数据结构:图

Data Structure

主讲教师：屈卫兰

Office number: 基地203

tel: 13873195964

图

- 术语及表示法
- 图的实现
- 图的周游
- 拓扑排序
- 最短路径问题
- 最小支撑树

术语及表示法

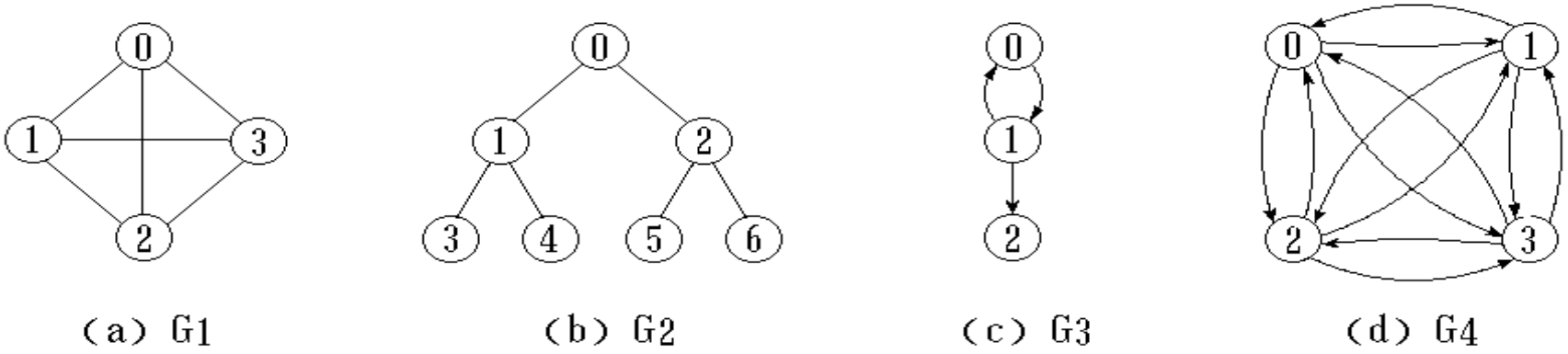
- 数据的逻辑结构可以表示为二元组B=(K,R)，在数据结构中研究一个关系的情况，R={r}。
  - 如果关系r不限制结点之间的关系，任意一对结点间都允许有一个关系(边)，这样的结构就是图。
  - 图是最基本的数据结构，树和线性表可看作受限图。
- 图可用G=(V, E) 来表示，结点在图中称为顶点，顶点的非空有穷集合记为V；顶点(结点)的偶对称为边，边的集合记为E，E内的每条边都是V中某一对顶点的连接。顶点总数计为|V|，边的总数记为|E|，取值范围是0到$O(|V|^2)$。

术语

- 若图中的边限定为从一个顶点指向另一个顶点，则称此图为有向图。
  - 有向图中的顶点偶对用尖括号来表示，<v1, v2>和<v2,v1>代表不同的边。
- 若图中的边无方向性，则称之为无向图。
  - 无向图中的顶点偶对用圆括号来表示，(v1, v2)和(v2,v1)代表同一条的边。
- 边数较少的图称为稀疏图，边数较多的图称为密集图，包括所有可能边的图称为完全图

术语

- 有向图与无向图　在有向图中，顶点对<x, y>是有序的。在无向图中，顶点对(x, y)是无序的。
- 完全图　若有 *n* 个顶点的无向图有 *n*(*n*-1)/2 条边, 则此图为完全无向图。有 *n* 个顶点的有向图有*n*(*n*-1) 条边, 则此图为完全有向图。

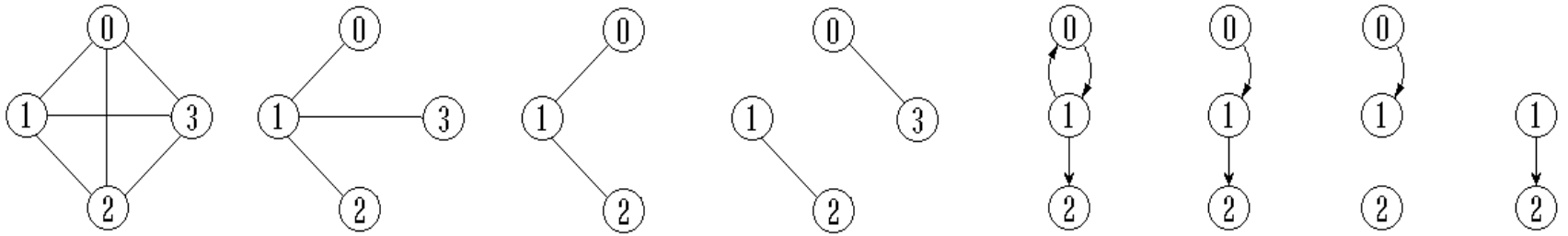


(a) G1　(b) G2　(c) G3　(d) G4

- 邻接顶点　如果 (u, v) 是 E(G) 中的一条边，则称 u 与 v 互为邻接顶点。

术语

- **权**　某些图的边具有与它相关的数，称之为权。这种带权图叫做网络。
- **子图**　设有两个图G＝(*V*, *E*)和G‘＝(*V*’, *E*‘)。若*V*’⊆*V*且*E*‘⊆*E*，则称图G’是图G的子图。



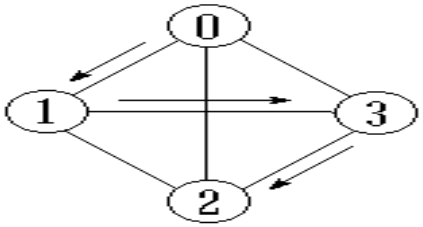
- **顶点的度**　一个顶点*v*的度是与它相关联的边的条数。记作TD(*v*)。在有向图中，顶点的度等于该顶点的入度与出度之和。

术语

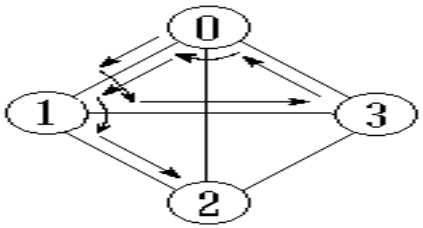
- 顶点 v 的入度是以 v 为终点的有向边的条数; 顶点 v 的出度是以 v 为始点的有向边的条数。
- 路径　在图 G＝(V, E) 中, 若从顶点 v_i 出发, 沿一些边经过一些顶点 v_{p1}, v_{p2}, …, v_{pm}, 到达顶点v_j。则称顶点序列 (v_i v_{p1} v_{p2} … v_{pm} v_j) 为从顶点v_i到顶点 v_j 的路径。它经过的边(v_i, v_{p1})、(v_{p1}, vp2)、…、(vpm, vj) 应是属于E的边。
- 路径长度
  - 非带权图的路径长度是指此路径上边的条数。
  - 带权图的路径长度是指路径上各边的权之和。

术语

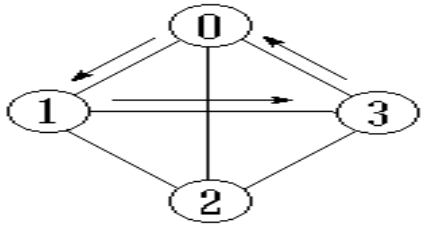
- **简单路径**　　若路径上各顶点 $v_1, v_2, \ldots, v_m$ 均不互相重复，则称这样的路径为简单路径。
- **回路**　　若路径上第一个顶点 v1 与最后一个顶点vm 重合，则称这样的路径为回路或环。



（a）简单路径



（b）非简单路径

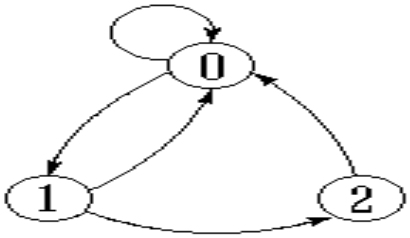


（c） 回路

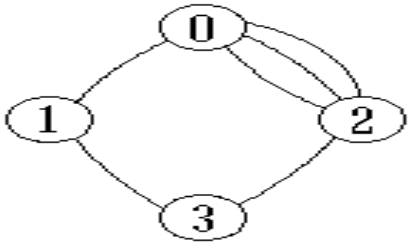
路径，则称顶点v1与v2是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图。非连通图的极大连通子图叫做连通分量

术语

- 强连通图与强连通分量　　在有向图中，若对于每一对顶点vi和vj，都存在一条从vi到vj和从vj到vi的路径，则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。
- 生成树　　一个连通图的生成树是它的极小连通子图，在n个顶点的情形下，有n-1条边。但有向图则可能得到它的由若干有向树组成的生成森林。
- 不讨论的图



（a）带自身环的图



（b）多重图

图的实现

图的抽象数据类型

```
class Graph { // Graph abstract class
private:
   void operator=(const Graph&) {}
   Graph(const Graph&) {}
public:
   Graph() {}
   virtual ~Graph() {}
   virtual void Init(int n)=0;
  virtual int n() =0; // # of vertices
  virtual int e() =0; // # of edges
  // Return index of first, next
     neighbor
  virtual int first(int v) =0;
  virtual int next(int v, int w) =0;
// Store new edge
  virtual void setEdge(int v1, int v2, int
     wght) =0;
  // Delete edge defined by two vertices
  virtual void delEdge(int v1, int v2) =0;
  // Weight of edge connecting two
     vertices
  virtual bool isEdge(int i, int j) =0;
  virtual int weight(int v1, int v2) =0;
  virtual int getMark(int v) =0;
  virtual void setMark(int v, int val) =0;
};
```

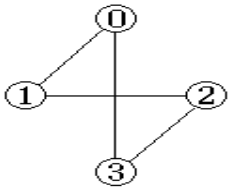
邻接矩阵 (Adjacency Matrix)

- 在图的邻接矩阵表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。
- 设图 A = (*V*, *E*)是一个有 *n* 个顶点的图，则图的邻接矩阵是一个二维数组 A.*edge*[*n*][*n*]，定义：

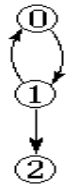
$$\mathbf{A}.Edge[i][j]=\begin{cases}1, & \text{如果} <i,j>\in E \text{ 或者 } (i,j)\in E\\ 0, & \text{否则}\end{cases}$$

- 无向图的邻接矩阵是对称的，有向图的邻接矩阵可能是不对称的。

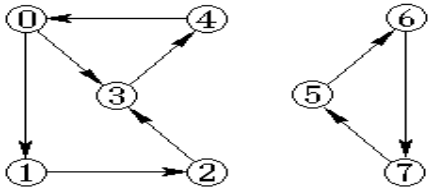
邻接矩阵



G8



G3



G7

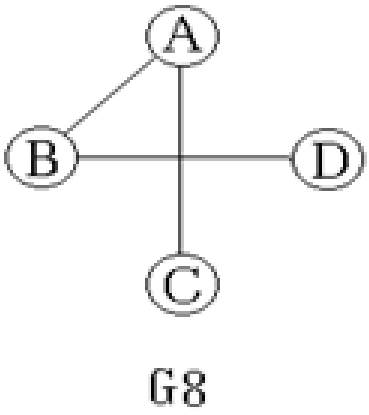
$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \end{matrix}$$

