



学习目标



Chapter 4

Graph Algorithm

- ◆ 图结构是一种非线性结构，反映了数据对象之间的任意关系，在计算机科学、数学和工程中有着非常广泛的应用。
- ◆ 了解图的定义及相关的术语，掌握图的逻辑结构及其特点；
- ◆ 了解图的存储方法，重点掌握图的邻接矩阵和邻接表存储结构；
- ◆ 掌握图的遍历方法，重点掌握图的遍历算法的实现；
- ◆ 了解图的应用，重点掌握最小生成树、双连通性、强连通性、最短路径、拓扑排序和关键路径算法的基本思想、算法原理和实现过程。



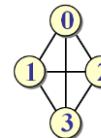
主要内容

- Definitions
- Physical structure design
- Traversing Graph
- Minimum Spanning Trees
- AOV net (Topological Sort)
- AOE net
- Shortest Path algorithms

4.1 Definitions



- A graph $\text{Graph} = (V, E)$ consists of a set of vertices V and a set of edges E .
- Each edge is a pair (v, w) , where $v, w \in V$.



Graph = (V, E)
V={0,1,2,3}
E={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}



Graph = (V, E)
V={0,1,2}
E={<0,1>,<1,0>,<1,2>}

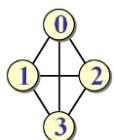
Directed and undirected 有向图与无向图

Edges are sometimes referred to as arcs(弧). If the pair is ordered, then the graph is directed.

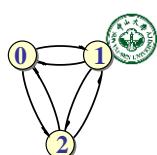
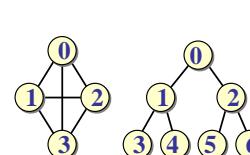
(x,y) and (y,x) are different edge.

For (x, y) , x : rear of arc(弧尾); y : head of arc(弧头)

If (x,y) and (y,x) are same edge, this graph is undirected.



Graph = (V, E)
V={0,1,2}
E={<0,1>,<1,0>,<1,2>}

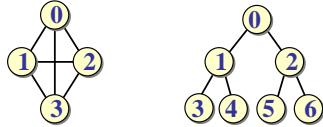


Complete graph 完全图

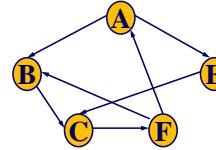
A complete graph is a graph in which there is an edge between every pair of vertices.

- ◆ 在具有 n 个顶点的有向图中，最大弧数为 $n(n-1)$
- ◆ 在具有 n 个顶点的无向图中，最大边数为 $n(n-1)/2$

- Degree of vertices in undirected graph 顶点的度 一个顶点 v 的度是与它相关的边的条数。记作 $TD(v)$ 。



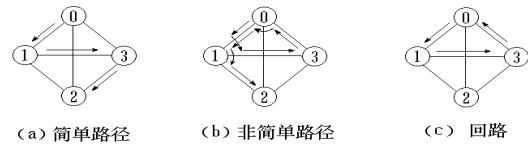
Degree of vertices in directed graph (有向图)



顶点的出度: 以顶点 v 为弧尾的弧的数目;
顶点的入度: 以顶点 v 为弧头的弧的数目。

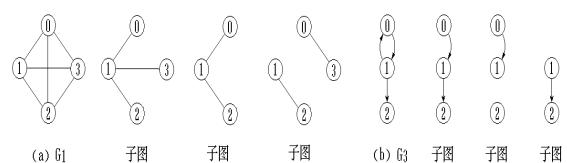
- Path(路径)** A path in a graph is a sequence of vertices w_1, w_2, \dots, w_n , $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.
- The length of such a path is the number of edges on the path, which is equal to $n-1$.
- If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred to as a loop.
- A simple path is a path such that all vertices are distinct, except that the first and last could be the same.

- Cycle (回路)** A cycle in a directed graph is a path of length at least 1 such that $w_1 = w_n$. This cycle is simple if the path is simple.



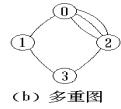
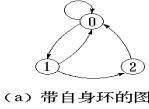
- Adjacent邻接点**
- Vertex w is adjacent to v if and only if $(v, w) \in E$.
- Weight 权** Sometimes an edge has a third component, known as either a weight or a cost.

- Subgraph 子图** Graph $G = (V, E)$ and $G' = (V', E')$. If $V' \subseteq V$ 且 $E' \subseteq E$, then G' is the subgraph of G .





本章不予讨论的图

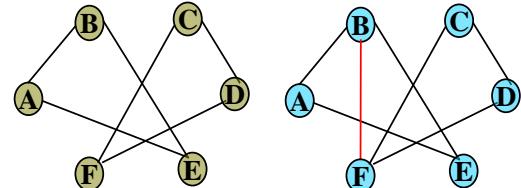


连通图与连通分量

顶点的连通性: 在无向图中, 若从顶点 v_i 到顶点 v_j ($i \neq j$)有路径, 则称顶点 v_i 与 v_j 是连通的。

连通图: 如果一个无向图中任意一对顶点都是连通的, 则称此图是连通图。

连通分量: 非连通图的极大连通子图叫做连通分量。

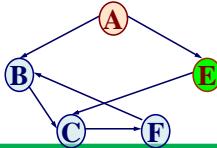
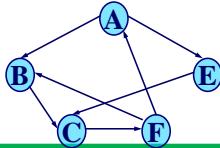


强连通图与强连通分量

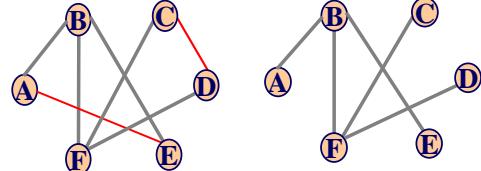
顶点的强连通性: 在有向图中, 若对于每一对顶点 v_i 和 v_j ($i \neq j$), 都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的有向路径, 则称顶点 v_i 与 v_j 是强连通的。

强连通图: 如果一个有向图中任意一对顶点都是强连通的, 则称此有向图是强连通图。

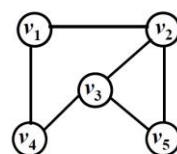
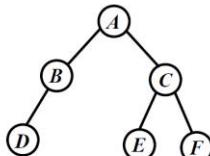
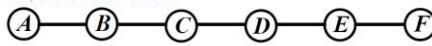
强连通分量: 非强连通图的极大强连通子图叫做强连通分量。



Spanning tree 生成树: 假设一个连通图有 n 个顶点和 e 条边, 其中 $n-1$ 条边和 n 个顶点构成一个极小连通子图, 称该**极小**连通子图为此连通图的**生成树**。

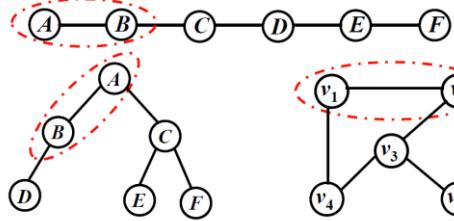


不同逻辑结构之间的比较



- ◆ 在线性结构中, 数据元素之间仅具有**线性关系(1:1)**;
- ◆ 在树型结构中, 结点之间具有**层次关系(1:m)**;
- ◆ 在图型结构中, 任意两个顶点之间都可能有关系($m:n$)。

不同逻辑结构之间的比较



- ◆ 在线性结构中, 元素之间的关系为**前驱**和**后继**;
- ◆ 在树型结构中, 结点之间的关系为**双亲**和**孩子**;
- ◆ 在图型结构中, 顶点之间的关系为**邻接**。

Operations 图的操作



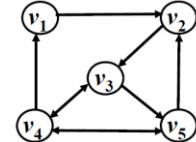
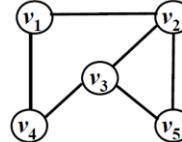
- 顶点定位
- 取顶点
- 求第一个邻接点
- 求下一个邻接点
- 插入顶点
- 插入弧
- 删除顶点
- 删除弧

4.2 Physical design图的存储表示



图的特点：顶点之间的关系是 $m:n$ ，即任何两个顶点之间都可能存在关系（边），无法通过存储位置表示这种任意的逻辑关系，所以，图无法采用连续存储结构。

图是由顶点和边组成的，考虑如何存储顶点、如何存储边与顶点之间的关系。



1. 邻接矩阵 (Adjacency Matrix) 表示法



- **顶点表：**一个记录各个顶点信息的一维数组，
- **邻接矩阵：**一个表示各个顶点之间的关系（边或弧）的二维数组。
- 设图 $G = (V, E)$ 是一个有 n 个顶点的图，则图的邻接矩阵 $G.arcs[n][n]$ 定义为：
- $G.arcs[i][j] = \begin{cases} 1 & \text{若 } \langle v_i, v_j \rangle \text{ 或 } (v_i, v_j) \in E \\ 0 & \text{反之} \end{cases}$

1) 类型定义

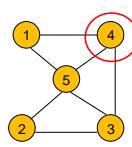
```
#define MAX_VERTEX_NUM 20 //最大顶点数
typedef int
AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

//邻接矩阵类型
typedef struct {
    VertexType vexs[MAX_VERTEX_NUM]; //顶点表
    AdjMatrix arcs; //邻接矩阵
    int vexnum,arcnum; //图的顶点数和弧数
} MGraph;
```

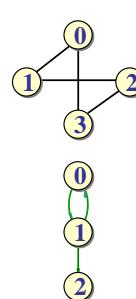
在无向图中，第 i 行（列）1 的个数就是顶点 i 的度。



vexs	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> </table>	0	1	2	3	4	1	2	3	4	5
0	1	2	3	4							
1	2	3	4	5							



$$\text{arcs} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$



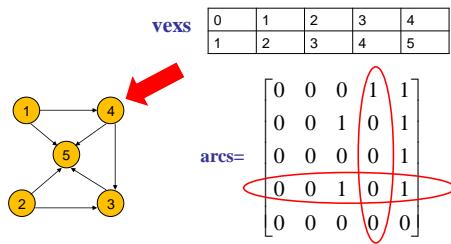
$$\text{A.arcs} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

