数据结构与算法

第 7 章 图

主讲：赵海燕

北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

主要内容

- 图的基本概念
- 图的抽象数据类型
- 图的存储结构
- 图的周游
- 最短路径问题
- 最小生成树
- 图知识点总结

基本概念

- 逻辑结构二元组

  B = (K, R)

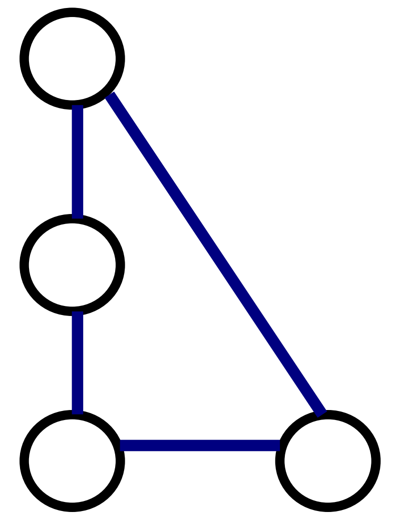
  中，研究其中一个关系 r 的情况
  - 若关系 r 不限制结点间的关系，任意一对不同结点间都允许一个关系（边）存在的话，就形成图（graph）。包括
    - 无向图、有向图、带权图、稀疏图、稠密图、完全图、连通图
- 图是 **最通用** 的数据结构，广泛应用于各种问题，树型结构和线性结构均可看作受限图

图的定义和术语

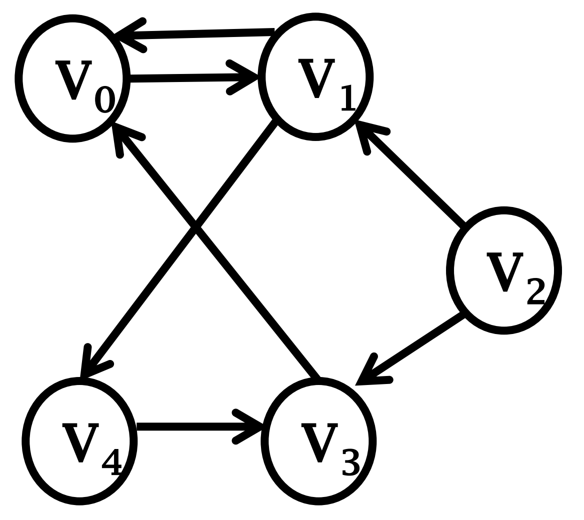
- 用G = (V, E) 来表示
  - V 是顶点 (结点： vertex)的非空有限集合
  - E 是边 (edge) 的集合，边是顶点的偶对
- 通常
  - 顶点总数记为|V|
  - 边的总数记为|E|，则|E|的取值范围在0到$|V|^2$之间

图的定义和术语

- 稀疏图 (sparse graph)
  - 稀疏度（稀疏因子）
  - 边数小于完全图的5%
- 密集图 (dense graph）

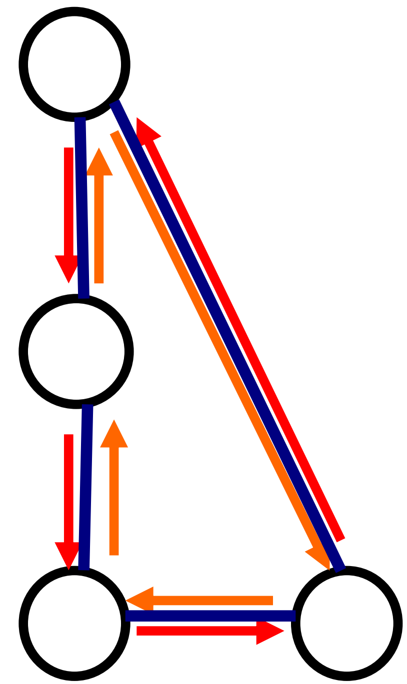


- 完全图 (complete graph)
  - 有向完全图有n·(n-1) 条边
  - 无向完全图有n·(n-1)/2条边



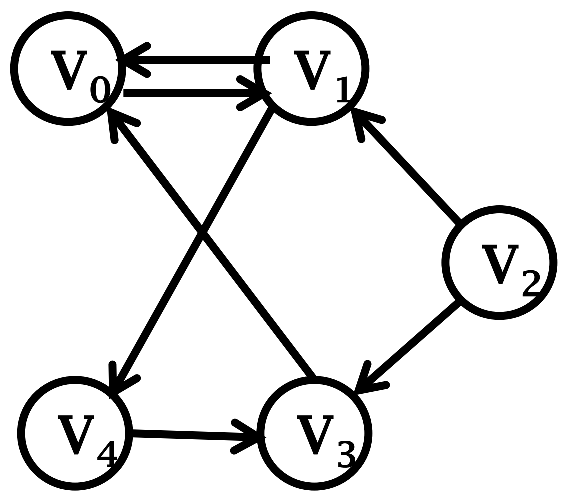
无向图

- 边涉及顶点偶对无序，
  - undirected graph
  - 双通
  - 无向图中的顶点偶对用圆括号来表示，(x, y) 和 (y, x) 代表同一条边
- 两个顶点 x 和 y 是相邻的（adjacent），若 (x, y) 是 E 中的一条边，两点彼此称为邻接点（neighbors）
  - 连接一对邻接点x、y的边被称为与顶点 x、y 相关联(incident)的边，记作(x, y)



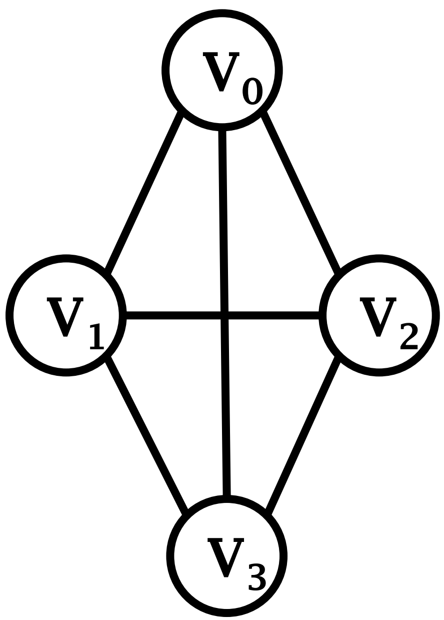
有向图

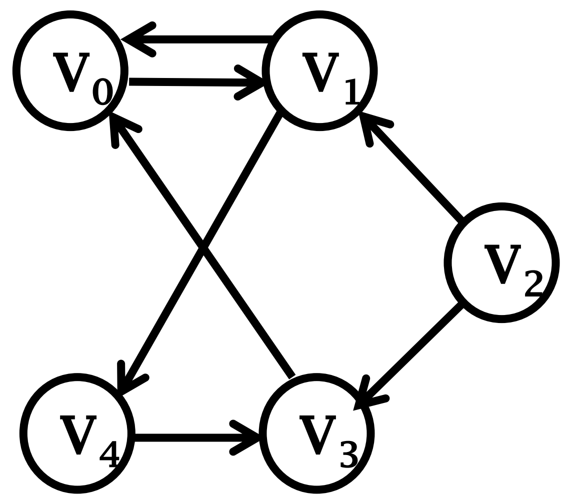
- 图中的边有向，限定从一个顶点指向另一个顶点
  - directed graph / digraph
  - 边涉及顶点的偶对是 有序 的
  - 顶点偶对用尖括号来表示， <x, y > 和 <y, x> 代表不同的边



标号图

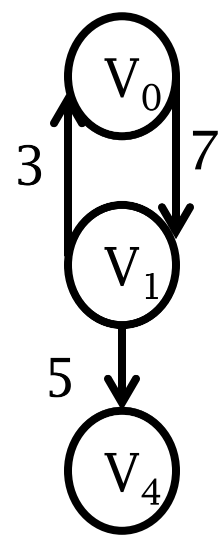
- labeled graph

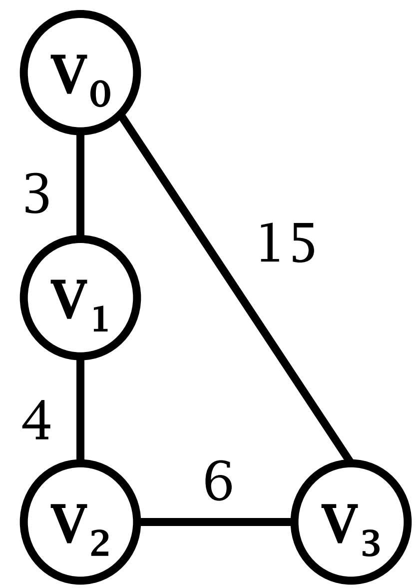




带权图

- 某些应用中，每条边都可能附有一个称其为值或权的数值（通常为非负整数），这样的图称为带权图（weighted graph）
- 带权的连通图称为网络

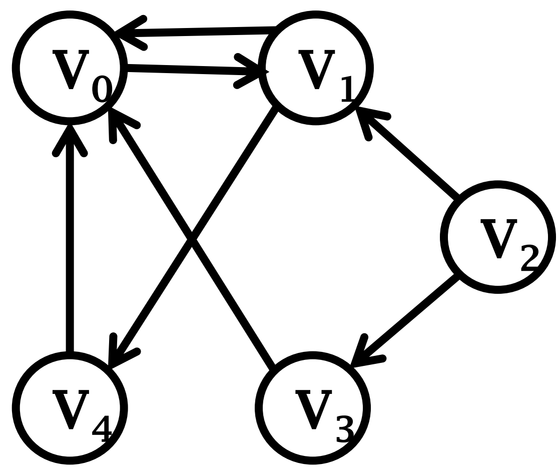


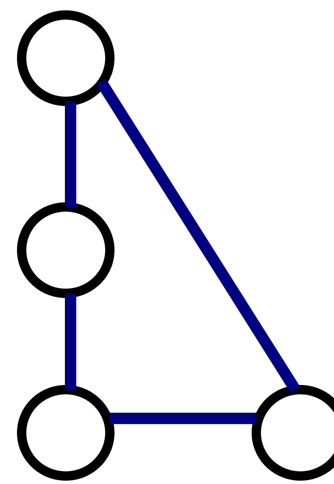


顶点的度数(degree)

- 与该顶点相关联的**边的数目**，记为TD(v)
  - 入度 (in degree)， ID(v)
  - 出度 (out degree)， OD(v)
- 图G（或有向或无向）若有 n个顶点， e条边，顶点 v_i 的度数为TD(v_i)，则有

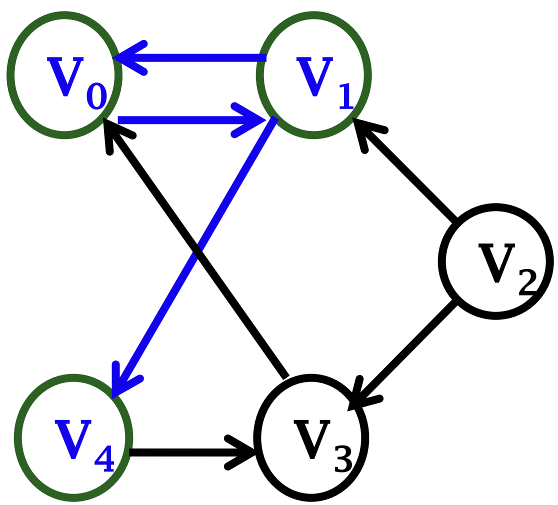
$$e = \frac{1}{2}\sum_{i=0}^{n-1} TD(v_i)$$