$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} ⓪ \\ ① \\ ② \end{matrix}$$

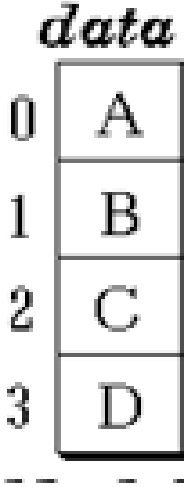
$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ & ④ & ⑤ & ⑥ & ⑦ \end{matrix} \\ \left(\begin{array}{ccccc:ccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \end{array} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \\ ⑦ \end{matrix}$$

- 在有向图中, 统计第 *i* 行 1 的个数可得顶点 *i* 的出度，统计第 *j* 行 1 的个数可得顶点 *j* 的入度。
- 在无向图中, 统计第 *i* 行 (列) 1 的个数可得顶点*i* 的度。

邻接矩阵-举例



无向图



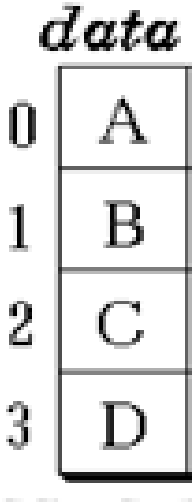
顶点数组

0	1	1	0
1	0	0	1
1	0	0	0
0	1	0	0

邻接矩阵

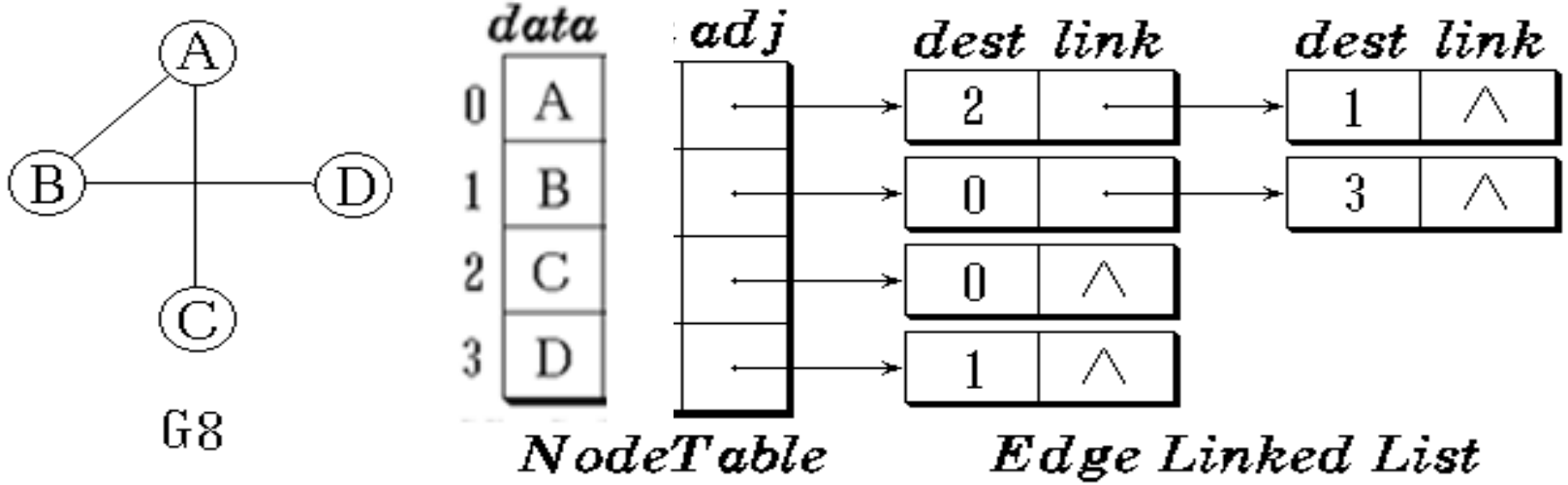
图的相邻矩阵实现

```
class Graphm:public Graph {
private:
    int numVertex, numEdge;
    int **matrix;
    int *mark;
public:
```



邻接表 (Adjacency List)

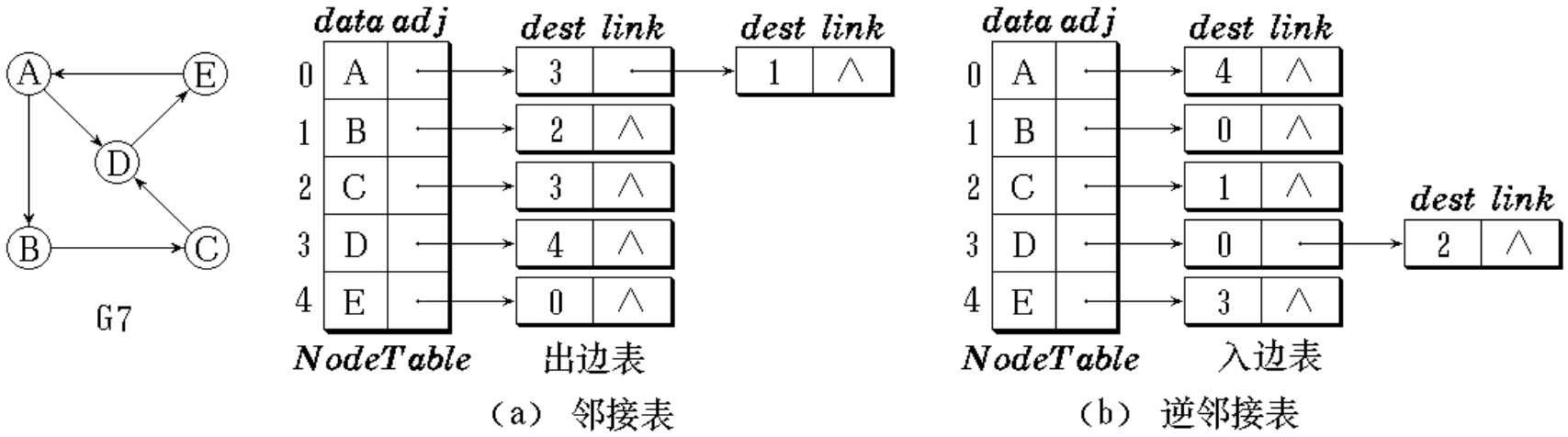
- 无向图的邻接表



把同一个顶点发出的边链接在同一个边链表中，链表的每一个结点代表一条边，叫做边结点，结点中保存有与该边相关联的另一顶点的顶点下标 ***dest*** 和指向同一链表中下一个边结点的指针 ***link***。

邻接表

- 有向图的邻接表和逆邻接表



- 在有向图的邻接表中，第 *i* 个边链表链接的边都是顶点 *i* 发出的边。也叫做出边表。
- 在有向图的逆邻接表中，第 *i* 个边链表链接的边都是进入顶点 *i* 的边。也叫做入边表。

邻接表

- 带权图的边结点中保存该边上的权值 ***cost***。
- 顶点 ***i*** 的边链表的表头指针 ***adj*** 在顶点表的下标为 ***i*** 的顶点记录中，该记录还保存了该顶点的其它信息。
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定。
- 设图中有 ***n*** 个顶点，***e*** 条边，则用邻接表表示无向图时，需要 ***n*** 个顶点结点，***2e*** 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 ***n*** 个顶点结点，***e*** 个边结点。

图的邻接表实现

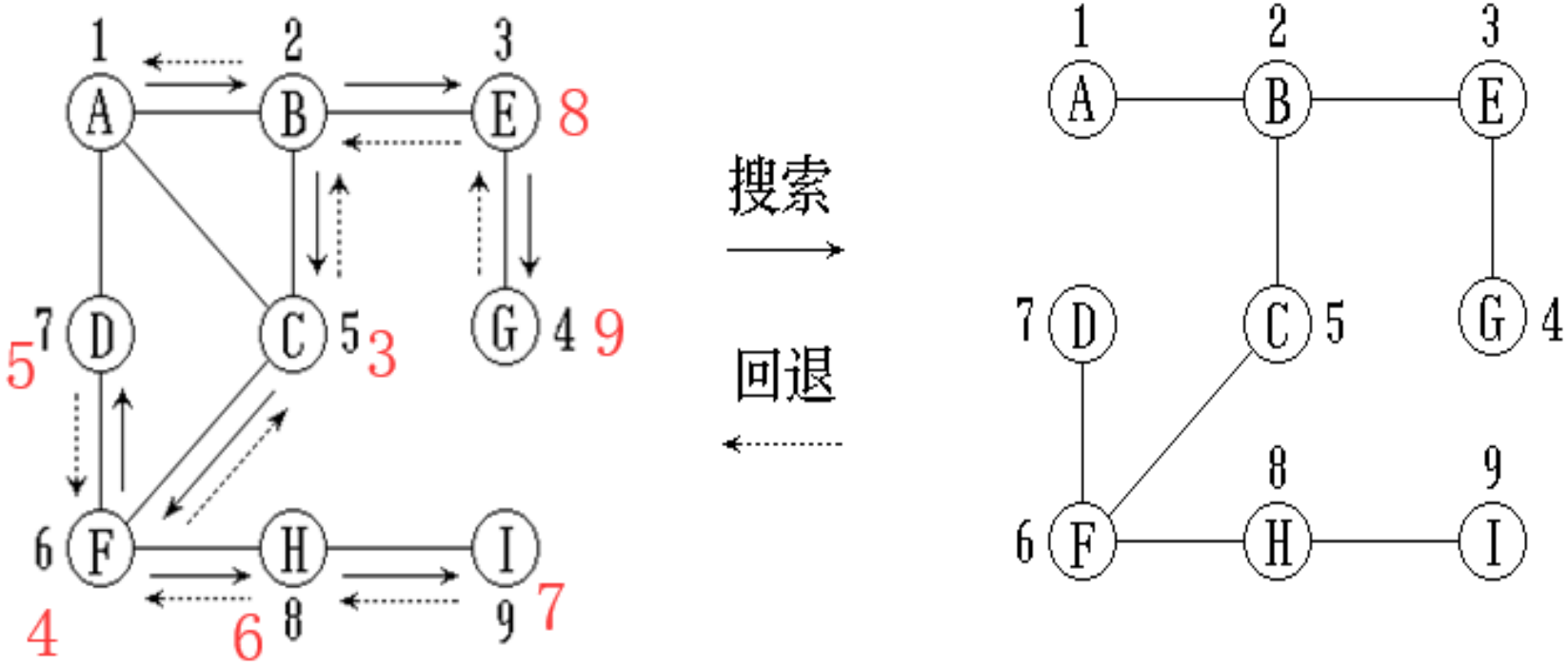
```
class Graph1:public Graph {
private:
    List<Edge> ** Vertex;
    int numVertex, numEdge;
    int *mark;
public:
```

