, bool &flag, std::vector<char> &answer)

Функция поиска пути при помощи жадного алгоритма. from - имя вершины, с которой начинаем, to - имя вершины, до которой ищем путь, edges - вектор, хранящий все ребра графа, flag - переменная равна 1, если ответ нашелся, и 0, если ответа нет, answer - вектор для записи ответа.

Задание 2. Алгоритм A*:

1)
```
struct Vertex
{
    char name;
    std::map<char, double> adjacentV;
    bool seen;
    double g;
    double f;
    Vertex *prev;
};
```

Структура для хранения информации о вершине. name - имя вершины, adjacentV - хранит данные о смежных вершинах (имя вершины - длина ребра между ними), seen - для отметки просмотренных вершин, g - дистанция от стартовой вершины, f - сумма g и оценки расстояния при помощи эвристической функции, *prev - указатель на предыдущую вершину, используется при восстановлении найденного пути.

2) void input(char &start, char &end1, char &end2, std::vector<Vertex *> &vertices)

Функция для записи введенных данных, а также заполнения вектора указателей на вершины. start - начальная вершина, end1 и end2 - конечные

вершины (если конечная вершина одна, то вторая приравнивается первой), vertices - вектор, хранящий указатели на все вершины.

3) `double h(char start, char end)`

Эвристическая функция оценки расстояния между вершинами с именами start и end.

4)`void fAdj(char end, std::vector<Vertex *> &open, Vertex *curr, std::vector<Vertex *> vertices)`

Функция обработки смежных с текущей вершин. end - конечная вершина, open - вектор, хранящий указатели на “открытые” вершины, *curr - указатель на текущую (рассматриваемую) вершину, vertices - вектор, хранящий указатели на все вершины.

5)`Vertex *retVer(char ver, std::vector<Vertex *> vertices)`

Функция возвращает указатель на вершину с переданным именем. ver - имя вершины, указатель на которую возвращает функция, vertices - вектор, хранящий указатели на все вершины.

6) `bool isIn(Vertex *vert, std::vector<Vertex *> &vertices)`

Функция проверяет наличие вершины в векторе. *vert - указатель на вершину, которую нужно проверить, vertices - вектор, хранящий указатели на все вершины.

7)`void printAns(Vertex *ans)`

Функция, восстанавливающая путь до вершины. *ans - указатель на конечную вершину.

8)`bool cmp(Vertex *a, Vertex *b)`

Вспомогательная функция для сортировки вектора “открытых” вершин. *a и *b - указатели на вершины, которые нужно сравнить.

9)`bool aStar(char start, char end, std::vector<Vertex *> &vertices)`

Функция поиска пути. `start` - начальная вершина, `end` - конечная вершина, `vertices` - вектор, хранящий указатели на все вершины.

Описание алгоритма.

Задание 1. Жадный алгоритм:

Начиная со стартовой вершины и пока текущая вершина не равна конечной, рассматриваются все смежные с актуальной вершины, записываются и сортируются в порядке возрастания длины ребра до текущей. После этого выбирается первая (с наименьшей длиной ребра) вершина из смежных, если такие есть. Все то же самое делается с ней. Если выбор кратчайших ребер привел в тупик, то происходит возврат на 1 вершину назад, там рассматривается другое смежное ребро.

Задание 2. Алгоритм A*:

В A* похожий алгоритм, но при выборе вершины учитывается не только расстояние, но также число, даваемое эвристической функцией оценки расстояния. В данном случае в качестве эвристической функции было использовано расстояние в алфавите между именами вершин.

Относительно индивидуализации использовалась схема: если конечная вершина одна, то путь ищется до нее, если таких вершин две, то случайным образом выбирается до какой происходит поиск пути.

Оценка сложности.

Обозначения: E - количество ребер, V - количество вершин.

Задание 1. Жадный алгоритм:

По памяти O(V).

По времени O(V+E), так как в худшем случае придется обойти все вершины, каждая из которых будет соединена со всеми смежными.

Задание 2. Алгоритм A*:

По памяти:

1) в хорошем случае, когда будет хранится путь от начальной до конечной вершины - O(E(E+V))
2) в плохом случае, когда все пути будут хранится, сложность будет экспоненциальной.

По времени:

1) в хорошем случае, когда эвристическая функция делает выбор в правильном направлении - O(E+V)
2) В плохом случае число вершин, исследуемых алгоритмом, растет 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 $h^*(x)$ - оптимальная эвристика, т.е. точная оценка расстояния от вершины x к цели. Иначе говоря, ошибка $h^*(x)$ не должна расти быстрее, чем логарифм от оптимальной эвристики.

В плохом случае, когда придется обойти все пути - O(E^2).

Тестирование.