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

- 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。

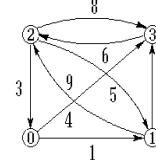


- 在有向图中：第 i 行 1 的个数就是顶点 i 的出度，第 j 列 1 的个数就是顶点 j 的入度。



网的邻接矩阵(adjacency matrix)

$$G.\text{arcs}[i][j] = \begin{cases} W_{ij} & \text{若 } (v_i, v_j) \text{ 或 } (v_j, v_i) \in E \\ \infty & \text{反之} \end{cases}$$



$$A.\text{arcs} = \begin{pmatrix} ① & ① & ② & ③ \\ 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{pmatrix} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{matrix}$$

2. 邻接表 (Adjacency List)



- 邻接表：是图的一种链式存储结构。

▪ 边(弧)的结点结构

adjvex	nextarc	info
--------	---------	------

adjvex; // 该边(弧)所指向的顶点的位置
nextarc; // 指向下一条边(弧)指针
info; // 该边(弧)相关信息的指针

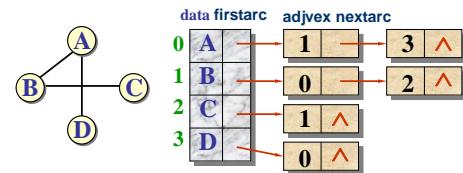
▪ 顶点的结点结构

data	firstarc
------	----------

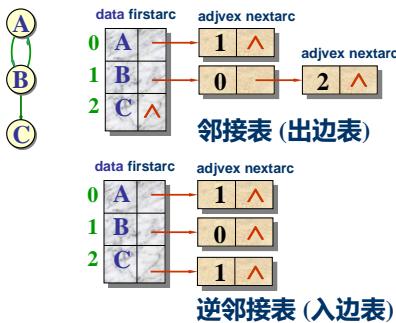
data; // 顶点信息

firstarc; // 指向第一条依附该顶点的边(弧)

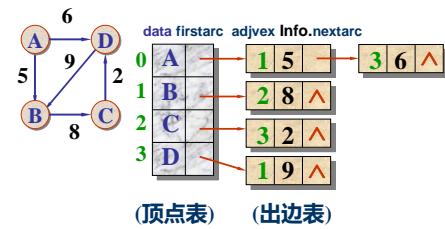
▪ 无向图的邻接表



▪ 有向图的邻接表和逆邻接表



▪ 网络 (带权图) 的邻接表





图的邻接表存储表示

```
#define MAX_VERTEX_NUM 20
typedef struct ArcNode {
    int adjvex; // 该边(弧)所指向的顶点的位置
    struct ArcNode *nextarc; // 指向下一条边(弧)指针
    InfoType *info; // 该边(弧)相关信息的指针
} ArcNode;
typedef struct VNode {
    VertexType data; // 顶点信息
    ArcNode *firstarc; // 指向第一条依附该顶点的边(弧)
} VNode, AdjList[MAX_VERTEX_NUM];
```



```
typedef struct {
    AdjList vertices;
    Int vexnum, arcnum; // 图的当前顶点数和边(弧)数
    Int kind; // 图的种类标志
} ALGraph;
```

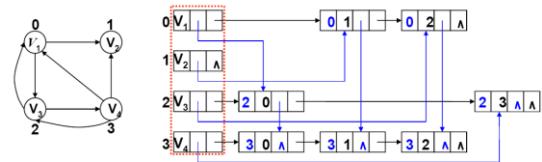


- 带权图的边结点中 **info** 保存该边上的权值。
- 顶点 v_i** 的边链表的头结点存放在下标为 i 的顶点数组中。
- 在邻接表的边链表中，**各个边结点的链入顺序任意**，视边结点输入次序而定。
- 设图中有 n 个顶点， e 条边，则**用邻接表表示无向图时**，需要 n 个顶点结点， $2e$ 个边结点；**用邻接表表示有向图时**，若不考虑逆邻接表，只需 n 个顶点结点， e 个边结点。
- 建立邻接表的时间复杂度为 $O(n^2)$ 。若顶点信息即为顶点的下标，则时间复杂度为 $O(n+e)$ 。



3、有向图的十字链表 (Orthogonal List) 表示

- ↓ **十字链表**，是有向图的另一种链式存储结构。
- 可以看成是将有向图的正邻接表和逆邻接表结合起来得到的一种链式存储结构。
 - 也即弧头相同的弧在同以一链表上，弧尾相同的弧也在同一链表上。
 - 从横向上看是正邻接表，从纵向上看是逆邻接表。



有向图的十字链表 (Orthogonal List) 表示

◆ **结点结构:** 弧结点结构

tailvex	headvex	hlink	tlink	info
---------	---------	-------	-------	------

tailvex: 尾域，指示弧尾顶点在图中的位置
headvex: 头域，指示弧头顶点在图中的位置
hlink: 链域，指向弧头相同的下一条弧。
tlink: 链域，指向弧尾相同的下一条弧。
info: 数据域，指向该弧的相关信息。

◆ **头结点 (顶点结点) 结构**

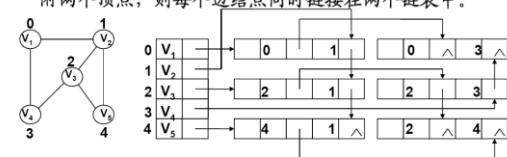
data	firstin	firstout
------	---------	----------

data: 数据域，存储和顶点相关的信息，如顶点名称等。
firstin: 链域，指向以该顶点为弧头的第一个弧结点。
firstout: 链域，指向以该顶点为弧尾的第一个弧结点。

Diagram illustrating the Orthogonal List structure for a directed graph with 4 nodes (V0 to V3) and 6 arcs. The grid shows the connections between nodes. Each node has a vertical column of pointers (firstin) and a horizontal row of pointers (firstout). The connections are: V0 to V1, V0 to V2, V1 to V2, V1 to V3, V2 to V3, and V2 to V0.

4、无向图的邻接多重表 (Adjacency Multilist) 表示

- ◆ **邻接多重表**，是无向图的另一种链式存储结构。
- 邻接多重表可以看作是对无向图的**邻接矩阵**的一种压缩表示，这种结构在**边的操作**上会方便很多。
 - 邻接多重表的结构与十字链表类似。在邻接多重表中，所有依附于同一顶点的边串联在同一链表中，由于每条边依附两个顶点，则每个边结点同时链接在两个链表中。





无向图的邻接多重表(Adjacency Multilist)表示

◆ 结点结构：边结点结构

mark	ivex	ilink	jvex	jlink	info
------	------	-------	------	-------	------

mark: 标志域，用以标记该条边是否被搜索过

ivex和**jvex**: 为该边依附的两个顶点在图中的位置

ilink: 链域，指向下一条依附于顶点ivex的边

jlink: 链域，指向下一条依附于顶点jvex的边

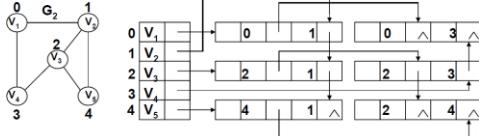
info: 数据域，指向和边相关的各种信息的指针域

头结点(顶点结点)结构

data	firstedge
------	-----------

Data: 数据域，存储和该顶点相关的信息

firstedge: 链域，指示第一条依附于该顶点的边



4.3 Traversing Graph图的遍历

- 从图中某一顶点出发访遍图中其余顶点，且使每个顶点仅被访问一次，就叫做图的遍历 (Traversing Graph)。
- 图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等算法的基础。



图的遍历要解决的关键问题

- 在图中，如何选取遍历的起始顶点？
 - 解决办法：从编号小的顶点开始。
- 从某个起点始可能到达不了所有其它顶点，怎么办？
 - 解决办法：多次调用从某顶点出发遍历图的算法。
- 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访问过的顶点。如何避免某些顶点可能会被重复访问？
 - 解决办法：附设访问标志数组visited[n]。
- 在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，如何选取下一个要访问的顶点？



- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 **visited []**，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 **visited [i]** 为 1，防止它被多次访问。

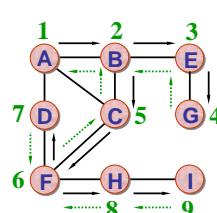
两个遍历图的路径：深度优先搜索、广度优先搜索。



4.3.1 深度优先搜索DFS (Depth First Search)

- 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 *w₁*；再从 *w₁* 出发，访问与 *w₁* 邻接但还没有访问过的顶点 *w₂*；然后再从 *w₂* 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。

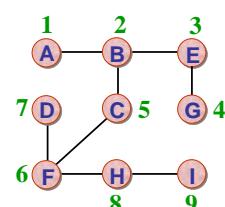
▪ 深度优先搜索过程



前进 ——————

回退 ——————

深度优先生成树





• DFS 的进一步描述

- (1) 从图中的某个顶点 v 出发, 访问之;
- (2) 依次从顶点 v 的未被访问过的邻接点出发, 深度优先遍历图, 直到图中所有和顶点 v 有路径相通的顶点都被访问到;
- (3) 若此时图中尚有顶点未被访问到, 则另选一个未被访问过的顶点作起始点, 重复上述 (1) (2) 的操作, 直到图中所有的顶点都被访问到为止。



图的深度优先搜索算法

```
void Graph_Traverse (AdjGraph G) {
    int * visited = new int [NumVertices];
    for ( int i = 0; i < G.n; i++ )
        visited [i] = 0; //访问数组 visited 初始化
    for ( int i = 0; i < G.n; i++ )
        if ( ! visited[i] ) DFS (G, i, visited );
            //从顶点 i 出发开始搜索
    delete [ ] visited; //释放 visited
}
```

```
void DFS (AdjGraph G, int v, int visited []) {
    cout << GetValue (G, v) << ' ';
    visited[v] = 1; //顶点 v 作访问标记
    int w = GetFirstNeighbor (G, v);
    //取 v 的第一个邻接顶点 w
    while ( w != -1 ) { //若邻接顶点 w 存在
        if ( !visited[w] ) DFS (G, w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor (G, v, w );
        //取顶点 v 排在 w 后的下一个邻接顶点
    }
}
```