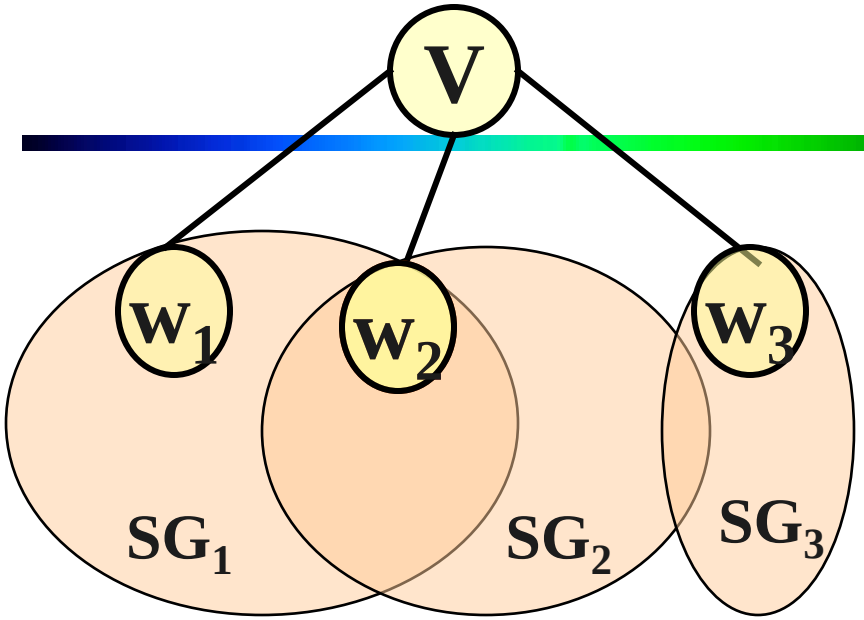
图的周游

- 从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的周游 (Graph Traversal)。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的标志位，它的初始状态为 0，在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即让该顶点标志位为 1，防止它被多次访问。

深度优先搜索DFS (Depth First Search)

- 深度优先搜索的示例：





W_1、W_2和W_3 均为 V 的邻接点，SG_1、SG_2 和 SG_3 分别为含顶点W_1、W_2和W_3 的子图。

访问顶点 V：

for (W_1、W_2、W_3)

若该邻接点W未被访问，

则从它出发进行深度优先搜索遍历。

从上页的图解可见:

1. 从深度优先搜索遍历连通图的过程类似于树的先根遍历;

2. 如何判别V的邻接点是否被访问?

解决的办法是:为每个顶点设立一个“访问标志 visited[w]”。

深度优先搜索DFS

DFS 在访问图中某一起始顶点 *v* 后，由 *v* 出发，访问它的任一邻接顶点 w_1；再从 w_1 出发，访问与 w_1 邻 接但还没有访问过的顶点 w_2；然后再从 w_2 出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 *u* 为止。接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

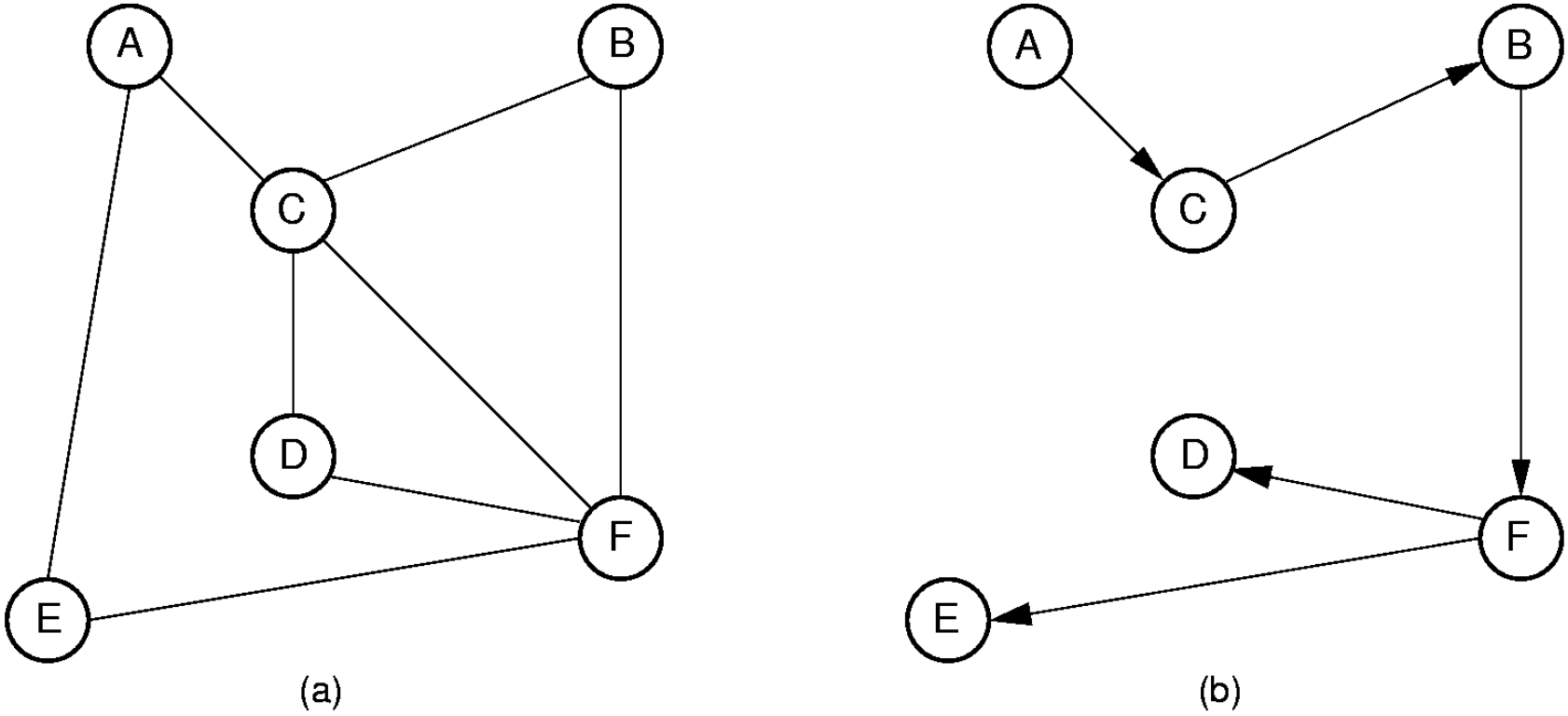
图的深度优先搜索算法

```
// Depth first search
void DFS(Graph* G, int v) {
  PreVisit(G, v);
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G-
    >n();w = G->next(v,w))
    if (G->getMark(w) ==
    UNVISITED)
      DFS(G, w);
}
```

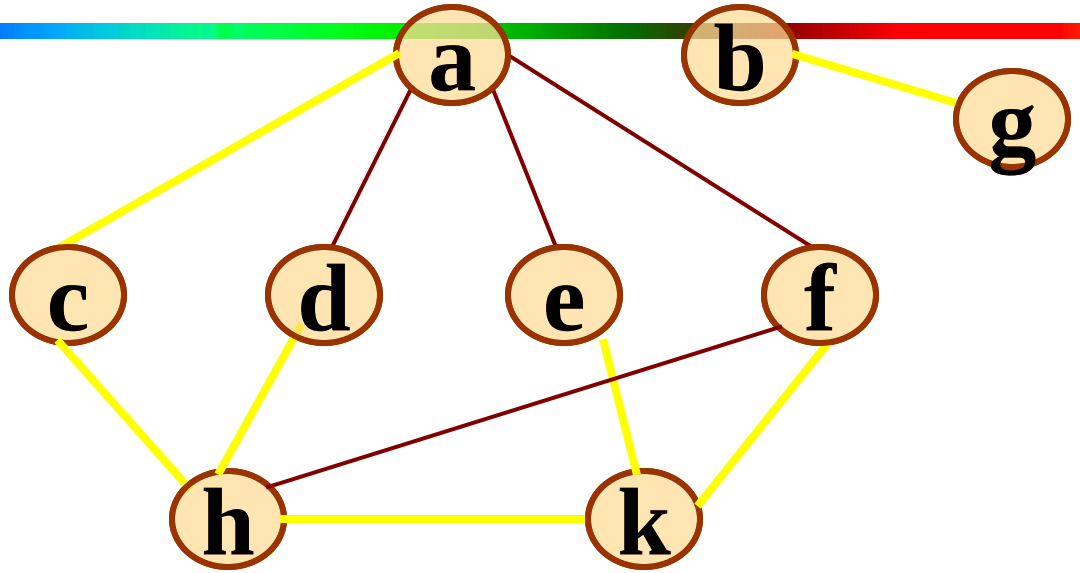
```
void graphTraverse(const
    Graph* G) {
  for (v=0; v<G->n(); v++)
    G->setMark(v, UNVISITED);
    // Initialize
  for (v=0; v<G->n(); v++)
    if (G->getMark(v) ==
    UNVISITED)
      DFS(G, v);
}
```

图的深度优先搜索算法
DFS (Depth First Search)

Cost: $\Theta(|V| + |E|)$.



