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 _____ Моисейченко К.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить и применить на практике жадный алгоритм поиска пути и алгоритм A*. Реализовать программу, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма A*.

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

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 O-нотацию).

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 (начальной) к другой (целевой, конечной). Порядок обхода вершин определяется эвристической функцией “расстояние + стоимость”.

Выполнение работы.

Реализация жадного алгоритма:

Все вершины помещаются в двоичную кучу. Приоритет всех вершин, кроме стартовой, изначально равен максимальному числу, а стартовой - нулю. Начинается цикл, который прекратится тогда, когда в двоичной куче не останется элементов:

1. Из кучи достается и удаляется вершина с наименьшим приоритетом.
2. Приоритеты соседних к удаленной вершине пересчитываются в соответствие с длиной рёбер, проведенных к этим вершинам, если новый приоритет меньше текущего. Вершины, у которых поменялись приоритеты, получают информацию о вершине, из пункта 1.
3. Алгоритм останавливается, если вершина, полученная из кучи, является конечной. Путь восстанавливается от конечной вершины до стартовой.

Реализация алгоритма A*:

Единственное отличие от реализации жадного алгоритма состоит в пересчете приоритета: при пересчете мы приравниваем приоритет к длине текущего пути + длине ребра + эвристике вершины. При одинаковых приоритетах, приоритет имеет та вершина, у которой меньше эвристика.

Сложность.

Временная сложность алгоритмов равна O(n log n). Для оптимизации алгоритма по памяти, вместо построения очереди приоритетов из имеющегося вектора, мы пользуемся стандартной функцией std::make_heap(). Эта функция перестраивает уже имеющийся вектор в кучу, и таким образом не нужны дополнительные затраты по памяти. Максимальная емкостная сложность алгоритма - O(n^2), т.к. граф хранится с помощью матрицы смежности.

Описание функций и структур данных.

struct Node - структура, прототип вершины графа. В ней содержится информация о приоритете вершины, символьном значении вершины и информация о смежных вершинах и численном значении пути до них.

void readGraph() - функция, создает граф, в зависимости от входных данных.

int heuristic_func() - функция, рассчитывает эвристику для алгоритма A*.

int findIndex() - функция, ищет индекс элемента в списке смежности для обновления соседей вершины или создания новой вершины.

string Astar() - функция, реализующая алгоритм A*.

string greedy() - функция, реализующая жадный алгоритм.

string vecToString() - функция, выводит список смежности.

void freeMemory() - функция, очищает память, занимаемой списком смежности.

Примеры работы программы.

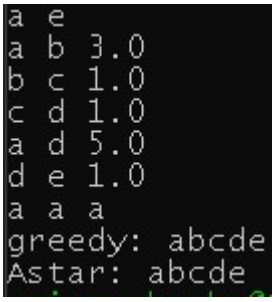

Рисунок 1 - Пример работы программы №1.

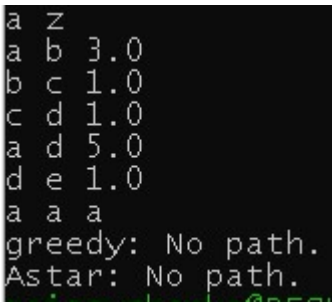

Рисунок 2 - Пример работы программы №2.

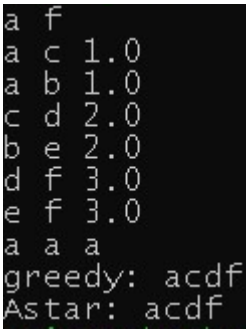

Рисунок 3 - Пример работы программы №3.

Выводы.

Были изучены и применены на практике жадный алгоритм поиска пути и алгоритм A*. Реализована программа, которая считывает граф и находит в нем путь от стартовой вершины к конечной с помощью жадного алгоритма и алгоритма A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: graph.h

```
#pragma once

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


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

void readGraph(std::vector<Node*>& graph, char start, char finish, std::istream& in);
int heuristic_func(char start, char finish);
int findIndex(std::vector<Node*>& graph, char value);
std::string Astar(std::vector<Node*>& graph, char finish);
std::string greedy(std::vector<Node*>& graph, char finish);
std::string vecToString(std::vector<Node*>& vec);
void freeMemory(std::vector<Node*>& vec);
```

