МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритмы на графах

Студентка гр. 8304 Сани З. Б.

Преподаватель Размочаева Н. В.

Санкт-Петербург

2020

Вариант 3.

Цель работы.

Научиться применять жадный алгоритм и алгоритм A* поиска пути в графе и оценивать их сложность.

Постановка задачи.

Вариант 3. Написать функцию, проверяющую эвристику на допустимость и монотонность.

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.

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.

Описание алгоритма.

Стартовая вершина помечается «открытой». Пока существуют открытые вершины:

1) Текущая вершина – открытая вершина, с наименьшей полной стоимостью.
2) Если текущая вершина конечная – алгоритм заканчивает работу.
3) Текущая вершина становится закрытой.
4) Для каждого незакрытого ребенка текущей вершины:
    - Рассчитывается функция пути для этой вершины.
    - Если вершина еще не открыта или рассчитанная функция меньше функции, рассчитанной для этой вершины ранее, рассчитанная функция становится функцией этой вершины, вершина-предок запоминается, как вершина, из которой совершен переход в ребенка. (необходимо для восстановления пути в будущем)

Если открытых вершин не осталось, а до конечной маршрут так и не был проложен, алгоритм заканчивает работу (пути не существует).

Анализ алгоритмов.

Временная сложность алгоритмов: $O(E + V^2) = O(V^2)$, где V – кол-во вершин, E – кол-во ребер. Обосновывается оценка тем, что в худшем случае будет совершен обход всех вершин и всех ребер. временная сложность в худшем случае экспоненциальная, в лучшем полиномиальная.

Сложность по памяти экспоненциальная.

Описание основных структур данных и функций.

Структуры.

Граф хранится в словаре `std::map<char, VertexInfo>`, где `VertexInfo` это структура, хранящая информацию о соседях и значение функции G.

```
struct VertexInfo {
    std::vector<std::pair<char, int>> neighbors;
    int length;
};
```

Функции из описания алгоритма:

$f(v)=g(v)+h(v)$

- $g(v)$ — наименьшая стоимость пути в vv из стартовой вершины,
- $h(v)$ — эвристическое приближение стоимости пути от vv до конечной цели.

```
int G(char from, std::map<char, VertexInfo > & graphDict);
int H(char from, char endVertex);
int F(char from, char endVertex, std::map<char, VertexInfo >& graphDict);
```

Функция ищет для какой из открытых вершин будет минимальным путь.

```
char minVertex(std::set<char> & open, char endVertex, std::map<char, VertexInfo >&
graphDict);
```

Функция возвращает понятный для пользователя путь в графе.

```
std::string constructPath (std::map<char,char> & from,char startVertex, char
endVertex);
```

Функция ищущая алгоритмом A* минимальный путь.

```
void astarAlgo(char startVertex, char endVertex,std::map<char, VertexInfo >& my_
graphDict);
```

monotone - функцияиспользуется для проверки данной эвристики на монотонность.

```
void monotone(std::map<char, VertexInfo > vertices,char endVertex);
```

Вывод промежуточной информации.

В ходе работы был реализован жадный алгоритм и алгоритм A* поиска пути в графе, была оценена их сложность.

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

Ввод	**Вывод**
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade
a b a e 0 a c 0 e b 0	aeb

a b a d 3.0 a c 2.1 d b 1.0	adb

Вывод.

В ходе работы был построен и анализирован алгоритм A* на основе решения задачи о нахождении минимального пути в графе. Исходный код программы представлен в приложении 1.

ИСХОДНЫЙ КОД

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <set>

struct VertexInfo {
    std::vector<std::pair<char, int>> neighbors;
    int length;
};

bool cmp(const std::pair<char,int> &a, const std::pair<char,int> &b) {
    if(a.second == b.second) return a.first > b.first;
    return a.second < b.second;
}

int G(char from, std::map<char, VertexInfo> &graphDict) {
    return graphDict[from].length;
}

int H(char from, char endvertex) {
    return abs(from - endvertex);

}

int F(char from, char endvertex, std::map<char, VertexInfo>& graphDict){
    return G(from, graphDict) + H(from, endvertex);
}

char minVertex(std::set<char> & open, char endvertex, std::map<char, VertexInfo>& graphDict) {
    int min = INT_MAX;
    char curr = 0;
    for(char i: open)
        if(F(i, endvertex, graphDict) < min){
            min = F(i, endvertex, graphDict);
            curr = i;
        }
    return curr;
}

std::string constructPath(std::map<char,char> &from,char startVertex, char endVertex) {
    std::string path = {endVertex};
    char curr = endVertex;
    while (curr != startVertex) {
        curr = from[curr];
        path += curr;
    }
    std::reverse(path.begin(), path.end());
    return path;
}

void astarAlgo(char startVertex, char endVertex,std::map<char, VertexInfo>& graphDict) {
    char currentVertex = startVertex;
    std::set<char> closed;
    std::set<char> open = {startVertex};
    std::map<char,char> from;

    while(!open.empty()) {
        currentVertex = minVertex(open,endVertex,graphDict);

        if(currentVertex == endVertex){//endGame
            std::cout << constructPath(from, startVertex, endVertex) << std::endl;
            return;
        }
        open.erase(currentVertex);
        closed.insert(currentVertex);

        for(auto neighbour : graphDict[currentVertex].neighbors) {
            bool tentative_is_better;
            if(closed.find(neighbour.first) != closed.end())
                continue;
            int tentative_g_score = G(currentVertex, graphDict) + neighbour.second;
            if(open.find(neighbour.first) == open.end()){
                open.insert(neighbour.first);
                tentative_is_better = true;
            }
            else {
                tentative_is_better = tentative_g_score < G(neighbour.first, graphDict);
            }

            if(tentative_is_better) {
                from[neighbour.first] = currentVertex;
                graphDict[neighbour.first].length = tentative_g_score;
            }
        }
    }
    std::cout << "No way" << std::endl;
}
```

```cpp
bool monotone(std::map<char, VertexInfo> vertices,char endVertex){
    for(auto vertex : vertices){
        char vertice1 = vertex.first;
        for (int i = 0 ; i < vertex.second.neighbors.size(); ++i) {
            char vertice2 = vertex.second.neighbors[i].first;
            double weight = vertex.second.neighbors[i].second;
            if(abs(endVertex-vertice1) - abs(endVertex-vertice2) > weight){
                return false;
            }
        }
    }
    return true;
}

int main() {
    char startVertex, endVertex;

    std::cin >> startVertex >> endVertex;
    std::map<char, VertexInfo> graphDict;

    char from, to;
    float length;
    while(std::cin >> from >> to >> length) {
        if(length == -1)
            break;
        graphDict[from].neighbors.push_back({to,length});
        std::sort(graphDict[from].neighbors.begin(),graphDict[from].neighbors.end(), cmp);
    }


    auto time = clock();
    std::cout << "________AStar Algorith with Monotone function_______" << std::endl;
    std::cout << "Optima path : ";
    astarAlgo(startVertex, endVertex, graphDict);
    if(monotone(graphDict, endVertex)){
        std::cout<< "The heuristic function of the graph is Monotone\n";

    }else{
        std::cout << "Not Monotone\n";
    }
    std::cout << "_________________________________" << std::endl;
    std::cout << "Time taken : " << (double)(clock() - time) / CLOCKS_PER_SEC << std::endl;
    std::cout << "Algorith complexity G(V,E) : O(|V| * |V| + E) ~ O(V * V)" << std::endl;
    std::cout << "where V - Vertices and E - edges" << std::endl;
    return 0;
}
```