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. 9383 _____ Камзолов Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с алгоритмами поиска кратчайшего пути в ориентированном графе (A* и жадный). Применить знания на практике и написать программу, реализующую эти алгоритмы.

Задание.

1. 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

2. 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 5. Реализация алгоритма, оптимального по используемой памяти (абсолютно, не только через О-нотацию)

Основные теоретические положения.

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

A* алгоритм **-** алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины к другой. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость»

Описание алгоритма.

Реализация жадного алгоритма:

Все вершины помещаются в двоичную кучу, играющую роль очереди с приоритетами (все вершины кроме стартовой имеют приоритет равный бесконечно большому числу, стартовая имеет приоритет 0).

1. На каждом шаге из кучи достается и удаляется вершина, имеющая наименьший приоритет.
2. Приоритет соседних к вершине, которую достали в пункте 1, пересчитываются в соответствии с длиной ребра, проведённых к этим вершинам, в том случае, если новый приоритет меньше текущего. Также вершины, у которых поменялись приоритеты, получают информацию о вершине, из пункта 1 (прокладывается путь).
3. Если на очередном шаге вершина, полученная из кучи, является конечной, то алгоритм останавливается. Путь восстанавливается от конечной вершины к первой, а затем разворачивается.

Реализация алгоритма A*:

Реализация данного алгоритма не сильно отличается от реализации жадного алгоритма. Единственное отличие состоит в пересчете приоритета: вместо того, чтобы приравнивать приоритет к длине ребра, мы приравниваем приоритет к длине текущего пути + длине ребра + эвристике вершины. При одинаковых приоритетах приоритет имеет та вершина, у которой эвристика меньше.

Данные алгоритмы были выбраны, потому что они могут решить проблему построения минимального пути от одной вершине к другой в графе.

Сложность.

Временная сложность алгоритмов равна O(nlogn), так как мы рассматриваем все вершины за O(n), затем достаем и перестраиваем кучу за O(logn), а также обновляем соседние вершины за O(m), где m – кол-во смежных вершин к данной.

Для оптимизации по памяти вместо построения очереди приоритетов из имеющегося vector’а, мы пользуемся стандартной функцией языка C++ std::make_heap(). Таким образом эта функция перестраивает, уже имеющийся vector в кучу и не нужны дополнительные затраты по памяти. Максимальная емкостная сложность алгоритма O(n^2), так как для хранения графа нужна матрица смежности.

Описание функций и структур данных.

struct Node – структура данных, которая является прототипом вершины графа. В ней содержится информация о текущем приоритете вершины, символьном значении вершины, а также здесь хранится информация о смежных вершинах и численном значении пути до этой вершины.

int customFind() – функция, которая ищет индекс элемента в списке смежности для обновления соседей вершины или создания новой вершины.

void initGraph() – функция, которая создает граф в зависимости от входных данных.

void algo() – функция, реализующая алгоритм (A* или жадный) в зависимости от файла.

int heuristic() – функция, рассчитывающая эвристику для алгоритма A*.

void freeMemory() – функция для очистки памяти, занимаемой списком смежности.

string vecToString() – функция для форматного вывода списка смежности.

Также для функций написано тестирование с помощью библиотеки Catch2.

Для файлов тестирования и основной программы написан Makefile.

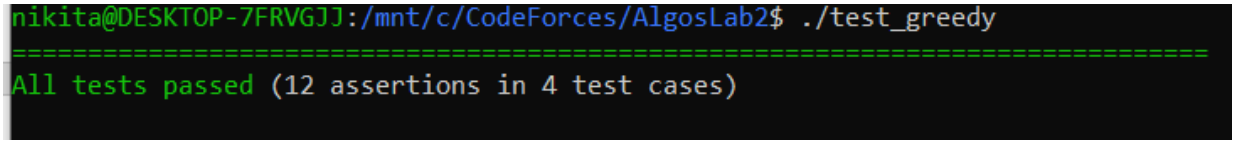

Рисунок 1 – Демонстрация корректного отрабатывания тестирования для жадного алгоритма.

