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: Алгоритмы на графах

Студент гр. 9383 _____ Арутюнян С.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы поиска пути на графах и применить их на практике.

Основные теоретические положения.

Путь межу вершиной V и вершиной U в графе — это такая последовательность вершин X1, X2, …, Xn, что X1 = V, Xn = U.

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.

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 для обоих заданий:

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
```

Вариант 3.

Написать функцию, проверяющую эвристику на допустимость и монотонность.

Ход работы:

1. Был разработан класс Graph, представляющий собой обертку над весовой матрицей.

2. Был разработан жадный алгоритм из задания 1. Алгоритм его работы таков:

1) Если текущая вершина равна конечной, завершаем алгоритм. Иначе переходим к шагу 2.

2) Если у текущей вершины нет еще не посещенных соседей, убираем ее из текущего пути, помечаем ее как посещенную и возвращаемся к предыдущей вершине (переходим к шагу 1). Иначе переходим к шагу 3.

3) Инициализируем локальный минимум (т. е. вершина, к которой ведет наименьшее ребро от текущей). Сначала локальный минимум равен первому попавшемуся соседу текущей вершины.

4) Проходимся по всем соседям текущей вершины.

5) Если вес ребра между текущей вершиной и текущим соседом меньше, чем вес ребра между текущей вершиной и локальным минимумом, заменяем локальный минимум на текущего соседа.

6) Когда мы прошлись по всем соседям, инициализируем текущую вершину как локальный минимум.

7) Добавляем локальный минимум в путь. Переходим к шагу 1.

3. Был разработан алгоритм A* из задания 2. Алгоритм его работы таков:

1) Инициализируются вектора weights (вектор float, весь заполняется бесконечностью), visited (булевый вектор, заполняется false) и ancestors (вектор char, заполняется 0).

2) Добавляем в очередь с приоритетами стартовую вершину. weights[start] = 0.

3) Если очередь с приоритетами пуста, завершаем алгоритм. Иначе переходим к шагу 4.

4) Достаем вершину из очереди с приоритетом.

5) Если текущая вершина уже была посещена, переходим к шагу 3.

6) Если текущая вершина равна конечной, завершаем алгоритм.

7) Проходимся по всем соседям текущей вершины. Если соседи закончились, переходим к шагу 10.

8) Если (weigths[neighbour] = беск.) или (путь от стартовой вершины до текущей + путь от текущей вершины до текущего соседа меньше текущего пути до текущего соседа), переходим к шагу 9. Иначе переходим к следующему соседу и переходим к шагу 8.

9) weights[neighbour] становится равно путю от стартовой вершины до текущей + путь от текущей вершины до текущего соседа. В очередь с приоритетами добавляется текущий сосед с приоритетом weights[neighbour] + эвристика(neighbour, end). Переход к шагу 8.

10) Текущая вершина помечается как посещенная. Переход к шагу 3.

Описание функций и структур данных:

1. Класс Graph. Он имеет поле std::vector<std::vector<float>> weight_matrix. Это представление весовой матрицы, в которой weight_matrix[v][u] равняется весу ребра между вершинами v и u.

У класса Graph есть следующие методы:

1) void AddEdge(v1, v2, weight) – добавляет ребро между v1 и v2 с весом weight.

2) float GetEdgeWeight(v1, v2) – возвращает вес ребра между v1 и v2.

3) bool HasEdge(v1, v2) – возвращает true, если между v1 и v2 существует ребро.

4) std::queue<char> GetNeighbours(v, predicate) – возвращает очередь соседей вершины v, удовлетворяющих условию predicate.

2. Структура Vertex. Хранит имя вершины и ее приоритет в очереди с приоритетами в алгоритме A*. Служит для более удобной работы с очередью с приоритетами.

Примеры работы программы

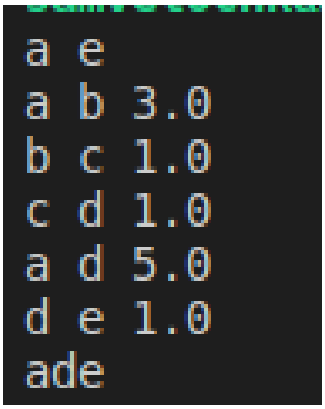

Рисунок 1. Пример работы программы (1).

Рисунок 2. Пример работы программы (2).

Рисунок 3. Пример работы программы (3).

Выводы.

В выполненной лабораторной работе были изучены и применены на практике алгоритмы жадного поиска кратчайшего пути и A*.

ПРИЛОЖЕНИЕ А

GRAPH.H

```
#pragma once

#include <vector>
#include <queue>
#include <functional>

class Graph {
public:
    Graph();

    void AddEdge(char u, char v, float weight);
    void AddEdge(int u, int v, float weight);

    std::queue<char> GetNeighbours(char u, const std::function<bool(char)>& predicate = [](char c)
{ return true; }) const;
    float GetEdgeWeight(char u, char v) const;
    float GetEdgeWeight(int u, int v) const;
    bool HasEdge(char u, char v) const;
    bool HasEdge(int u, int v) const;



    void PrintEdges() const;

private:
    std::vector<std::vector<float>> incident_matrix;
};
```

GRAPH.CPP

```
#include "graph.h"
#include <iostream>

Graph::Graph() {
    incident_matrix.resize(26, std::vector<float>(26, -1));
}

void Graph::AddEdge(char u, char v, float weight) {
    incident_matrix[u - 97][v - 97] = weight;
}

void Graph::AddEdge(int u, int v, float weight) {
    incident_matrix[u][v] = weight;
}

std::queue<char> Graph::GetNeighbours(char u, const std::function<bool(char)>& predicate) const
{
    std::queue<char> neighbours;

    for (char i = 'a'; i <= 'z'; ++i) {
        if (incident_matrix[u - 97][i - 97] != -1 && predicate(i))
            neighbours.push(i);
    }

    return neighbours;
}

float Graph::GetEdgeWeight(char u, char v) const {
    return incident_matrix[u - 97][v - 97];
}

float Graph::GetEdgeWeight(int u, int v) const {
    return incident_matrix[u][v];
}
```