邻接矩阵表示

```
void DFS (AdjGraph G, int v, int visited []) {
    cout << GetValue (G, v) << ' ';
    visited[v] = 1;
    int w = GetFirstNeighbor (G, v);
    while ( w != -1 ) {
        if ( !visited[w] ) DFS (G, w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor (G, v, w );
        //取顶点 v 排在 w 后的下一个邻接顶点
    }
}
```



邻接表表示

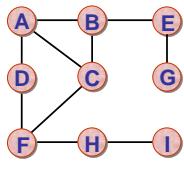
```
void DFS (AdjGraph G, int v, int visited []) {
    cout << GetValue (G, v) << ' ';
    visited[v] = 1;
    int w = GetFirstNeighbor (G, v);
    while ( w != -1 ) {
        if ( !visited[w] ) DFS (G, w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor (G, v, w );
        //取顶点 v 排在 w 后的下一个邻接顶点
    }
}
```



算法分析

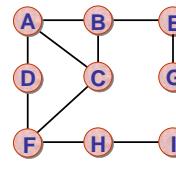
- 图中有 n 个顶点, e 条边。
- 如果用邻接表表示图, 沿 $Firstarc \rightarrow nextarc$ 链可以找到某个顶点 v 的所有邻接顶点 w 。由于总共有 $2e$ 个边结点, 所以扫描边的时间为 $O(e)$ 。而且对所有顶点递归访问1次, 所以遍历图的时间复杂性为 $O(n+e)$ 。
- 如果用邻接矩阵表示图, 则查找每一个顶点的所有边, 所需时间为 $O(n)$, 则遍历图中所有的顶点所需的时间为 $O(n^2)$ 。

给出不同存储状态下的结果(1)



1	A	→	2	→	3	→	4	↑
2	B	→	1	→	3	→	5	↑
3	C	→	1	→	2	→	6	↑
4	D	→	1	→	6	↑		
5	E	→	2	→	7	↑		
6	F	→	3	→	4	→	8	↑
7	G	→	5	↑				
8	H	→	6	→	9	↑		
9	I	→	8	↑				

给出不同存储状态下的结果(2)



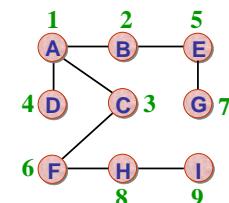
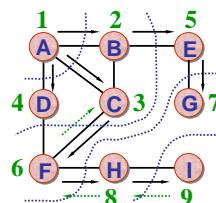
1	A	→	4	→	3	→	2	↑
2	B	→	1	→	3	→	5	↑
3	C	→	6	→	1	→	2	↑
4	D	→	1	→	6	↑		
5	E	→	2	→	7	↑		
6	F	→	4	→	3	→	8	↑
7	G	→	5	↑				
8	H	→	6	→	9	↑		
9	I	→	8	↑				

4.3.2 广度优先搜索BFS (Breadth First Search)



- BFS在访问了起始顶点 v 之后, 由 v 出发, 依次访问 v 的各个未被访问过的邻接顶点 w_1, w_2, \dots, w_t , 然后再顺序访问 w_1, w_2, \dots, 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。

▪ 广度优先搜索过程



广度优先生成树

• BFS 的进一步描述



- 从图中的某个顶点 v 出发, 访问之;
- 依次访问顶点 v 的各个未被访问过的邻接点, 将 v 的全部邻接点都访问到;
- 分别从这些邻接点出发, 依次访问它们的未被访问过的邻接点, 并使“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, 直到图中所有已被访问过的顶点的邻接点都被访问到。

- 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下一层的顶点, 以便于向下一层访问。

图的广度优先搜索算法

```
void Graph_Traverse (AdjGraph G) {
    .....
    for ( int i = 0; i < G.n; i++ )
        if ( ! visited[i] ) BFS (G, i, visited );
    .....
}
```

```

void BFS (AdjGraph G, int v, int visited[ ]) {
    cout << GetValue (v) << ' '; visited[v] = 1;
    Queue<int> q; InitQueue(&q);
    EnQueue (&q, v); //进队列
    while ( ! QueueEmpty (&q) ) { //队空搜索结束
        DeQueue (&q, v);
        int w = GetFirstNeighbor (G, v);
        while ( w != -1 ) { //若邻接顶点 w 存在
            if ( !visited[w] ) { //未访问过
                cout << GetValue (w) << ' ';

```



```

        visited[w] = 1; EnQueue (&q, w);
    }
    w = GetNextNeighbor (G, v, w);
}
//重复检测 v 的所有邻接顶点
} //外层循环，判队列空否
delete [ ] visited;
}

```



算法分析

- 如果使用邻接表表示图，则循环的总时间代价为 $d_0 + d_1 + \dots + d_{n-1} = O(e)$ ，其中的 d_i 是顶点 i 的度。
- 如果使用邻接矩阵，则对于每一个被访问过的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为 $O(n^2)$ 。



邻接矩阵表示

```

void BFS (AdjGraph G, int v, int visited[ ]) {
    cout << GetValue (v) << ' '; visited[v] = 1;
    Queue<int> q; InitQueue(&q);
    EnQueue (&q, v); //进队列
    while ( ! QueueEmpty (&q) ) { //队空搜索结束
        DeQueue (&q, v);
        int w = GetFirstNeighbor (G, v);
        while ( w != -1 ) { //若邻接顶点 w 存在
            if ( !visited[w] ) { //未访问过
                cout << GetValue (w) << ' ';

```



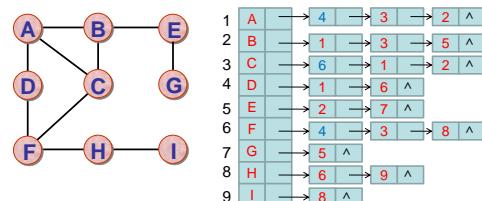
```

        visited[w] = 1; EnQueue (&q, w);
    }
    w = GetNextNeighbor (G, v, w);
}
//重复检测 v 的所有邻接顶点
} //外层循环，判队列空否
delete [ ] visited;
}

```



给出不同存储状态下的结果(2)



4.3.3 图的连通性问题



口无向图的连通分量与生成树

对于无向图，对其进行遍历时：

- ◆ 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
- ◆ 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

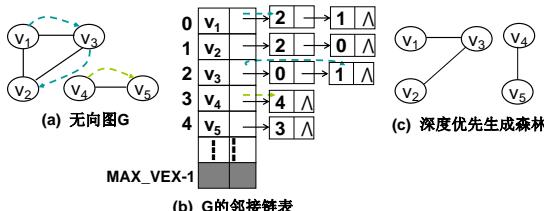
对非连通图，从某个顶点出发进行遍历，只能遍历到它所在的连通子图上的所有顶点。

依次从每个未访问过的顶点出发进行遍历，就可以遍历完所有的顶点，并且可以得到非连通图的连通分量个数。

```
//依次从每个未访问过的顶点
//出发DFS
subnets=0;
for(int n=1; n<=nodes; n++)
{
    if(!visited[n])
    {
        DFS(n);
        subnets++;
    }
}
```

```
//从顶点n出发，DFS遍历
int DFS(int n)
{
    visited[n]=1;
    for(int i=1; i<=nodes; i++)
    {
        if( node[n][i]==1 && !visited[i] )
            DFS(i);
    }
    return 0;
}
```

如图所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，2次调用DFS所得到的顶点访问序列集是：{ v1 ,v3 ,v2 }和{ v4 ,v5 }



无向图及深度优先生成森林

(1) 若 $G=(V,E)$ 是无向连通图，顶点集和边集分别是 $V(G)$ ， $E(G)$ 。若从 G 中任意点出发遍历时， $E(G)$ 被分成两个互不相交的集合：

$T(G)$ ：遍历过程中所经过的边的集合；

$B(G)$ ：遍历过程中未经过的边的集合；

显然： $E(G)=T(G) \cup B(G)$ ， $T(G) \cap B(G)=\emptyset$

显然，图 $G'=(V, T(G))$ 是 G 的极小连通子图，且 G' 是一棵树。 G' 称为图 G 的一棵生成树。

从任意点出发按DFS算法得到生成树 G' 称为深度优先生成树；按BFS算法得到的 G' 称为广度优先生成树。

(2) 若 $G=(V,E)$ 是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集： $V_1(G), V_2(G), \dots, V_n(G)$ 和相应所经过的边集： $T_1(G), T_2(G), \dots, T_n(G)$ 。

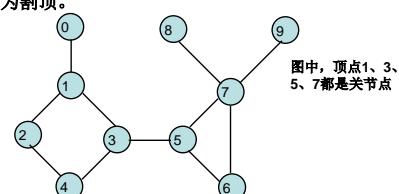
则对应的顶点集和边集的二元组： $G_i=(V_i(G), T_i(G))$

($1 \leq i \leq n$) 是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。

说明：当给定无向图要求画出其对应的生成树或生成森林时，必须先给出相应的邻接表，然后才能根据邻接表画出其对应的生成树或生成森林。

口 关节点及重连通图

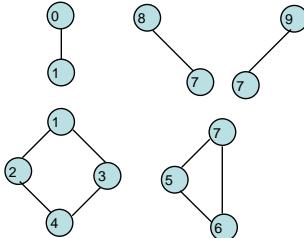
关节点：在一个无向连通图 G 中，当且仅当删去 G 中的顶点 v 及其所关联的边后，可将图分割成2个或2个以上的连通分量，则称顶点 v 为关节点(Articulation Point)，或者称为割顶。