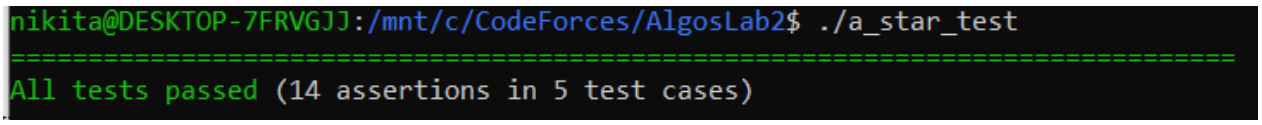

Рисунок 2 – Демонстрация корректного отрабатывания тестирования для алгоритма A*.

Тестирование.

Ввод	Вывод
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Жадный: abcde A*: ade
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	Жадный: abefg A*: ag
a b a b 1.0 a c 1.0	Жадный: ab A*: ab
a f a c 1.0 a b 1.0 c d 2.0	Жадный: acdf A*: acdf

b e 2.0 d f 3.0 e f 3.0	
a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	Жадный: aed A*: aed

Демонстрация работы.

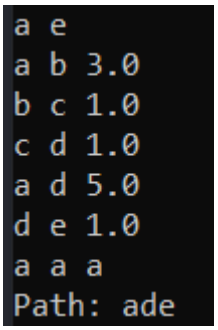

Рисунок 3 – Демонстрация работы программы, реализующей алгоритм A*.

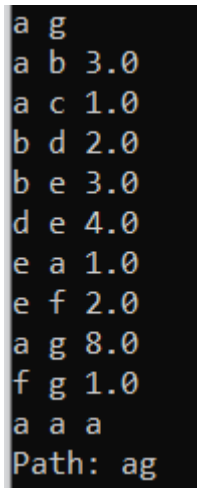

Рисунок 4 – Демонстрация работы программы, реализующей алгоритм A*.

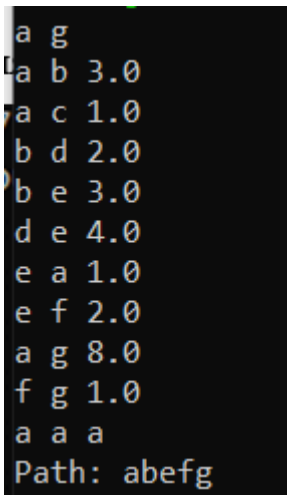

Рисунок 5 – Демонстрация работы программы, реализующей жадный алгоритм.

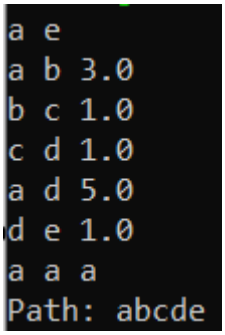

Рисунок 6 – Демонстрация работы программы, реализующей жадный алгоритм.

Выводы.

Произошло ознакомление с алгоритмами поиска кратчайшего пути в ориентированном графе (A* и жадный). Применены знания на практике и написаны программы, которые реализуют данный алгоритм и успешно ищут в введенном графе путь от стартовой вершины до конечной. Была проведена оптимизация по памяти, в ходе которой исходный std::vector перестраивается в бинарную кучу, и не происходит копирование его элементов в очередь с приоритетами.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл Greedy/main.cpp:

```
#include "greedy.h"

int main() {
    char start, finish;
    std::cin >> start >> finish;

    std::vector<Node*> graph;
    std::vector<Node*> vecToDelete;

    initGraph(graph, start, std::cin);

    vecToDelete = graph;

    std::cout << "Path: "<< algo(graph, finish) << '\n';
    freeMemory(vecToDelete);
}
```

Файл greedy.h:

```
#ifndef GREEDY_H
#define GREEDY_H

#include <vector>
#include <iostream>
#include <map>
#include <algorithm>
#include <sstream>

#define INF 3.40282e+038

struct Node {
    std::vector<std::pair<Node*, float>> neighbours;
    char vertex;
    float priority;
    Node* prev = nullptr;
    Node(char vertex, float priority = INF) : vertex(vertex),
priority(priority) {}
};

std::string vecToString(std::vector<Node*>& vec);
int customFind(std::vector<Node*>& graph, char& vertex);
void initGraph(std::vector<Node*>& graph, char& start, std::istream& in);
std::string algo(std::vector<Node*>& graph, char& finish);
void freeMemory(std::vector<Node*>& vec);

#endif
```