例如:

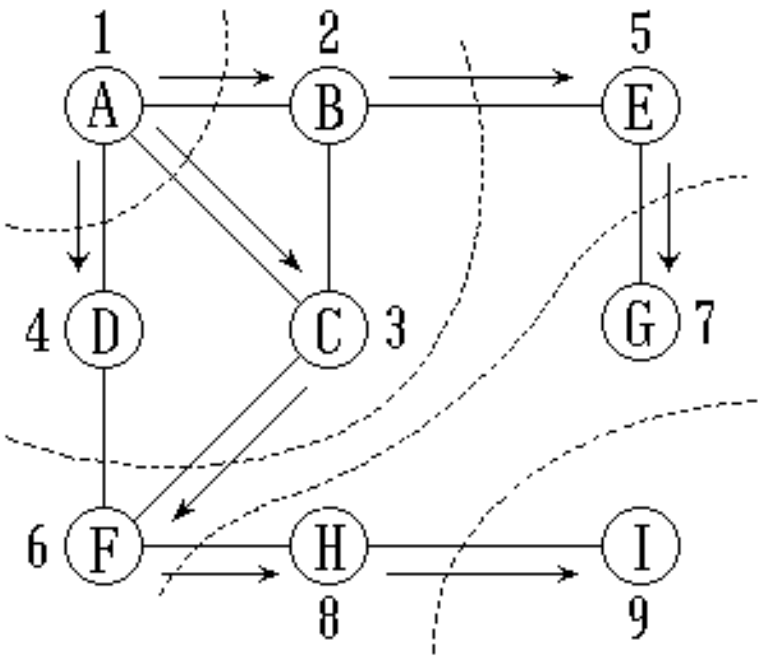


	0	1	2	3	4	5	6	7	8
访问标志:	T	T	T	T	T	T	T	T	T

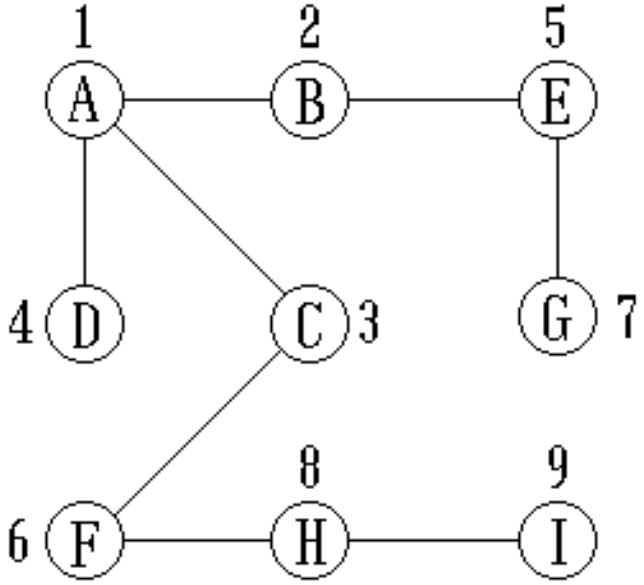
访问次序: a c h d k f e b g

广度优先搜索BFS (Breadth First Search)

- 广度优先搜索的示例



搜索



广度优先搜索过程

广度优先生成树

广度优先搜索BFS

- 使用广度优先搜索在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未曾被访问过的邻接顶点 $w_1, w_2, \ldots, w_t$，然后再顺序访问 $w_1, w_2, \ldots, w_t$ 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，… 如此做下去，直到图中所有顶点都被访问到为止。
- 广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。

广度优先搜索DFS

为了实现逐层访问，算法中使用了一个队列，以记忆正在访问的这一层和上一层的顶点，以便于向下一层访问。

```
void BFS(Graph* G, int start,Queue<int>*Q) {
  int v, w;
  Q->enqueue(start);        // Initialize Q
  G->setMark(start, VISITED);
  while (Q->length() != 0) { // Process Q
    Q->dequeue(v);    PreVisit(G, v);   // Take action
    for(w=G->first(v);w<G->n();w=G->next(v,w))
      if (G->getMark(w) == UNVISITED) {
     G->setMark(w, VISITED);       Q->enqueue(w);      }
    PostVisit(G, v);  // Take action
  }
}
```

拓扑排序

- 问题提出：学生选修课程问题

  顶点——表示课程

  有向弧——表示先决条件，若课程i是课程j的先决条件，则图中有弧<i,j>

  学生应按怎样的顺序学习这些课程，才能无矛盾、顺利地完成学业——拓扑排序

- 定义

  AOV网——用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网(Activity On Vertex network)，简称AOV网

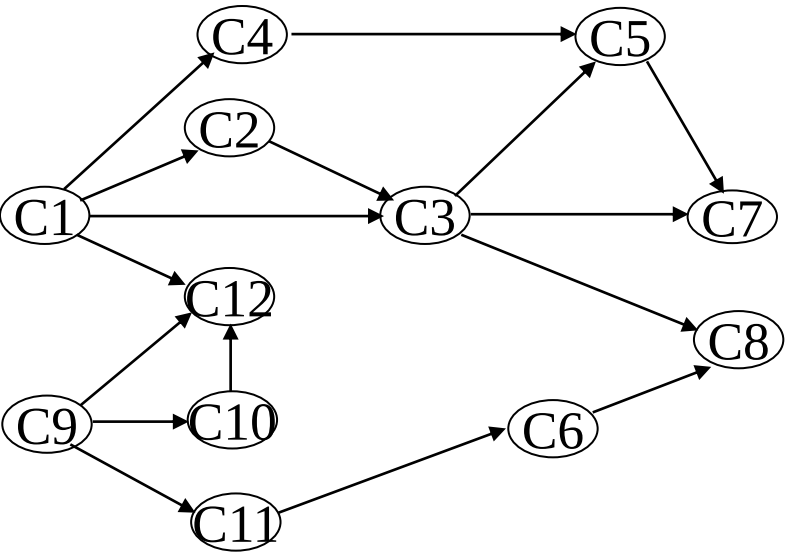
  - 若<vi,vj>是图中有向边，则vi是vj的直接前驱；vj是vi的直接后继
  - AOV网中不允许有回路，这意味着某项活动以自己为先决条件

拓扑排序

- 拓扑排序——把AOV网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程叫拓扑排序
  - 检测AOV网中是否存在环方法：对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该AOV网必定不存在环
- 拓扑排序的方法
  - 在有向图中选一个没有前驱的顶点且输出之
  - 从图中删除该顶点和所有以它为尾的弧
  - 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止

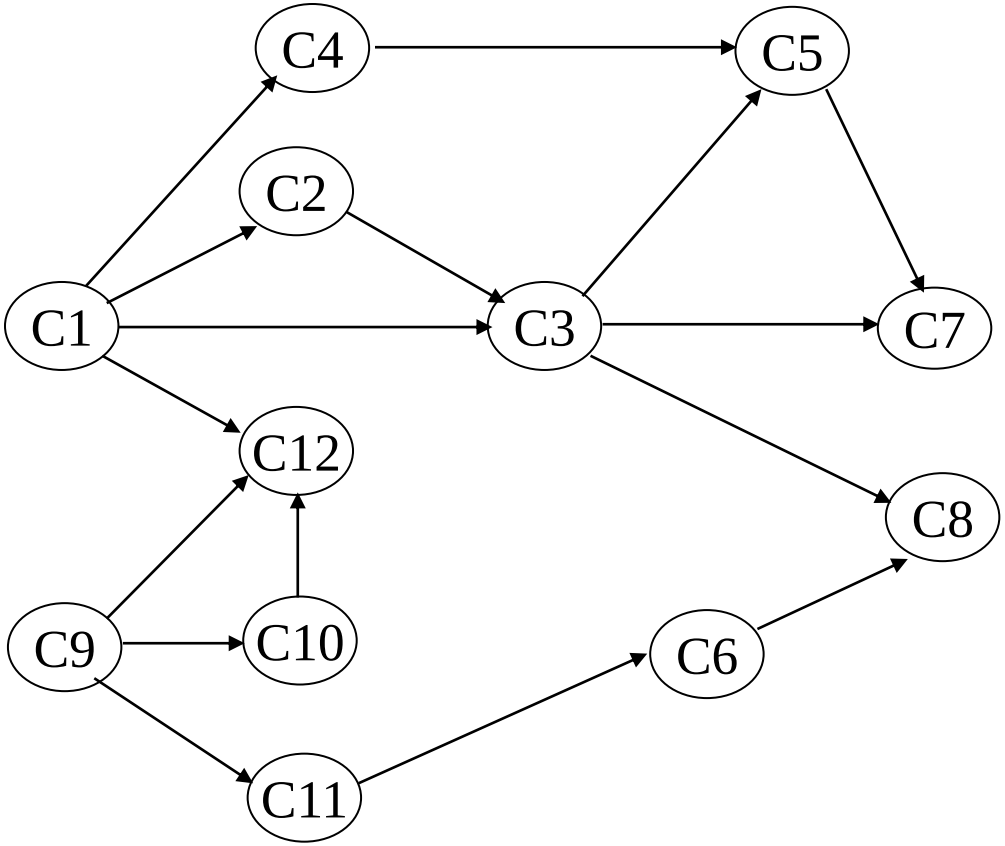
举例

例



课程代号	课程名称	先修棵
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构	C1,C2
C4	汇编语言	C1
C5	语言的设计和分析	C3,C4
C6	计算机原理	C11
C7	编译原理	C3.C5
C8	操作系统	C3,C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1,C9,C10

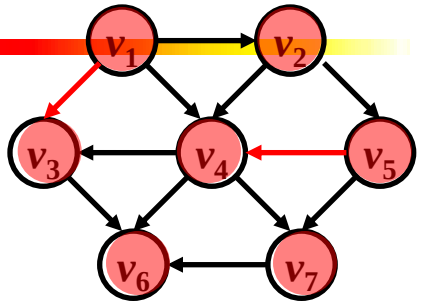
举例

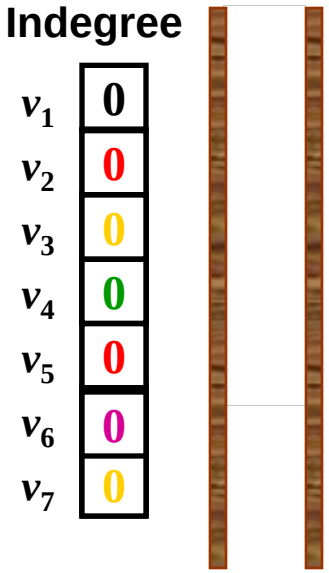


拓扑序列：C1--C2--C3--C4--C5--C7--C9--C10--C11--C6--C12--C8
或　　：C9--C10--C11--C6--C1--C12--C4--C2--C3--C5--C7--C8

基于队列算法实现

```
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];
  int v, w;
  for (v=0; v<G->n(); v++) Count[v] = 0;
  for (v=0; v<G->n(); v++) // Process edges
    for (w=G->first(v); w<G->n();w = G->next(v,w))
      Count[w]++;   // Add to v2's count
  for (v=0; v<G->n(); v++) // Initialize Q
    if (Count[v] == 0)     // No prereqs
      Q->enqueue(v);
  while (Q->length() != 0) {
    Q->dequeue(v);
    printout(v);     // PreVisit for V
    for (w=G->first(v); w<G->n();w = G->next(v,w)) {
      Count[w]--;    // One less prereq
      if (Count[w] == 0) // Now free
        Q->enqueue(w);
    }}}
```