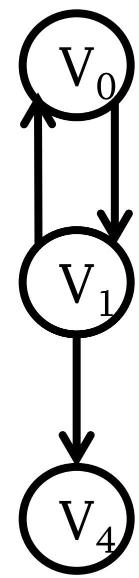




子图

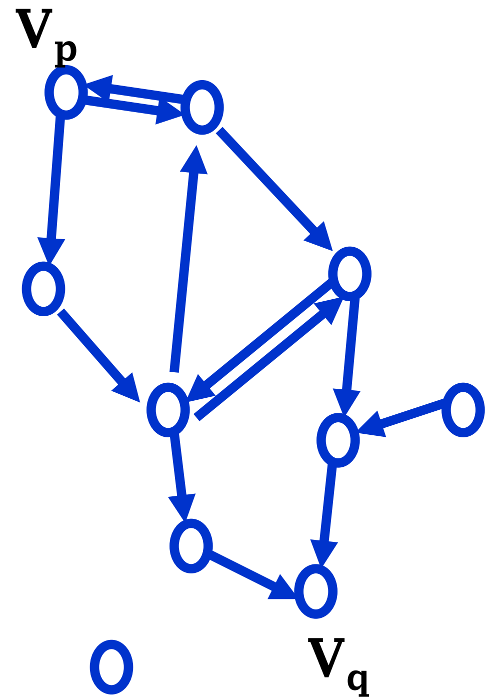
- 图G = <V,E>中，若E′是E的子集，V′是V的子集，且E′中的边仅与V′中顶点相关联，则图G′= (V′,E′)称为图G的**子图**(subgraph)





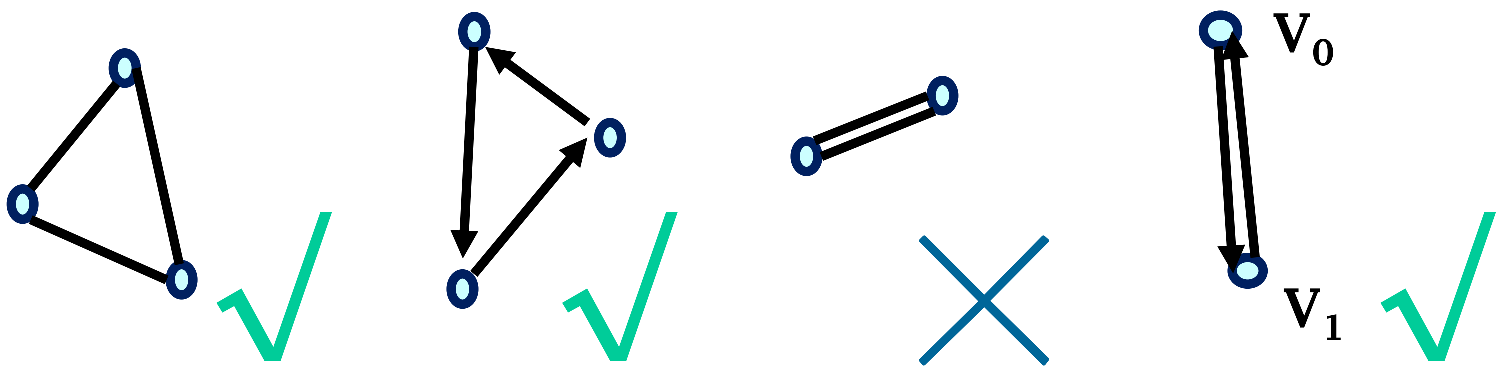
路径 (path)

- 路径(path)： G 中的一条路径是一列相邻接的不同顶点。若从v_i到v_{i+1}($1 \le i \le k$) 的边都存在，则称**顶点序列** $v_1, \ldots, v_k, v_{k+1}$ 构成一条 **长度为 k** 的 **路径**
  - 简单路径(simple path)：路径上各顶点均不同
  - 路径长度：路径所包含的边的条数



回路 (cycle)

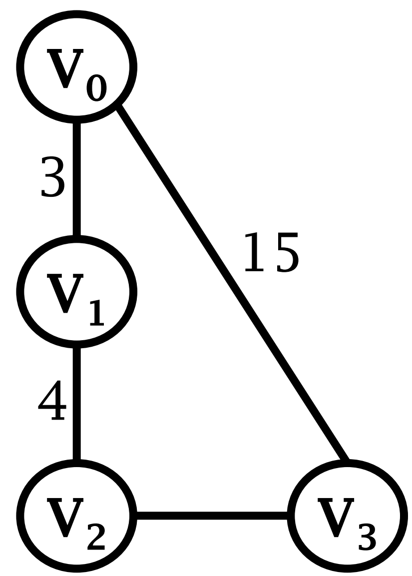
- 也称为环
  - 简单回路 (simple cycle)
  - 无环图 (acyclic graph)
    - 有向无环图（directed acyclic graph，简写为DAG)



- 无向图路径长度大于等于 3，两个结点间有平行边不构成“环”
- 有向图两条边可以构成环，$<V_0$，$V_1>$和$<V_1$，$V_0>$ 构成环

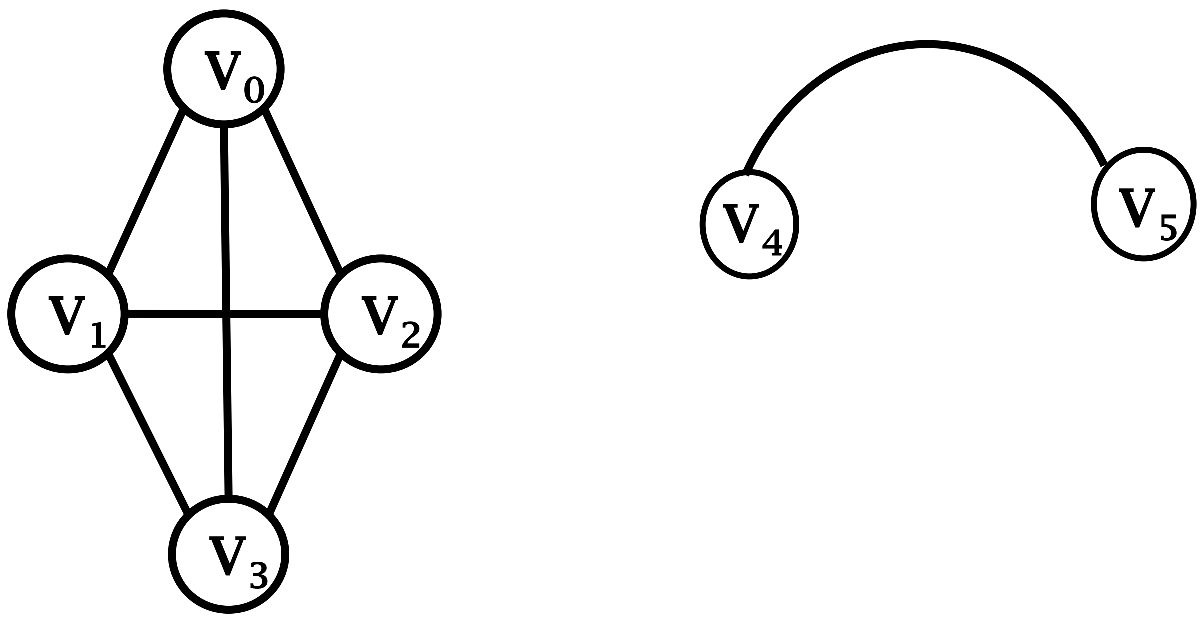
连通图

- 对无向图 G= (V，E) 而言，若从 V_1 到 V_2 有一条路径 (i.e., 从 V_2 到 V_1 也一定有一条路径)，则称 V_1 和 V_2 是 连通的 (connected)
- 若图G中任何两个顶点x、y之间都至少存在一条路径， G是连通图



无向图连通分量

- 连通分量（connected component）： 非连通无向图的极大连通子图
  - 也称连通分支

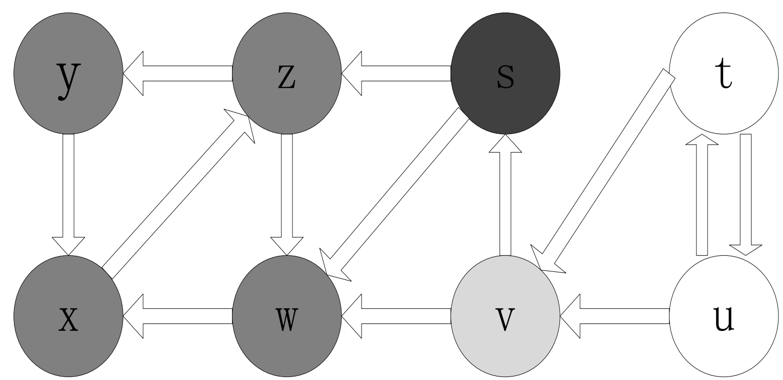


有向图的强连通

- 有向图 G (V,E)，两个不同顶点 v_i, v_j 间存在一条从 v_i 到 v_j 的有向路径，同时存在一条从 v_j 到 v_i 的有向路径，则称两个顶点强连通(strongly connected)
- 若任意两个不同顶点v_i和v_j，都同时存在从v_i到v_j的（有向）路径和从v_j到v_i的的（有向）路径（即，任意两个顶点均是强连通的），则G是强连通图

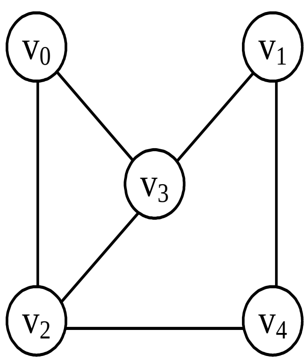
有向图的强连通分量

- 非强连通有向图的极大强连通子图，称为 强连通分量 (strongly connected components)

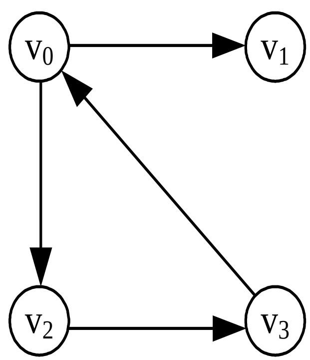


图的生成树

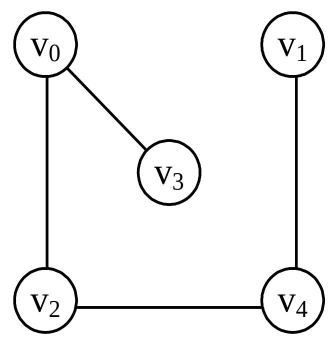
- 一个连通图的生成树是含有其全部顶点的一个极小连通子图
  - 若连通图G的顶点个数为n，则G的生成树的边数为n-1；反之，n-1条边的图不一定是生成树
  - 若无向图G的一个生成树 G′上添加一条边，则G′中一定有环，因依附于这条边的两个顶点间有另一条路径。相反，如果G′的边数小于n-1，则G′一定不连通



