图论基础知识

在本文中，我们将介绍图论的基础知识。 **图论**是数学的一个分支，旨在研究与称为图的结构有关的问题。

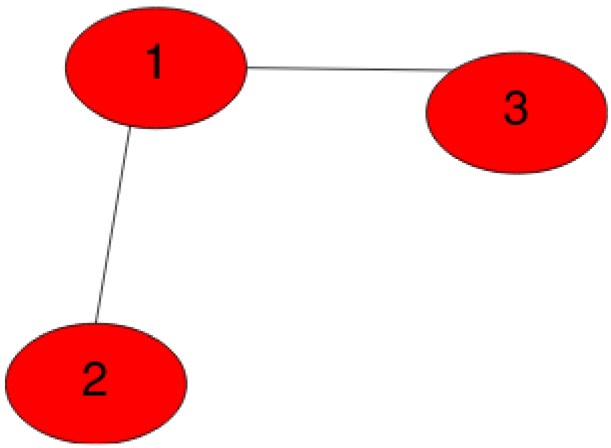
什么是图?

图是由映射到一组边的一组节点/顶点组成的结构。 他们如何一起绘制地图?

节点/顶点是图形上的一个点，边通过线段将两个顶点连接在一起。

我们用=>　**edge =（u，v）**表示连接一对顶点的**边** .