✓ 重连通图：没有关节点的连通图。在重连通图上，任何一对顶点之间至少存在有2条路径，在删去某个顶点及其所关联的边时，也不破坏图的连通性。

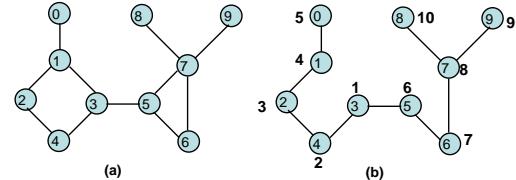
重连通分量

- 如果连通图G不是重连通图，那么它可以包括几个重连通分量。一个连通图的重连通分量是该图的极大连通子图。
- 图中包含了6个连通分量



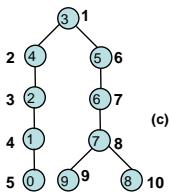
求关节点的算法

- 从顶点3出发进行深度优先搜索，得到图(b)所示的生成树，并改画成图(c)所示的树形形状。
- 图(c)中每个顶点外侧的数字标明了进行深度优先搜索时各顶点访问的次序，称为顶点的深度优先数。



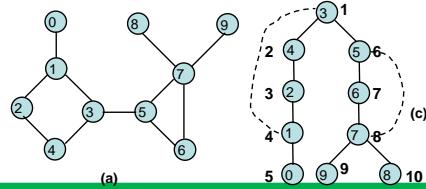
注意：

- 如果 u 和 v 是2个顶点，且在深度优先搜索生成树中 u 是 v 的祖先，则 u 的深度优先数小于 v ， u 先于 v 被访问。



回边与交叉边

- 回边：当且仅当 u 在生成树中是 v 的祖先，或者 v 是 u 的祖先，非生成树的边 (u, v) 才成为一条回边。如图(a)中的(1,3)、(5,7)都是回边。
- 交叉边：除生成树的边、回边外，原图中的其他边称为交叉边。
- 一旦生成树确定以后，那么原图中的边只可能有回边和生成树的边，交叉边实际上是不存在的。

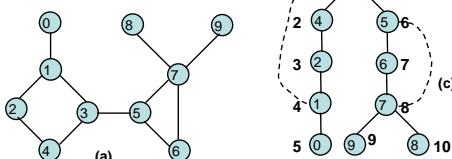


顶点 u 是关节点的充要条件：

- 如果顶点 u 是深度优先搜索生成树的根，则 u 至少有2个子女；
✓ 删除 u ，它的子女所在的子树就断开了。
- 如果 u 不是生成树的根，则它至少有一个子女 w ，从 w 出发，不可能通过 w 、 w 的子孙，以及一条回边组成的路径到达 u 的祖先。(这时删去顶点 u 及其所关联的边，则以顶点 w 为根的子树就从搜索树中脱离了。)

顶点5是关节点

顶点6不是关节点



有向图的强连通分量

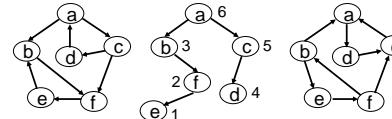
对于有向图，在其每一个**强连通分量**中，**任何两个顶点都是可达的**。 $\forall V \in G$ ，与 V 可相互到达的所有顶点就是包含 V 的强连通分量的所有顶点。

设从 V 可到达(**以 V 为起点的所有有向路径的终点**)的顶点集合为 $T_1(G)$ ，而到达 V (**以 V 为终点的所有有向路径的起点**)的顶点集合为 $T_2(G)$ ，则包含 V 的强连通分量的顶点集合是： $T_1(G) \cap T_2(G)$ 。



求有向图G的强连通分量的基本步骤是：

- (1) 对G进行深度优先遍历，生成G的深度优先生成森林T。
- (2) 对森林T的顶点按中序遍历顺序进行编号。
- (3) 改变G中每一条弧的方向，构成一个新的有向图G'。
- (4) 按(2)中标出的顶点编号，从编号最大的顶点开始对G'进行深度优先搜索，得到一棵深度优先生成树。若一次完整的搜索过程没有遍历G'的所有顶点，则从未访问的顶点中选择一个编号最大的顶点，由它开始再进行深度优先搜索，并得到另一棵深度优先生成树。在该步骤中，每一次深度优先搜索所得到的生成树中的顶点就是G的一个强连通分量的所有顶点。
- (5) 重复步骤(4)，直到G'中的所有顶点都被访问。



(a) 有向图G (b) 执行步骤(1)和(2) (c) 执行步骤(3) (d) 执行步骤(4)和(5)
利用深度优先搜索求有向图的强连通分量

在算法实现时，建立一个数组in_order[n]存放深度优先生成森林的中序遍历序列。对每个顶点v，在调用DFS函数结束时，将顶点依次存放在数组in_order[n]中。图采用十字链表作为存储结构最合适。

4.4 最小生成树—连通网的最小代价生成树



Minimum Spanning Trees

◆ 生成树的代价

- 设 $G = (V, E)$ 是一个无向连通网， E 中每一条边 (u, v) 上的权值 $c(u, v)$ ，称为 (u, v) 的边长。
- 图G的生成树上各边的权值（边长）之和称为该生成树的代价。

◆ 最小生成树(Minimum-Cost Spanning Tree, MST)

- 在图G所有生成树中，代价最小的生成树称为最小生成树
- ◆ 最小生成树的概念可以应用到许多实际问题中。

例如，在n个教室之间建造局域网络，至少要架设n-1条通信线路，而两个教室之间的距离可能不同，从而架设通信线路的造价就是不一样的，那么如何设计才能使得总造价最小？



最小生成树（代价之和最小的生成树）

• 构造准则：

- 尽可能用网络中权值最小的边；
- 必须使用且仅使用 $n-1$ 条边来联结网络中的 n 个顶点；
- 不能使用产生回路的边。

算法：

- 1) 普里姆算法
- 2) 克鲁斯卡尔算法

1) 普里姆(Prim)算法



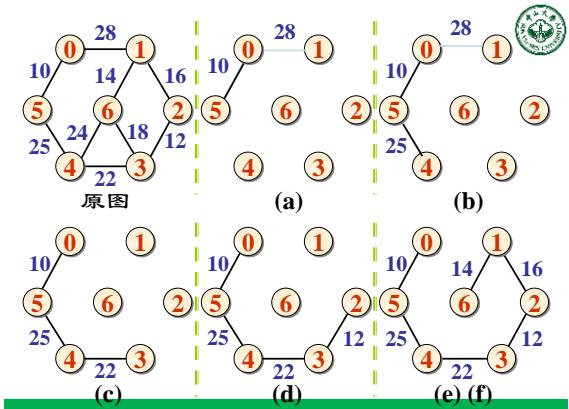
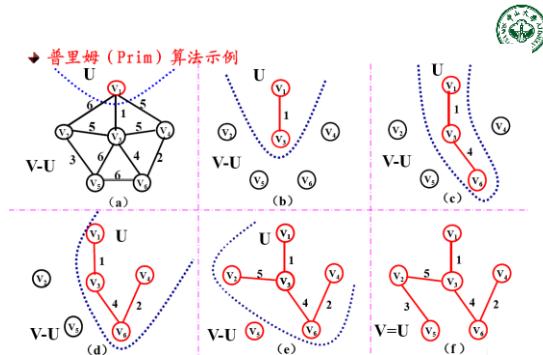
• 普里姆算法的基本思想：

从连通网络 $N = \{V, E\}$ 中的某一顶点 u_0 出发，选择与它关联的具有最小权值的边 (u_0, v) ，将其顶点加入到生成树的顶点集合U中。
以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边 (u, v) ，把该边加入到生成树的边集TE中，把它的顶点加入到集合U中。如此重复执行，直到网络中的所有顶点都加入到生成树顶点集合U中为止。



• 普里姆算法进一步描述：

- (1) 在连通网的顶点集合V中，任选一个顶点，构成最小生成树的初始顶点集合U；
- (2) 在U和V-U中各选一个顶点，使得该边的权值最小，把该边加入到最小生成树的边集TE中，同时将V-U中的该顶点并入到U中；
- (3) 反复执行第(2)步，直至 $V-U=\emptyset$ 为止。



注意：

- 若候选轻权边集中的轻权边不止一条，可任选其中的一条扩充到T中。
- 连通图的最小生成树不一定是唯一的，但它们的权相等。

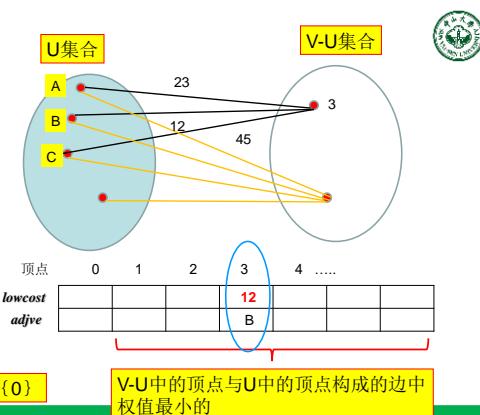
• 设置一个辅助数组closedge[]：

- lowcost域**: 存放生成树顶点集合内顶点到生成树外各顶点的当前最小权值;
- adjvex域**: 记录生成树顶点集合外各顶点距离集合内哪个顶点最近(即权值最小, 对应生成树中的顶点)。

顶点	0	1	2	3	4
lowcost						
adjvex						

U= {0}

V-U中的顶点与U中的顶点构成的边中权值最小的



0	28	∞	∞	∞	10	∞
28	0	16	∞	∞	∞	14
∞	16	0	12	∞	∞	∞
∞	∞	12	0	22	∞	18
∞	∞	∞	22	0	25	24
10	∞	∞	∞	25	0	∞
∞	14	∞	18	24	∞	0

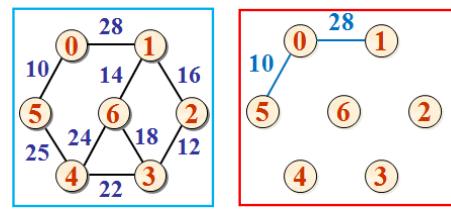


- 若选择从顶点0出发，即 $u_0 = 0$ ，则两个辅助数组的初始状态为：

lowcost	0	1	2	3	4	5	6
adjvex	-1	0	0	0	0	0	0

- 然后反复做以下工作：

- 在 lowcost [] 中选择 $\text{adjvex}[i] \neq -1$ & & $\text{lowcost}[i]$ 最小的边，用 v 标记它。则选中的权值最小的边为 $(\text{adjvex}[v], v)$ ，相应的权值为 $\text{lowcost}[v]$ 。

