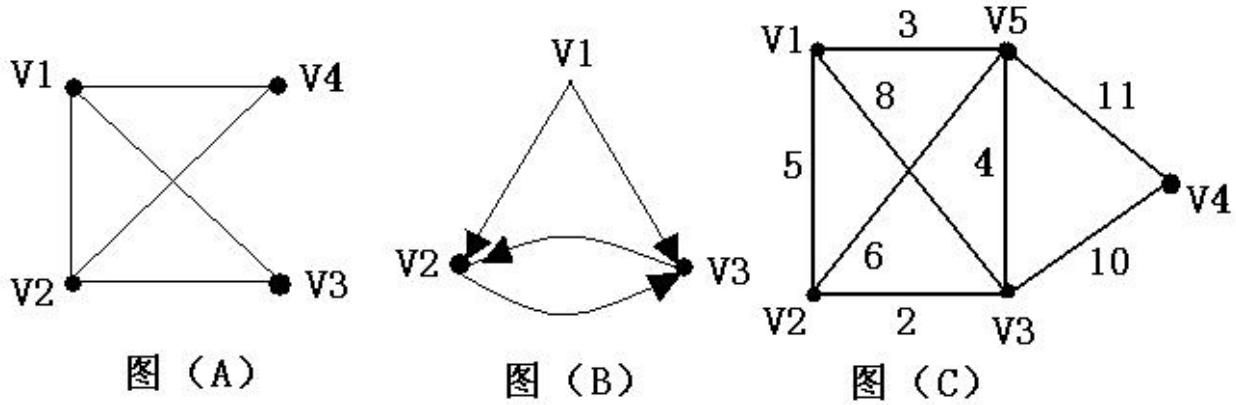
# 基础图论讲解

图论(Graph Theory )，它以图为研究对象。图论中的图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

## 图论基本概念

1. 顶点 vertex： 图中所抽象出的节点。
2. 边 edge： 连接顶点的关系，用 <u,v> 代表顶点 u 和 v 之间存在一条边。




3. 图 graph： 一系列顶点和边的集合。
4. 子图 sub-graph： 图的部分子集称为子图
5. 无向图 bidirectional graph or non-directional graph： 指边没有方向的图。
6. 有向图 directional graph： 指边具有指向性，由一个顶点指向一个顶点的图。
7. 点度 degree： 只顶点所连边的条数，无向图统一称为度数，有向图分为 出度/入度 ，出度指有 n 条边从顶点 u 指出，顶点 u 的出度为 n，同理，入度指顶点被有向边指的条数。
8. 无权图 non-weighted graph： 指边没有权值的图（大部分情况下默认权值为1）。
9. 带权图 weighted graph： 指边带有权值的的图，特别的，如果有边权为负，称为负权图。
10. 连通块/连通分量 Connected components： 通常指无向图的一个顶点集，其中所有顶点两两之间至少存在 1 条路径将顶点连通，称为1个连通分量。