Файл greedy.cpp:

```
#include "greedy.h"

std::string vecToString(std::vector<Node*>& vec) {
    std::string ans = "";
    std::string vertex = "";
    for (auto& node : vec) {
        vertex = node->vertex;
        ans += "vertex: " + vertex + ", "
            + "neighbours: [";
        for (auto& neighbour : node->neighbours) {
            vertex = neighbour.first->vertex;
            ans += vertex + ", ";
        }
        ans += " ]; \n";
    }
    return ans;
}

int customFind(std::vector<Node*>& graph, char& vertex) {
    for (size_t i = 0; i < graph.size(); i++) {
        if (graph[i]->vertex == vertex) return i;
    }
    return -1;
}

void initGraph(std::vector<Node*>& graph, char& start, std::istream& in)
{
    char curVertex, neighbourVertex;
    float edgeLength;

    Node* startNode = nullptr;
    Node* endNode = nullptr;

    while (in >> curVertex >> neighbourVertex >> edgeLength) {
        if(!isalpha(curVertex) || !isalpha(neighbourVertex)) {
            continue;
        }
        if(edgeLength < 0) {
            continue;
        }

        int startIndex = customFind(graph, curVertex);
        int endIndex = customFind(graph, neighbourVertex);
        if (startIndex == -1) {
            if (curVertex == start)
                startNode = new Node(curVertex, 0);
            else
                startNode = new Node(curVertex);
            graph.push_back(startNode);

            startIndex = graph.size() - 1;
        }
        else startNode = graph[startIndex];

        if (endIndex == -1) {
```

```
            if (neighbourVertex == start)
                endNode = new Node(neighbourVertex, 0);
            else
                endNode = new Node(neighbourVertex);
            graph.push_back(endNode);
            //count++;
        }
        else endNode = graph[endIndex];
        graph[startIndex]->neighbours.push_back(std::make_pair(endNode,
edgeLength));
    }
}

std::string algo(std::vector<Node*>& graph, char& finish) {

    auto nodeComparator = [](
        const Node* lhs,
        const Node* rhs
    ) {
        if (lhs->priority > rhs->priority)
            return true;
        if (lhs->priority == rhs->priority && lhs->vertex > rhs->vertex)
            return true;
        return false;
    };

    std::make_heap(
        std::begin(graph),
        std::end(graph),
        nodeComparator
    );

    std::string answer = "";
    while(!graph.empty()) {
        Node* node = graph.front();
        if (node->vertex == finish)
        {
            while (node->prev) {
                answer += node->vertex;
                node = node->prev;

            }
            answer += node->vertex;
            break;
        }
        for (auto& neighbour : node->neighbours) {
            if (neighbour.first->priority > neighbour.second)
            {
                neighbour.first->priority = neighbour.second;
                neighbour.first->prev = node;
            }
        }
        graph.erase(graph.begin());
        std::make_heap(
            std::begin(graph),
            std::end(graph),
            nodeComparator
```

```
        );
    }
    reverse(answer.begin(), answer.end());
    if(answer.empty())
        answer = "No path.";
    return answer;
}

void freeMemory(std::vector<Node*>& vec) {
    for(auto node : vec) {
        delete node;
    }
}
```

Файл A_star/main.cpp:

```
#include "a_star.h"

int main() {
    char start, finish;
    std::cin >> start >> finish;

    std::vector<Node*> graph;
    std::vector<Node*> vecToDelete;

    initGraph(graph, start, finish, std::cin);

    vecToDelete = graph;

    std::cout << "Path: "<< algo(graph, finish) << '\n';

    freeMemory(vecToDelete);
}
```