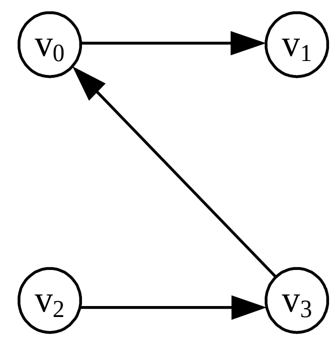
(a) 无向图G_1



(b) 有向图G_2



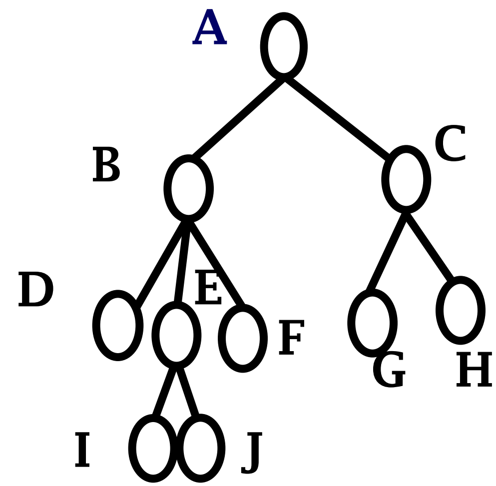
(a) 无向图G_1的生成树



(b) 有向图G_2的生成树

有根图

- 一个有向图中，若存在一个顶点 V_0，从此顶点出发有路径可达图中其它所有顶点，则称此有向图为有根的图，V_0 称作图的根
  - 树、森林



- 若一个有向图入度为0 的顶点只有1个，其余顶点的入度均为1，则称为有向树
  - 一个有向图的生成森林由若干棵有向树组成，这些树的并集包含了原图所有顶点，各有向树的边不相交

自由树

- 一棵自由树（free tree，或称无向树）
  - 不带简单回路的无向图
  - 连通
  - 有|V|-1条边
- 树是有根、有向、无环连通图

图的基本运算

- 整个图相关
  - 创建一个空图/销毁给定的图
  - 判断一个给定的图是否为空图
  - 周游
- 与顶点相关
  - 在图中查找顶点
    - 第1个顶点
    - 给定顶点的下一个顶点
    - 值给定值的顶点
  - 在图中增加顶点
  - 删除顶点及其相关的边

图的基本运算

- 与边相关
  - 删除边
  - 增添边
  - 判断是否存在一条指定的边
  - 找图中与给定顶点相邻的第1个顶点
  - 找图中与给定边相邻的下一条边

图的基本运算

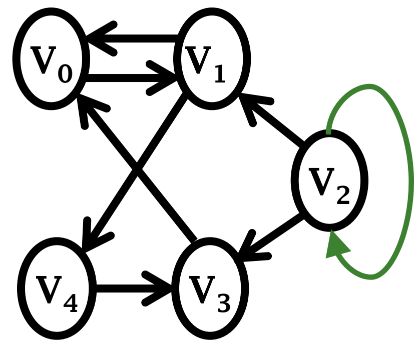
- 运算的具体实现依赖于图的具体表示
- 实际应用中根据具体情况实现其中某些运算即可，并非全部实现

图的抽象数据类型

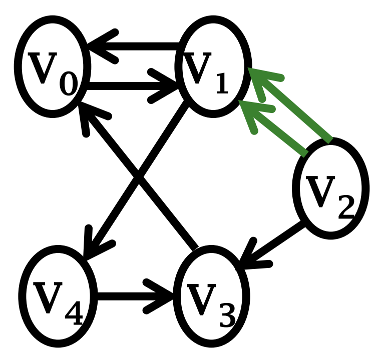
```
class Graph {                                          // 图的ADT
public:
    int  VerticesNum();                                // 返回图的顶点个数
    int  EdgesNum();                                   // 返回图的边数
    Edge FirstEdge(int oneVertex);                     // 第一条关联边
    Edge NextEdge(Edge preEdge);                       // 下一条兄弟边
    bool setEdge(int fromVertex, int toVertex,
                 int weight);                          // 添一条边
    bool delEdge(int fromVertex,int toVertex); // 删边
    bool IsEdge(Edge oneEdge);                         // 判断oneEdge是否
    int  FromVertex(Edge oneEdge);                     // 返回边的始点
    int  ToVertex(Edge oneEdge);                       // 返回边的终点
    int  Weight(Edge oneEdge);                         // 返回边的权
};
```

思考

- 为何不允许一条边的起点与终点都是同一个顶点?



- 是否存在多条起点与终点都相同的边?



图的存储结构

设|V| = n ，且各顶点依次记为$v_0, v_1, \ldots, v_{n-1}$，图的常用表示法（存储方式）有：

- 相邻矩阵表示法（Adjacency Table/Matrix Representation）
- 邻接表表示法（Adjacency List Representation）
- 十字链表表示法 （Orthogonal List）

相邻矩阵和邻接表均可用于存储有向图或无向图

- 无向图中连接某两个顶点u和v的边可用两条有向边代替：一条从u到v，一条从v到u

相邻矩阵表示法

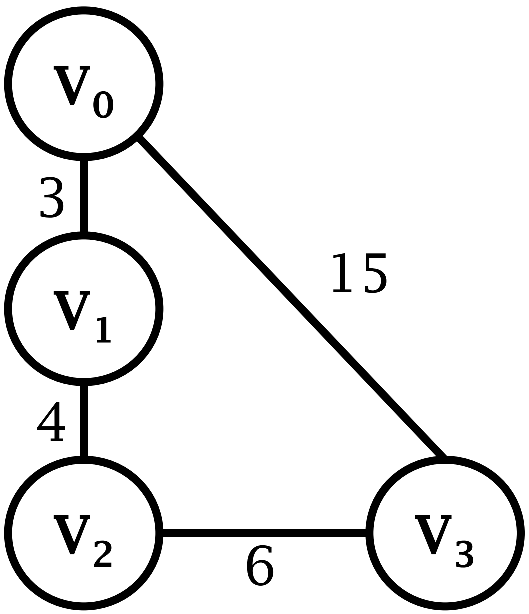
一个具有n个顶点的图G的相邻矩阵M为n×n矩阵：

$$A[i,j]=\begin{cases}1, & 若(V_i, V_j)\in E 或 \langle V_i, V_j\rangle \in E\\ 0, & 若(V_i, V_j)\notin E 或 \langle V_i, V_j\rangle \notin E\end{cases}$$

对于带权图来讲，用数字来标记每条边的权值

相邻矩阵表示法

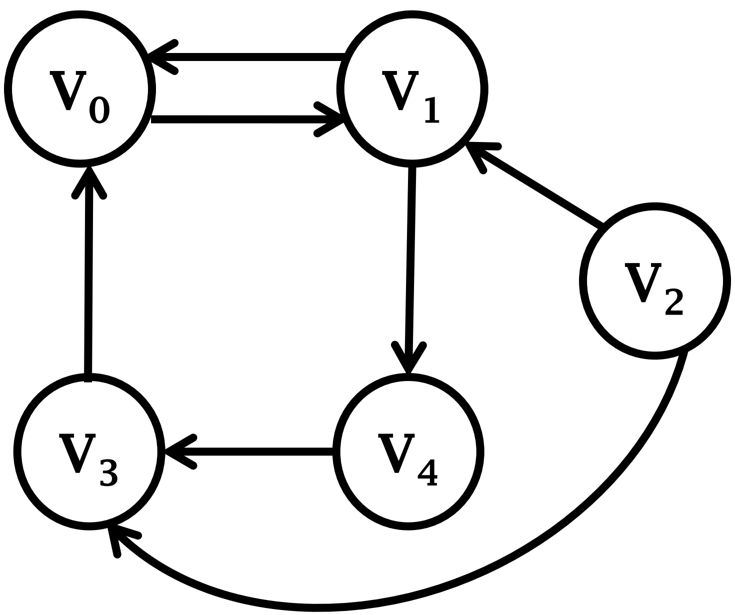
- 无向图的相邻矩阵是对角线对称的（对称矩阵）
  - 可只存相邻矩阵的下三角或上三角



$$A = \begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

相邻矩阵表示

- 有向图的相邻矩阵



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

相邻矩阵空间代价

- 相邻矩阵的空间代价为$O(n^2)$
  - 只与图中的顶点数目有关，与边的数目无关
  - 边较少，相邻矩阵就会出现大量的零元素
- 稀疏因子
  - m × n 的矩阵中，有 t 个非零元素，则稀疏因子 δ 为:

$$\delta = \frac{t}{m \times n}$$

  - 若 δ 小于 0.05 ，可认为是稀疏矩阵

邻接矩阵的运算

- 无向图
  - 对称矩阵
  - 第i行（或第i列）非零元素（或非∞元素）个数为第i个顶点的度$D(v_i)$
- 有向图
  - 第i行非零元素（或非∞ 元素）个数为第i个顶点的出度$OD(v_i)$
  - 第i列非零元素（或非∞元素）个数是第i个顶点的入度$ID(v_i)$
- 易于确定图中任意两个顶点间是否有边

相邻矩阵上的运算

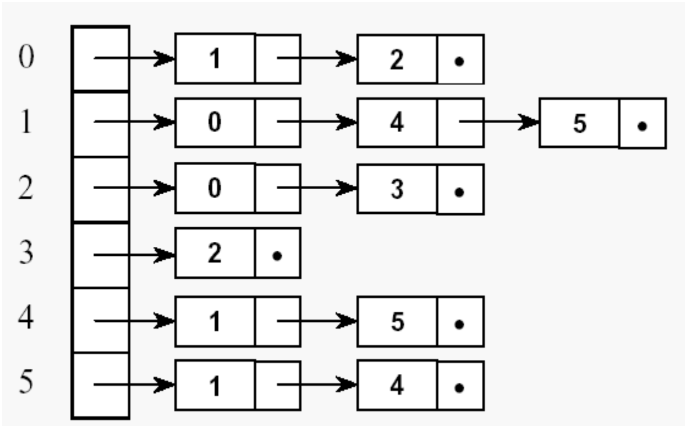
- 判断一个指定的边存在与否需要$\Theta(1)$的时间
- 找到一个指定顶点的所有的邻接点需要$\Theta(n)$的时间
- 增加或删除一条边需要$\Theta(1)$的时间
- 增加或删除一个顶点不太容易，较适合于静态的图结构

相邻矩阵

```
class Edge  {                                   // 边类
public:
    int from, to, weight;                       // 边的始点, 终点, 权
    Edge()  {                                   // 缺省构造函数
        from = -1;  to = -1;  weight = 0;       }
    Edge(int f,int t,int w) {                   // 给定参数的构造函数
        from = f;   to = t;   weight = w;     }
};
class Graph  {
public:
    int numVertex;                              // 图中顶点的个数
    int numEdge;                                // 图中边的条数
    int *Mark;                                  // 图的顶点访问标记
    int *Indegree;                              // 存放图中顶点的入度
};
```

