图

在数学上，一个**图**（Graph）是表示物件与物件之间的关系的方法，是图论的基本研究对象。一个图看起来是由一些小圆点（称为**顶点**或**结点**）和连结这些圆点的直线或曲线（称为**边**）组成的。

定义

图又有各种变体，包括简单图/多重图;有向图/无向图等，但大体上有以下两种定义方式。

二元组的定义

图G是一个二元组(V, E)，其中V称为顶点集，E称为边集。它们亦可写成$V(G)$和$E(G)$。E的元素是一个二元组数对，用(x, y)表示，其中$x, y \in V$。

三元组的定义

一个**图**，是指一个三元组(V, E, I)，其中V称为顶集（Vertices set），E称为边集（Edges set），E与V不相交；I称为关联函数，I将E中的每一个元素映射到$V \times V$。如果$I(e) = (u, v)(e \in E, u, v \in V)$那么称边$e$连接顶点$u, v$，而$u, v$则称作$e$的端点，$u, v$此时关于$e$相邻。同时，若两条边$i, j$有一个公共顶点$u$，则称$i, j$关于$u$相邻。

分类

有/无 向图

如果给图的每条边规定一个方向，那么得到的图称为有向图，其边也称为有向边。在有向图中，与一个节点相关联的边有出边和入边之分，而与一个有向边关联的两个点也有始点和终点之分。相反，边没有方向的图称为**无向图**。

简单图

一个图如果

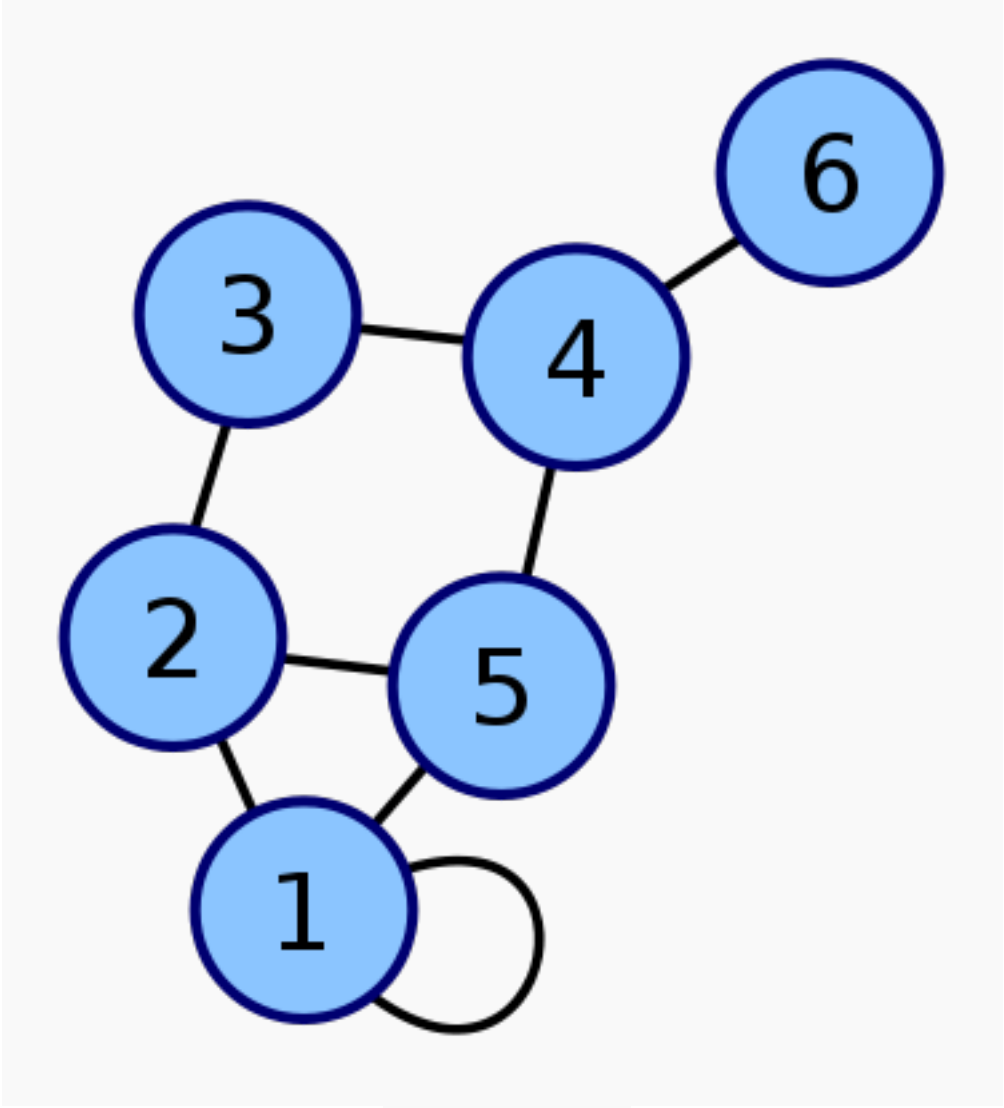
1. 没有两条边，它们所关联的两个点都相同（在有向图中，没有两条边的起点终点都分别相同）；
2. 每条边所关联的是两个不同的顶点