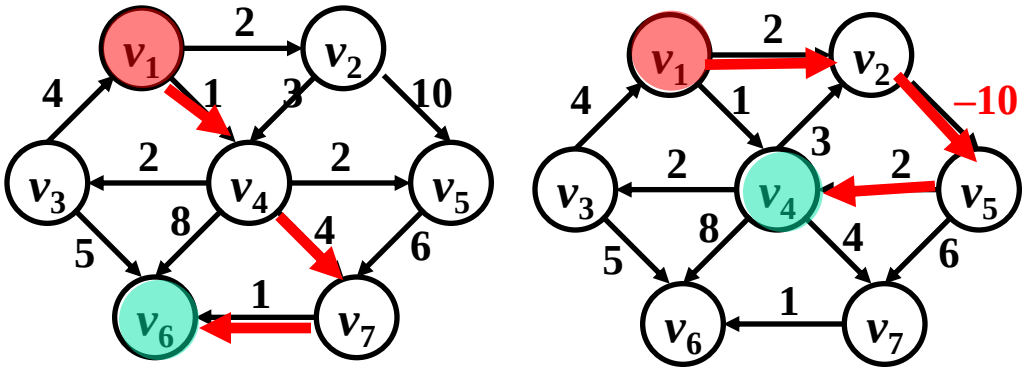


最短路径问题

- **最短路径问题：**如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
- 问题解法
  - 边上权值非负情形的单源最短路径问题
    — Dijkstra算法
  - 边上权值为任意值的单源最短路径问题
    — Bellman和Ford算法
  - 所有顶点之间的最短路径
    — Floyd算法

最短路径问题

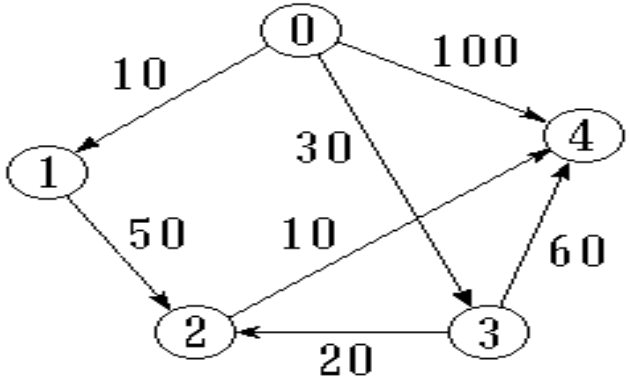
- **最短路径问题：** 如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
- 问题解法
  - 边上权值非负情形的单源最短路径问题: Dijkstra算法
  - 边上权值为任意值的单源最短路径问题:Bellman和Ford算法
  - 所有顶点之间的最短路径:Floyd算法



边上权值非负情形的单源最短路径问题

- 问题的提法： 给定一个带权有向图*D*与源点*v*，求从*v*到*D*中其它顶点的最短路径。限定各边上的权值大于或等于0。
- 为求得这些最短路径，Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点v到其它各顶点的最短路径全部求出为止。

举例



（a）带权有向图

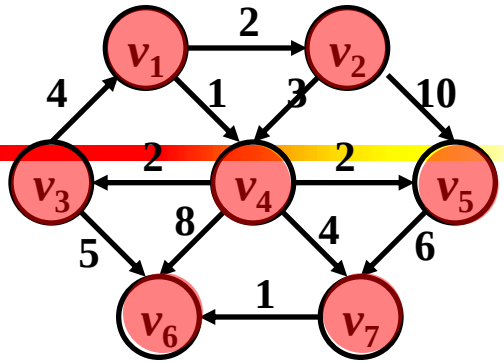
	0	1	2	3	4	
	0	10	∞	30	100	0
	∞	0	50	∞	∞	1
	∞	∞	0	∞	10	2
	∞	∞	20	0	60	3
	∞	∞	∞	∞	0	4

（b）邻接矩阵

Dijkstra逐步求解的过程

源点	终点	最短路径				路径长度		
v_0	v_1	(v_0, v_1)				10		
	v_2	—	(v_0, v_1, v_2)	(v_0, v_3, v_2)		—	60	50
	v_3	(v_0, v_3)				30		
	v_4	(v_0, v_4)		(v_0, v_3, v_4)	(v_0, v_3, v_2, v_4)	100	90	60

Dijkstra算法可描述如下：



☆ 初始化： $S \leftarrow \{ v_0 \}$;

$dist[j] \leftarrow Edge[0][j]$, $j = 1, 2, \ldots, n\text{-}1$;

// n为图中顶点个数

- 求出最短路径的长度：

  $dist[k] \leftarrow \min\{ dist[i] \}$, $i \in V\text{-}S$;

  $S \leftarrow S \cup \{ k \}$;

- 修改：

  $dist[i] \leftarrow \min\{ dist[i], dist[k] + Edge[k][i] \}$,

  对于每一个 $i \in V\text{-}S$;

- 判断： 若$S = V$, 则算法结束，否则转。

	Dist	Path
v_1	0	0
v_2	2	v_1
v_3	3	v_4
v_4	1	v_1
v_5	3	v_4
v_6	6	v_7
v_7	5	v_4

Dijkstra算法的实现

```
// Compute shortest path distances from s,
// return them in D
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;
  for (i=0; i<G->n(); i++) { // Do vertices
    v = minVertex(G, D);
    if (D[v] == INFINITY) return;
    G->setMark(v, VISITED);
    for (w=G->first(v); w<G->n();
                        w = G->next(v,w))
      if (D[w] > (D[v] + G->weight(v, w)))
        D[w] = D[v] + G->weight(v, w);
  }
}
```

minVertex

```
// Find min cost vertex
int minVertex(Graph* G, int* D) {
  int i, v;
  // Set v to an unvisited vertex
  for (i=0; i<G->n(); i++)
  if (G->getMark(i) == UNVISITED)
    { v = i; break; }
  // Now find smallest D value
  for (i++; i<G->n(); i++)
    if ((G->getMark(i) == UNVISITED) &&
        (D[i] < D[v]))
      v = i;
  return v;
}
```

因为扫描需进行|V|次，且每条边需要相同的次数来更新D值，所以总时间代价为$O(|V|^2+|E|)=O(|V|^2)$。适合于密集图

利用优先队列实现的Dijkstra算法

```
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;       // v is current vertex
  DijkElem temp;
  DijkElem E[G->e()]; // Heap array
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;     // Initialize heap array
  minheap<DijkElem, DDComp> H(E, 1, G->e());
  for (i=0; i<G->n(); i++) {// Get distances
   do { if(!H.removemin(temp)) return;
        v = temp.vertex;
      } while (G->getMark(v) == VISITED);
   G->setMark(v, VISITED);
   if (D[v] == INFINITY) return;
   for(w=G->first(v); w<G->n(); w=G->next(v,w))
     if (D[w] > (D[v] + G->weight(v, w))) {
       D[w] = D[v] + G->weight(v, w);
       temp.distance = D[w]; temp.vertex = w;
       H.insert(temp); // Insert in heap
     }}}
```

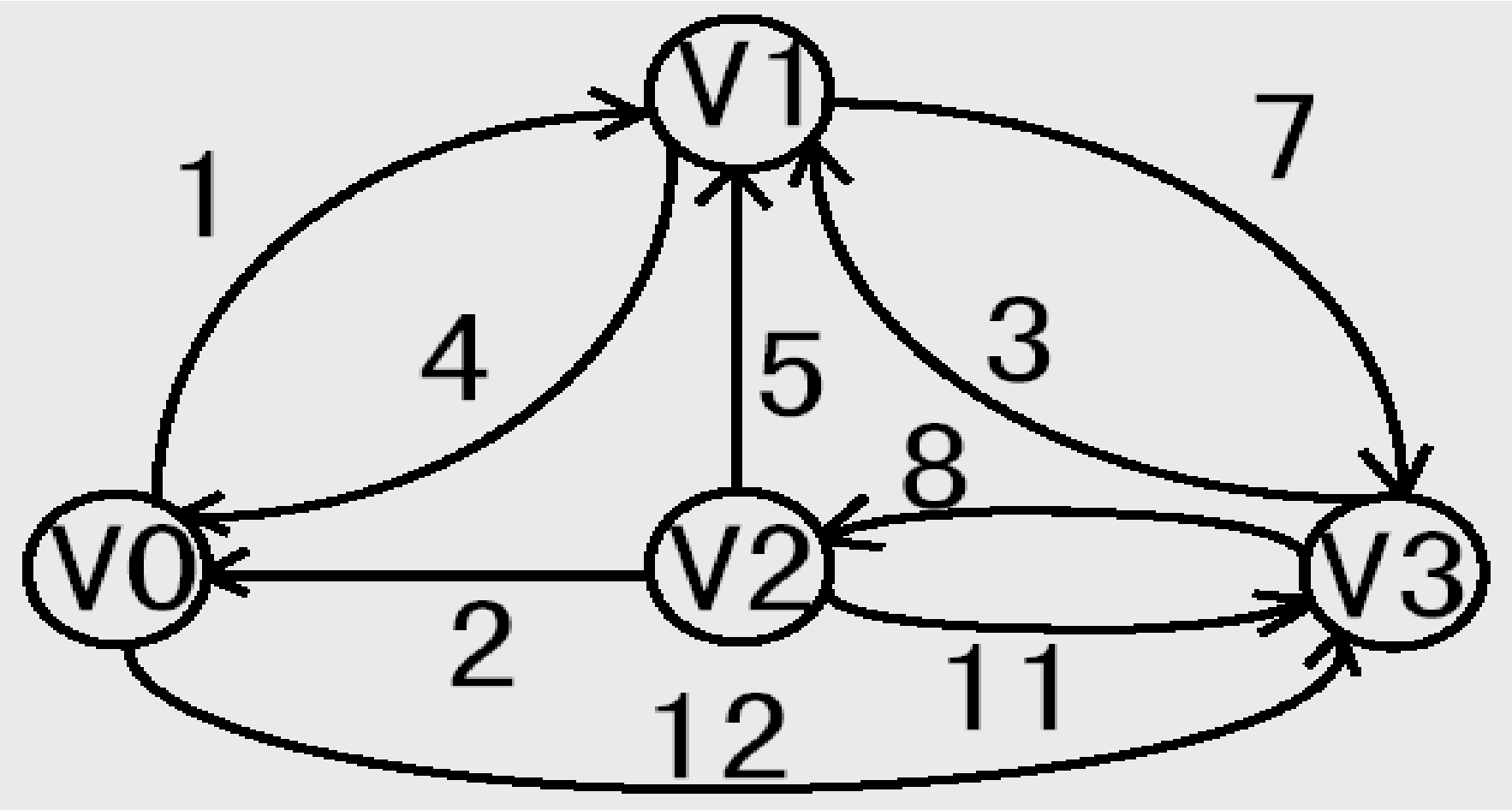
优先对列实现的Dijkstra算法

- 将未被处理的顶点的D值保存在一个最小堆中
- 可用O(log|V|) 次搜索找出下一个最近顶点
- 每次改变D(x) 值时，都可以通过先删除再重新插入的方法改变顶点x在堆中的位置
- 重复插入最短路径长度的唯一缺点是：在最坏情况下，它将使堆中元素数目由O(|V|)增加到O(|E|) 。总时间开销为O((|V|+|E|) ·log|E|))，因为处理每条边时，都必须对堆进行一次重排
- 适合于稀疏图