下图表示具有**3**个顶点和**2个**边的图。



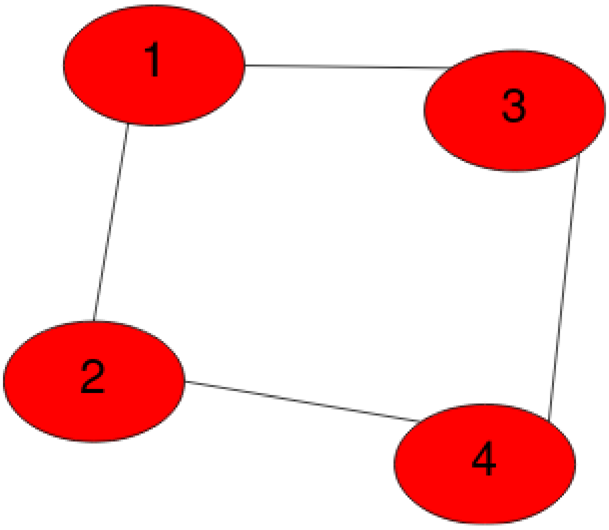
图的类型

分类两种类型的图表： **无向**和**有向** 。

无向图

无向图是其中每个边缘都表示为*无序对* **e =（1，2）的图** 。 这意味着顺序对我们来说无关紧要，因为 **（1，2）**与**（2，1）相同** 。

所有在顶点之间没有箭头的图称为无向图。

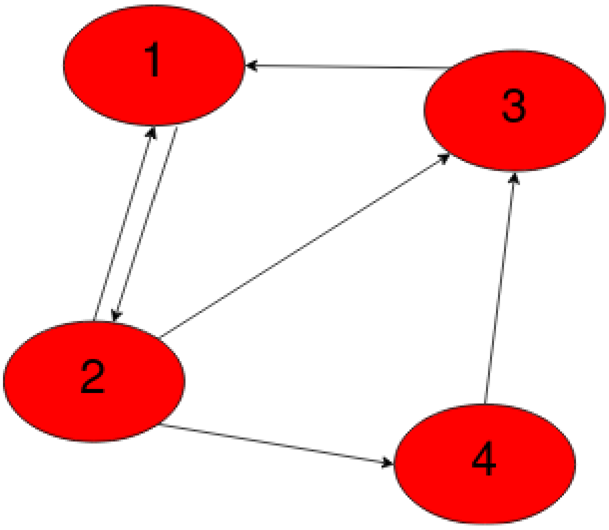


有向图

有向图是每个边都指向（单向）的图。 这意味着边e1 = （1，2）**和**e2 = （2，1）不同。

它们代表介于**1**和**2**之间的相反方向的边 。

我们用箭头标记表示任何连接以显示边的方向。

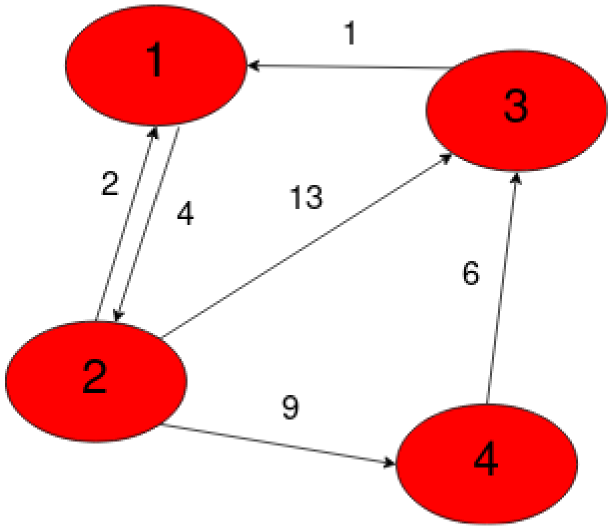


这是图的两种主要类型，它们本身具有不同的划分。 将来，我们将继续涵盖这个主题。

加权图

图还有另一种类型，称为**加权图**， 我们在很多问题中都使用了它。

每个边都有与之关联的权重/成本。



图的实现和存储

三种实现图的方法。

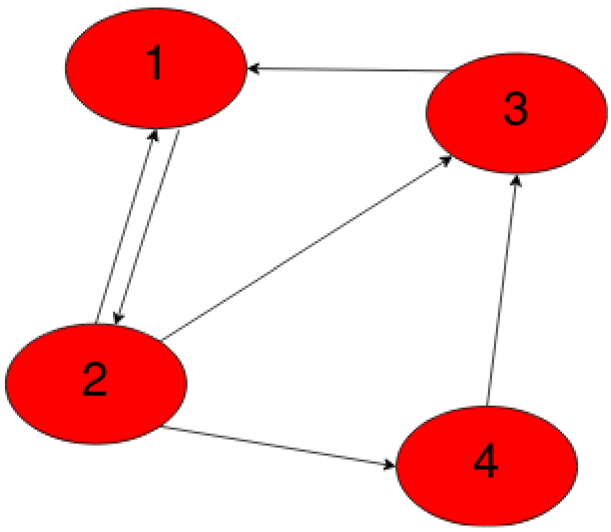
邻接矩阵

邻接表

动态数组

使用邻接矩阵

对于下面的有向图，让我们找出邻接矩阵。



我们表示只有在从i到j的有*向边的情况下*，才使元素**adj [i] [j] = 1的**矩阵。

	Node 1	Node 2	Node 3	Node 4
Node 1	0	1	0	0
Node 2	1	0	1	1
Node 3	1	0	0	0
Node 4	0	0	1	0

邻接矩阵的好处:

（1）直观、简单、好理解

（2）方便检查任意一对定点间是否存在边

（3）方便找任一顶点的所有"邻接点"（有边直接相连的顶点）

（4）方便计算任一顶点的度

对于无向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的度。

对于有向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的出度（或入度）。

邻接矩阵的局限性：时间复杂度O(n^2)

空间复杂度O(n^2)

（1）浪费空间。对于稠密图还是很合算的。但是对于稀疏图（点很多而边很少）有大量无效元素。

（2）浪费时间。要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。

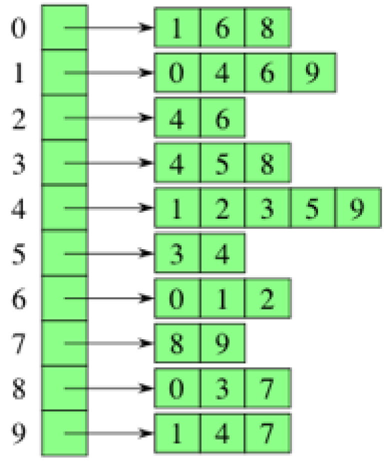
```
bool map[5001][5001];
int main()
{
    int n,m;
    int u,v;
    int q;
    while(~scanf("%d %d",&n,&m))
    {
        memset(map,false,sizeof(map));
        while(m--)
        {
            scanf("%d %d",&u,&v);
            map[u][v]=true;
        }
        scanf("%d",&q);
        while(q--)
        {
            scanf("%d %d",&u,&v);
            if(map[u][v])
            {
                printf("Yes\n");
            }else
            {
                printf("No\n");
            }
```

```
        }
    }
    return 0;
}
```

使用邻接表

邻接列表是可用于表示连接的顶点的列表。

这个想法是存储顶点的链表，该链表包括直接连接到它的所有顶点。



1、结构

这里用两个东西:

1 结构体数组edge存边，edge[i]表示第i条边,

2 head[i]存以i为起点的第一条边(在edge中的下标)

```
struct EDGE{
    int next;   //下一条边的存储下标(默认0)
    int to;     //这条边的终点
    int w;      //权值
};
EDGE edge[500010];
```

2、增边

若以点i为起点的边新增了一条，在edge中的下标为j.