№ теста	Ввод	Вывод Жадный алгоритм	Вывод Алгоритм A*
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	ade
2	a g	abcdefg	ag

	a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 a g 9.0 e f 7.0 f g 3.0		
3	a x a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 a g 9.0 e f 7.0 f g 3.0	No way from a to x.	Такого пути нет
4	a a a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 a g 9.0 e f 7.0 f g 3.0	a	a
5	a e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	abge	ac

6	a e d a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0		abd

Выводы.

В ходе выполнения работы была написана программа, реализующая жадный алгоритм поиска пути в графе, а также программа, реализующая поиск пути в графе при помощи алгоритма A*.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ greedy.cpp.

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

using vectPair = std::vector<std::pair<char, double>>;

#define COMMENTS

struct Edge
{
      char v1; //структура для хранения имен двух вершин, соединненых
ребром, и длины ребра
    char v2;
    double len;
    Edge(char a, char b, double c) : v1(a), v2(b), len(c) {}
};

//функции
void                    input(std::vector<Edge>                    &);
//ok
void findWay(char from, char to, std::vector<Edge> &edges, bool &flag,
std::vector<char> &answer); //функция поиска пути в графе
vectPair     adjacentVer(char     ver,     std::vector<Edge>     edges);
//функция поиска смежных с данной вершин
//ok

int main()
{
    char fromEdge, toEdge;
    std::vector<Edge> edges;
    std::string in;
    getline(std::cin, in);
    fromEdge = in[0];
    toEdge = in[2];
    input(edges);
    std::vector<char> answer;
    bool flag = false;
    findWay(fromEdge, toEdge, edges, flag, answer);
    if (flag)
    {
#ifdef COMMENTS
        std::cout << "Ответ:\n";
#endif
        for (auto i : answer)
        {
            std::cout << i;
        }
```

```
    }
    else
    {
         std::cout << "No way from " << fromEdge << " to " << toEdge <<
"."; 
    }
    std::cout << '\n';
    return 0;
}

void input(std::vector<Edge> &edges) //ok
{
    std::string in;
    while (getline(std::cin, in))
    {
        if (in.empty())
            break;
        edges.push_back({in[0], in[2], std::stod(in.substr(3))});
    }
}

vectPair adjacentVer(char ver, std::vector<Edge> edges)
//находит смежные и сортирует их по длине ребра в порядке возрастания
{
    vectPair adjVer;
    for (auto i : edges)
    {
        if (i.v1 == ver)
        {
            adjVer.push_back({i.v2, i.len});
        }
    }
    std::sort(adjVer.begin(), adjVer.end(), [](std::pair<char, double> a,
std::pair<char, double> b) {
        return a.second < b.second;
    });
    return adjVer;
}

void findWay(char from, char to, std::vector<Edge> &edges, bool &flag,
std::vector<char> &answer)
//функция поиска пути, если таковой имеется
{
    std::map<char, vectPair> dict;
    std::vector<char> vertices;
    vectPair vect;
    int ansSize = 0;
    char ver = from, ver2;
    while (ver != to)
    {
        if (dict.count(ver) == 0)
        {
            vertices.push_back(ver);
```

```
            dict[ver] = adjacentVer(ver, edges);
        }
#ifdef COMMENTS
        std::cout << "Смежные с [" << ver << "]\n";
        for (auto i : dict[ver])
        {
            std::cout << '[' << i.first << ',' << i.second << ']';
        }
        std::cout << '\n';
#endif
        if (dict[ver].size() > 0)
        {

             if (ansSize > 0 && ver != answer[ansSize - 1] || ansSize ==
0)
            {
/*if (find(answer.begin(), answer.end(), ver) != answer.end())
                {
                    answer.clear();
                    ansSize = 0;
                }*/
#ifdef COMMENTS
                    std::cout << "Добавили в ответ вершину [" << ver <<
"]\n";
#endif
                answer.push_back(ver);
                ansSize++;
            }
            ver2 = ver;
            ver = dict[ver][0].first;
#ifdef COMMENTS
            std::cout << "------------------------\n";
               std::cout << "Новая рассматриваемая вершина [" << ver <<
"]\n";
#endif
#ifdef COMMENTS
            std::cout << "В данный момент в ответ записано:\n";
            if (ansSize > 0)
            {
                for (auto i : answer)
                {
                    std::cout << i;
                }
            }
            else
            {
                std::cout << "nothing";
            }
            std::cout << "\n";
#endif
            dict[ver2].erase(dict[ver2].begin());
        }
        else
```

