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*

Студентка гр. 9383 _____ Лихашва А.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы поиска кратчайшего пути в ориентированном графе (жадный алгоритм и алгоритм A*). Написать программу, которая реализует данные алгоритмы, используя полученные знания.

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
```

Вариант 2.

В A* эвристическая функция для каждой вершины задается неотрицательным числом во входных данных.

Основные теоретические положения.

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным

*Алгоритм A** — алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины к другой.

Описание алгоритмов.

Жадный алгоритм:

1. Начиная со стартовой вершины просматриваются смежные вершины от текущей. Среди этих смежных вершин выбирается та, у которой вес ре-

бра наименьший. Данная новая вершина прибавляется к текущему пути, а просматриваемая вершина считается пройденной.
2. Далее происходит то же самое для вершины, которая была выбрана на предыдущем шаге.
3. Если все смежные вершины от текущей пройдены, то нужно вернуться в пути на одну вершину назад.
4. Алгоритм считается завершенным, как только будет рассматриваться конечная вершина.

Алгоритм A*:

1. На каждом шаге выбирается вершина с наименьшим приоритетом. Приоритет определяется с помощью функции для оценки приоритета, которая состоит из эвристической функции и текущего расстояния от начальной вершины. Эвристическая функция вводится пользователем (по моему варианту).
2. Далее для данной вершины рассматриваются смежные ей вершины.
3. Для каждой смежной вершины проверяется ее кратчайший путь до начальной вершины.
4. Если текущий путь короче, чем кратчайший путь, то текущий путь становится кратчайшим.
5. Далее данная смежная вершина помещается в очередь с приоритетом, где значение приоритета определяется при помощи функции оценки приоритета.
6. Алгоритм считается завершенным, как только будет рассматриваться конечная вершина.

Сложность.

Жадный алгоритм:

Сложность по времени работы O(V * E), где V – количество вершин, E — количество ребер, так как на каждом шаге алгоритма рассматриваются смежные ребра.

Для хранения графа используется список смежности, поэтому в этом случае сложность O(E), где E — количество ребер в графе. При этом используется стек с вершинами, следовательно сложность будет O(V + E), где V – количество вершин в графе.

Алгоритм A*:

Лучший случай, когда имеется более подходящая эвристическая функция, которая позволяет делать каждый шаг в нужном направлении. Сложность по времени будет O(V + E), где V – количество вершин, E – количество ребер графа.

Худший случай, когда эвристическая функция не совсем подходящая, и определение нужного направления происходит достаточно долго, тогда нужно проходить всевозможные пути. Следовательно, время работы будет расти экспоненциально по сравнению с длиной оптимального пути.

В лучшем случае для каждой вершины будет храниться путь от начала до самой вершины. Оценка сложности по памяти будет O(V * (V + E)), где V — количество вершин, E — количество ребер графа.

В худшем случае все пути будут храниться в очереди, и сложность по памяти будет экспоненциальной.

Функции и структуры данных:

Структуры данных:

class FindingPath – класс для поиска кратчайшего пути.

map<char, vector<pair<char, double>>> graph — структура данных для хранения графа.

map<char, bool> visited — структура данных для отслеживания посещенных вершин.

map<char, double> heuristic — структура данных для хранения эвристических функций, которых вводит пользователь (алгоритм A*).

map<char, pair<vector<char>, double>> ShortPathes — структура данных, отвечающая за текущие кратчайшие пути от начальной вершины (алгоритм A*).

priority_queue<pair<char, double>, vector<pair<char, double>>, Sorting> PriorityQueue — очередь в алгоритме A*. Состоит из названия вершины и оценочной функции(кратчайшее расстояние до вершины + эвристическая функция). Для очереди есть специальный компаратор *Sorting*, который определяет приоритет.

Функции:

FindingPath::Read() — функция для считывания данных. Также для алгоритма A* считывается эвристические функции (по заданию).

vector<char> FindingPath::AlgorithmAStar() — функция, которая реализует алгоритм A*. Функция возвращает вектор, состоящий из вершин, которые входят в кратчайший путь.

vector<char> FindingPath::GreedyAlgorithm() — функция, которая реализует жадный алгоритм. Функция возвращает вектор, состоящий из вершин, которые входят в кратчайший путь.

Тестирование.

Жадный алгоритм:

Входные данные	Выходные данные
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	abdefg
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	abdefg
a d	abcad

a b 1.0 b c 1.0 c a 1.0 a d 8.0	

Алгоритм A*:

Входные данные	Выходные данные
a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 a 5 b 4 c 3 d 2 e 1	ade
a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0 a 1 b 2 c 3 d 4 e 5	aed
a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0 a 6 b 5 c 4 d 3 e 2 f 1	acdf
a b 3.0	abd

b c 2.0 b d 2.0 c d 4.0 a c 5.0 a 4 b 3 c 2 d 1	

Заключение.

В результате выполнения работы были изучены алгоритмы поиска кратчайшего пути в ориентированном графе (жадный алгоритм и алгоритм A*). Основываясь на полученных знания, была написана программа, реализующая данные алгоритмы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл GreedyAlg.h

```
#pragma once

#include <iostream>
#include <vector>
#include <map>
#include <queue>
using namespace std;


class FindingPath {
public:
    FindingPath() {};
    vector<char> GreedyAlgorithm();
    void Read();

private:
    map<char, vector<pair<char, double>>> graph;
    map<char, bool> visited;
    char start;
    char end;
    int number;
};
```

Файл GreedyAlg.cpp