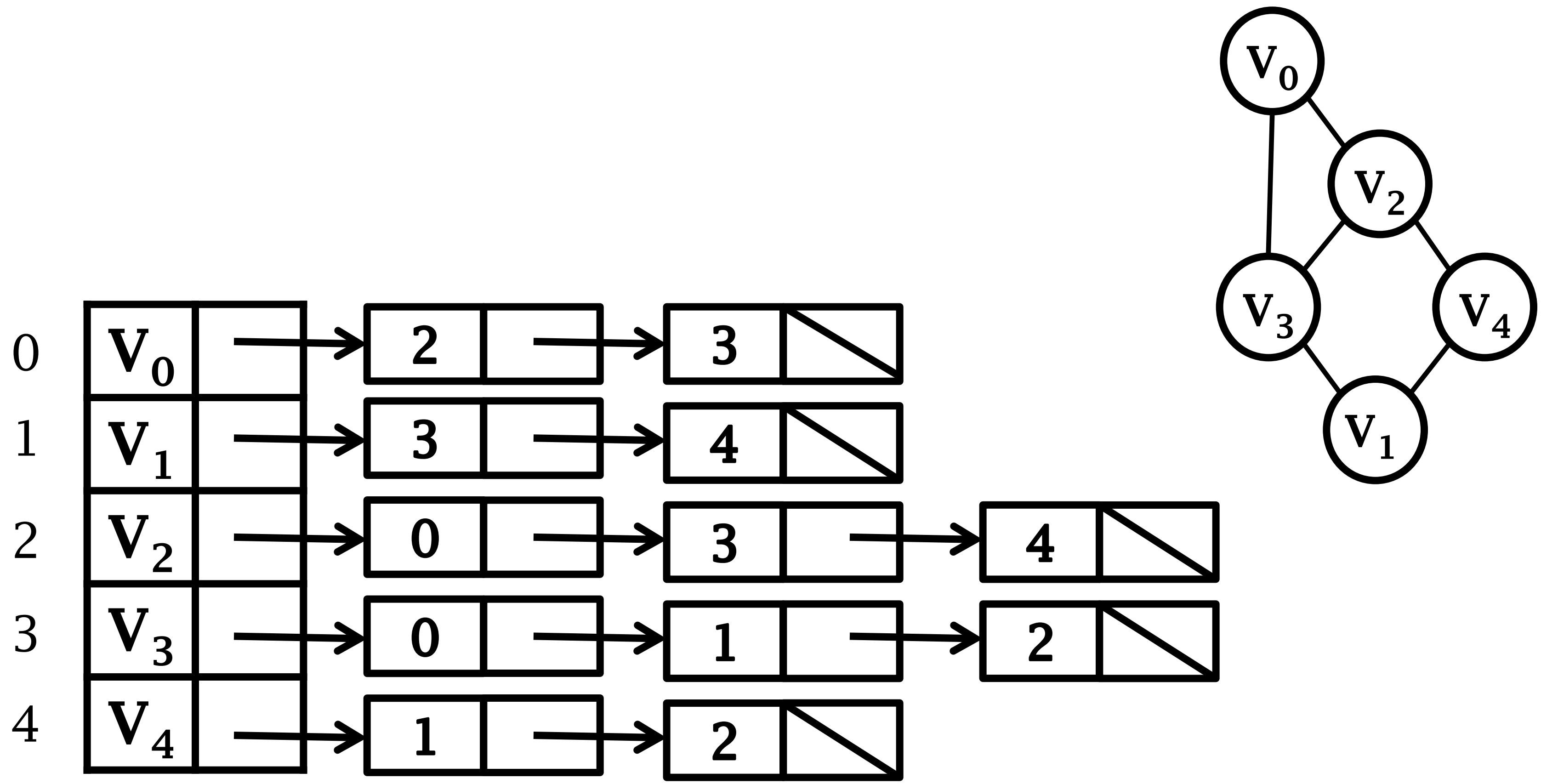
邻接表表示法

- 保存一个顺序存储的顶点表和 n 个链接存储的边表
  - 顶点表的每个表目对应于图的一个顶点，包括两个字段：
    - 数据（或指向数据的指针）
    - 指向边表的指针
  - 边表的每个表目对应于与该顶点相关联的一条边，包括两个字段：
    - 另一顶点的序号
    - 指向边表的下一表目的指针



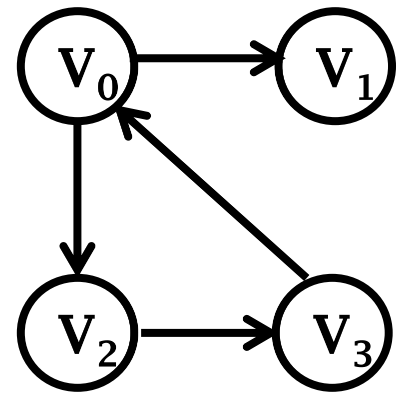
无向图的邻接表表示

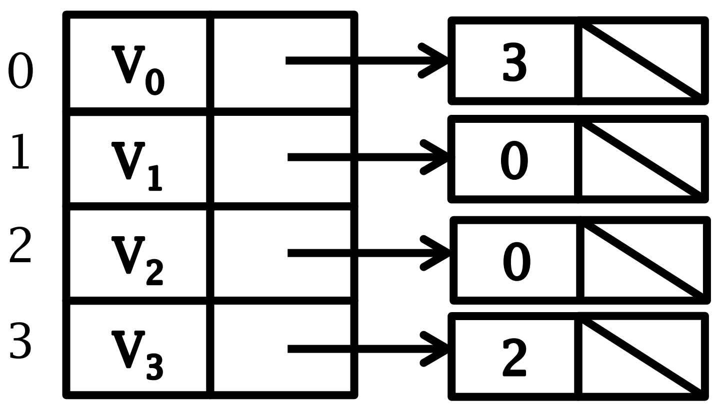
- 每条边出现2次，所占空间为 $n+2|E|$



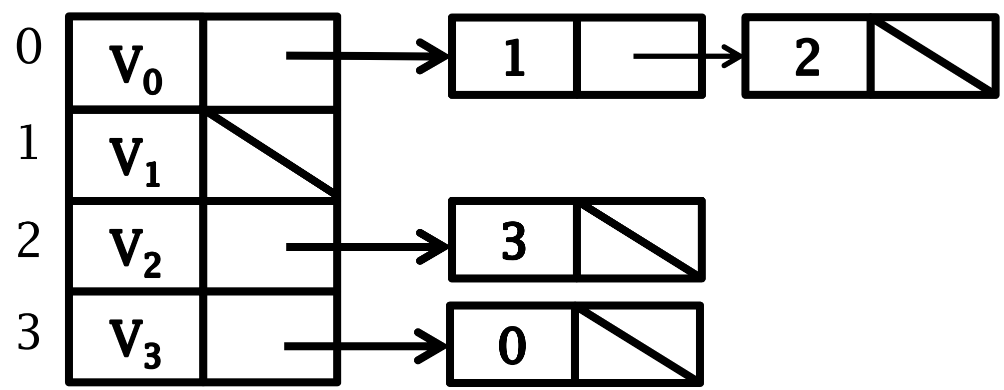
有向图的邻接表表示

- 根据需要可保存每个顶点的出边表，或入边表之一或两者
  - 需要空间为(n+|E|)，仍为O(n +|E|)





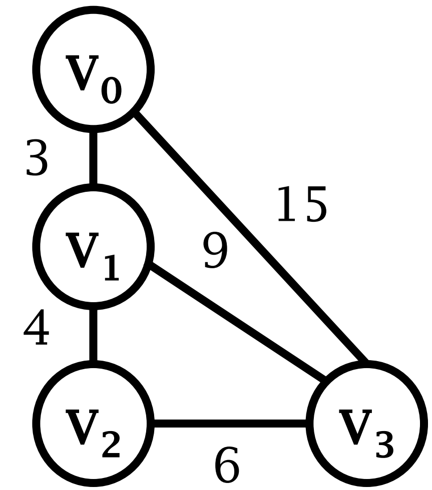
有向图的逆邻接表（入边表）

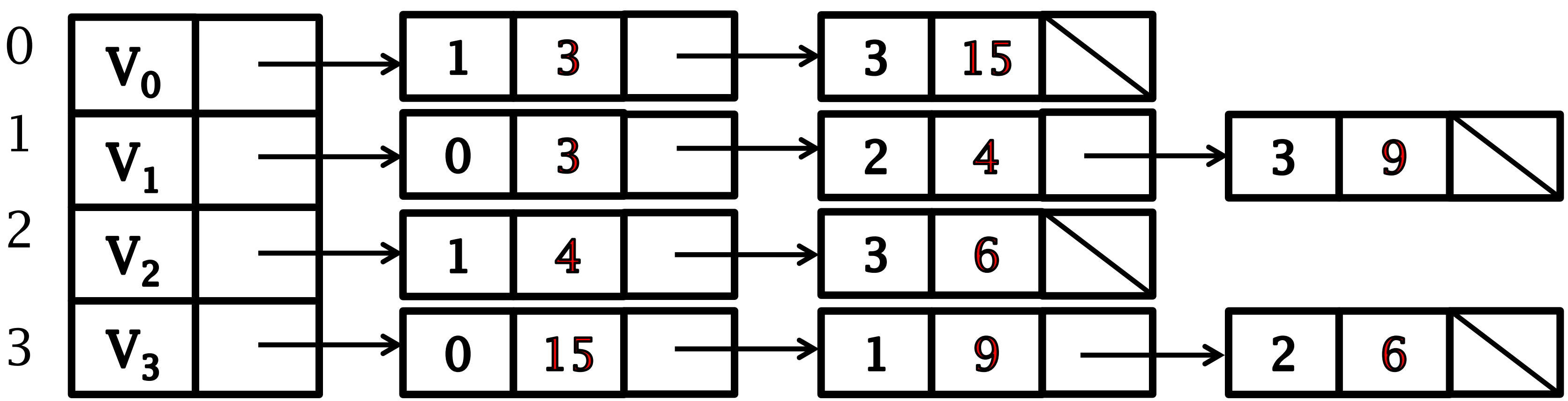


有向图的邻接表（出边表）

带权图的邻接表表示

- 边表结点中加入权值信息（增加一个字段）





邻接表的空间代价

- 与图的边数及顶点数均有关
  - 每个顶点占一个数组元素的位置（若该顶点无邻接点，则其边表无元素）
  - 每条边须出现在某个顶点的边表中
- 代价为$O(n+|E|)$
  - 每条边在其所关联的两个顶点的边表里各占一个表目，故需空间为$(n+2|E|)$
  - 对有向图而言，若只保存出边表和入边表之一，则需要空间为$(n+|E|)$
- 当$|E| \ll n^2$时，节省了存储单元，同时也因与一个顶点相关联的所有边都链接在同一个边表里，也给某些运算提供了便利

邻接表的时间代价

- 最差情况下
  - 判断一个指定的边存在与否需要Θ(n)的时间
  - 找到一个指定顶点的所有的邻接点需要Θ(n)的时间
  - 增加或删除一条边需要Θ(1)~ Θ(n)的时间
- 增加或删除顶点仍不太容易，但可采用链表代替数组来表示顶点表

十字链表

- 可看成邻接表和逆邻接表的结合，针对**有向图**的另一种链式存储结构
- 由顶点表和边表组成
  - 顶点表可以顺序结构存储，表目由3个域组成：
    - 存放顶点相关信息的数据域
    - 指向第一条以该顶点为终点的弧的指针
    - 指向第一条以该顶点为始点的弧的指针
  - 边表的每个表目对应于有向图的一条弧，由5个域组成：
    - 表示弧头（终点）顶点序号
    - 表示弧尾（始点）顶点序号
    - 指向下一条顶点以tailvex为弧尾的弧的指针
    - 指向下一条以顶点headvex为弧头的弧的指针
    - 表示弧权值等信息的info域

