DFS 和BFS 的认识

简单的区别 一下 树和图

图的遍历

概念：从图的某一点出发访问其余顶点，且使得每一个顶点仅仅被访问一次，这过程叫图的遍历。

1、深度优先遍历：不到黄河不死心，死心后返回；关键在于所有点都被访问一遍，可能有回退（一般都有）。

2、广度优先遍历：之前我以为同层之间的相邻节点必须有边，但是这理解是错误的，实际上像树的层次遍历，只要求把其所有子节点入队，不要求相邻子节点之间有边。

树的遍历

1、前序、中序、后序遍历，其实都采用的是DFS思想，DFS思想常用递归实现；

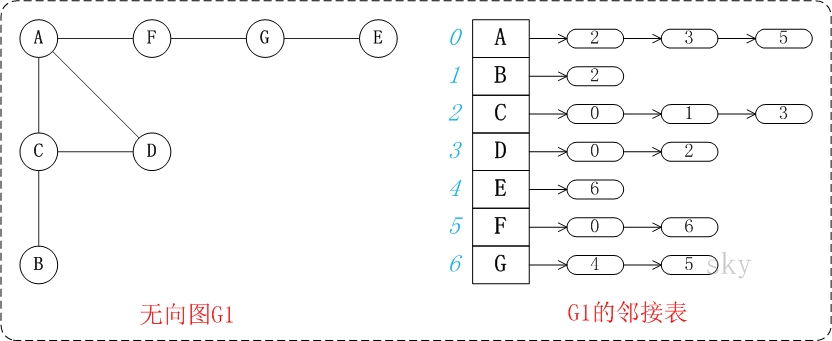
2、层次遍历，采用的是BFS思想，采用队列这种数据结构实现

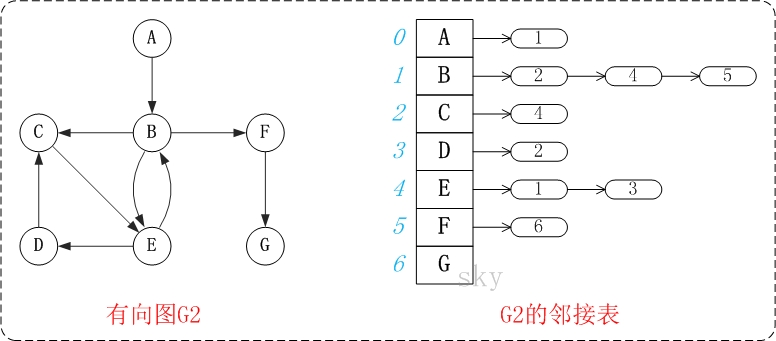
关于图的认识

定义：图(graph)是由一些点(vertex)和这些点之间的连线(edge)所组成的；其中，点通常被成为"顶点(vertex)"，而点与点之间的连线则被成为"边或弧"(edege)。通常记为，G=(V,E)

并且一般可以分为 有向图和无向图

两种存储结构 一是邻接矩阵 二是邻接表





算法思路：

从顶点V开始，访问这个顶点，然后依次从V的未被访问的邻接点出发深度优先遍历图，直至图中所有和V有路径的相通的顶点都被访问了，如果此时还有顶点未被访问，则选择图中未被访问的那个顶点作为起点，重复上述动作

是临街矩阵的存储方式

具体两种实现 看github <https://github.com/wudeape/algorithm>

struct Graph\_array {

int vexnum; //图的顶点数

int edge; //图的边数

int \*\* arc; //邻接矩阵

int kind; //0,为有向图，1，为无向图

string \* infromation; //表示每个顶点的信息

}; 自定义结构体

//进行DFS遍历，

void DFS\_store\_array(Graph\_array g,int v,bool \* & visit) {

cout << g.infromation[v] << " ";

visit[v] = true;

for (int i = 0; i < g.vexnum; i++) {//找出下一个位被访问的顶点

if (g.arc[v][i] == 0 || g.arc[v][i] == INT\_MAX) { //如果两个顶点不存在边

continue;