Название файла: graph.cpp

```
#include "graph.h"

void readGraph(std::vector<Node*>& graph, char start, char finish, std::istream& in)
{
    char curValue, neighbourValue;
    float edgeLength;
    Node* startNode = nullptr;
    Node* endNode = nullptr;

    while (in >> curValue >> neighbourValue >> edgeLength)
    {
        if (!isalpha(curValue) || !isalpha(neighbourValue))
        {
            continue;
        }
        if (edgeLength < 0)
        {
            continue;
        }
        int startIndex = findIndex(graph, curValue);
```

```
            int endIndex = findIndex(graph, neighbourValue);
            if (startIndex == -1)
            {
                if (curValue == start)
                {
                    startNode = new Node(curValue, heuristic_func(start,
finish));
                }
                else
                {
                    startNode = new Node(curValue);
                }
                graph.push_back(startNode);
                startIndex = graph.size() - 1;
            }
            else
            {
                startNode = graph[startIndex];
            }
            if (endIndex == -1)
            {
                if (neighbourValue == start)
                {
                    endNode       =       new       Node(neighbourValue,
heuristic_func(start, finish));
                }
                else
                {
                    endNode = new Node(neighbourValue);
                }
                graph.push_back(endNode);
            }
            else
            {
                endNode = graph[endIndex];
            }

graph[startIndex]->neighbours.push_back(std::make_pair(endNode,
edgeLength));
        }
    }

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

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

```
    }

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

        auto node_cmp = [](const Node* left, const Node* right)
        {
            if (left->priority > right->priority)
            {
                return true;
            }
            if  (left->priority  ==  right->priority  &&  left->value  <
right->value)
            {
                return true;
            }
            return false;
        };

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

        std::string answer;
        while (!graph.empty())
        {
            //std::cout << vecToString(graph);
            Node* node = graph.front();
            if (node->value == finish)
            {
                while (node->prev)
                {
                    answer += node->value;
                    node = node->prev;
                }
                answer += node->value;
                break;
            }
            for (auto& neighbour : node->neighbours)
            {
                float temp = node->priority - heuristic_func(node->value,
finish) + neighbour.second;
                if (temp < neighbour.first->priority)
                {
                    neighbour.first->priority       =       temp       +
heuristic_func(neighbour.first->value, finish);
                    neighbour.first->prev = node;
                }
            }
            std::swap(graph[0], graph[graph.size() - 1]);
            graph.pop_back();
            std::make_heap(std::begin(graph), std::end(graph), node_cmp);
        }
        reverse(answer.begin(), answer.end());
        if (answer.empty())
        {
            answer = "No path.";
        }
        return answer;
```

```
    }

    std::string greedy(std::vector<Node*>& graph, char finish)
    {
        auto node_cmp = [](const Node* left, const Node* right)
        {
            if (left->priority > right->priority)
            {
                return true;
            }
            if (left->priority == right->priority && left->value >
right->value)
            {
                return true;
            }
            return false;
        };

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

        std::string answer;
        while (!graph.empty())
        {
            Node* node = graph.front();
            if (node->value == finish)
            {
                while (node->prev)
                {
                    answer += node->value;
                    node = node->prev;
                }
                answer += node->value;
                break;
            }
            for (auto& neighbour : node->neighbours)
            {
                if (neighbour.first->priority > neighbour.second)
                {
                    neighbour.first->priority = neighbour.second;
                    neighbour.first->prev = node;
                }
            }
            std::swap(graph[0], graph[graph.size() - 1]);
            graph.pop_back();
            std::make_heap(std::begin(graph), std::end(graph), node_cmp);
        }
        reverse(answer.begin(), answer.end());
        if (answer.empty())
        {
            answer = "No path.";
        }
        return answer;
    }

    std::string vecToString(std::vector<Node*>& vec)
    {
        std::string ans;
```

```
    std::string value;
    for (auto& node : vec)
    {
        value = node->value;
        ans += value + " = [";
        for (auto& neighbour : node->neighbours)
        {
            value = neighbour.first->value;
            ans += value + ", ";
        }
        ans += " ]; \n";
    }
    return ans;
}

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

Название файла: main.cpp

```
#include "graph.h"

int main() {
    char start, finish;
    std::cin >> start >> finish;
    std::vector<Node*> graph;
    std::vector<Node*> vecToDelete;
    readGraph(graph, start, finish, std::cin);
    std::cout << greedy(graph, finish) << '\n';
    std::cout << Astar(graph, finish) << '\n';
    freeMemory(graph);
}
```