```
bool Graph::HasEdge(char u, char v) const {
    return incident_matrix[u - 97][v - 97] != -1;
}

bool Graph::HasEdge(int u, int v) const {
    return incident_matrix[u][v] != -1;
}

void Graph::PrintEdges() const {
    for (char i = 'a'; i <= 'z'; ++i) {
        for (char j = 'a'; j <= 'z'; ++j) {
            if (HasEdge(i, j))
                std::cout << i << " " << j << " " << GetEdgeWeight(i, j) << std::endl;
        }
    }
}
```

SEARCH_ALGORITHMS.H

```
#pragma once

#include <string>
#include "graph.h"

std::string GreedyAlgorithm(const Graph& graph, char start, char end);
std::string AStarAlgorithm(const Graph& graph, char start, char end);
```

ASTAR.CPP

```
#include "search_algorithms.h"
#include <queue>
#include <algorithm>
#include <string>
#include <vector>


struct Vertex {
    char name;
    float priority;

    bool operator<(const Vertex& v) const {
        return priority != v.priority ? priority > v.priority : name < v.name;
    }
};

float Heuristics(char v, char u) {
    return static_cast<float>(std::abs(v - u));
}

static std::string FindPath(char start, char end, const std::vector<char>& ancestors) {
    std::string path;

    char current = end;
    while (current != start) {
        path += current;
        current = ancestors[current - 97];
    }
    path += start;
    std::reverse(path.begin(), path.end());

    return path;
}
```

```
std::string AStarAlgorithm(const Graph& graph, char start, char end) {
    std::vector<float> weights(26, -1);
    std::vector<bool> visited(26);
    std::vector<char> ancestors(26);

    std::priority_queue<Vertex> to_visit;
    to_visit.push({start, 0});
    weights[start - 97] = 0;

    while (!to_visit.empty()) {
        auto current_vertex = to_visit.top();
        to_visit.pop();

        if (visited[current_vertex.name])
            continue;
        if  (current_vertex.name == end)
            break;


        auto neighbours = graph.GetNeighbours(current_vertex.name);
        while (!neighbours.empty()) {
            auto neighbour = neighbours.front();
            neighbours.pop();

                auto cost = weights[current_vertex.name - 97] +
graph.GetEdgeWeight(current_vertex.name, neighbour);
            if (weights[neighbour - 97] == -1 || cost < weights[neighbour - 97]) {
                weights[neighbour - 97] = cost;
                to_visit.push({neighbour, cost + Heuristics(neighbour, end)});
                ancestors[neighbour - 97] = current_vertex.name;
            }
        }

        visited[current_vertex.name - 97] = true;
```

```
    }

    return FindPath(start, end, ancestors);
}
```

GREEDY.CPP

```
#include "search_algorithms.h"
#include <vector>
#include <set>


std::string GreedyAlgorithm(const Graph& graph, char start, char end) {
    std::vector<char> path = {start};
    std::set<char> visited = {start};
    char current_vertex = start;

    while (current_vertex != end) {
        auto neighbours = graph.GetNeighbours(current_vertex,
                                              [&visited](char c) { return visited.find(c) ==
visited.end(); });
        if (neighbours.empty()) {
            visited.insert(current_vertex);
            path.pop_back();
            current_vertex = path.back();
            continue;
        }

        char local_minimum = neighbours.front();

        while (!neighbours.empty()) {
            char neighbour = neighbours.front();
                                if (graph.GetEdgeWeight(current_vertex, neighbour) <
graph.GetEdgeWeight(current_vertex, local_minimum))
                local_minimum = neighbour;
```

```
            NEIGHBOURS.POP();
        }

        CURRENT_VERTEX = LOCAL_MINIMUM;
        PATH.PUSH_BACK(CURRENT_VERTEX);
    }

    RETURN STD::STRING{PATH.BEGIN(), PATH.END()};
}
```

MAIN.CPP

```
#INCLUDE <IOSTREAM>
#INCLUDE "GRAPH.H"
#INCLUDE "SEARCH_ALGORITHMS.H"

INT MAIN() {
    GRAPH GRAPH;

    CHAR START, END;
    STD::CIN >> START >> END;

    WHILE (TRUE) {
        CHAR U; STD::CIN >> U;
        IF (STD::CIN.EOF())
            BREAK;

        CHAR V;       STD::CIN >> V;
        FLOAT WEIGHT; STD::CIN >> WEIGHT;
        GRAPH.ADDEDGE(U, V, WEIGHT);
    }

    AUTO PATH = ASTARALGORITHM(GRAPH, START, END);
    STD::COUT << PATH << STD::ENDL;
    RETURN 0;
}
```