```
#include "GreedyAlg.h"

void FindingPath::Read() {
    char start, end;
    cin >> start >> end;
    this->start = start;
    this->end = end;
    int count = 0;

    while (cin >> start) {
        if (start == '0')  //символ остановки ввода данных
            break;
        double weight;
        cin >> end >> weight;
        graph[start].push_back({ end,weight });
        visited[start] = false;
        visited[end] = false;
        count++;
    }
    this->number = count;
}
```

```
vector<char> FindingPath::GreedyAlgorithm() {
    double min;
    vector<char> result;
    result.reserve(this->number);
    result.push_back(this->start);

    char CurVertex = this->start;

    while (CurVertex != this->end) {
        min = 100;
        char NextVertex;
        bool found = false;

        for (auto& i : this->graph[CurVertex]) {
            if (!visited[i.first] && i.second < min) {
                min = i.second;
                NextVertex = i.first;
                found = true;
            }
        }

        visited[CurVertex] = true;

        if (!found) {
            if (!result.empty()) {
                result.pop_back();
                CurVertex = result.back();
            }
            continue;
        }

        CurVertex = NextVertex;
        result.push_back(CurVertex);
    }

    return result;
}


int main() {
    setlocale(LC_ALL, "Russian");
    FindingPath answer;
    answer.Read();
    vector<char> out = answer.GreedyAlgorithm();
    for (auto& i : out) {
        cout << i;
    }
    return 0;
}
```

Файл Astar.h

```
#pragma once

#include <iostream>
#include <vector>
#include <map>
#include <queue>
using namespace std;


struct Sorting { //функция сортировки для приоритетной очереди
    bool operator() (pair<char, double> a, pair<char, double> b)
{
        //если стоимость двух вершин равна, то возвращается
меньшая из них в алфавитном порядке, если стоимость разная, то
большая из них
        if (a.second == b.second)
            return (a.first < b.first);
        else
            return (a.second > b.second);
    }
};



class FindingPath {   //класс для поиска кратчайшего пути
public:
    FindingPath() {};
    vector<char> AlgorithmAStar();  //функция, реализующая алго-
ритм A*
    void Read();  //функция считывания

private:
    map<char, vector<pair<char, double>>> graph;  //хранение
графа
    map<char, bool> visited;  //посещенные вершины
    char start;
    char end;
    map<char, double> heuristic; //эвристические функции
};
```

Файл Astar.cpp

```
#include “AStar.h”

void FindingPath::Read() {
    cout << "Введите начальную и конечную вершины:\n";
    char start, end, heur, elem;
    cin >> start >> end;
    this->start = start;
    this->end = end;
```

```
    cout << "Введите ребра графа и их вес:\n";
    while (cin >> start) {
        if(start == '0')  //символ остановки ввода данных
            break;
        double weight;
        cin >> end >> weight;
        graph[start].push_back({ end, weight });
        visited[start] = false;
        visited[end] = false;
    }
    cout << "Введите эвристические функции для вершин (<вершина>
<эврист. функция>):\n";
    while (cin >> elem) {
        if (elem == '0')
            break;
        cin >> heur;
        if (heur > 0)
            heuristic[elem] = heur;
        else {
            cout << "Эвристическая функция должна быть не отри-
цательной!\n";
            break;
        }
    }
}


vector<char> FindingPath::AlgorithmAStar() {  //A*
    map<char, pair<vector<char>, double>> ShortPathes;  //теку-
щие кратчайшие пути
    vector<char> vertex;
    priority_queue<pair<char, double>, vector<pair<char,
double>>, Sorting> PriorityQueue; //очередь в алгоритме

    PriorityQueue.push({ start, 0 });
    vertex.push_back(start);
    ShortPathes[start].first = vertex;

    while (!PriorityQueue.empty()) {  //пока очередь не пуста
        if (PriorityQueue.top().first == end) {  //если найдена
конечная вершина
            return ShortPathes[end].first;  //то заканчивается
поиск
        }

        auto TmpVertex = PriorityQueue.top();  //достается прио-
ритетная вершина из очереди
        PriorityQueue.pop();

        for (auto& i : graph[TmpVertex.first]) {  //рассматрива-
ются все вершины, которые соединены с текущей вершиной
            double CurLength =
ShortPathes[TmpVertex.first].second + i.second;
```

```
            if (ShortPathes[i.first].second == 0 ||
ShortPathes[i.first].second > CurLength) { //если пути нет или
найденный путь короче
                vector<char> path =
ShortPathes[TmpVertex.first].first;  //добавляется в путь роди-
тельской вершины текущая вершина с кратчайшим путем
                path.push_back(i.first);
                ShortPathes[i.first] = { path, CurLength };  //
обновление пути и расстояния
                //double heuristic = abs(end - i.first);
                //cout << i.first << ' ' << heuristic[i.first]
<< '\n';
                PriorityQueue.push({ i.first, heuristic[i.first]
+ ShortPathes[i.first].second }); //записывается в очередь теку-
щая вершина
            }

        }

    }
    return ShortPathes[end].first;
}


int main() {
    setlocale(LC_ALL, "Russian");
    FindingPath answer;
    answer.Read();
    vector<char> out = answer.AlgorithmAStar();
    cout << "Результат работы алгоритма A*:\n";
    for (auto& i : out) {
        cout << i;
    }
    return 0;
}
```