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 поиска пути в графах

Студентка гр. 8304 _____ Мельникова О.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с алгоритмами поиска пути в графах на примере жадного алгоритмом на графе и алгоритма A*. Научиться применять данные алгоритмы для решения задач, а также оценивать временную сложность алгоритмов.

Вариант 1.

Постановка задачи.

1) 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.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

2) Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе именуется целыми числами (в т. ч. отрицательными), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Выполнение работы.

Описание жадного алгоритма.

На первом шаге рассматривается стартовая вершина. На каждом последующем шаге будут рассматриваться вершины, в которые можно попасть напрямую из текущей вершины. Далее среди этих вершин выбирается вершина, расстояние до которой от текущей является наименьшим. Эта вершина помечается рассмотренной и выбирается текущей. Предыдущий шаг повторяется. Если из текущей вершины не существует пути в еще не рассмотренные до этого вершины, то происходит возврат в вершину, из которой был совершен переход в текущую.

Алгоритм заканчивает работу в двух случаях:

1) Когда текущей вершиной становится конечная (путь найден).
2) Когда все вершины были рассмотрены.

Описание алгоритма A*.

Стартовая вершина помечается заносится в «открытую». Пока существуют «открытые» вершины:

1) В качестве текущей выбирается открытая, с наименьшим значением f (полной стоимостью).
2) Если текущая вершина является конечной, то алгоритм заканчивает свою работу (путь существует).
3) Переносим текущую вершину в «закрытые».

Для каждого ребенка текущей вершины, который не является «закрытым» находим длину пути.

Если вершина еще не закрыта или рассчитанная функция пути меньше функции, рассчитанной для этой вершины ранее, то присваиваем значение рассчитанной функции этой вершине. Запоминаем вершину, из которой мы пришли в ребенка для того, чтобы потом восстановить путь.

Если открытых вершин не осталось, алгоритм заканчивает работу (пути не существует).

Описание функций и структур данных.

Структуры для описания вершины: VertexI содержит имя вершины в int, массив ее детей — хранящихся с помощью структуры ChildI, которая содержит дочернюю вершину и цену пути до нее от родительской. Кроме того в VertexI есть поля g — стоимость пути и f — эвристическая оценка.

```
struct ChildI{
    ChildI(VertexI* child, double cost) : child(child), cost(cost) { }
    VertexI* child;
    double cost;
};
struct VertexI{
    VertexI(int name) : name(name){}
    int name;
    std::vector <ChildI*> children;
    double g;
    double f;
};
```

Функция для вычисления эвристической функции (в данном решении – расстояние между символами в таблице ASCII).

```
int h(VertexI* vertex1, VertexI* vertex2) ;
```

Функция для поиска вершины с самой низкой оценкой.

```
VertexI* vertexWithMinF(std::set<VertexI*> open) {
```

Функция для вывода результата.

```
std::string printResA(VertexI* end, std::map<VertexI*, VertexI*> from){
```

Функция алгоритма A*. Принимает начальную и конечную вершину.

```
std::string aStar(VertexI* start, VertexI* end)
```

Анализ алгоритмов.

Оценим временную сложность алгоритмов: $O(E+V^2)=O(V^2)$, где V – кол-во вершин, E – кол-во ребер. В худшем случае для поиска пути потребуется обход всех ребер и вершин.

Оценим затраты на хранение: $O(E+V)$, где V – кол-во вершин, E – кол-во ребер, затрачивается на хранение графа. Для хранения в программе используется V – вершин и E – указателей на смежные вершины.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	1 4 1 2 1.0 2 3 1.1 3 4 0.5	1234
2.	1 5 1 2 3.0 2 3 1.0 3 4 1.0 1 4 5.0 5 6 1.0	Путь не найден
3.	-1 6 -5 4 5.0 -1 0 4.0 0 2 1.0 2 6 0.1 0 6 5.0	-5-2013
4.	-1 6 -5 4 5.0 -1 0 4.0 0 2 1.0	-106

	2 6 0.1 0 6 5.0	

Выводы.