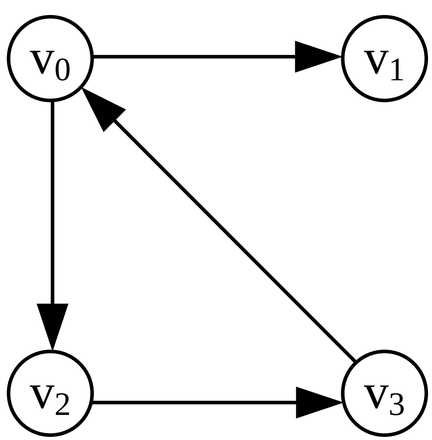
十字链表

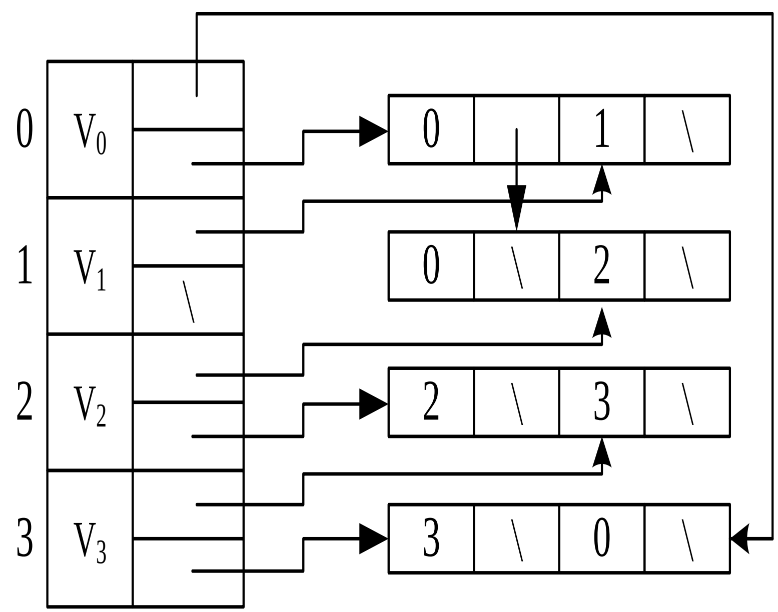
data	firstinarc
	firstoutarc

顶点结点

tailvex	tailnextarc	headvex	headnextarc	info

弧(有向边)结点

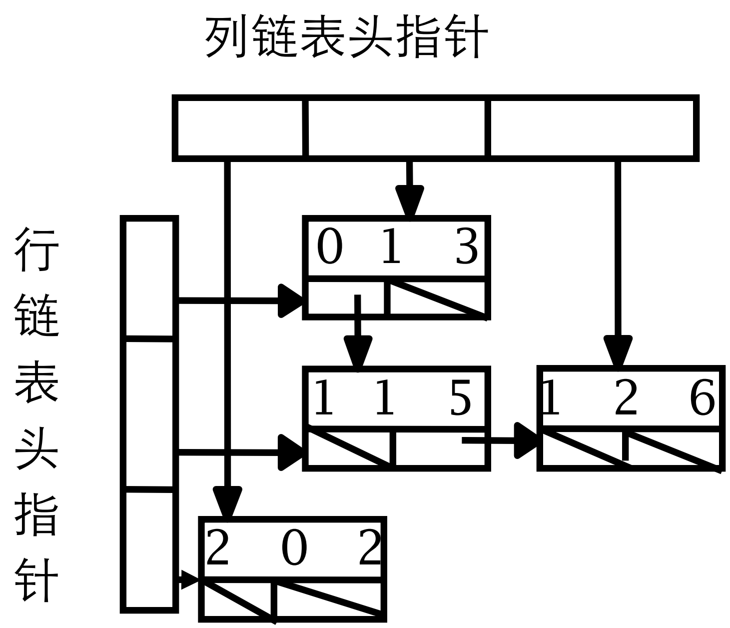




稀疏矩阵的十字链表

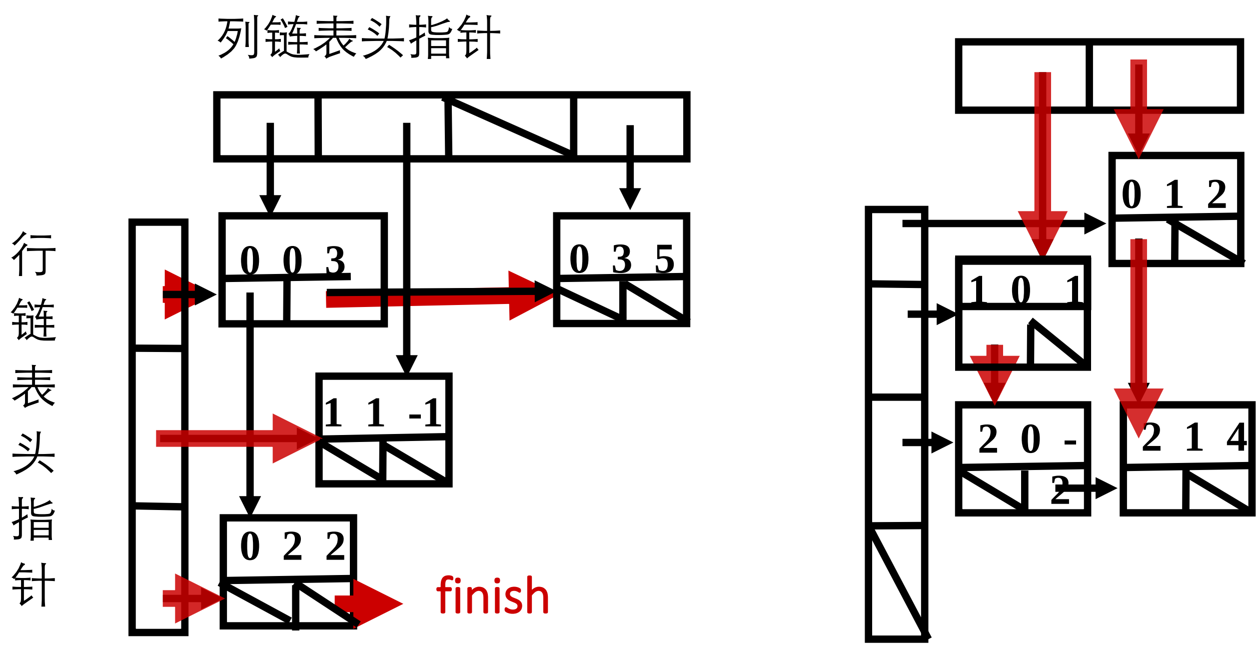
- 由两组链表组成
  - 行和列的指针序列
  - 每个结点都包含两个指针：同一行的后继，同一列的后继

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 5 & 6 \\ 2 & 0 & 0 \end{bmatrix}$$



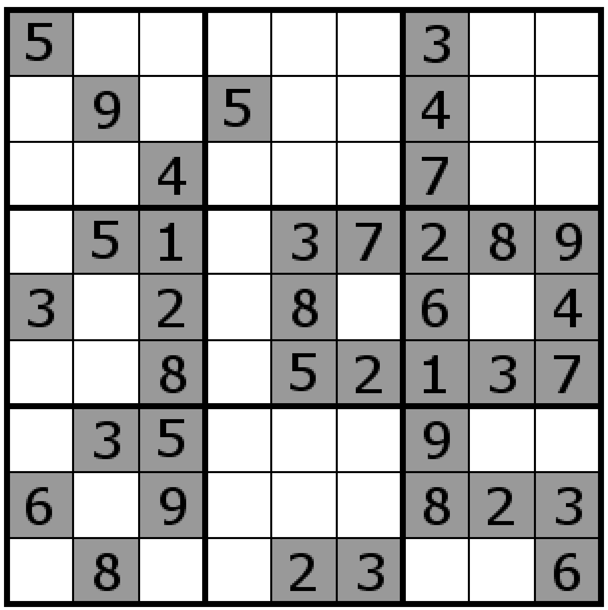
稀疏矩阵乘法

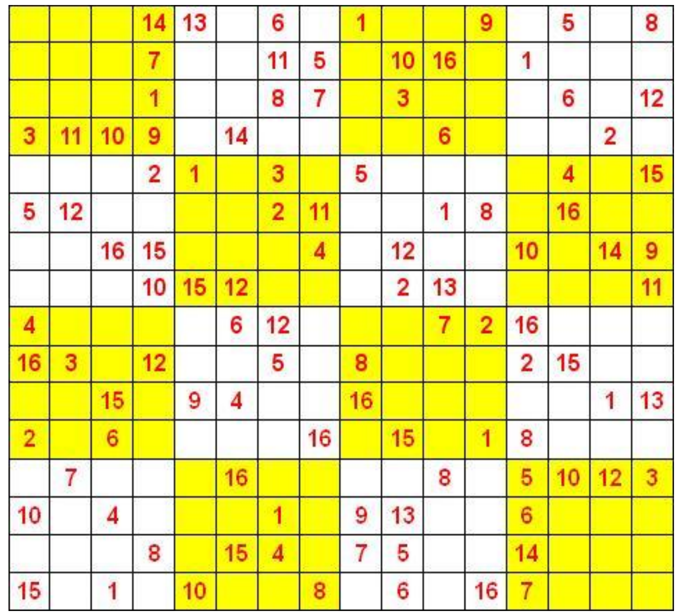
$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$



矩阵：数独 Sudoku

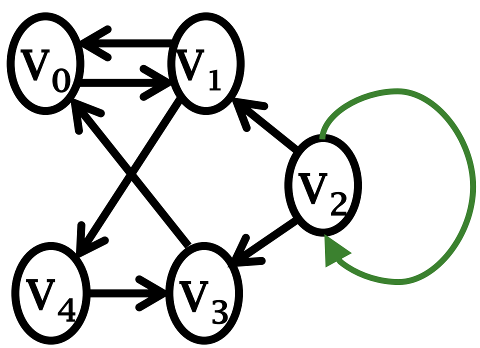
- n×n 个 n×n 的子矩阵拼接而成
  - 每行、每列的数字不重复
  - 每个子矩阵中的数字不重复

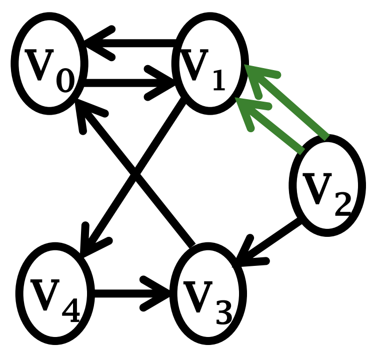




思考

- 对于以下两种扩展的复杂图结构，存储结构应做怎样的改变?