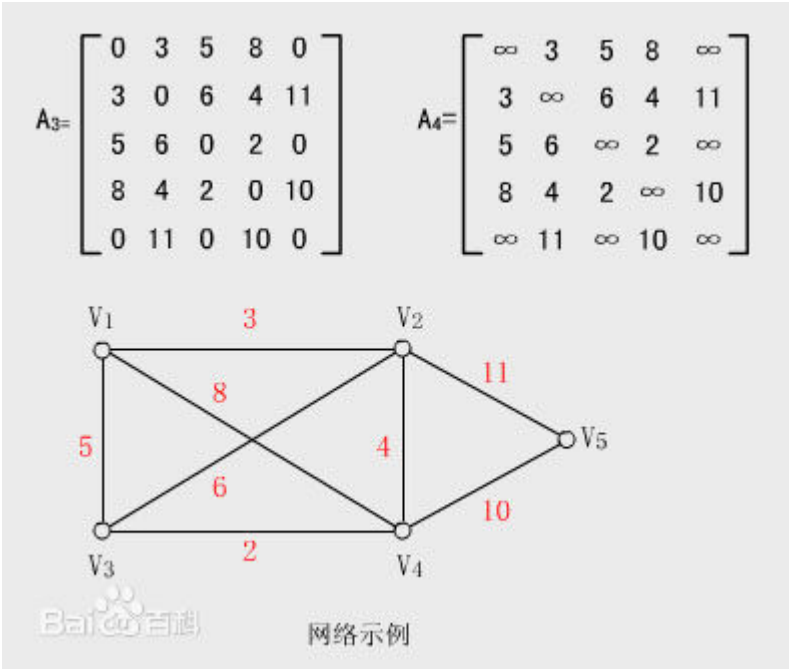
# 图的存储方式

以下皆用 V 代表顶点数，E 代表边数。

## 邻接矩阵

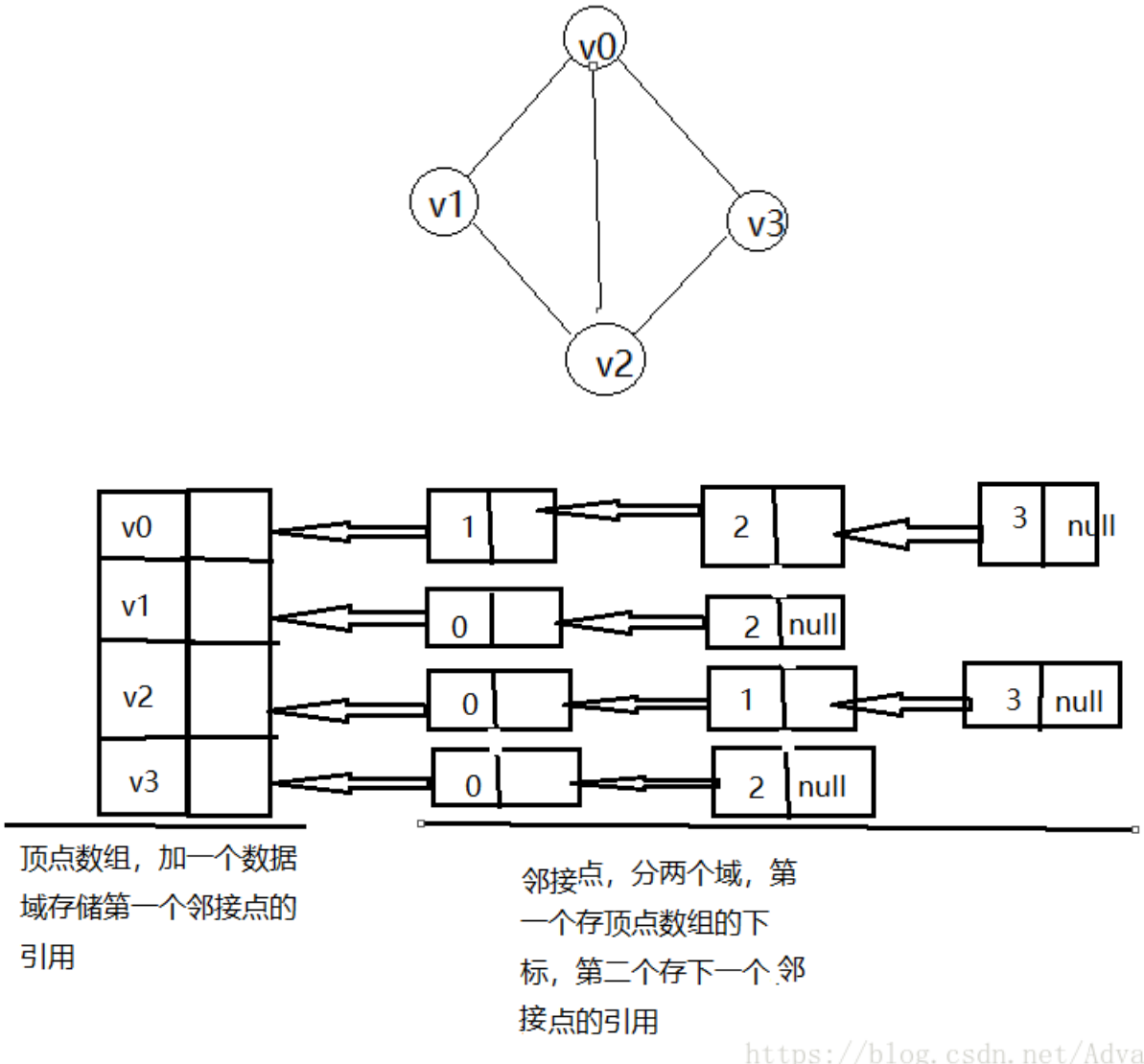
通常使用二维数组存储图矩阵，空间复杂度为O(V^2)，直接上图！

遍历的时间复杂度为O(V^2)



## 邻接表

维护 V 个单链表，第 u 个链表存储与节点 u 所相邻或指向的边的节点和权值。




https://blog.csdn.net/Advance_

有2种常见的实现方式，均可认为存图和遍历的复杂度为O(V+E)：

1. std::vector 实现，由于vector的长度不固定，不会像邻接矩阵一样浪费空间，本质上还是一个二维数组。

```
//边的结构体
struct edge{
    int to, cost;
    //此处省略构造函数
}

图的声明：vector<edge>graph[V]

加边操作：对于带权的边 <u,v> 权值为w 的无向边，有向边则只加1条
graph[u].push_back(edge(v, w));
graph[v].push_back(edge(u, w));
```

2. 链式前向星，可认为是数组版的链表实现，编写代码的复杂度相对较低，不需要用到指针，省内存，速度快。

```
struct node{int nxt,v,w;};
node edge[2*E];
int idx, head[V];

//加边
void addedge(int u,int v,int w){
    edge[idx].v = v;
    edge[idx].w = w;
    edge[idx].nxt = head[u];
    head[u] = idx++;
}
//链表初始化
void init(){
    idx = 1;
    memset(head,0,sizeof(head));
}
//遍历以节点u的相邻边
for(int i=head[u];i;i=edge[i].nxt) {
    int v = edge[i].v;  //后继点
    int w = edge[i].w;  //边的权值
    /*do something...*/
}
```

# 图的遍历

## 图的深度优先遍历

图的深度优先搜索DFS(depth first search)(~~敌法师算法~~)

它的思想：假设初始状态是图中所有顶点均未被访问，则从某个顶点v出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和v有路径相通的顶点都被访问到。 若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

显然，深度优先搜索是一个递归的过程。

```
//伪代码
bool vis[V] = {false}
def travel(u):
    if vis[u] : return
    //do something...
    for vertex v in neighbors of u:
        //do something...
        vis[v] = true
        travel(v)
        //do something...
```

```
//以vector为例
bool vis[V];
struct edge{
    int to, cost;
    //此处省略构造函数
}
vector<edge>g[V];
void dfs(int u) {
    if(vis[u]) return;
    //do something...
    for(int i=0;i<g[u].size();i++) {    //本句根据存图方式不同进行更改
        int v = g[u][i].to;
        //do something...
        vis[v] = true;
        dfs(v);
        //do something...
    }
    //do something...
}
```