每对顶点间最短路径

- 对任意的u、v∈V，计算d(u,v) 的值。
- 使用|V|次Dijkstra 算法
  - 每次从不同顶点出发计算最短路径。
  - 如图G是稀疏图，这不失为一种好方法。对于基于优先队列的Dijkstra 算法，总的时间代价为$O(|V|^2+|V||E|\log|V|)$。
  - 对密集图来说，基于优先队列的Dijkstra 算法时间代价为$O(|V|^3\log|E|)$；而时间代价为$O(|V|^2)$的Dijkstra 算法在这儿的时间代价将为$O(|V|^3)$。

Floyd 算法例图



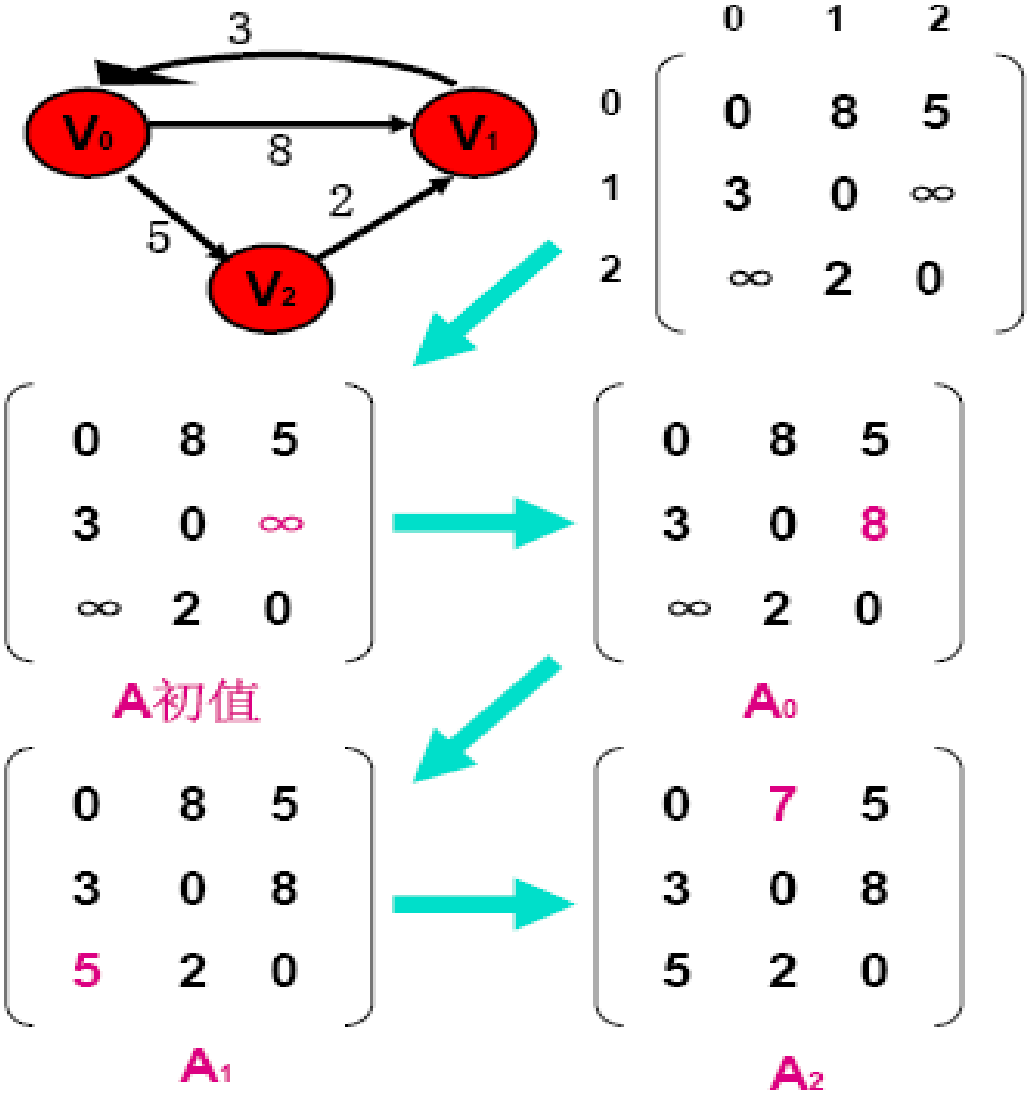
Floyd 算法思想

- 定义k-path 为任意一条从顶点v到u的，中间顶点序号小于k的路径。
  - 0-path 即为直接地从v到u的边。
- 定义$D_k(v, u)$为从v到u的长度最小的k-path
- 假设我们已知从v到u的最短k-path path，则最短的(k+1)-path 经过，或不经过顶点k
  - 如果它经过顶点k，短则最短(k+1)-path 的一部分是从v到k的最短k-path ，另一部分是从k到u的最短k-path 。
  - 否则，我们保持其值为最短k-path不变。

Floyd 算法特点

- Floyd 算法仅通过一个三重循环检查了所有的可能性。
- 下面给出Floyd 算法的实现。在算法结束时，D数组存储了所有成对顶点间的最短路径长度。
- 显然这个算法的时间代价为$O(|V|^3)$
- Floyd 算法也是一种动态规划法

实例及求解


3
V0
8
V1
2
5
V2
0 1 2
0 0 8 5
1 3 0 ∞
2 ∞ 2 0
0 8 5
3 0 ∞
∞ 2 0
A初值
0 8 5
3 0 8
∞ 2 0
A0
0 8 5
3 0 8
5 2 0
A1
0 7 5
3 0 8
5 2 0
A2

Floyd 算法

```
//Floyd's all-pairs shortest paths algorithm
void Floyd(Graph* G) {
  int D[G->n()][G->n()];  // Store distances
  for (int i=0; i<G->n(); i++) // Initialize
    for (int j=0; j<G->n(); j++)
      D[i][j] = G->weight(i, j);
  // Compute all k paths
  for (int k=0; k<G->n(); k++)
    for (int i=0; i<G->n(); i++)
      for (int j=0; j<G->n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```

最小支撑树

- 给定一个连通的无向图G，且它的每条边均有相应的长度或权值
- 最小支撑树（ MST MST）是一个包括G的所有顶点和一个边的子集的图，边的子集满足下列条件：
  - (1) 子集中所有边的权之和为类似子集中最小的；
  - (2) 子集中的边能保证图是连通的。
- MST 的应用
  - 使连接电路板上一系列接头所需焊接的线路最短
  - 使在几个城市间建立电话网所需的线路最短

MST 的性质

- MST 中没有回路
  - 若MST 的边集中有回路，显然可通过去掉回路中某条边而得到花销更小的MST
- MST 是一棵有有|V| -1条边的自由树
- 称之为最小支撑树
  - 满足MST 要求的边集所构成的树支撑起了所有的顶点(即把它们联接起来了)
  - 此边集的代价最小

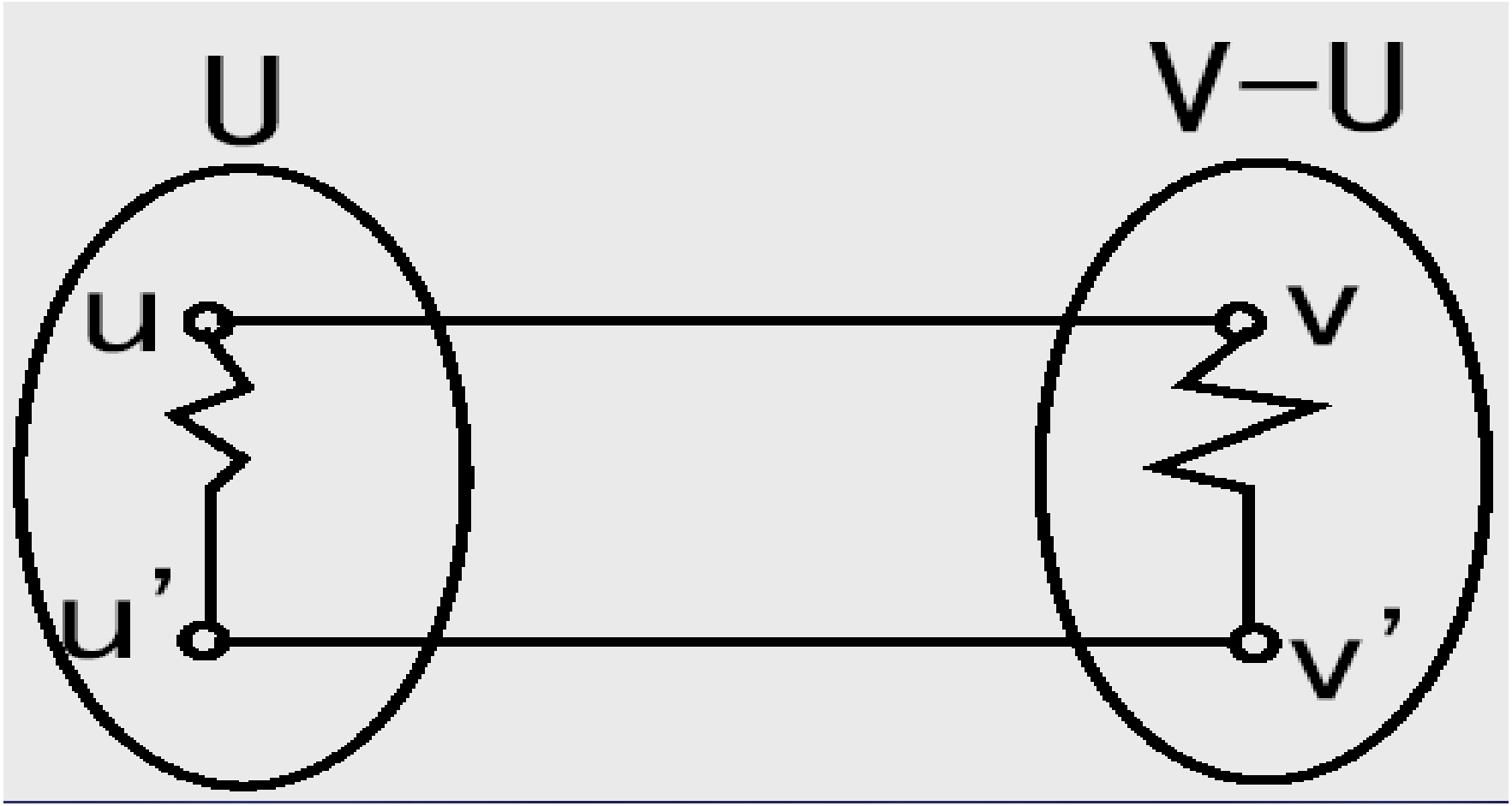
最小支撑树的性质

- 两种构造最小支撑树算法都利用了最小支撑树的性质，简称MST 性质
- 假设G=(V,E) 是一个带权连通图， U是顶点集V的一个非空子集；若(u,v) 有一条具有最小权值的边，其中u∈U，v∈V-U，则必存在一棵包边(u,v) 的最小支撑树。

反证证明法最小支撑树性质

- 假设连通图G的所有最小支撑树都不包含(u,v)
- 设T是G上的一颗最小支撑树，如果把边(u,v)加入到T，由于T是包含图G所有顶点的自由树，加入边(u,v) 后必然存在一条包含含(u,v)的回路。
- 由于T是支撑树，则在T上必存在另一条边(u',v')，其中u'∈U，v'∈V-U；且u和u'之间，v和v'之间有路径相通。删去边(u',v') 就可以消除回路，同时得到另一颗支撑树T'。因为为(u,v) 的权不高于(u',v') ，则T'的代价也不高于T。因此T'是包含(u,v)的一颗最小支撑树。由此与假设产生矛盾。

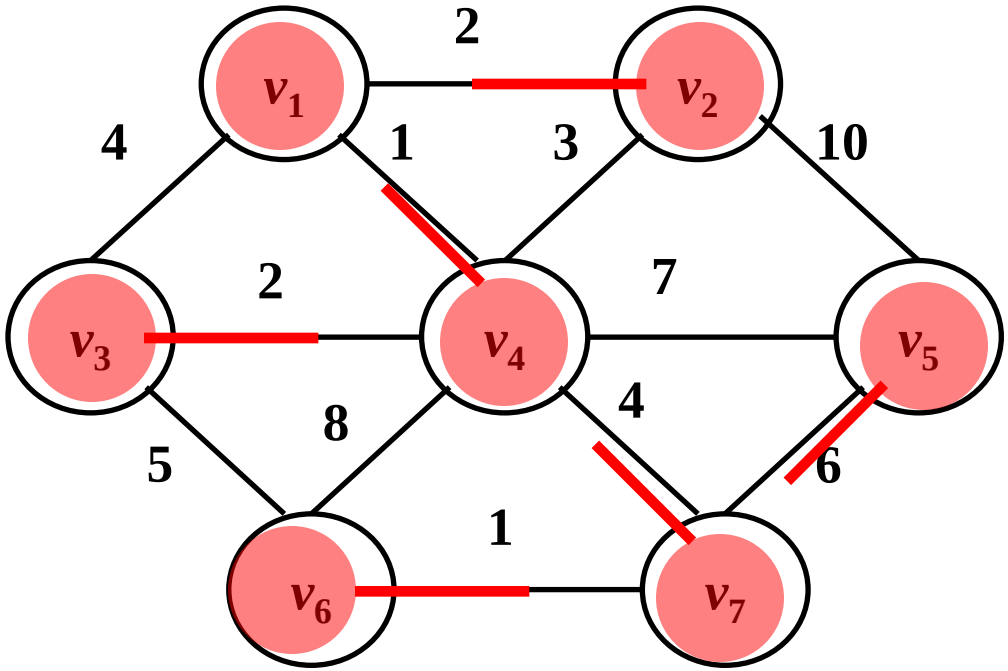
MST性质反证法图示



Prim 算法

- **Prim 算法是贪心法**
  - 由图中任一顶点N开始，初始化MST 为N
  - 选出与N相关联的边中权最小的一条，设其连接N与另一顶点M。把顶点M和边(N，M) 加入MST中
  - 然后选出与N或M相关联的边中权最小的一条，设其连接另一新顶点，将此边和新顶点加入MST中
  - 反复这样处理，每一步都通过选出连接目前已在MST中的某个顶点及另一不在MST中顶点的代价最小的边而扩展MST

Prim 算法图示



Prim 算法