存储结构的遴选

- 哪种表示法的存储效率更高取决于图中边的数目和运算要求
- 譬如说，建立一个稀疏图的存储结构，若输入的顶点信息为顶点的编号
  - 邻接表仅需查找O(n +|E|) 次
  - 相邻矩阵却共需查找$O(n^2)$次

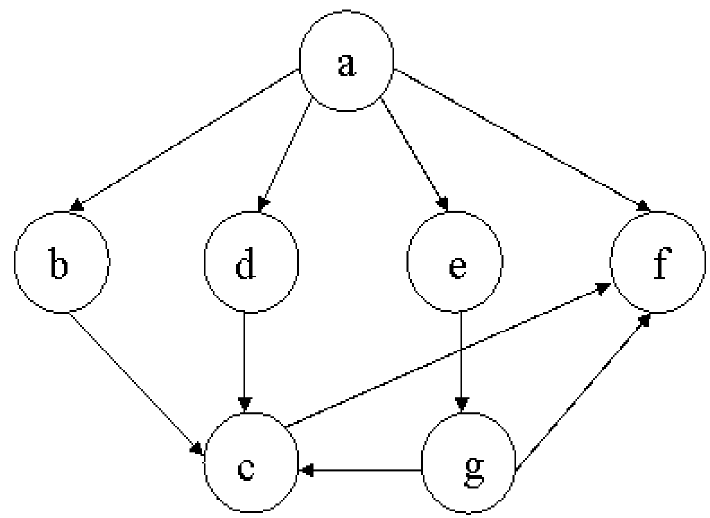
图的运算

- 图的周游
- 最短路径
- 最小生成树
- 关键路径

图的周游

- 给定一个图G和其中任一顶点V_0，从V_0出发按照某种方式系统地访问G中所有的顶点，每个顶点访问且仅被访问一次

- 连通图/强连通图



图的周游

- 典型方法
  - 从一个顶点出发，试探性访问其余顶点，须考虑到下列情况：
    - 从一个顶点出发，不能到达所有其它的顶点，如非连通图
    - 陷入死循环，如存在回路的图
  - 解决办法：顶点保留一个标志位(mark bit)
    - 算法开始时，所有顶点的标志位置为未被访问（零）
    - 周游的过程中，当某个顶点被访问时，其标志位就被标记为已访问

图的周游框架

```
// do_traverse函数用深度优先或者广度优先
void graph_traverse(Graph& G) {
    // 对图所有顶点的标志位进行初始化
    for (int i=0; i<G.VerticesNum(); i++)
        G.Mark[i] = UNVISITED;
    // 检查图的所有顶点是否被标记过，如果未被标记，
    // 则从该未被标记的顶点开始继续遍历
        for (int i=0; i<G.VerticesNum(); i++)
        if (G.Mark[i] == UNVISITED)
            do_traverse(G, i);
}
```

图的周游

- 图的周游算法是求解图的连通性问题、拓扑排序和关键路径等问题的基础

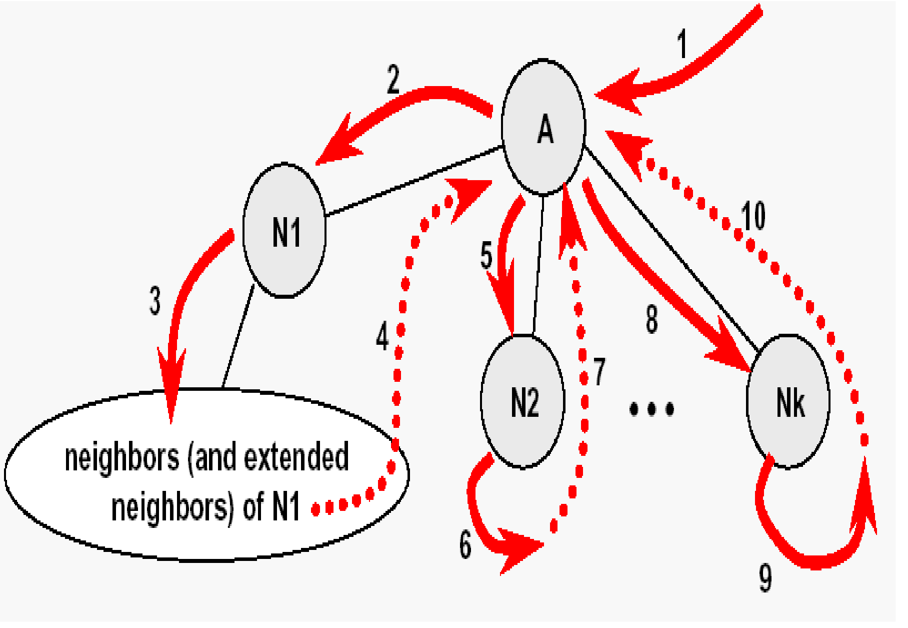
- 顶点的次序在周游中很重要，并依赖于特定的周游算法，常用的周游方法有：
  - 深度优先（depth-first search， 简称DFS）
  - 广度优先（breadth-first search，简称BFS）
  - 拓扑排序（topological sort）

深度优先周游

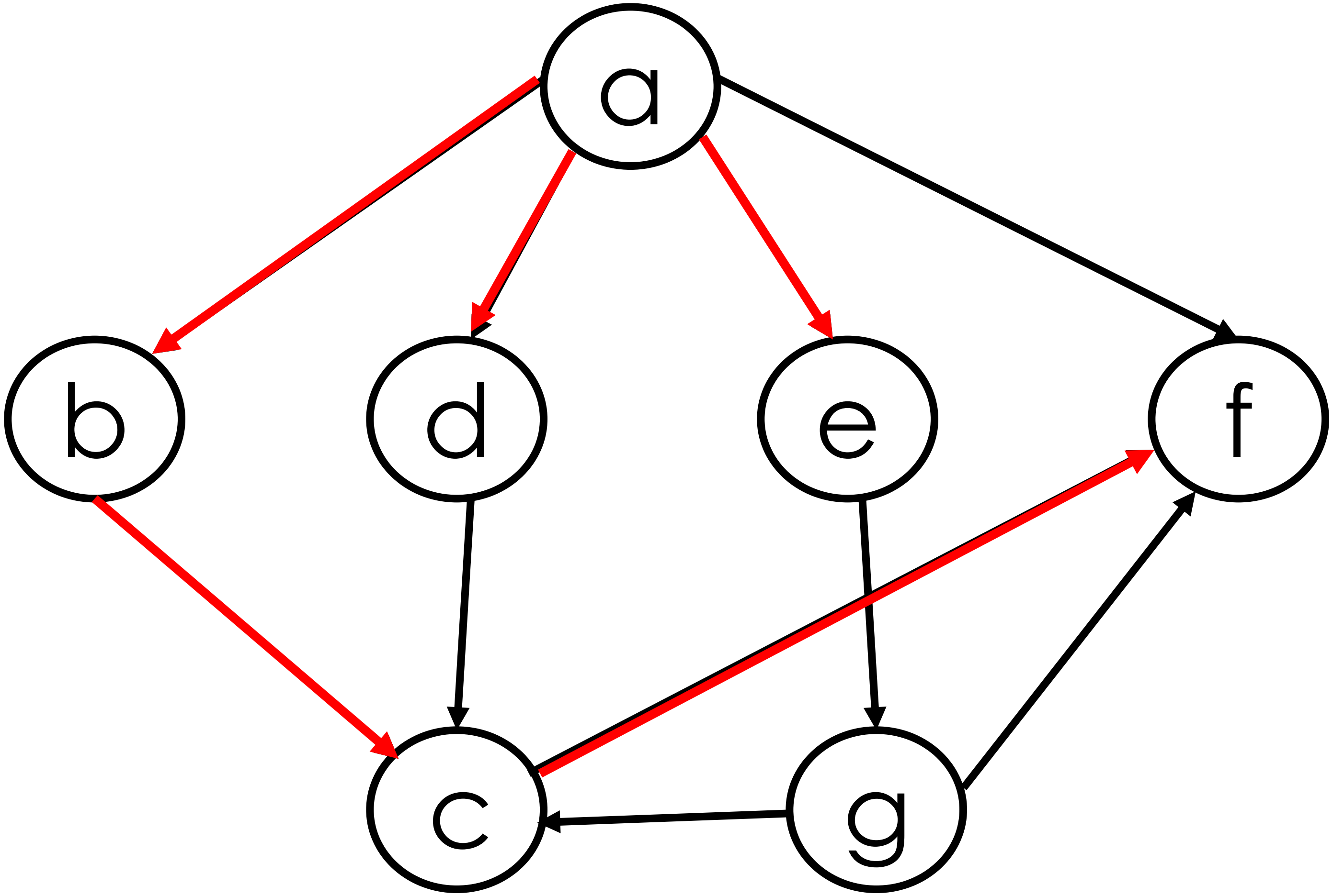
- 基本思想
  - 访问一个顶点V
  - 访问该顶点邻接到的未被访问过的顶点W，再从W出发递归地按照深度优先方式周游其邻接点
  - 遇到一个其所有邻接点都被访问过了的顶点U 时，则回到已访问顶点序列中最后一个拥有未被访问邻接点的顶点X，再从X 出发按照深度优先方式递归周游
  - 当任何已被访问过的顶点都没有未被访问的邻接点时，则周游结束
- 形成
  - 深度优先搜索序列（DFS序列）
  - 深度优先搜索树（depth-first search tree）

深度优先周游

- 假设A是最近被访问的顶点，且A有邻接点N_1, N_2, …, N_k
- 深度优先周游次序
  1. 访问邻接点N_1；
  2. 访问N_1的所有尚未访问过的邻接点；
  3. 按同样的方式，访问A 的其它邻接点

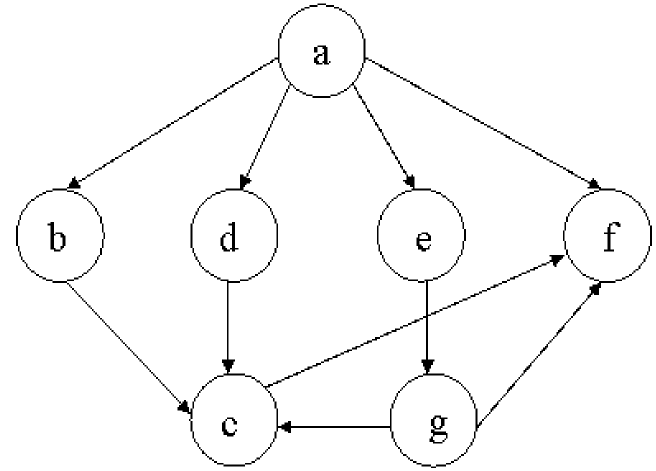


深度优先周游

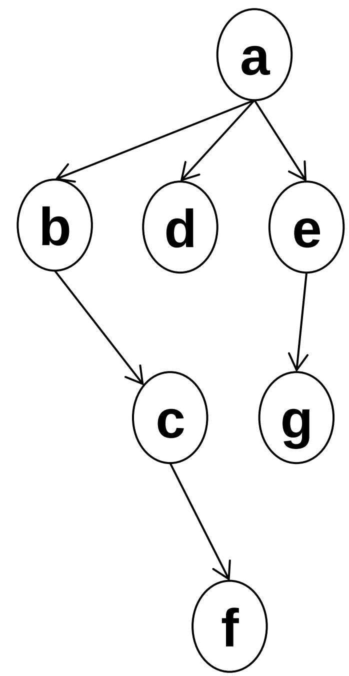


深度优先搜索的顺序是： a→b→c→f→d→e→g

深度优先周游



- 若把周游过程中所经边加以标记，则形成一棵包含图中所有顶点的树
  - 图的生成树(spanning tree)