## 图的广度优先搜索

广度优先搜索算法(Breadth First Search)，又称为"宽度优先搜索"或"横向优先搜索"，简称BFS。

它的思想是：从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

换句话说，广度优先搜索遍历图的过程是以v为起点，由近至远，依次访问和v有路径相通且路径长度为1,2...的顶点。

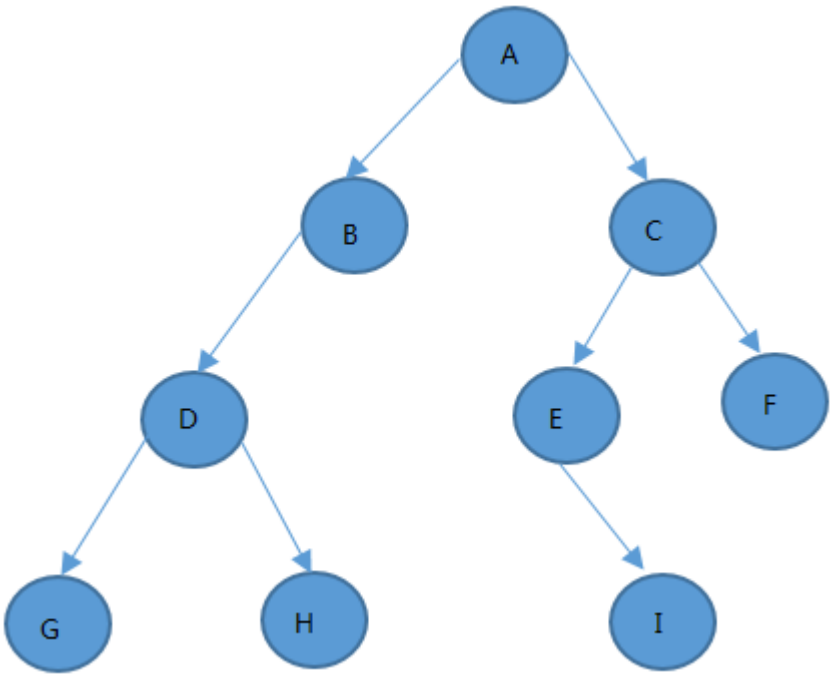
```
//伪代码
queue Q
bool vis[V] = {false}
def bfs(start):
    vis[start] = true
    start -> Q
    while Q is not empty:
        u = front of Q
        pop Q
        //do something...
        for vertex v in neighbors of u:
            if vis[v] = false:
                vis[v] = true
                v -> Q
```

```
//以vector为例
bool vis[V];
struct edge{
    int to, cost;
    //此处省略构造函数
}
vector<edge>g[V];
queue<int>Q;
void bfs(int start) {
    vis[start] = true;
    Q.push(start);
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop();
        // do something...
        for(int i=0;i<g[u].size();i++) {    //本句根据存图方式不同进行更改
            int v = g[u][i].to;
            if (!vis[v]) {
                vis[v] = true;
                Q.push(v);
            }
        }
    }
}
```

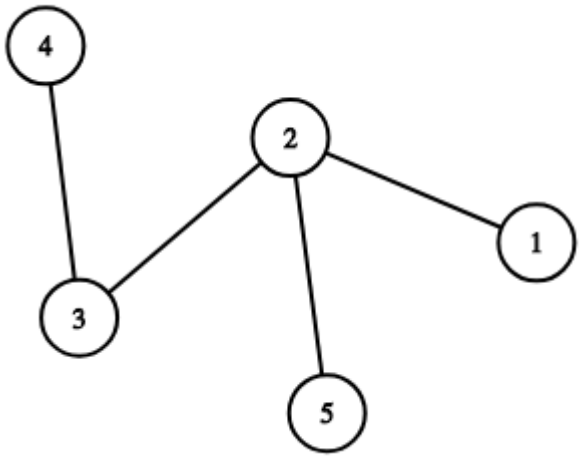
# 简单树形结构讲解

树：由 n 个节点和 n-1 条边组成的特殊的连通图，节点两两之间只存在一条路径

有根树：




无根树：




## 相关概念：

叶节点 leaf： 无根树中度数为1的节点，或有根树中没有子节点的节点

父节点 parent： 若一个节点含有子节点，则这个节点称为其子节点的父节点

孩子节点或子节点 child：一个节点含有的子树的根节点称为该节点的子节点

节点的祖先 ancestor： 从根到该节点所经分支上的所有节点

子孙： 以某节点为根的子树中任一节点都称为该节点的子孙

森林 forest： 由m（m>=0）棵互不相交的树的集合称为森林

# 参考资料

图的遍历之 深度优先搜索和广度优先搜索

基本算法——深度优先搜索（DFS）和广度优先搜索（BFS）

Wikipedia相关词条

百度百科相关词条