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: Жадный алгоритм и A*

Студент гр. 7304 _____ Овчинников Н.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2019

Цель работы

Изучить жадный алгоритм и алгоритм A*, а так же реализовать данный алгоритмы на языке программирования C++.

Задание

1. Разработать 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.
2. Разработать 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 таблицу ASCII.

Ход работы

1. Была реализована программа, которая ищет путь в ориентированном взвешенном графе с помощью жадного алгоритма. В начале программы производится считывание входных данных и инициализация вершин графа (a-z). Каждая вершина графа содержит все ребра от себя до других вершин. После инициализации графа сортируются ребра у всех вершин по возрастанию веса. Затем в цикле от начальной вершины идём по самому дешевому ребру, пока не дойдём до конечной.
2. Была реализована программа, которая ищет кратчайший путь в ориентированном взвешенном графе с помощью алгоритма A*. В начале программы производится считывание входных данных и инициализация вершин графа (a-z). Каждая вершина графа содержит все ребра от себя до других вершин. Для работы алгоритма с помощью вектора в главной функции были реализованы множества просмотренных вершин и вершин, которые требуется рассмотреть, а также вспомогательные функции: эвристическая, которая вычисляет близость символов в таблице ASCII; функция, которая определяет принадлежность вершины множеству; функция поиска в множестве минимальной по значению эвристической функции вершины; функция вывода результата.

Вывод

В ходе выполнения лабораторной работы был изучен и реализован жадный алгоритм поиска пути в графе. Жадный алгоритм ищет путь из данной стартовой вершины в заданную конечную вершину, при этом жадный алгоритм не гарантирует нахождение кратчайшего пути в графе. Жадность алгоритма заключается в том, что если на каждом шаге из вариатива вершин выбрать минимальную, то удастся дойти до конечной вершины по короткому пути.

Также в ходе работы был реализован алгоритм A*, который находит кратчайший путь в графе из заданной стартовой вершины в конечную. Алгоритм A* является модификацией алгоритма Дейкстры. Модификация заключается в том, что вводится эвристическая функция, и в результате путь до вершины вычисляется по формуле $f(x)=g(x)+h(x)$, где – путь до текущей вершины, а $h(x)$ – эвристическая функция, которая может быть различной для разных задач.

Приложение: исходный код программы

Piaa_lr2_1.cpp:

```
#include <iostream>
#include <stdio.h>
#include <cstdlib>
#include <limits>
#include <ctype.h>
#include <vector>
#include <math.h>

#define EPS 0.000001

using namespace std;

class Edge;
bool comp(Edge &a, Edge &b);

class Path
{
public:
    char from;
    char to;
    char *path;
};

class Edge
{
public:
    double weight;
    char to;
};

class Vertices
{
public:
    char name;
    int visitedEdgesCounter = 0;
    vector<Edge> edges;
};

class Graph
{
public:
    vector<Vertices> vertices;
    int size;
    Path path;

    Graph(char from, char to)
    {
        path.from = from;
        path.to = to;
        size = 26;
        path.path = new char[size*2];
        char tmp = 'a';
        for(int i=0; i < size; i++, tmp++)
        {
            Vertices tmpVertex;
            tmpVertex.name = tmp;
            vertices.push_back(tmpVertex);
            path.path[0] = '\0';
        }
```

```
    }

    ~Graph()
    {
        delete path.path;
        path.path = NULL;
    }

    void greed()
    {
        double weight;
        char one, two = '\0';
        int index;
        while(1)
        {
            if(cin >> one && cin >> two && cin >> weight){
                Edge tmpEdge;
                tmpEdge.to = two;
                tmpEdge.weight = weight;
                index = static_cast<int>(one) - static_cast<int>('a');
                Vertices *tmpVertex = &(vertices.at(index));
                tmpVertex->edges.push_back(tmpEdge);
            }
            else
                break;
        }

        for(int i=0; i<size; i++)
        {
            Vertices *tmpVertex = &(vertices.at(i));
            tmpVertex->edges.shrink_to_fit();
            if(tmpVertex->edges.empty() == false)
                sort(tmpVertex->edges.begin(), tmpVertex->edges.end(), comp);
        }

        int k=0;
        one = path.from;
        for(; one != path.to; k++)
        {
            path.path[k] = one;
            index = static_cast<int>(one) - static_cast<int>('a');
            Vertices *tmpVertex = &(vertices.at(index));
            if(tmpVertex->edges.empty())
            {
                //cout << "Vector reber u vershini  " << index << " pust\n";
                path.path[k--] = '\0';
                index = static_cast<int>(path.path[k]) - static_cast<int>('a');
                tmpVertex = &(vertices.at(index));
            }
            Edge tmpEdge = tmpVertex->edges.at(tmpVertex-
>visitedEdgesCounter++);
            one = tmpEdge.to;
        }

        for(; k<size*2; k++)
            path.path[k] = '\0';
        cout << path.path << path.to << endl;
     }
};

bool comp(Edge &a, Edge &b)
{
    if(fabs(a.weight - b.weight) < EPS)
    {
```

```
        return a.to < b.to;
    }
    return a.weight < b.weight;
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    char from, to;
    scanf("%c %c", &from, &to);
    Graph go(from, to);
    go.greed();

    return a.exec();
}
```