- 图的生成树不唯一，从不同的顶点出发可能得到不同的生成树

深度优先周游的一种实现

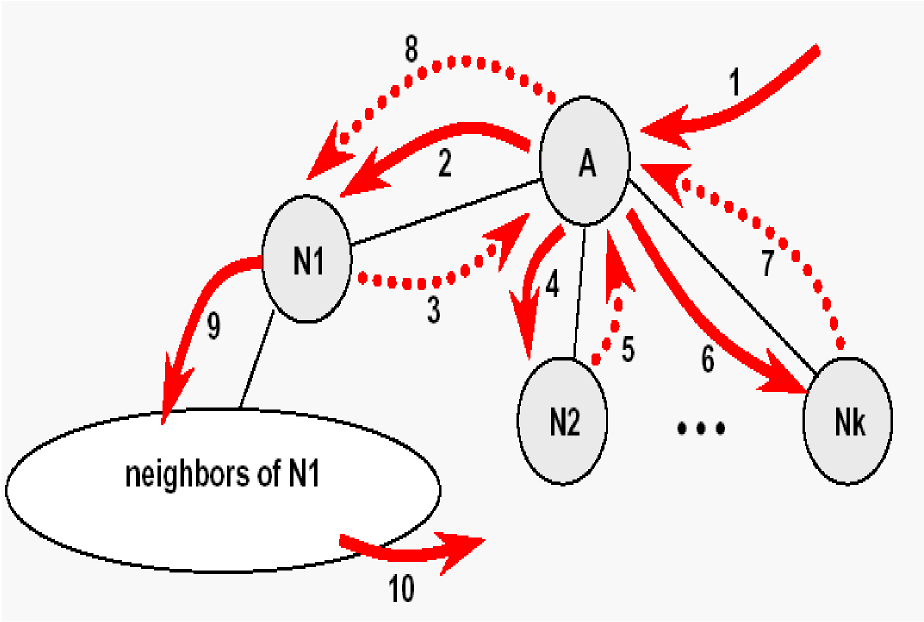
```
void DFS(Graph& G, int v)  {        // 深度优先搜索的递归实现
    G.Mark[v] = VISITED;                        // 把标记位设置为 VISITED
    Visit(G,v);                                 // 访问顶点v
    for (Edge e = G.FirstEdge(v); G.IsEdge(e);
                e = G.NextEdge(e))
        if (G.Mark[G.ToVertex(e)] == UNVISITED)
            DFS(G, G.ToVertex(e));
    PostVisit(G,v);                             // 对顶点v的后访问
}
```

广度优先周游

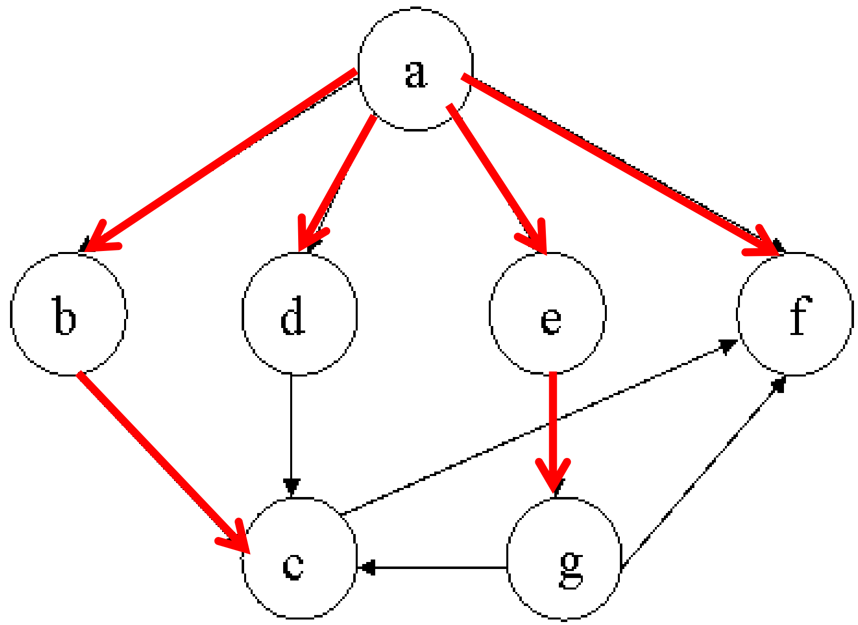
- 基本思想
  - 访问顶点V_0
  - 尔后访问V_0邻接到的所有未被访问过的顶点V_{01}，V_{02}，…V_{0i}
  - 再依次访问V_{01}，V_{02}，…V_{0i}邻接到的所有未被访问的顶点
  - 如此，直到访问遍所有的顶点
- 形成
  - 广度优先搜索序列（BFS序列）
  - 广度优先搜索树（breadth-first search tree）

广度优先周游

- 设A为是最近被访问的顶点，且A有邻接点N_1, N_2, …, N_k
- 广度优先的周游次序
  1. 访问顶点N_1，然后访问N_2，如此直到N_k;
  2. 访问顶点N_1邻接到的所有尚未访问过的顶点;
  3. 按同样的方式，依次访问N_2,…, N_k邻接到的所有尚未访问过的顶点



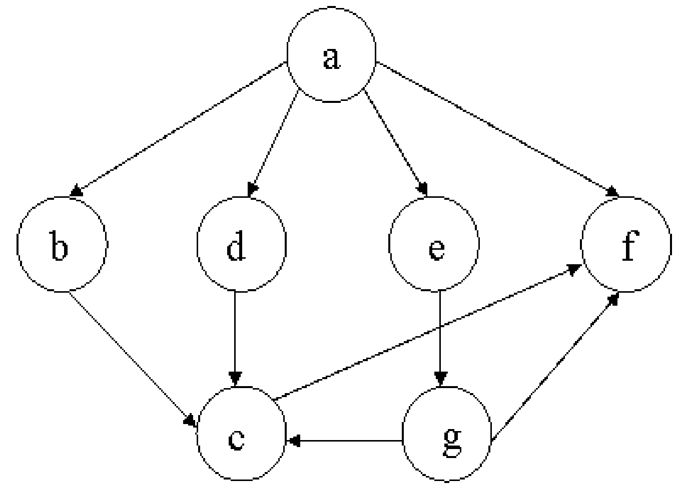
广度优先周游

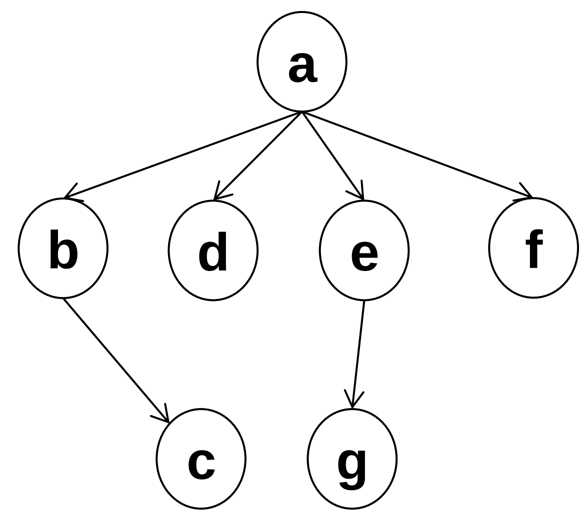


广度优先搜索序列： a , b , d , e , f , c , g

广度优先周游

- 同样地，广度优先周游也生成一棵图的生成树
- 一般情况下，广度优先周游生成的生成树与深度优先周游生成的生成树不同





广度优先周游的一种实现

```
void BFS(Graph& G, int v)  {
    using std::queue; queue<int> Q;          // 使用STL中的队列
    Visit(G,v);                              // 访问顶点v
    G.Mark[v] = VISITED; Q.push(v);          // 标记,并入队列
    while (!Q.empty()) {                     // 如果队列非空
        int u = Q.front ();                  // 获得队列顶部元素
        Q.pop();                             // 队列顶部元素出队
        for (Edge e = G.FirstEdge(u); G.IsEdge(e);
                e = G.NextEdge(e))           // 所有未访问邻接点入队
            if (G.Mark[G.ToVertex(e)] == UNVISITED){
                    Visit(G, G.ToVertex(e));
                    G.Mark[G.ToVertex(e)] = VISITED;
                    Q.push(G.ToVertex(e));
            }
    }
}
```

图搜索的时间复杂度

- 实质上为搜索每个顶点的邻接点，时间代价主要体现在从某个顶点出发，搜索其所有邻接点上
- DFS 和 BFS 每个顶点访问一次，对每一条边处理一次 (无向图的每条边从两个方向处理)
  - 采用邻接表表示，有向图总代价为 $\Theta(n + e)$，无向图为 $\Theta(n + 2e)$
  - 采用相邻矩阵表示，处理所有的边需要$\Theta(n^2)$ 时间，所以总代价为

    $$\Theta(n + n^2) = \Theta(n^2)$$

拓扑排序（Topological Sort）

- 有向图上的一种重要运算，将并在实际中被广泛应用：
  - 课程间的先修关系
  - 术语表中各技术术语定义的依赖关系
  - 课程或书中各主题间的组织
  - 工程的施工图
  - 产品的生产流程图

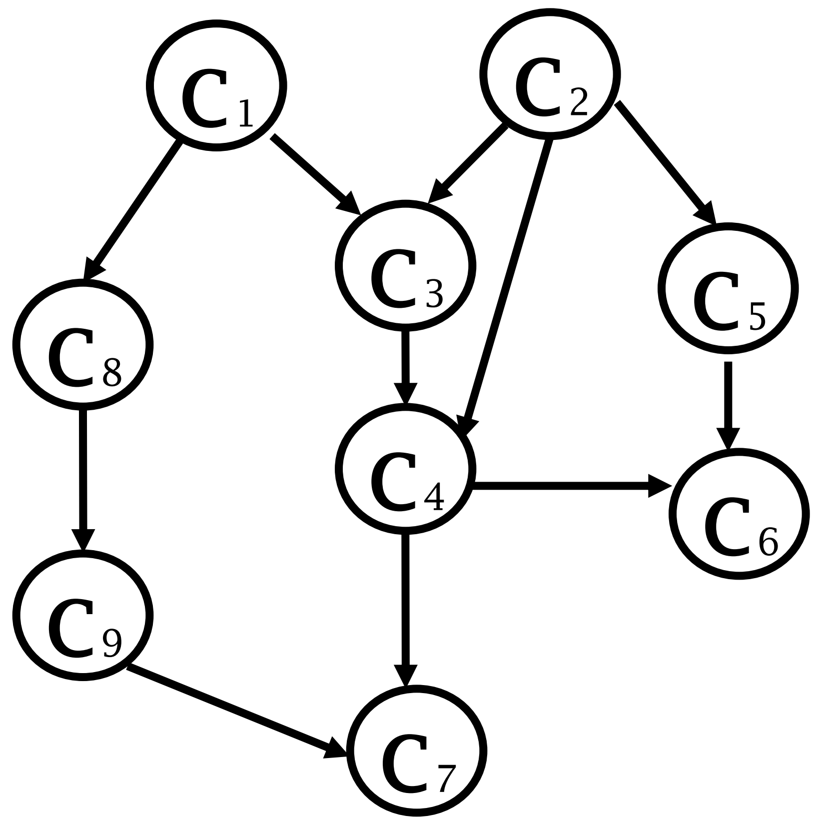
将一个有向无环图中所有顶点在不违反先决条件关系的前提下排成线性序列的过程称为拓扑排序