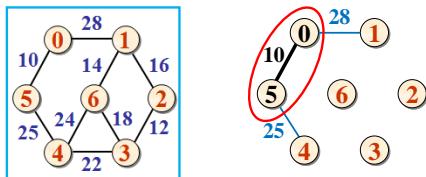


lowcost	0	1	2	3	4	5	6
adjvex	-1	0	0	0	0	0	0

选 v=5 | 选边 (0,5)

顶点v=5加入生成树顶点集合：

lowcost	0	1	2	3	4	5	6
adjvex	-1	0	0	0	0	-1	0



- 将 $\text{adjvex}[v]$ 改为 -1，表示它已加入生成树顶点集合。
- 将边 $(\text{adjvex}[v], v, \text{lowcost}[v])$ 加入生成树的边集合。
- 取 $\text{lowcost}[i] = \min\{ \text{lowcost}[i], \text{Edge}[v][i] \}$ ，即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 v 的距离 $\text{Edge}[v][i]$ 与原来它们到生成树顶点集合中顶点的最短距离 $\text{lowcost}[i]$ 做比较，取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。

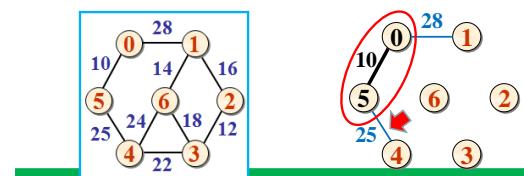
- 如果生成树顶点集合外顶点 i 到刚加入该集合的新顶点 v 的距离比原来它到生成树顶点集合中顶点的最短距离还要近，则修改 $\text{adjvex}[i]$ ： $\text{adjvex}[i] = v$ 。表示生成树外顶点 i 到生成树内顶点 v 当前距离最近。



lowcost	0	1	2	3	4	5	6
adjvex	-1	0	0	0	0	-1	0

顶点v=5加入生成树顶点集合：

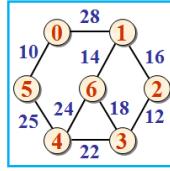
lowcost	0	1	2	3	4	5	6
adjvex	-1	0	0	0	0	5	0



顶点v=5加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞
adjvex	-1	0	0	0	5	-1	0

选 v=4 | 选边 (5,4)



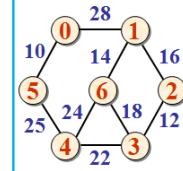
边 (0,5,10) 加入生成树



顶点v=4加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
adjvex	-1	0	0	4	-1	-1	4

选 v=3 | 选边 (4,3)



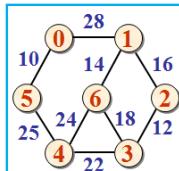
边 (5,4,25) 加入生成树



顶点v=3加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18
adjvex	-1	0	3	-1	-1	-1	3

选 v=2 | 选边 (3,2)



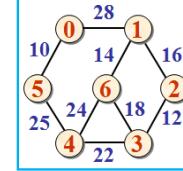
边 (4,3,22) 加入生成树



顶点v=2加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	18
adjvex	-1	2	-1	-1	-1	-1	3

选 v=1 | 选边 (2,1)



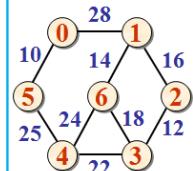
边 (3,2,12) 加入生成树



顶点v=1加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
adjvex	-1	-1	-1	-1	-1	-1	1

选 v=6 | 选边 (1,6)

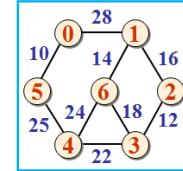


边 (2,1,16) 加入生成树



顶点v=6加入生成树顶点集合:

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
adjvex	-1	-1	-1	-1	-1	-1	-1



边 (1,6,14) 加入生成树



最后生成树中边集合里存入得各条边为：

(0, 5, 10), (5, 4, 25), (4, 3, 22),
(3, 2, 12), (2, 1, 16), (1, 6, 14)



利用普里姆算法建立最小生成树

```
void Prim ( Graph G, MST& T, int u ) {
    float * lowcost = new float[NumVertices];
    int * nearvex = new int[NumVertices];
    for ( int i = 0; i < NumVertices; i++ ) {
        lowcost[i] = G.Edge[u][i]; //V到各点代价
        adjvex[i] = u;           //及最短带权路径
    }
}
```

```
adjvex[u] = -1;      //加到生成树顶点集合
int k = 0;           //存放最小生成树结点的指针
for ( i = 0; i < G.n; i++ )
    if ( i != u ) { //循环n-1次, 加入n-1条边
        EdgeData min = MaxValue; int v = 0;
        for ( int j = 0; j < NumVertices; j++ )
            if ( adjvex[j] != -1 && //=-1不参选
                  lowcost[j] < min )
                { v = j; min = lowcost[j]; }
        //求生成树外顶点到生成树内顶点具有最
        //小权值的边, v是当前具最小权值的边
```



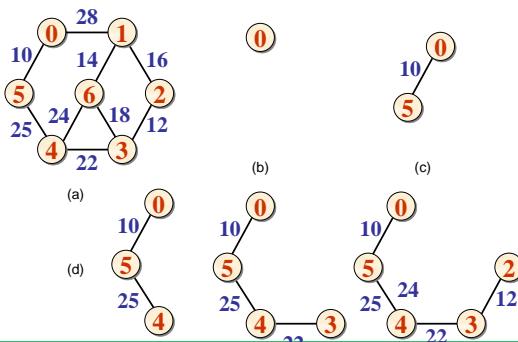
```
if ( v ) { //v=0表示再也找不到要求顶点
    T[k].tail = adjvex[v]; //选边加入生成树
    T[k].head = v;
    T[k++].cost = lowcost[v];
    adjvex[v] = -1; //该边加入生成树标记
    for ( j = 0; j < G.n; j++ )
        if ( adjvex[j] != -1 &&
             G.Edge[v][j] < lowcost[j] ) {
            lowcost[j] = G.Edge[v][j]; //修改
            adjvex[j] = v;
        }
    }
}
```

} //循环n-1次, 加入n-1条边
}

- 分析以上算法, 设连通网络有 n 个顶点, 则该算法的时间复杂度为 $O(n^2)$, 它适用于边稠密的网络。
- 注意: 当各边有相同权值时, 由于选择的随意性, 产生的生成树可能不唯一。当各边的权值不相同时, 产生的生成树是唯一的。



普里姆算法构造最小生成树的过程



2) 克鲁斯卡尔 (Kruskal) 算法



• 克鲁斯卡尔算法的基本思想:

设有一个有 n 个顶点的连通网络 $N = \{ V, E \}$, 最初先构造一个只有 n 个顶点, 没有边的非连通图 $T = \{ V, \emptyset \}$, 图中每个顶点自成一个连通分量。当在 E 中选到一条具有最小权值的边时, 若该边的两个顶点落在不同的连通分量上, 则将此边加入到 T 中; 否则将此边舍去, 重新选择一条权值最小的边。如此重复下去, 直到所有顶点在同一个连通分量上为止。



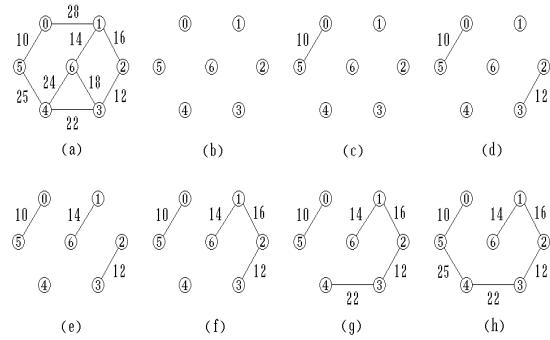
应用克鲁斯卡尔算法构造最小生成树的过程



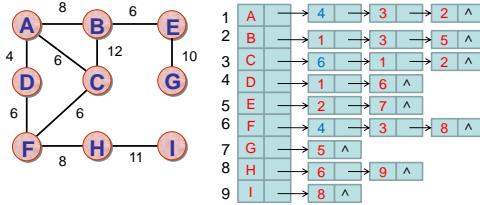
◆ 克鲁斯卡尔 (Kruskal) 算法

■ 实现步骤:

1. 初始化: $U=V$; $TE=\{\}$;
2. 循环直到 T 中的连通分量个数为 1
 - 2.1 在 E 中选择最短边 (u, v) ;
 - 2.2 如果顶点 u, v 位于 T 的两个不同连通分量, 则
 - 2.2.1 将边 (u, v) 并入 TE ;
 - 2.2.2 将这两个连通分量合为一个;
 - 2.3 在 E 中标记边 (u, v) , 使得 (u, v) 不参加后续最短边的选取



给出不同存储状态下的结果



4.5 AOV网(Activity On Vertices)



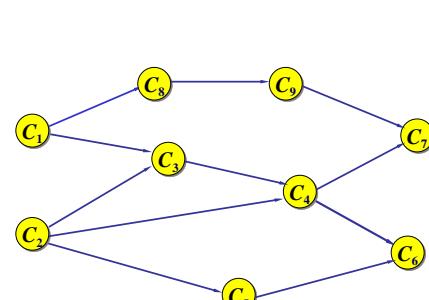
- 用顶点表示活动, 用有向边 $<v_i, v_j>$ 表示活动间的优先关系。 v_i 必须先于活动 v_j 进行。这种有向图叫做顶点表示活动的AOV网络(Activity On Vertices)。

课程代号

课程名称

先修课程

C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	
C_4	数据结构	
C_5	高级语言程序设计	
C_6	编译方法	C_1, C_2
C_7	操作系统	C_3, C_2
C_8	普通物理	C_2
C_9	计算机原理	C_5, C_4

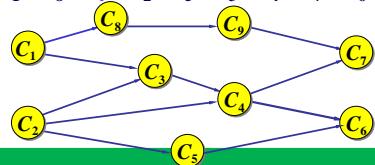


学生课程学习工程图

- (Topological sequence) 拓扑序列：即将各个顶点（代表各个活动）排列成一个线性有序的序列，使得所有弧尾结点都排在弧头结点的前面。
- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做拓扑排序(Topological Sort)。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会有现有向环。

- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。
- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为

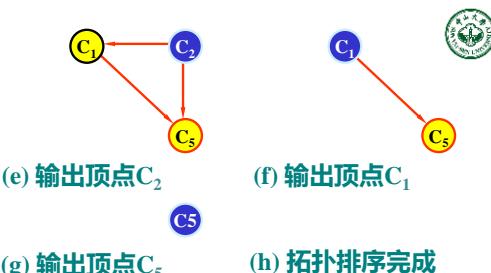
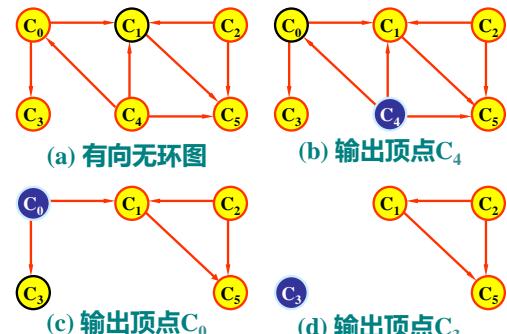
$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$
或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



拓扑排序的方法

- ① 输入AOV网络。令 n 为顶点个数。
- ② 在AOV网络中选一个没有直接前驱的顶点，并输出之；
- ③ 从图中删去该顶点，同时删去所有它发出的有向边；
- ④ 重复以上 ②、③步，直到
 - 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
 - 图中还有未输出的顶点，但已跳出处理循环。说明图中还剩下一些顶点，它们都有直接前驱。这时网络中必存在有向环。

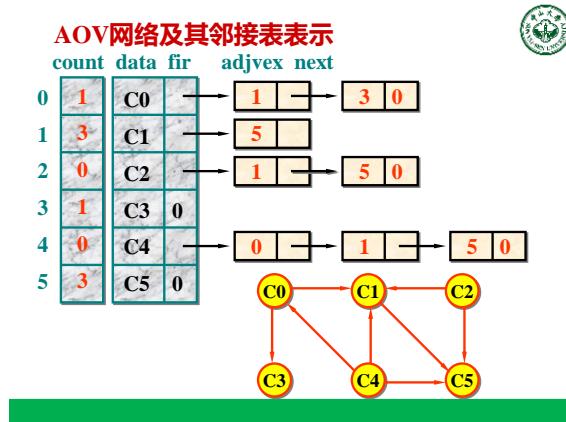
拓扑排序的过程



最后得到的拓扑有序序列为 $C_4, C_0, C_3, C_2, C_1, C_5$ 。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如 C_4 和 C_2 ，也排除了先后次序关系。

- 如果图采用邻接表存储，则在邻接表中增设一个数组 $count[]$ ，记录各顶点入度。入度为零的顶点即无前驱顶点。
- 在输入数据前，顶点表 $data[]$ 和入度数组 $count[]$ 全部初始化。在输入数据时，每输入一条边 $\langle j, k \rangle$ ，就需要建立一个边结点，并将它链入相应边链表中，统计入度信息：

```
EdgeNode * p = new EdgeNode;
p->advex = k; //建立边结点
p->nextarc = G.VexList[j].firstarc;
data[j].firstarc = p;
//链入顶点 j 的边链表的前端
count[k]++;
//顶点 k 入度加一
```

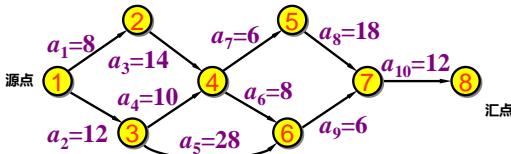


- 在算法中，使用一个存放入度为零的顶点的链式栈，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：
 - 建立入度为零的顶点栈；
 - 当入度为零的顶点栈不空时，重复执行
 - 从顶点栈中退出一个顶点，并输出之；
 - 从AOV网络中删去这个顶点和它发出的边，边的终顶点入度减一；
 - 如果边的终顶点入度减至0，则该顶点进入度为零的顶点栈；
 - 如果输出顶点个数少于AOV网络的顶点个数，则报告网络中存在有向环。

• 分析拓扑排序算法

如果AOV网络有 n 个顶点， e 条边，在拓扑排序的过程中，搜索入度为零的顶点，建立链式栈所需要的时间是 $O(n)$ 。在正常的情况下，有向图有 n 个顶点，每个顶点进一次栈，出一次栈，共输出 n 次。顶点入度减一的运算共执行了 e 次。所以总的时间复杂度为 $O(n+e)$ 。

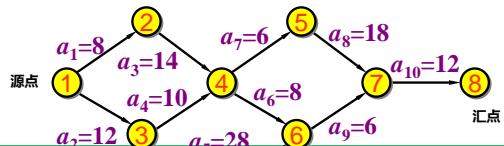
- 在AOE网络中，有些活动顺序进行，有些活动并行进行，入度为零的点叫源点，出度为零的点叫汇点。



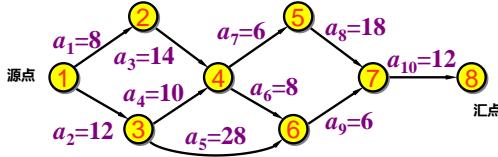
4.6 AOE网络(Activity On Edges)

- 如果在无向环的带权有向图中
 - 用有向边表示一个工程中的各项活动(Activity)
 - 用边上的权值表示活动的持续时间(Duration)
 - 用顶点表示事件(Event)
- 则这样的有向图叫做用边表示活动的网络，简称AOE(Activity On Edges)网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
 - (1)完成整个工程至少需要多少时间(假设网络中没有环)？
 - (2)为缩短完成工程所需的时间，应当加快哪些活动？

- 从源点到各个顶点，以至从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。
- 因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)。



- 要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。
- 关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。



定义几个与计算关键活动有关的量：

- 事件 v_i 的最早可能开始时间 $ve[i]$**
是从源点 v_0 到顶点 v_i 的最长路径长度。
- 事件 v_i 的最迟允许开始时间 $vl[i]$**
是在保证汇点 v_{n-1} 在 $ve[n-1]$ 时刻完成的前提下，事件 v_i 的允许的最迟开始时间。
- 活动 a_k 的最早可能开始时间 $e[k]$**
设活动 a_k 在边 $\langle v_i, v_j \rangle$ 上，则 $e[k]$ 是从源点 v_0 到顶点 v_i 的最长路径长度。因此， $e[k] = ve[i]$ 。



- ④ 活动 a_k 的最迟允许开始时间 $l[k]$**
 $l[k]$ 是在不会引起时间延误的前提下，该活动允许的最迟开始时间。

$$l[k] = vl[j] - dur(\langle i, j \rangle)$$
 其中， $dur(\langle i, j \rangle)$ 是完成 a_k 所需的时间。
- ⑤ 时间余量 $l[k] - e[k]$**
表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。 $l[k] == e[k]$ 表示活动 a_k 是没有时间余量的**关键活动**。
- 为找出关键活动，需要求各个活动的 $e[k]$ 与 $l[k]$ ，以判别是否 $l[k] == e[k]$ 。

- 为求得 $e[k]$ 与 $l[k]$ ，需要先求得从源点 v_0 到各个顶点 v_i 的 $ve[i]$ 和 $vl[i]$ 。

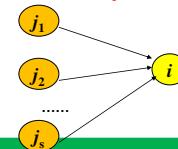
求 $ve[i]$ 的递推公式

- 从 $ve[0] = 0$ 开始，向前递推

$$ve[i] = \max_j \{ ve[j] + dur(\langle v_j, v_i \rangle) \},$$

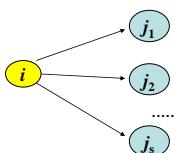
$\langle v_j, v_i \rangle \in S_2, i = 1, 2, \dots, n-1$

S_2 是所有指向 v_i 的有向边 $\langle v_j, v_i \rangle$ 的集合。



- 从 $vl[n-1] = ve[n-1]$ 开始，反向递推

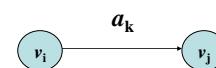
$$vl[i] = \min_j \{ vl[j] - dur(\langle v_i, v_j \rangle) \},$$
 $\langle v_i, v_j \rangle \in S_1, i = n-2, n-3, \dots, 0$
 S_1 是所有源自 v_i 的有向边 $\langle v_i, v_j \rangle$ 的集合。



- 设活动 $a_k (k=1,2,\dots,e)$ 在带权有向边 $\langle v_i, v_j \rangle$ 其持续时间用 $dur(\langle v_i, v_j \rangle)$ 表示，则有

$$e[k] = ve[i];$$

$$l[k] = vl[j] - dur(\langle v_i, v_j \rangle); k = 1, 2, \dots, e.$$
 这样就得到计算关键路径的算法。

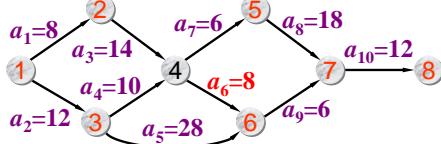




求 $ve[i]$ 的递推公式
从 $ve[0] = 0$ 开始, 向前递推
 $ve[i] = \max_j \{ ve[j] + dur(< v_j, v_i >) \},$
 $< v_j, v_i > \in S_2, i = 1, 2, \dots, n-1$

从 $vl[n-1] = ve[n-1]$ 开始, 反向递推
 $vl[i] = \min_j \{ vl[j] - dur(< v_i, v_j >) \},$
 $< v_i, v_j > \in S_1, i = n-2, n-3, \dots, 0$