```
        {
            if (ver == from)
            {
#ifdef COMMENTS
                std::cout << "Дошли до начальной вершины => пути нет\n";
#endif
                break;
            }
            ver = answer[ansSize - 1];
#ifdef COMMENTS
            std::cout << "Смежных нет, возвращаемся на шаг назад\n";
            std::cout << "------------------------\n";
               std::cout << "Новая рассматриваемая вершина [" << ver <<
"]\n";
#endif
            answer.pop_back();
            ansSize--;
#ifdef COMMENTS
            std::cout << "В данный момент в ответ записано:\n";
            if (ansSize > 0)
            {
                for (auto i : answer)
                {
                    std::cout << i;
                }
            }
            else
            {
                std::cout << "nothing";
            }
            std::cout << "\n";
#endif
        }
    }
    answer.push_back(ver);
    if (ver == to)
    {
#ifdef COMMENTS
        std::cout << "Добавили в ответ вершину [" << ver << "]\n";
#endif
        flag = 1;
        return;
    }
}
```

ПРИЛОЖЕНИЕ Б.

ИСХОДНЫЙ КОД ПРОГРАММЫ astar.cpp.

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <algorithm> // std::sort

//#define COMMENTS

struct Vertex
{
    char name;
    std::map<char, double> adjacentV;
    bool seen;
    double g;
    double f;
    Vertex *prev;
};

//функции
void input(char &start, char &end1, char &end2, std::vector<Vertex *> &);
//функция считывающая ввод
double h(char start, char end1, char end2);
//эвристическая фукнция оценки расстояния между вершинами
void fAdj(char end, std::vector<Vertex *> &open, Vertex *curr,
std::vector<Vertex *> vertices);
//функция поиска вершины с мин. f
Vertex *retVer(char ver, std::vector<Vertex *> vertices);
//функция возвращает указатель на вершину с именем ver
bool isIn(Vertex *vert, std::vector<Vertex *> &vertices);
//проверяет есть ли вершина в списке вершин
void printAns(Vertex *ans);
//функция вывода ответа
bool aStar(char start, char end, std::vector<Vertex *> &vertices);
//функция поиска пути при помощи алгоритма A* с 1 end

int main()
{
      char fromVer, toVer1, toVer2;    //для записи вершины, с которой
начинаем, и вершин(ы), до которой(ых) нужно искать путь
    std::vector<Vertex *> vertices; //для записи вершин
    input(fromVer, toVer1, toVer2, vertices);
    if (fromVer == toVer1 || fromVer == toVer2)
    {
        std::cout << fromVer << '\n';
        return 0;
    }
     if (retVer(toVer1, vertices) == nullptr && retVer(toVer2, vertices)
== nullptr)
```

```
    {
        std::cout << "Такого пути нет\n";
        return 0;
    }
    bool ans = 0;
    char name;
    char random = rand() % 2 + 1;

    if (toVer1 == toVer2 || random == 1)
    {
#ifdef COMMENTS
        std::cout << "Будем искать путь до вершины [" << toVer1 << "]\n";
#endif
        ans = aStar(fromVer, toVer1, vertices);
        name = toVer1;
    }
    else
    {
#ifdef COMMENTS
        std::cout << "Будем искать путь до вершины [" << toVer2 << "]\n";
#endif
        ans = aStar(fromVer, toVer2, vertices);
        name = toVer2;
    }
    if (!ans)
    {
        std::cout << "Такого пути нет\n";
    }
    else
    {
#ifdef COMMENTS
        std::cout << "Ответ:\n";
#endif
        printAns(retVer(name, vertices));
        std::cout << '\n';
    }
    return 0;
}

void input(char &start, char &end1, char &end2, std::vector<Vertex *>
&vertices)
//функция считывающая ввод
{
    std::string in;
    getline(std::cin, in);
    //std::cout << "\t\tin [" << in << "]\n";
    start = in[0];
    end1 = in[2];
    if (in[4] >= 'a' && in[4] <= 'z')
    {
        end2 = in[4];
    }
    else
```

```
{
    end2 = end1;
}
char ver1, ver2;
double len;
std::map<char, double> adjV;
bool flag = 0;
while (getline(std::cin, in))
{
    if (in.empty())
        break;
    ver1 = in[0];
    ver2 = in[2];
    len = std::stod(in.substr(3));
    flag = 0;
    for (auto &i : vertices)
    {
        if (i->name == ver1)
        {
            flag = 1;
            i->adjacentV[ver2] = len;
        }
    }
    if (!flag)
    {
        adjV[ver2] = len;
        auto *v = new Vertex;
        v->name = ver1;
        v->adjacentV[ver2] = len;
        v->seen = 0;
        v->prev = nullptr;
        v->f = 100000;
        v->g = 0;
        vertices.push_back(v);
        adjV.clear();
    }
    flag = 0;
    for (auto &i : vertices)
    {
        if (i->name == ver2)
        {
            flag = 1;
        }
    }
    if (!flag)
    {
        adjV[ver2] = len;
        auto *v = new Vertex;
        v->name = ver2;