则称为简单图（Simple graph）。简单的有向图和无向图都可以使用以上的“二元组的定义”，但形如(x, x)的序对不能属于E。而无向图的边集必须是对称的，即如果$(x, y) \in E$，那么$(y, x) \in E$。

多重图

若允许两结点间的边数多于一条，又允许顶点通过同一条边和自己关联，则为多重图的概念。它只能用“三元组的定义”。

基本术语



在顶点1有一个环

- **阶**（Order）：图G中顶集V的大小称作图G的阶。
- **子图**（Sub-Graph）：图G'称作图G的子图如果$V(G') \subseteq V(G)$以及$E(G') \subseteq E(G)$。
- **生成子图**（Spanning Sub-Graph）：指满足条件$V(G') = V(G)$的G的子图G。

- **度**（Degree）是一个顶点的度是指与该顶点相关联的总边数，顶点v的度记作$d(v)$。度和边有如下关系：$\sum_{v \in V} d(v) = 2|E|$。
- **出度**（Out-degree）和**入度**（In-degree）：对有向图而言，顶点的度还可分为出度和入度。一个顶点的出度为d_o，是指有d_o条边以该顶点为起点，或说与该点关联的出边共有d_o条。入度的概念也类似。
- 邻接矩阵
- **自环**（Loop）：若一条边的两个顶点相同，则此边称作自环。
- **路径**（Path）：从顶点u到顶点v的一条路径是指一个序列$v_0, e_1, v_1, e_2, v_2, ...e_k, v_k$，$e_i$的起点终点为$v_{i-1}$及$v_i$；$k$称作路径的长度；$v_0 = u$，称为路径的起点；$v_k = v$，称为路径的终点。如果$u = v$，称该路径是**闭**的，反之则称为**开**的；如果$v_1, ..., v_k$两两不等，则称之为**简单路径**（Simple path，注意，$u = v$是允许的）。
- **行迹**（Trace）：如果路径$P(u, v)$中边各不相同，则该路径称为u到v的一条行迹。
- **轨道**（Track）：即简单路径。
- 闭的行迹称作**回路**（Circuit），闭的轨道称作**圈**（Cycle）。

（现存文献中的命名法并无统一标准。比如在另一种定义中，walk对应上述的path，path对应上述的track，trail对应上述的trace。）

- **距离**（Distance）： 从顶点u出发到顶点v的最短路径若存在，则此路径的长度称作从u到v的距离。若从u到v根本不存在路径，则记该距离为无穷（∞）。
- 距离矩阵
- **桥**（Bridge）：若去掉一条边，便会使得整个图不连通，该边称为桥。

图的存储表示

- 数组（邻接矩阵）存储表示（有向或无向）
- 邻接表存储表示
- 前向星存储表示
- 有向图的十字链表存储表示
- 无向图的邻接多重表存储表示

一个不带权图中若两点不相邻，邻接矩阵相应位置为0，对带权图（网），相应位置为∞。一个图的邻接矩阵表示是唯一的，但其邻接表表示不唯一。在邻接表中，对图中每个顶点建立一个单链表（并按建立的次序编号），第i个单链表中的结点表示依附于顶点vi的边（对于有向图是以顶点vi为尾的弧）。每个结点由两个域组成：邻接点域（Adjvex），用以指示与vi邻接的点在图中的位