Piaa_lr2_2.cpp:

```
#include <iostream>
#include <stdio.h>
#include <cstdlib>
#include <limits>
#include <ctype.h>
#include <vector>
#include <math.h>

#define EPS 0.000001

using namespace std;

class Edge
{
public:
    double weight;
    char to;
};

class Vertices
{
public:
    char name;
    double costFromBegin = numeric_limits<double>::max();   //стоимость пути от
начала до этой точки
    double heuristic = 0;   //значение эвристической функции
    vector<Edge> edges;
    Vertices *parent = nullptr;
};

class Graph
{
public:
    vector<Vertices> vertices;
    vector<char> resultPath;
    char from;
    char to;

    Graph(char from, char to)
    {
        this->from = from;
        this->to = to;
        char tmp = 'a';
        for(int i=0; i < 26; i++, tmp++)
```

```
        {
            Vertices tmpVertex;
            tmpVertex.name = tmp;
            vertices.push_back(tmpVertex);
        }

        double weight;
        char one, two = '\0';
        int index;
        while(1)
        {
            if(cin >> one && cin >> two && cin >> weight){
                Edge tmpEdge;
                tmpEdge.to = two;
                tmpEdge.weight = weight;
                index = static_cast<int>(one) - static_cast<int>('a');
                Vertices *tmpVertex = &(vertices.at(index));
                tmpVertex->edges.push_back(tmpEdge);
            }
            else
                break;
        }
    }

    double h(char from) //эвристическая функция
    {
        return abs(static_cast<int>(this->to) - static_cast<int>(from));
    }

    int min_Q(vector<Vertices> Q)
    {
        if(Q.size() == 1)
            return 0;
        unsigned int i = 1;
        int min = 0;
        Vertices tmp1;
        Vertices tmp2;
        while(i < Q.size())
        {
            tmp1 = Q.at(i-1);
            tmp2 = Q.at(i);
            if(tmp1.heuristic < tmp2.heuristic || fabs(tmp1.heuristic -
tmp2.heuristic)<EPS)
                min = i-1;
            else
                min = i;
            i++;
        }
        return min;
    }

    bool belongToVector(vector<Vertices> &A, Vertices &a) const
    {
        Vertices tmp;
        for(unsigned int i=0; i<A.size(); i++)
        {
            tmp = A.at(i);
            if(tmp.name == a.name && tmp.costFromBegin == a.costFromBegin &&
tmp.heuristic == a.heuristic)
                return true;
        }
        return false;
    }
```

```
    bool aStar(char from, char to)
    {
        Edge tmpEdge;
        Vertices *tmpVertex;
        vector<Vertices> Q; //мн-во вершин,которые требуется рассмотреть
        vector<Vertices> U; //мн-во рассмотренных вершин
        vector<Vertices>::iterator itQ;
        int tentativeScore = 0;
        int index = static_cast<int>(from) - static_cast<int>('a');
        tmpVertex = &(vertices.at(index));
        tmpVertex->costFromBegin = 0;
        tmpVertex->heuristic = tmpVertex->costFromBegin + h(from);
        Q.push_back(vertices.at(index));
        while(Q.size() != 0)
        {
            int min = min_Q(Q);
            Vertices current = Q.at(min);//*(itQ + min);
            if(current.name == to)
            {
                return true;
            }
            U.push_back(current);
            itQ = Q.begin();
            itQ = itQ + min;
            Q.erase(itQ);
            for(unsigned int i=0; i<current.edges.size(); i++)
            {
                tmpEdge = current.edges[i];
                tentativeScore = current.costFromBegin + tmpEdge.weight;
                index = static_cast<int>(tmpEdge.to) - static_cast<int>('a');
                tmpVertex = &(vertices[index]);
                if(belongToVector(U, *tmpVertex) && tentativeScore >= tmpVertex-
>costFromBegin)
                    continue;
                if(!belongToVector(U, *tmpVertex) || tentativeScore < tmpVertex-
>costFromBegin)
                {
                    tmpVertex->costFromBegin = tentativeScore;
                    tmpVertex->heuristic = tmpVertex->costFromBegin +
h(tmpVertex->name);
                    tmpVertex->parent =
&vertices.at(static_cast<int>(current.name) - static_cast<int>('a'));
                    if(!belongToVector(Q, *tmpVertex))
                        Q.push_back(*tmpVertex);
                }
            }
        }
        return false;
    }

    void print()
    {
        Vertices *tmpVertex = &(vertices.at(static_cast<int>(this->to) -
static_cast<int>('a')));
        vector<char>::iterator it = resultPath.begin();
        do
        {
            it = resultPath.begin();
            resultPath.insert(it, tmpVertex->name);
            tmpVertex = tmpVertex->parent;
        }
        while(tmpVertex != nullptr);
        for(it = resultPath.begin(); it != resultPath.end(); it++)
            cout << *it;
```

```
        cout << endl;
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    char from, to;
    scanf("%c %c", &from, &to);
    Graph go(from, to);
    go.aStar(go.from, go.to);
    go.print();

    return a.exec();
}
```