Файл a_star.cpp:

```
#include "a_star.h"



int heuristic(char& start, char& finish) {
    return abs(start - finish);
}

std::string vecToString(std::vector<Node*>& vec) {
    std::string ans = "";
    std::string vertex = "";
    for (auto& node : vec) {
        vertex = node->vertex;
        ans += "vertex: " + vertex + ", "
            + "neighbours: [";
        for (auto& neighbour : node->neighbours) {
            vertex = neighbour.first->vertex;
```

```
            ans += vertex + ", ";
        }
        ans += " ]; \n";
    }
    return ans;
}

int customFind(std::vector<Node*>& graph, char& vertex) {
    for (size_t i = 0; i < graph.size(); i++) {
        if (graph[i]->vertex == vertex) return i;
    }
    return -1;
}

void initGraph(std::vector<Node*>& graph, char& start, char& finish,
std::istream& in) {
    char curVertex, neighbourVertex;
    float edgeLength;

    Node* startNode = nullptr;
    Node* endNode = nullptr;

    while (in >> curVertex >> neighbourVertex >> edgeLength) {
        if(!isalpha(curVertex) || !isalpha(neighbourVertex)) {
            continue;
        }
        if(edgeLength < 0) {
            continue;
        }

        int startIndex = customFind(graph, curVertex);
        int endIndex = customFind(graph, neighbourVertex);
        if (startIndex == -1) {
            if (curVertex == start)
                startNode = new Node(curVertex, heuristic(start,
finish));
            else
                startNode = new Node(curVertex);
            graph.push_back(startNode);

            startIndex = graph.size() - 1;
        }
        else startNode = graph[startIndex];

        if (endIndex == -1) {
            if (neighbourVertex == start)
                endNode = new Node(neighbourVertex, heuristic(start,
finish));
            else
                endNode = new Node(neighbourVertex);
            graph.push_back(endNode);
            //count++;
        }
        else endNode = graph[endIndex];
        graph[startIndex]->neighbours.push_back(std::make_pair(endNode,
edgeLength));
    }
```

```
}

std::string algo(std::vector<Node*>& graph, char& finish) {

    auto nodeComparator = [](
        const Node* lhs,
        const Node* rhs
    ) {
        if (lhs->priority > rhs->priority)
            return true;
        if (lhs->priority == rhs->priority && lhs->vertex < rhs->vertex)
            return true;
        return false;
    };

    std::make_heap(
        std::begin(graph),
        std::end(graph),
        nodeComparator
    );

    std::string answer = "";
    while(!graph.empty()) {
        //std::cout << vecToString(graph);
        Node* node = graph.front();
        if (node->vertex == finish)
        {
            while (node->prev) {
                answer += node->vertex;
                node = node->prev;
            }
            answer += node->vertex;
            break;
        }
        //std::cout << node->vertex << ':' << node->cost << std::endl;
        for (auto& neighbour : node->neighbours) {
            float temp = node->priority - heuristic(node->vertex, finish)
+ neighbour.second;
            if (temp < neighbour.first->priority)
            {
                neighbour.first->priority = temp +
heuristic(neighbour.first->vertex, finish);
                neighbour.first->prev = node;
            }
        }
        graph.erase(graph.begin());
        std::make_heap(
            std::begin(graph),
            std::end(graph),
            nodeComparator
        );
    }
    reverse(answer.begin(), answer.end());
    if(answer.empty())
        answer = "No path.";
    return answer;
}
```

```
void freeMemory(std::vector<Node*>& vec) {
    for(auto node : vec) {
        delete node;
    }
}
```

Файл a_star.h:

```
#ifndef ASTAR_H
#define ASTAR_H

#include <vector>
#include <iostream>
#include <map>
#include <algorithm>
#include <sstream>

#define INF 3.40282e+038

struct Node {
    std::vector<std::pair<Node*, float>> neighbours;
    char vertex;
    float priority;
    Node* prev = nullptr;
    Node(char vertex, float priority = INF) : vertex(vertex),
priority(priority) {}
};

int customFind(std::vector<Node*>& graph, char& vertex);
void initGraph(std::vector<Node*>& graph, char& start, char& finish,
std::istream& in);
std::string algo(std::vector<Node*>& graph, char& finish);
int heuristic(char& start, char& finish);
void freeMemory(std::vector<Node*>& vec);
std::string vecToString(std::vector<Node*>& vec);

#endif
```