В ходе выполнения лабораторной работы были реализованы алгоритмы поиска пути в графе, дана оценка времени работы алгоритмов и требуемой памяти.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <stack>
#include <set>

struct VertexI;

struct ChildI{
    ChildI(VertexI* child, double cost) : child(child), cost(cost) { }
    VertexI* child;
    double cost;
};

struct VertexI{
    VertexI(int name) : name(name){}
    int name;
    std::vector <ChildI*> children;
    double g;
    double f;
};

int h(VertexI* vertex1, VertexI* vertex2) { //эвр. ф. - близость символов
    return abs(vertex1->name - vertex2->name);
}

VertexI* vertexWithMinF(std::set<VertexI*> open) {
    VertexI* vertexWithMinF = nullptr;

    for (auto i : open) {
        if (!vertexWithMinF) {
            vertexWithMinF = i;
        }
        else if (vertexWithMinF->f > i->f) {
            vertexWithMinF = i;
        }
        else if (vertexWithMinF->f == i->f) {
```

```
            if (vertexWithMinF->name < i->name) {
                vertexWithMinF = i;
            }
        }
    }
    return vertexWithMinF;
}

std::string printResA(VertexI* end, std::map<VertexI*, VertexI*> from){
    std::string res = "";

    std::stack<int> stack;

    while (from.find(end) != from.end()) { //формируем стек с конца
        stack.push(end->name);
        end = from[end];
    }

    stack.push(end->name);
    int top = stack.top();
    if(top<0){
        res.push_back('-');
        top = -top;
    }
    res.push_back(top + '0');
    stack.pop();
    while (!stack.empty()) {
        top = stack.top();
        if(top<0){
            res.push_back('-');
            top = -top;
        }
        res.push_back(top + '0');
        stack.pop();
    }
    return res;
}

std::string aStar(VertexI* start, VertexI* end) {
    std::set<VertexI*> open; //раскрыты
    std::set<VertexI*> closed; //нужно обработать
    std::map<VertexI*, VertexI*> from; //необходимые
```

```
    //заполняем св-ва вершины старт
    start->g = 0; //стоимость пути от начальной
    start->f = start->g + h(start, end); //эвр. оценка

    open.insert(start); //обработана

    while (!open.empty()) {
           VertexI* current = vertexWithMinF(open); //вершина с самой низкой
оценкой

        if (current == end) {
            return printResA(current, from);
        }

        open.erase(open.find(current)); //уд. из очереди
        closed.insert(current); //доб. в список обработанных

        for (auto child : current->children) { //проверяем каждого соседа
            if (closed.find(current) != closed.end()) {
                double tmpChildG = current->g + child->cost;

                 if (child->child->g > tmpChildG || closed.find(child->child)
== closed.end())  {
                   from[child->child] = current;
                   child->child->g = tmpChildG;
                   child->child->f = child->child->g + h(child->child, end);

                   if (open.find(child->child) == open.end()) {
                       open.insert(child->child);
                   }
               }
            }

        }
    }
    return "Путь не найден";
}

int main(){
    int startVertex;
    int endVertex;

    std::cin >> startVertex >> endVertex;
```

```
    VertexI* start = new VertexI(startVertex);
    VertexI* end = new VertexI(endVertex);

    std::map<int, VertexI*> connectionMap;
    connectionMap[start->name] = start;
    connectionMap[end->name] = end;

    int firstVertex;
    int secondVertex;
    double length = 0;

    while (std::cin >> firstVertex >> secondVertex >> length) {
//        std::cin >> firstVertex >> secondVertex >> length;

        if (connectionMap.find(firstVertex) != connectionMap.end() &&
                    connectionMap.find(secondVertex) != connectionMap.end())
{ //оба есть - устанавливаем связь
                      ChildI* child = new ChildI(connectionMap[secondVertex],
length);
                    connectionMap[firstVertex]->children.push_back(child);
                }
                                  else  if  (connectionMap.find(firstVertex)  !=
connectionMap.end()) {  //есть родитель - доб. ребенка, устанавл. связь
                    VertexI* node = new VertexI(secondVertex);
                    connectionMap[secondVertex] = node;

                      ChildI* child = new ChildI(connectionMap[secondVertex],
length);
                    connectionMap[firstVertex]->children.push_back(child);
                }
                                  else  if  (connectionMap.find(secondVertex)  !=
connectionMap.end()) { //есть ребенок - доб. реб., уст.связь
                    VertexI* node = new VertexI(firstVertex);
                    connectionMap[firstVertex] = node;

                      ChildI* child = new ChildI(connectionMap[secondVertex],
length);
                    node->children.push_back(child);
                }
                else { //ничего нет - добавляем
                    VertexI* node = new VertexI(firstVertex);
                    connectionMap[firstVertex] = node;
```

```
                    VertexI* node2 = new VertexI(secondVertex);
                    connectionMap[secondVertex] = node2;

                      ChildI* child = new ChildI(connectionMap[secondVertex],
length);
                    connectionMap[firstVertex]->children.push_back(child);
                }
    }

    std::string way = aStar(start, end);

    std::cout << way;
}
```