```
void Prim(Graph* G, int* D, int s) {
  int V[G->n()];  // Who's closest
  int i, w;
  for (i=0; i<G->n(); i++) {// Do vertices
    int v = minVertex(G, D);
    G->setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return;
    for (w=G->first(v); w<G->n();
                        w = G->next(v,w))
      if (D[w] > G->weight(v,w)) {
        D[w] = G->weight(v,w);// Update dist
        V[w] = v; // Update who it came from
      }
  }
}
```

优先队列实现的Prim 算法

```
void Prim(Graph* G, int* D, int s) {
int i, v，  w;                    // The current vertex
int V[G-〉 n()];                   // Who's closest
DijkElem temp;  DijkElem E[G->e()]; // Heap array with lots of space
temp.distance = 0; temp.vertex = s;
E[0] = temp;                      // Initialize heap array
minheap< DijkElem,DDComp> H(E, 1, G->e()); // Create the heap
for (i=0; i<G->n(); i++) {        // Now build MST
  do { if (!H.removemin(temp)) return; v = temp.vertex;
       } while (G.getMark(v) == VISITED);
  G->setMark(v, VISITED);
  if (v != s)  AddEdgetoMST(V[v], v);
  if (D[v] == INFINITY) return; // Remaining vertices
  for (w = G->first(v); w<G->n(); w = G->next(v,w))
     if (D[w] > G->weight(v,w)) { D[w] = G->weight(v,w);
          V[w] = v; temp.distance = D[w]; temp.vertex = w;
          H.insert(temp);    // Insert new distance in heap
     }
  }
}
```

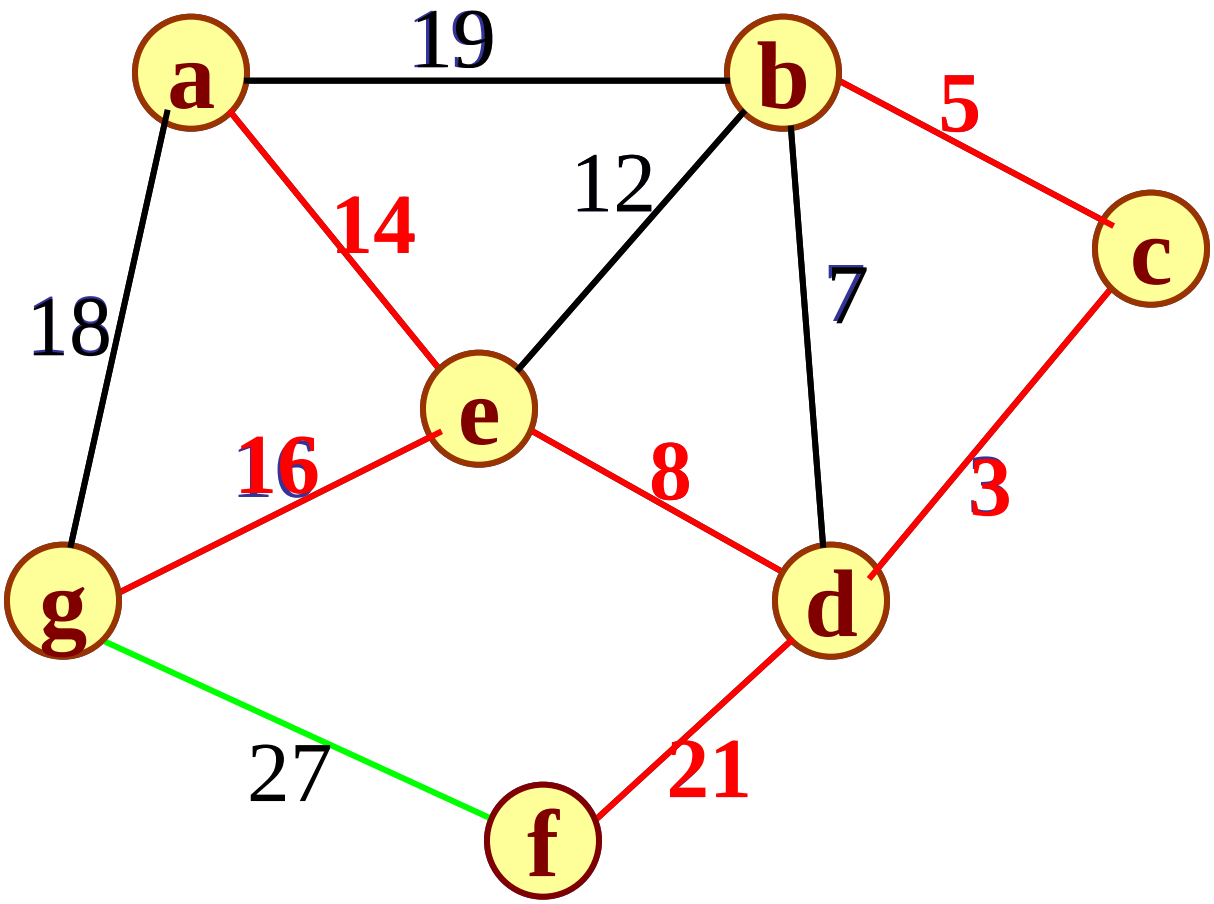
Kruskal 算法

- Kruskal 算法也是一个简单的贪心算法
- 首先，将顶点集分为|V| 个等价类，每个等价类包括一个顶点
- 然后，以权的大小为序处理各条边。如果某条边连接两个不同等价类的顶点，则这条边被加入MST ，两个等价类也被合为一个
- 反复执行此过程直至只余下一个等价类。

Kruskal 算法的关键技术

- 取权值最小的边
  - 首先对边进行完全排序
  - 使用最小值堆来实现，一次取一条边。实际上在完成MST 前仅需访问一小部分边。
- 确定两个顶点是否属于同一等价类
  - 可以树的基于父指针表示法的UNION/FIND算法。

例如:



Kruskal 算法

```
Class KruskElem{
Public:
  int from,to,distance;
  KruskElem() {from=to=distance=-1;}
  KreskElem(int f,int t,int d)
    {from=f;to=t;distance=d;}
};
Kruskel(Graph* G){
  Gentree A(G->n());
  KruskElem E[G->e()];
  int I;
  int edgecnt=0;
```

Kruskal 算法