拓扑序列

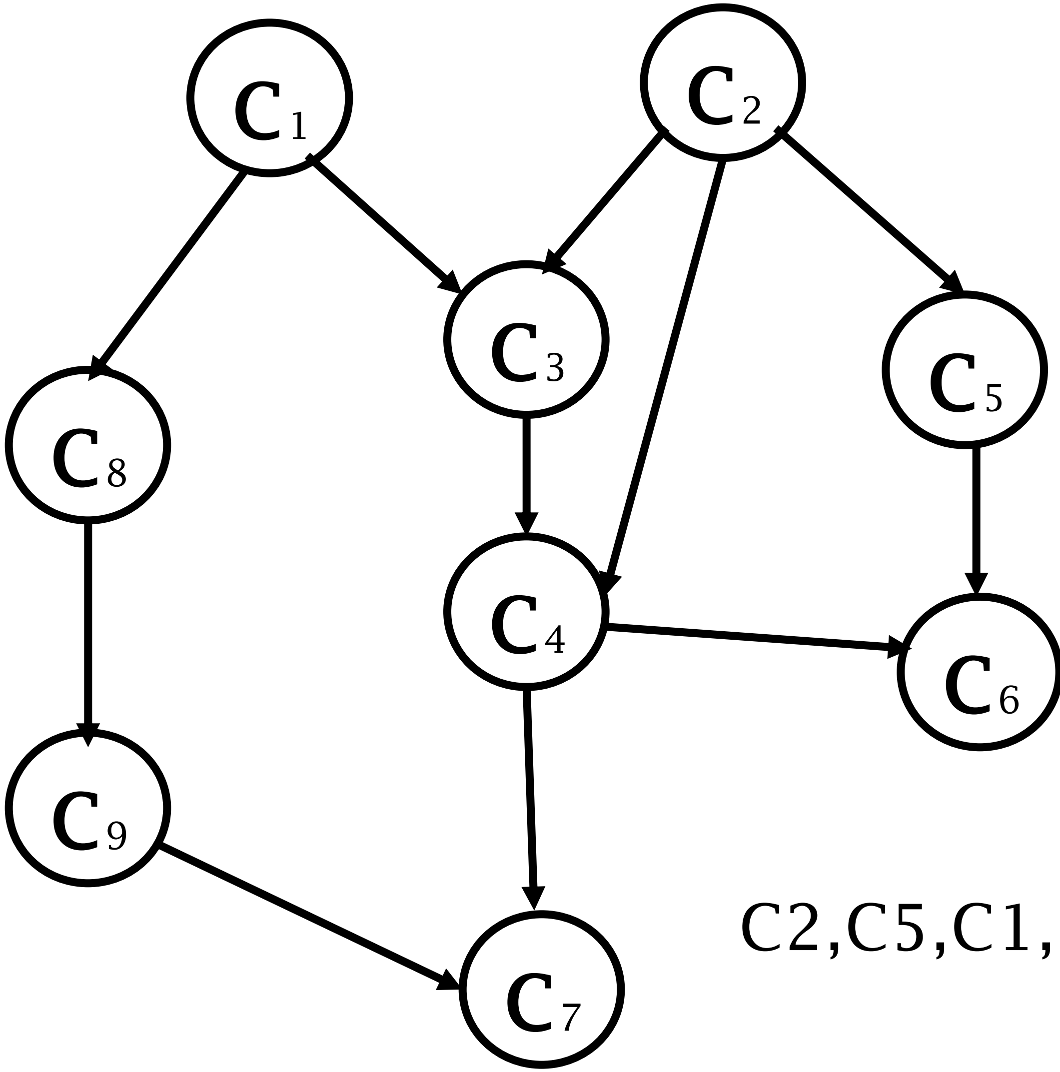
- 无环有向图G中顶点的线性序列称作一个**拓扑序列**（topological ordering），即顶点的一个如下序列：
  - 任何一对顶点 v 和w，若<v, w>为G的一条边，则 v 在序列中应出现在 w 的前面；
  - 推而广之，任何一对顶点v 和w，若存在从 v 到w的一条路经，则v在序列中应出现在w的前面

拓扑序列示例

课程代号	课程名称	先修课程
C1	高等数学	
C2	程序设计	
C3	离散数学	C1，C2
C4	数据结构	C2，C3
C5	算法分析	C2
C6	编译技术	C4，C5
C7	操作系统	C4，C9
C8	普通物理	C1
C9	计算机原理	C8



拓扑序列示例



C2,C5,C1,C8,C9,C3,C4,C7,C6

计算拓扑序列

- 一个有向图的结点的拓扑序列不唯一
- 并非任何有向图都可排成拓扑序列，有环图例外
- 任何 **有向无环图 (DAG)**，其结点都可排成一个拓扑序列，方法为：
  1. 从图中选择一个入度为0 的顶点且输出之；
  2. 从图中删除此结点及其所有的出边，并把对应的邻接点的入度减 1；
  3. 反复前两个步骤，直到所有的顶点都输出为止

基于队列计算拓扑排序

```
void TopsortbyQueue(Graph& G)  {
    for (int i = 0; i < G.VerticesNum(); i++) G.Mark[i] = UNVISITED; // 初始化
    using std::queue; queue<int> Q;                    // 使用STL中的队列
    for (i = 0; i < G.VerticesNum(); i++)              // 入度为0的顶点入队
        if (G.Indegree[i] == 0)  Q.push(i);
    while (!Q.empty()) {                               // 如果队列非空
        int v = Q.front(); Q.pop();                    // 获得队列顶部元素， 出队
        Visit(G,v);       G.Mark[v] = VISITED;         // 将标记位设置为VISITED
        for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) {
            G.Indegree[G.ToVertex(e)]--;               // 相邻的顶点入度减1
            if (G.Indegree[G.ToVertex(e)] == 0)        // 顶点入度减为0则入队
                Q.push(G.ToVertex(e));
        }
    }
    for (i = 0; i < G.VerticesNum(); i++)              // 判断图中是否有环
        if (G.Mark[i] == UNVISITED) {
            cout<<" 此图有环！ ";          break;
        }
}
```

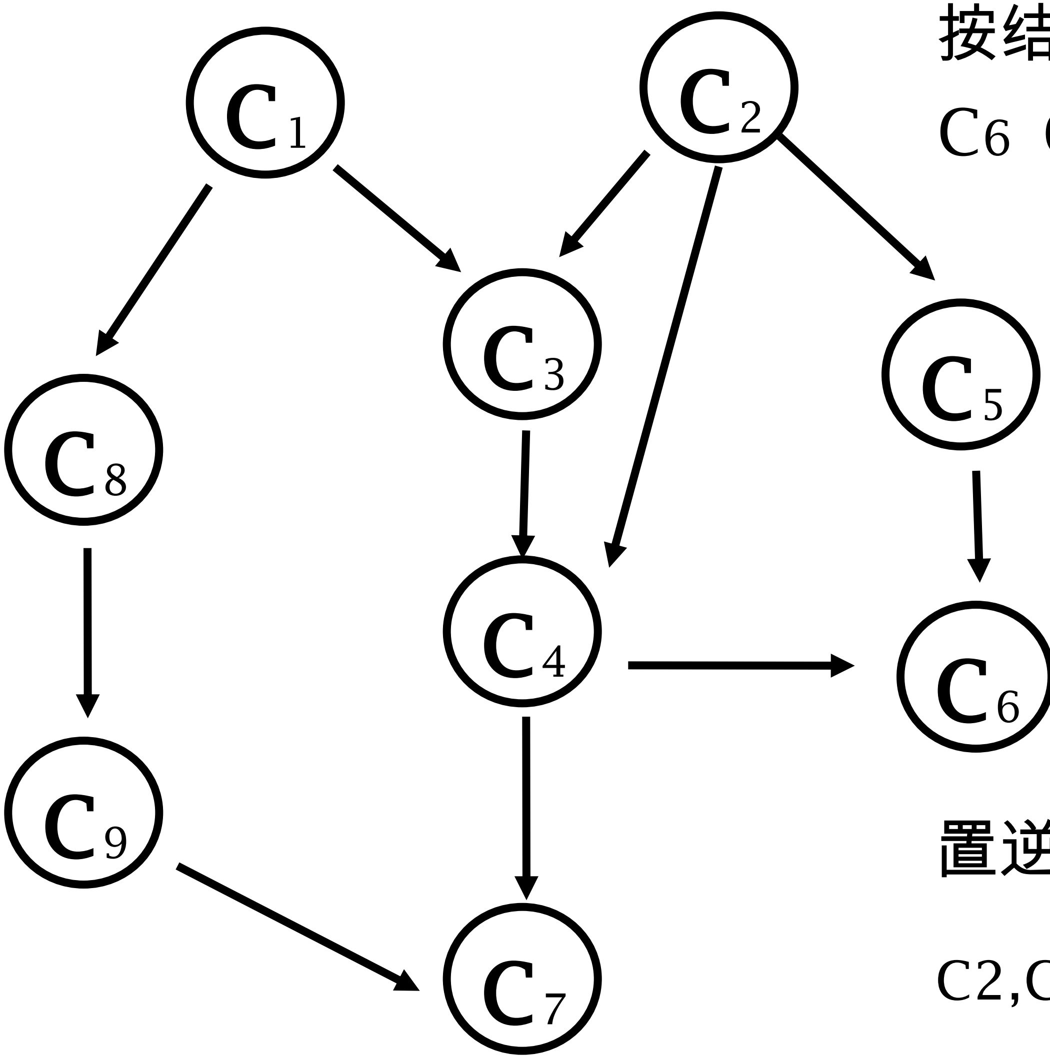
深度优先搜索计算拓扑排序

```
int  *TopsortbyDFS(Graph& G)  {                    // 结果是颠倒的
    for (int i=0; i<G.VerticesNum(); i++)          // 初始化
        G.Mark[i] = UNVISITED;
    int *result = new int[G.VerticesNum()];
    int index = 0;
    for (i=0; i<G.VerticesNum(); i++)              // 对所有顶点
        if (G.Mark[i] == UNVISITED)
            Do_topsort(G, i, result, index);       // 递归函数
    for (i=G.VerticesNum()-1; i>=0; i--)           // 逆序输出
        Visit(G, result[i]);
    return result;
}
```

深度优先搜索计算拓扑排序

```
// 拓扑排序递归函数
void Do_topsort(Graph& G, int V, int *result, int& index) {
    G.Mark[V] = VISITED;
    for (Edge e = G.FirstEdge(V);
          G.IsEdge(e); e=G.NextEdge(e)) {
            if (G.Mark[G.ToVertex(e)] == UNVISITED)
                Do_topsort(G, G.ToVertex(e),  result,  index);
    }
    result[index++]=V;                      // 相当于后处理
}
```

拓扑序列示例



按结点编号深度优先：

C6 C7 C4 C3 C9 C8 C1 C5 C2

置逆，拓扑序列为：

C2,C5,C1,C8,C9,C3,C4,C7,C6

拓扑排序代价分析

- 与图的深度优先搜索方式遍历相同
  - 图的每条边处理一次
  - 图的每个顶点访问一次
- 采用邻接表存储表示，时间代价 $O(|V| + |E|)$
  - 建立 入度为 0 的顶点队列的代价 $O(|V|)$
  - 排序过程中每个顶点输出一次，更新顶点的入度需要检查每条边总计|E|次，$O(|E|)$
- 采用相邻矩阵表示时，为 $\Theta(|V|^2)$

递归与非递归的拓扑排序

- 须为有向图
- 须为无环图
- 支持非连通图
- 不用考虑权值
- 回路
  - 非递归的算法，最后判断 (若还有顶点没有输出，肯定有回路)
  - 递归的算法要求判断有无回路

图算法需要考虑的问题

- 是否支持
  - 有向图、无向图
  - 有回路的图
  - 非连通图
  - 权值为负
- 如果不支持
  - 则修改方案?