置，链域（Nextarc）用以指向依附于顶点vi的下一条边所对应的结点。如果用邻接表存放网（带权图）的信息，则还需要在结点中增加一个存放权值的域（Info）。每个顶点的单链表中结点的个数即为该顶点的出度（与该顶点连接的边的总数）。无论是存储图或网，都需要在每个单链表前设一表头结点，这些表头结点的第一个域data用于存放结点vi的编号i，第二个域firstarc用于指向链表中第一个结点。

图的遍历

图的遍历方法有深度优先搜索法和广度（宽度）优先搜索法。

深度优先搜索法是树的先根遍历的推广，它的基本思想是：从图G的某个顶点v0出发，访问v0，然后选择一个与v0相邻且没被访问过的顶点vi访问，再从vi出发选择一个与vi相邻且未被访问的顶点vj进行访问，依次继续。如果当前被访问过的顶点的所有邻接顶点都已被访问，则退回到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点的顶点w，从w出发按同样的方法向前遍历，直到图中所有顶点都被访问。其递归算法如下：

```
Boolean visited[MAX_VERTEX_NUM];        //访问标志数组
Status (*VisitFunc)(int v);     //VisitFunc是访问函数，对图的每个顶点调用该函数
void DFSTraverse (Graph G, Status(*Visit)(int v)){
        VisitFunc = Visit;
        for(v=0; v<G.vexnum; ++v)
                visited[v] = FALSE;     //访问标志数组初始化
        for(v=0; v<G.vexnum; ++v)
                if(!visited[v])
                        DFS(G, v);      //对尚未访问的顶点调用DFS
}
void DFS(Graph G, int v){ //从第v个顶点出发递归地深度优先遍历图G
visited[v]=TRUE;        VisitFunc(v);   //访问第v个顶点
for(w=FirstAdjVex(G,v); w>=0; w=NextAdjVex(G,v,w))
//FirstAdjVex返回v的第一个邻接顶点，若顶点在G中没有邻接顶点，则返回空（0）。
//若w是v的邻接顶点，NextAdjVex返回v的（相对于w的）下一个邻接顶点。
//若w是v的最后一个邻接点，则返回空（0）。
                if(!visited[w])
                        DFS(G, w);      //对v的尚未访问的邻接顶点w调用DFS
}
```

图的**广度优先搜索**是树的按层次遍历的推广，它的基本思想是：首先访问初始点vi，并将其标记为已访问过，接着访问vi的所有未被访问过的邻接点

vi1,vi2,…, vi t，并均标记已访问过，然后再按照vi1,vi2,…, vi t的次序，访问每一个顶点的所有未被访问过的邻接点，并均标记为已访问过，依次类推，直到图中所有和初始点vi有路径相通的顶点都被访问过为止。其非递归算法如下：

```
Boolean visited[MAX_VERTEX_NUM];        //访问标志数组
Status (*VisitFunc)(int v);     //VisitFunc是访问函数，对图的每个顶点调用该函数
void BFSTraverse (Graph G, Status(*Visit)(int v)){
        VisitFunc = Visit;
for(v=0; v<G.vexnum, ++v)
                visited[v] = FALSE;
                initQueue(Q);   //置空辅助队列Q
        for(v=0; v<G.vexnum; ++v)
                if(!visited[v]){
                        visited[v]=TRUE;        VisitFunc(v);
                        EnQueue(Q, v);  //v入队列
                        while(!QueueEmpty(Q)){
                                DeQueue(Q, u);  //队头元素出队并置为u
                                for(w=FirstAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w))
                                        if(!Visited[w]){        //w为u的尚未访问的邻接顶点
                                                Visited[w]=TRUE;        VisitFunc(w);
                                                EnQueue(Q, w);
                                        }
                        }
                }
}
```