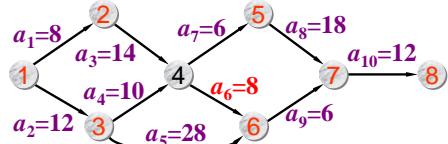
1	2	3	4	5	6	7	8	
ve	0	8	12	22	28	40	46	58
vl	0	8	12	22	28	40	46	58



ve	0	8	12	22	28	40	46	58
vl	0	8	12	22	28	40	46	58

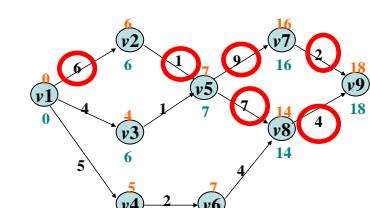
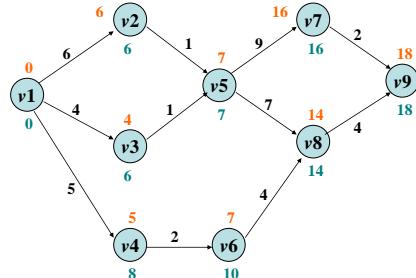
$e[k] = ve[i];$
 $l[k] = vl[j] - dur(< v_i, v_j >); k = 1, 2, \dots, e$

1	2	3	4	5	6	7	8	9	10	
e	0	0	8	12	12	22	22	28	40	46
l	0	0	8	12	12	32	22	28	40	46

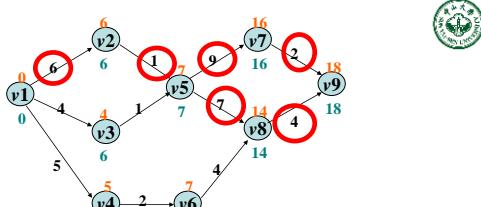


求 $ve[i]$ 的递推公式
从 $ve[0] = 0$ 开始, 向前递推
 $ve[i] = \max_j \{ ve[j] + dur(< v_j, v_i >) \},$
 $< v_j, v_i > \in S_2, i = 1, 2, \dots, n-1$

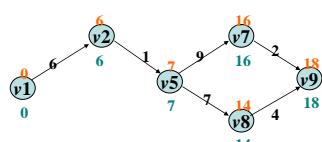
从 $vl[n-1] = ve[n-1]$ 开始, 反向递推
 $vl[i] = \min_j \{ vl[j] - dur(< v_i, v_j >) \},$
 $< v_i, v_j > \in S_1, i = n-2, n-3, \dots, 0$



活动	1-2	1-3	1-4	2-5	3-5	4-6	5-7	5-8	6-8	7-9	8-9
e	0	0	0	6	4	5	7	7	7	16	14
l	0	2	3	6	6	8	7	7	10	16	14
e-l	0	2	3	0	2	3	0	0	3	0	0



adjvex nextarc info



修改后的拓扑排序算法

```
void TopologicalSort (AdjGraph G)
{
    Stack S; StackEmpty(S); int j;
    for (int i = 0; i < n; i++) //入度为零顶点
        if (count[i] == 0) Push(S, i); //进栈
```

```

While ! ( StackEmpty(S) ){
    Pop( S, j );           //退栈
    cout << j << endl; //输出 Push(T,j) //j号顶点入栈
    EdgeNode * p = data[j].firstarc;
    while ( p != NULL ) { //扫描出边表
        int k = p->adjvex; //另一顶点
        if ( --count[k] == 0 ) //顶点入度减一
            Push( S, k );
        //顶点的入度减至零, 进栈
        if ( ve[j]+p->info>ve[k] ) ve[k]=ve[j]+p->info;
        p = p->nextarc;
    }
}
}

```

```

CristicalPath(G)
{
    v[0..vexnum-1]=ve[vexnum-1];
    while !(StackEmpty(T))
    {
        Pop(T,j);      p=G.data[j].firstarc;
        while (p!=NULL)
        {
            k=p->adjvex; dut=p->info;
            if (vl[k]-dut<vl[j]) vl[j]=vl[k]-dut;
        }
    }
    for (j=0;j<vernum;j++)
    for (p=G.data[j].firstarc; p; p=p->next)
    {
        k=p->adjvex; dut=p->info;
        ee=ve[j]; el=vl[k]-dut;
        if (ee==el) printf("%d,%d");
    }
}

```

算法分析

在拓扑排序求 $Ve[i]$ 和逆拓扑有序求 $Vi[i]$ 时, 所需时间为 $O(n+e)$, 求各个活动的 $e[k]$ 和 $l[k]$ 时所需时间为 $O(e)$, 总共花费时间仍然是 $O(n+e)$ 。

说明:

1、并不是加快任何一个关键活动都可以缩短整个工程的工期。只有加快那些包括在所有关键路径上的关键活动才能达到这个目的。

2、关键活动的速度提高是有限度的

4.7 Shortest Path algorithms

最短路径

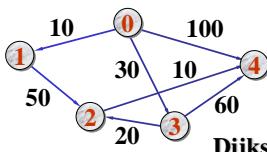
最短路径(Shortest Path)问题

- 如果图中从一个顶点可以到达另一个顶点, 则称这两个顶点间存在一条路径。
- 从一个顶点到另一个顶点间可能存在多条路径, 而每条路径上经过的边数并不一定相同。
- 如果图是一个带权图, 则路径长度为路径上各边的权值的总和, 两个顶点间路径长度最短的那条路径称为两个顶点间的最短路径, 其路径长度称为最短路径长度。
- 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?

- 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
 - 所有顶点之间的最短路径
 - Floyd算法

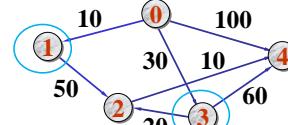
1、边上权值非负情形的单源最短路径问题

- 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。



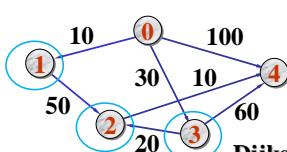
Dijkstra逐步求解的过程

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



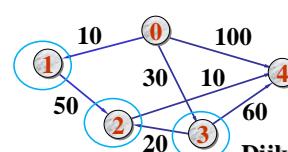
Dijkstra逐步求解的过程

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)	60
v_3		(v_0, v_3)	30
v_4		(v_0, v_4)	100



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)	50
v_3		(v_0, v_3)	30
v_4		(v_0, v_4) (v_0, v_3, v_4)	90



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)	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)	60



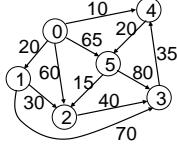
- 引入辅助数组 $dist[i]$ 。它的每一个分量 $dist[i]$ 表示当前找到的从源点 v_0 到终点 v_i 的最短路径的长度。初始状态：
- 若从源点 v_0 到顶点 v_i 有边，则 $dist[i]$ 为该边上的权值；
 - 若从源点 v_0 到顶点 v_i 无边，则 $dist[i]$ 为 ∞ 。

- 假设 S 是已求得的最短路径的终点的集合，则可证明：下一条最短路径必然是从 v_0 出发，中间只经过 S 中的顶点便可到达的那些顶点 v_x ($v_x \in V-S$) 的路径中的一条。
- 每次求得一条最短路径后，其终点 v_k 加入集合 S ，然后对所有的 $v_i \in V-S$ ，修改其 $dist[i]$ 值。





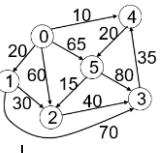
对带权有向图，用Dijkstra算法求从顶点0到其余各顶点的最短路径，数组dist和pre的各分量的变化如表所示。



带权有向图及其邻接矩阵

$$\begin{pmatrix} \infty & 20 & 60 & \infty & 10 & 65 \\ \infty & \infty & 30 & 70 & \infty & \infty \\ \infty & \infty & \infty & 40 & \infty & \infty \\ \infty & \infty & \infty & \infty & 35 & \infty \\ \infty & \infty & \infty & \infty & \infty & 20 \\ \infty & \infty & 15 & 80 & \infty & \infty \end{pmatrix}$$

步骤		顶点	1	2	3	4	5	S
初态	Dist pre	20 0	60 0	∞ 0	10 0	65 0	{0}	
1	Dist pre	20 0	60 0	∞ 0	10 0	30 4	{0, 4}	
2	Dist pre	20 0	50 1	90 1	10 0	30 4	{0, 4, 1}	
3	Dist pre	20 0	45 5	90 1	10 0	30 4	{0, 4, 1, 5}	
4	Dist pre	20 0	45 5	85 2	10 0	30 4	{0, 4, 1, 5, 2}	
5	Dist pre	20 0	45 5	85 2	10 0	30 4	{0, 4, 1, 5, 2, 3}	



Dijkstra算法



- ① 初始化： $S \leftarrow \{v_0\}$;
 $dist[j] \leftarrow Edge[0][j], j = 1, 2, \dots, n-1$;
// n为图中顶点个数
- ② 求出最短路径的长度：
 $dist[k] \leftarrow \min \{ dist[i] \}, i \in V - S$;
 $S \leftarrow S \cup \{k\}$;
- ③ 修改：
 $dist[i] \leftarrow \min \{ dist[i], dist[k] + Edge[k][i] \}$,
对于每一个 $i \in V - S$;
- ④ 判断：若 $S = V$ ，则算法结束，否则转 ②。

计算从单个顶点到其它各顶点最短路径

```
void ShortestPath (MTGraph G, int v) {
//MTGraph是一个有 n 个顶点的带权有向图,各边上的权值由
Edge[i][j]给出。
//dist[j], 0≤j<n, 是当前求到的从顶点v 到顶点j 的最短路径长
度, 用数组path[j], 0≤j<n, 存放求到的最短路径。
EdgeData dist[G.n]; //最短路径长度数组
int path[G.n]; //最短路径数组
int S[G.n]; //最短路径顶点集
```



```
for ( int i = 0; i < n; i++ ) {
    dist[i] = G.Edge[v][i]; //dist数组初始化
    S[i] = 0; //集合S初始化
    if ( i != v && dist[i] <.MaxValue )
        path[i] = v;
    else path[i] = -1; //path数组初始化
}
S[v] = 1; dist[v] = 0; //顶点v加入顶点集合
//选当前不在集合S中具有最短路径的顶点u
for ( i = 0; i < n-1; i++ ) {
    float min = ..MaxValue; int u = v;
```

```
for ( int j = 0; j < n; j++ )
    if ( !S[j] && dist[j] < min )
        { u = j; min = dist[j]; }
S[u] = 1; //将顶点u加入集合S
for ( int w = 0; w < n; w++ ) //修改
    if ( !S[w] && G.Edge[u][w] < ..MaxValue
        && dist[u] + G.Edge[u][w] < dist[w] ) {
        //顶点w未加入S,且绕过u可以缩短路径
        dist[w] = dist[u] + G.Edge[u][w];
        path[w] = u; //修改到w的最短路径
    }
} //选定各顶点到顶点 v 的最短路径
```



```
//打印各顶点的最短路径: 路径是逆向输出的
for ( i = 0; i < G.n; i++ ) {
    cout << endl;
    cout << "Distance: " << dist[i]
        << " Path: " << i;
//输出终点的最短路径长度和终点
int pre = path[i]; //取终点的直接前驱
while ( pre != v ) { //沿路径上溯输出
    cout << ", " << pre;
    pre = path[pre];
}
}
```