那么edge[j].next=head[i];然后head[i]=j.

即每次新加的边作为第一条边，最后倒序遍历

```
void Add(int u, int v, int w) {  //起点u, 终点v, 权值w
    //cnt为边的计数，从1开始计
    edge[++cnt].next = head[u];
    edge[cnt].w = w;
    edge[cnt].to = v;
    head[u] = cnt;    //第一条边为当前边
}
```

3、遍历

遍历以st为起点的边

`for(int i=head[st]; i!=0; i=edge[i].next)`

i开始为第一条边，每次指向下一条(以0为结束标志) （若下标从0开始，next应初始化-1）

```
#include <iostream>
using namespace std;

#define MAXM 500010
#define MAXN 10010

struct EDGE{
    int next;   //下一条边的存储下标
    int to;     //这条边的终点
    int w;      //权值
};
EDGE edge[MAXM];

int n, m, cnt;
int head[MAXN];  //head[i]表示以i为起点的第一条边

void Add(int u, int v, int w) {  //起点u, 终点v, 权值w
    edge[++cnt].next = head[u];
    edge[cnt].w = w;
    edge[cnt].to = v;
    head[u] = cnt;    //第一条边为当前边
}

void Print() {
    int st;
    cout << "Begin with[Please Input]: \n";
    cin >> st;
    for(int i=head[st]; i!=0; i=edge[i].next) {//i开始为第一条边，每次指向下一条(以0为结束标志)若下标从0开始，next应初始化-1
        cout << "Start: " << st << endl;
        cout << "End: " << edge[i].to << endl;
        cout << "W: " << edge[i].w << endl << endl;
    }
}

int main() {
    int s, t, w;
    cin >> n >> m;
    for(int i=1; i<=m; i++) {
        cin >> s >> t >> w;
        Add(s, t, w);
    }
    Print();
    return 0;
}
```

使用动态数组vector存图

```
#include<vector>
struct edge{
    int to,w;
};
vector<edge> v[maxn];

void addedge(int x,int y,int z)
{
    edge e;
    e.to=y;
    e.w=z;
    v[x].push_back(e);
}
//遍历与x相连的边
for(int i=0;i<v[x].size();i++)
{
    cout<<v[x][i].to<<" "<<v[x][i].w;
}
```

图的遍历

图的遍历一般采用搜索算法

深度优先遍历

深度优先遍历，从初始访问结点出发，我们知道初始访问结点可能有多个邻接结点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始结点，访问它的第一个邻接结点。总结起来可以这样说:每次都在访问完当前结点后首先访问当前结点的第一个邻接结点。

我们从这里可以看到，这样的访问策略是优先往纵向挖掘深入，而不是对一个结点的所有邻接结点进行横向访问.

```
bool vis[maxn];
void dfs(int x)
{
    cout<<x<<endl;
    vis[x]=1;
    for(int i=0;i<v[x].size();i++)
    {
        if(!vis[v[x][i].to])
            dfs(v[x][i].to);
    }
}
for(int i=1;i<=n;i++)
    if(!vis[i])//防止图不连通
        dfs(i);
```

广度优先遍历

类似于一个分层搜索的过程，广度优先遍历需要使用一个队列以保持访问过的结点的顺序，以便按这个顺序来访问这些结点的邻接结点。代码如下

```
queue<int> q;
void bfs(int start)
{
    q.push(start);
    vis[start]=1;
    while(!q.empty())
    {
        int x=q.front();
        q.pop();
        cout<<x<<endl;
        for(int i=0;i<v[x].size();i++)
        {
        if(!vis[v[x][i].to])
            {
                vis[start]=1;
                q.push(v[x][i].to);
            }
        }
    }
}
```

例题：NOIP 2015 信息传递　　NOIP2014 寻找道路　NOIP2012 文化之旅