        v->seen = 0;
        v->f = 100000;
        v->g = 0;
        vertices.push_back(v);
```

```
            adjV.clear();
        }
    }
#ifdef COMMENTS
     std::cout << "От вершины [" << start << "] до вершины [" << end1 <<
"] или [" << end2 << "]\n";
    std::cout << "Vertices:\n";
    for (auto &i : vertices)
    {
        std::cout << i->name << "----------------------";
        for (auto &j : i->adjacentV)
        {
            std::cout << '[' << j.first << ',' << j.second << "] ";
        }
        std::cout << std::endl;
    }
#endif
}

double h(char start, char end)
//эвристическая фукнция оценки расстояния между вершинами
{
    return abs(end - start);
}

void fAdj(char end, std::vector<Vertex *> &open, Vertex *curr,
std::vector<Vertex *> vertices)
// функция рассматривает смежные с curr вершины
//если смежной вершины нет в open, то добавляет
//меняет f, если она стала меньше
{
    Vertex *neighbour;
    double newF = 0, newG = 0;
    if (!curr->adjacentV.empty())
    {
        for (auto ver : curr->adjacentV)
        {
#ifdef COMMENTS
            std::cout << '\n';
#endif
            neighbour = retVer(ver.first, vertices);
            if (neighbour->seen != 1)
            {
                newG = curr->g + ver.second;
                newF = curr->g + ver.second + h(ver.first, end);
                if (!isIn(neighbour, open) || newF < neighbour->f)
                {
                    neighbour->prev = curr;
                    neighbour->g = newG;
                    neighbour->f = newF;
                }
                if (newF < neighbour->f)
                {
```

```
#ifdef COMMENTS
                                    std::cout << "Новая f меньше старой,
перезаписываем\n";
#endif
                }
                if (!isIn(neighbour, open))
                {
#ifdef COMMENTS
                    std::cout << "Добавляем вершину [" << ver.first << "]
в open\n";
#endif
                    open.push_back(neighbour);
                }

                newF = curr->f + ver.second + h(neighbour->name, end);
#ifdef COMMENTS
                  std::cout << "Новая f для [" << ver.first << "] равна "
<< newF << "\n";
#endif
                if (newF < neighbour->f)
                {
#ifdef COMMENTS
                        std::cout << "Новая f (" << newF << ") для [" <<
ver.first << "] меньше старой (" << neighbour->f << "). Обновление.\n";
#endif

                    neighbour->g += ver.second;
                    neighbour->f = neighbour->g + h(ver.first, end);
                    neighbour->prev = curr;
                }
#ifdef COMMENTS
                else
                {
                    std::cout << "Новая f не лучше старой\n";
                }
#endif
            }
        }
    }
}

Vertex *retVer(char ver, std::vector<Vertex *> vertices)
{
    for (auto &i : vertices)
    {
        if (i->name == ver)
        {
            return i;
        }
    }
    return nullptr;
}
```

```
bool isIn(Vertex *vert, std::vector<Vertex *> &vertices)
{
    for (auto &i : vertices)
    {
        if (i->name == vert->name)
        {
            return true;
        }
    }
    return 0;
}

void printAns(Vertex *ans)
{
    if (ans->prev == nullptr)
    {
        std::cout << ans->name;
        return;
    }
    printAns(ans->prev);
    std::cout << ans->name;
}

bool cmp(Vertex *a, Vertex *b)
// для сортировки open
{
    if (a->f == b->f)
    {
        return a->name > b->name;
    }
    return (a->f < b->f);
}

bool aStar(char start, char end, std::vector<Vertex *> &vertices)
{
    std::vector<Vertex *> open;
    Vertex *curr;
    open.push_back(retVer(start, vertices));
    open[0]->g = 0;
    open[0]->f = h(start, end);
    while (!open.empty())
    {
        curr = open[0];
#ifdef COMMENTS
        std::cout << "Выбрали вершину [" << curr->name << "]\n";
#endif
        if (curr->name == end)
        {
#ifdef COMMENTS
            std::cout << "Найден путь\n";
#endif
            return 1;
        }
```

```
        fAdj(end, open, curr, vertices);
#ifdef COMMENTS
        std::cout << "Open:\n";
        for (auto &i : open)
        {
            std::cout << i->name << " ";
        }
        std::cout << '\n';
#endif
        curr->seen = 1;
        open.erase(open.begin());
        std::sort(open.begin(), open.end(), cmp);
#ifdef COMMENTS
        std::cout << "Open after delete elem and sort:\n";
        for (auto &i : open)
        {
            std::cout << "[" << i->name << ", " << i->f << "] ";
        }
        std::cout << "\n------------------\n";
#endif
    }
    return 0;
}
```