}

else if (!visit[i]) {//如果没有被访问，访问该结点

DFS\_store\_array(g, i, visit);

}

}

}

//调用对应的DFS函数进行图的遍历

void DFS\_array\_travel(Graph\_array g,int begin) {

bool \* visit;

visit = new bool[g.vexnum];

int i;

for (i = 0; i < g.vexnum; i++) {

visit[i] = false;

}

cout << "图的DFS遍历结果：" << endl;

DFS\_store\_array(g,begin - 1,visit);

//如果图是非联通同，那么我们这里还需要对每个顶点遍历一次，保证

//全部顶点都被访问了

for (i = 0; i < g.vexnum; i++) {

if (!visit[i])

DFS\_store\_array(g, i,visit);

}

}

广度优先遍历

BFS就是我们所说的广度优先搜索，它的思路就是：假设从图中的顶点V出，在访问了v之后，依次访问v的各个未被访问的邻接点，然后，分别从这些邻接点出发，依次访问他们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的邻接点”先被访问，直至图中所有的顶点都被访问到为止，防止出现非连通图的情况，我们需要最后遍历一遍，看是否所有的点都被访问了，如果有未被访问的点，那么就把该点作为一个新的起点

//使用邻接矩阵完成图的遍历

struct Graph\_array {

int vexnum; //图的顶点数

int edge; //图的边数

int \*\* arc; //邻接矩阵

int kind; //0,为有向图，1，为无向图

string \* infromation; //表示每个顶点的信息

};

//使用邻接矩阵表示的图

void createGraph\_by\_array(int \*\*edge, Graph\_array & g) {

int i = 0;

g.arc = new int\*[g.vexnum];//为邻接矩阵开辟空间

for (i = 0; i < g.vexnum; i++)

{

g.arc[i] = new int[g.vexnum];

for (int j = 0; j < g.vexnum; j++)

g.arc[i][j] = 0;

}

for (i = 0; i < g.edge; i++)

{

//对矩阵进行赋值

g.arc[edge[i][0] - 1][edge[i][1] - 1] = 1;

}

}

//打印邻接矩阵

void print\_array(Graph\_array g) {

int i = 0;

for (i = 0; i <g.vexnum; i++) {

//cout << g.infromation[i] << " ";

for (int j = 0; j < g.vexnum; j++) {

cout << g.arc[i][j] << " ";

}

cout << endl;

}

}

//调用对应的BFS函数进行图的遍历

void BFS\_array\_travel(Graph\_array g, int begin) {

bool \* visit;

visit = new bool[g.vexnum];

int i;

for (i = 0; i < g.vexnum; i++) {

visit[i] = false;

}

cout << "图的BFS遍历结果：" << endl;

//通过我们之前说的算法思路，我们可以知道

//我们需要使用先进先出的数据存储结构实现我们的BFS，其实那就是队列

queue<int> q;

for (int v = 0; v < g.vexnum; v++) {//这重循环是为了保证非连通同的情况下，每个顶点都可以被访问

if (!visit[(begin-1 + v) % g.vexnum])//注意起点不一定是v1

{

cout << g.infromation[(begin - 1 + v) % g.vexnum] << " ";

visit[(begin - 1 + v) % g.vexnum] = true;

q.push((begin - 1 + v) % g.vexnum);//初始化我们的队列

while (!q.empty())

{

int u = q.front();

q.pop();

for (int j = 0; j < g.vexnum; j++) {

if (g.arc[u][j] == 0 || g.arc[u][j] == INT\_MAX) { //如果两个顶点不存在边

continue;

}

else if (!visit[j] ) {//先访问所有和u相连的顶点，并且把它们加入队列

cout << g.infromation[j] << " ";

visit[j] = true;

q.push(j);

}

}

}

}

}

cout << "完成" << endl;

}