```
for (i=0;i<G->n();i++)
  for (int w=G->first(I);w<G->weight(I,w)){
    E[edgecnt].distance=G->weight(i,w);
    E[edgecnt].from=i;
    E[edgecnt++].to=w;}
Minheap<KruskElem,KKComp>H(E,edgecnt,edgecnt);
int numMST=G->n():
For (i=0;numMST>1;i++){
  KruskElem temp;
  H.removemin(temp);
  int v=temp.from;int u=temp.to;
  if (A.differ(v,u)){ A.UNION(v,u);AddEdgetoMST(temp.from,temp.to);
numMST--;}
}
}
```

Kruskal 算法的代价

- 使用了路径压缩， differ 和UNION 函数几乎是常数
- 假设可能对几乎所有边都判断过了，则最坏情况下算法时间代价为O(|E|log(|E|))，即堆排序的时间
- 通常情况下只找了接近顶点数目那么多边的情况下，MST 就已经生成，时间代价接近于O(|V|log |E|)

关键路径

问题：

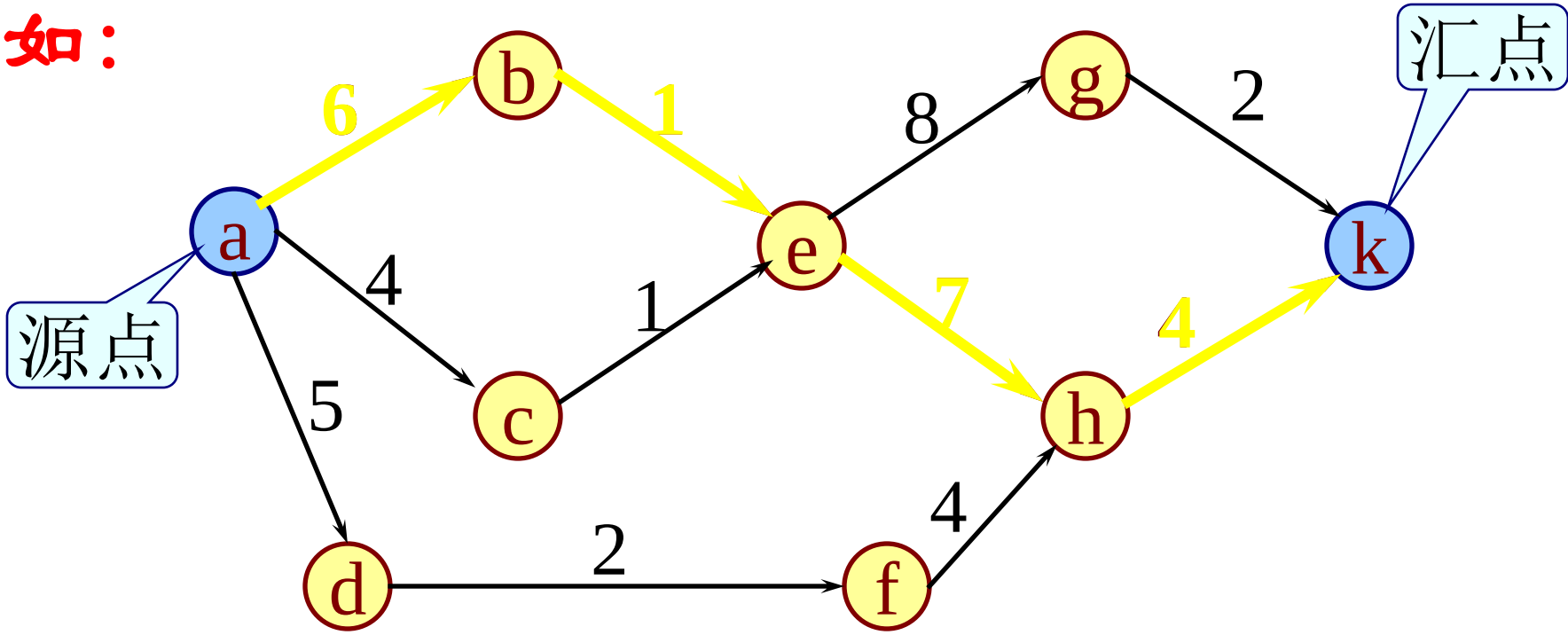
假设以有向网表示一个施工流图，弧上的权值表示完成该项子工程所需时间。

问：哪些子工程项是“关键工程”？

即：哪些子工程项将影响整个工程的完成期限的。

整个工程完成的时间为：从有向图的源点到汇点的最长路径。

例如：



“关键活动”指的是：该弧上的**权值增加** 将使有向图上的**最长路径的长度增加。**

如何求关键活动？

“事件(顶点)” 的 最早发生时间 ***ve*(j)**

ve(j) = 从源点到顶点j的最长路径长度；

“事件(顶点)” 的 最迟发生时间 ***vl*(k)**

vl(k) = 从顶点k到汇点的最短路径长度。

如何求关键活动？

假设第 i 条弧为 <j, k>

则 对第 i 项活动:

“活动(弧)”的 最早开始时间 *ee*(i)

ee(i) = ve(j);

“活动(弧)”的 最迟开始时间 *el*(i)

el(i) = vl(k) – dut(<j,k>);

如何求关键活动？

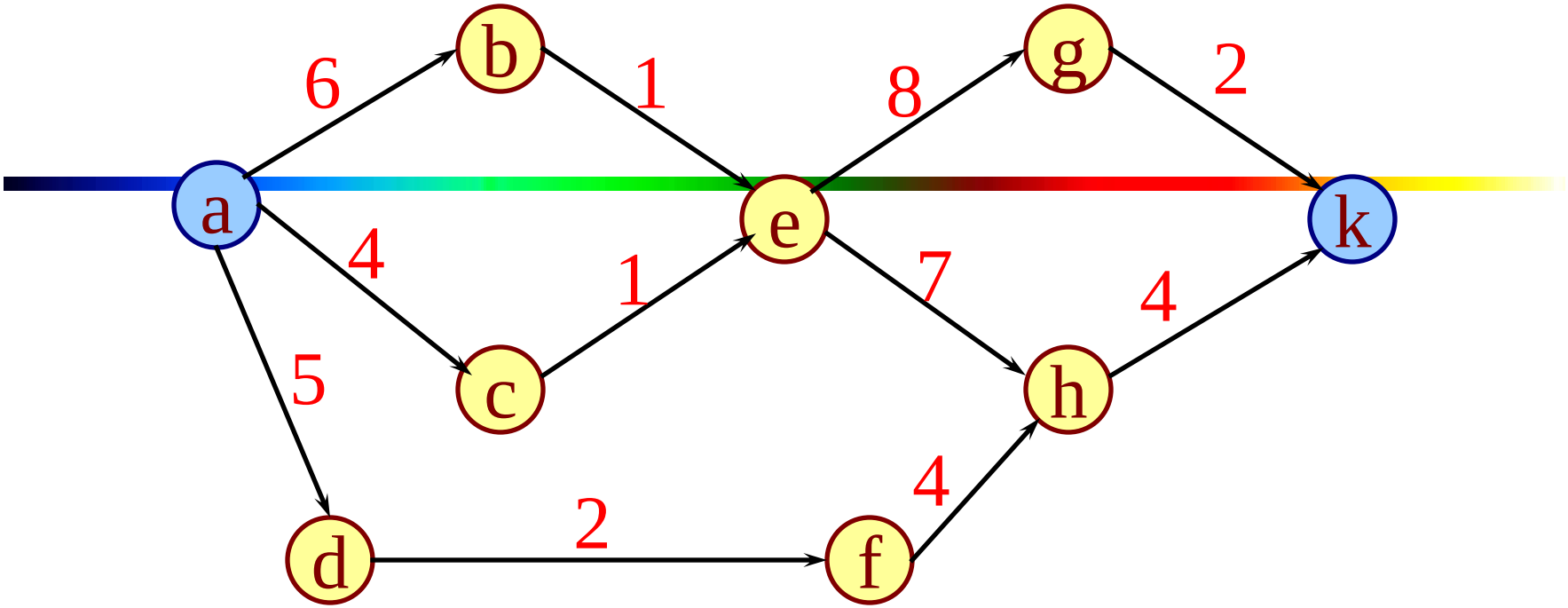
事件发生时间的计算公式：

ve(源点) = 0；

ve(k) = Max{ve(j) + dut(<j, k>)}

vl(汇点) = ve(汇点)；

vl(j) = Min{vl(k) – dut(<j, k>)}



								k
0	6	4	5	7	7	15	14	18
0	6	6	8	7	10	16	14	18

拓扑有序序列：a - d - f - c - b - e - h - g - k

	a	b	c	d	e	f	g	h	
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

算法的实现要点

显然，求ve的顺序应该是按拓扑有序的次序；

而 求vl的顺序应该是按拓扑逆序的次序；

因为 拓扑逆序序列即为拓扑有序序列的逆序列，

因此 应该在拓扑排序的过程中，另设一个“栈”记下拓扑有序序列。

```
Status TopologicalOrder(ALGraph G, Stack &T) {
 // 有向网G采用邻接表存储结构，求各顶点事件的最早发生时间ve(全局变量)。
 // T为拓扑序列定点栈，S为零入度顶点栈。
 // 若G无回路，则用栈T返回G的一个拓扑序列，且函数值为OK，否则为ERROR。
 Stack S;int count=0,k;
 char indegree[40];
 ArcNode *p;
 InitStack(S);
 FindInDegree(G, indegree);  // 对各顶点求入度indegree[0..vernum-1]
 for (int j=0; j<G.vexnum; ++j)     // 建零入度顶点栈S
   if (indegree[j]==0) Push(S, j);  // 入度为0者进栈
 InitStack(T);//建拓扑序列顶点栈T
 count = 0;
 for(int i=0; i<G.vexnum; i++) ve[i] = 0;  // 初始化
 while (!StackEmpty(S)) {
   Pop(S, j);  Push(T, j);  ++count;       // j号顶点入T栈并计数
   for (p=G.vertices[j].firstarc;  p;  p=p->nextarc) {
     k = p->adjvex;           // 对j号顶点的每个邻接点的入度减1
     if (--indegree[k] == 0) Push(S, k);   // 若入度减为0，则入栈
     if (ve[j]+p->info > ve[k])  ve[k] = ve[j]+p->info;
   }//for  *(p->info)=dut(<j,k>)
 }//while
 if (count<G.vexnum) return ERROR;  // 该有向网有回路
 else return OK;
} // TopologicalOrder
```

```
Status CriticalPath(ALGraph G) {   // G为有向网，输出G的各项关键活动。
  Stack T;
  int a,j,k,el,ee,dut;
  char tag;
  ArcNode *p;
  if (!TopologicalOrder(G, T)) return ERROR;
  for(a=0; a<G.vexnum; a++)
    vl[a] = ve[G.vexnum-1];    // 初始化顶点事件的最迟发生时间
  while (!StackEmpty(T))       // 按拓扑逆序求各顶点的vl值
    for (Pop(T, j), p=G.vertices[j].firstarc;  p;  p=p->nextarc) {
      k=p->adjvex;  dut=p->info;     //dut<j,k>
      if (vl[k]-dut < vl[j]) vl[j] = vl[k]-dut;
    }
  for (j=0; j<G.vexnum; ++j)           // 求ee,el和关键活动
    for (p=G.vertices[j].firstarc;  p;  p=p->nextarc) {
      k=p->adjvex;dut=p->info;
      ee = ve[j];  el = vl[k]-dut;
      tag = (ee==el) ? '*' : ' ';
      printf(j, k, dut, ee, el, tag);   // 输出关键活动
    }
  return OK;
} // CriticalPath
```