算法分析

Dijkstra算法的主要执行是：

- ◆ 数组变量的初始化：时间复杂度是 $O(n)$ ；
- ◆ 求最短路径的二重循环：时间复杂度是 $O(n^2)$ ；因此，整个算法的时间复杂度是 $O(n^2)$ 。

2、所有顶点之间的最短路径



- 如果希望求得图中任意两个顶点之间的最短路径，显然只要依次将每个顶点设为源点，调用迪杰斯特拉算法 n 次便可求出，其时间复杂度为 $O(n^3)$ 。弗洛伊德提出了另外一个算法，虽然其时间复杂度也是 $O(n^3)$ ，但算法形式更简单。

算法的基本思想就是：

从初始的邻接矩阵 A_0 开始，递推地生成矩阵序列 A_1, A_2, \dots, A_n

$$\begin{aligned} A_0[i][j] &= C[i][j] \\ A_{k+1}[i][j] &= \min\{A_k[i][j], A_k[i][k] + A_k[k][j]\} \end{aligned}$$

显然， **A 中记录了所有顶点对之间的最短路径长度**。
若要求得到最短路径本身，还必须设置一个路径矩阵 $P[n][n]$ ，在第 k 次迭代中求得的 $p_{path[i][j]}$ ，是从 i 到 j 的**中间点序号不大于 k** 的最短路径上顶点 i 的后继顶点。
算法结束时，由 $p_{path[i][j]}$ 的值就可以得到从 i 到 j 的最短路径上的各个顶点。

算法基本思想



对于顶点 i 和 j ：

- 1、首先，考虑从 i 到 j 是否有以顶点1为中间点的路径：即考虑图中是否有边 $\langle i, 1 \rangle$ 和 $\langle 1, j \rangle$ ，若有，则新路径 $i, 1, j$ 的长度是 $A[i][1] + A[1][j]$ ，**比较路径 i, j 和 $i, 1, j$ 的长度**，并以较短者为当前所求得的最短路径。该路径是**中间点序号不大于1的最短路径**。

- 2、其次，考虑从 i 到 j 是否包含顶点2为中间点的路径： $i, \dots, 2, \dots, j$ ，若没有，则从 i 到 j 的最短路径仍然是第一步中求出的，即从 i 到 j 的中间点序号不大于1的最短路径；若有，则 $i, \dots, 2, \dots, j$ 可分解成两条路径 $i, \dots, 2$ 和 $2, \dots, j$ ，而这两条路径是前一次找到的中间点序号不大于1的最短路径，将这两条路径相加就得到路径 $i, \dots, 2, \dots, j$ 的长度，将该长度与前一次求出的从 i 到 j 的中间点序号不大于1的最短路径长度比较，取其较短者作为当前求得的从 i 到 j 的中间点序号不大于2的最短路径。





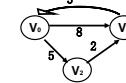
3、然后，再选择顶点3加入当前求得的从*i*到*j*中间点序号不大于2的最短路径中，按上述步骤进行比较，从未加入顶点3作中间点的最短路径和加入顶点3作中间点的新路径中选取较小者，作为当前求得的从*i*到*j*的中间点序号不大于3的最短路径。

依次类推，直到考虑了顶点*n*加入当前从*i*到*j*的最短路径后，选出从*i*到*j*的中间点序号不大于*n*的最短路径为止。由于图中顶点序号不大于*n*，所以从*i*到*j*的中间点序号不大于*n*的最短路径，已考虑了所有顶点作为中间点的可能性。因而它必是从*i*到*j*的最短路径。

示例

$$A_0[i][j] = C[i][j]$$

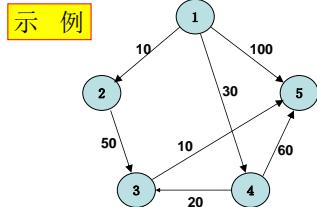
$$A_{k+1}[i][j] = \min\{ A_k[i][j], A_k[i][k] + A_k[k][j] \}$$



A_0	$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$	通过顶点0, $k=0$
A_1	$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$	通过顶点1, $k=1$
A_2	$\begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$	通过顶点2, $k=2$



$$A_0 = \begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad A_3 = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$



$$A_4 = \begin{bmatrix} 0 & 10 & 50 & 30 & 100 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

示例

A_0	$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$	$\text{path} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 0 & 3 \end{bmatrix}$
A_1	$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$	$\text{path} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$
A_2	$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$	$\text{path} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$
A_3	$\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$	$\text{path} = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$

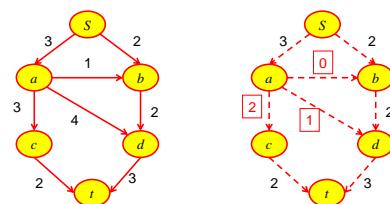


4.8 网络流问题



- 网络或容量网络:指的是一个连通的赋权有向图 $D = (V, E)$ ，其中 V 是该图的顶点集， E 是有向边(即弧)集。
- 网络上的流:是指定义在弧集合 E 上一个函数 $f = \{f(u,v)\}$ ，并称 $f(u,v)$ 为弧 (u,v) 上的流量。
- V 中有一个源点 s ，一个汇点 t ，网络上的流都是由源点流出最终流入汇点。
- E 中的每一条有向边 (u,v) 都有一个对应的容量上限 $c(u,v)$ ，通过这条边的流不能超过容量上限。

- 在既不是源点又不是汇点的任一顶点 v ，总的进入流必须等于总的发出流。最大流问题就是确定从 s 到 t 可以通过的最大流量。



一个简单的最大流算法:

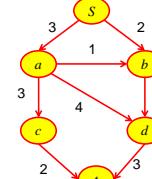


- G : 原图

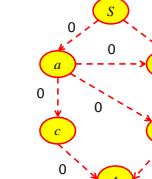
- G_f : 流图, 表示在算法的任意阶段已经达到的流, 初始时所有边没有流。

- G_r : 残余图, $G - G_f$ 。

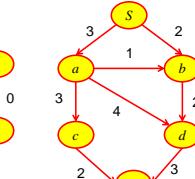
在每个阶段, 寻找图 G_r 中从 s 到 t 的一条路径, 称为增长路径, 这条路径上的最小边值就是可以添加到路径上边的流的量。通过调整 G_f 和重新计算 G_r 求解, 当 G_r 中没有从 s 到 t 的一条路径为止。



图

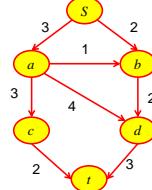


流图

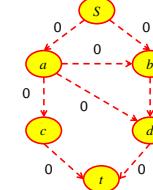


残余图

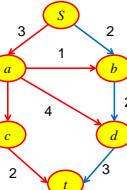
路径: $s \rightarrow b \rightarrow d \rightarrow t$



图

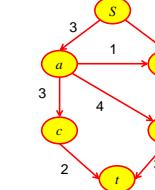


流图

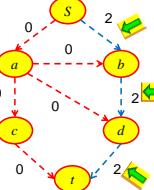


残余图

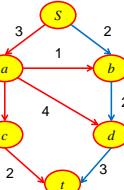
添加路径到流图: $s \rightarrow b \rightarrow d \rightarrow t$



图

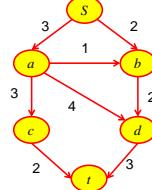


流图

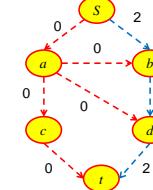


残余图

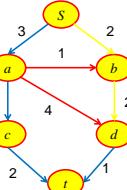
修改残余图



图

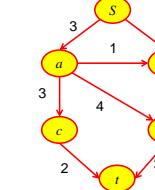


流图

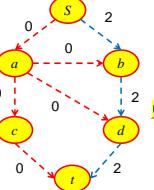


残余图

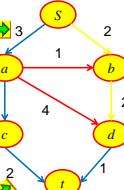
路径: $s \rightarrow a \rightarrow c \rightarrow t$



图



流图

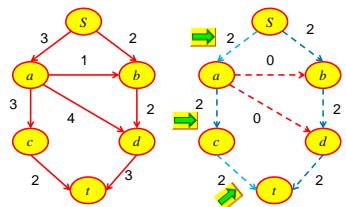


残余图



添加路径到流图: $s \rightarrow a \rightarrow c \rightarrow t$

修改残余图

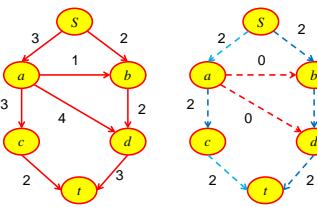


图

流图

残余图

路径: $s \rightarrow a \rightarrow d \rightarrow t$



图

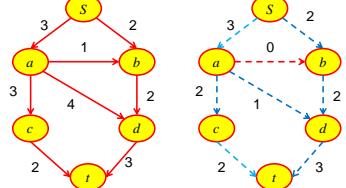
流图

残余图

添加路径到流图: $s \rightarrow a \rightarrow d \rightarrow t$



修改残余图



图

流图

残余图

添加路径到流图: $s \rightarrow a \rightarrow d \rightarrow t$

修改残余图



图

流图

残余图