



中国科学技术大学
University of Science and Technology of China

数据结构

图 (12 学时)

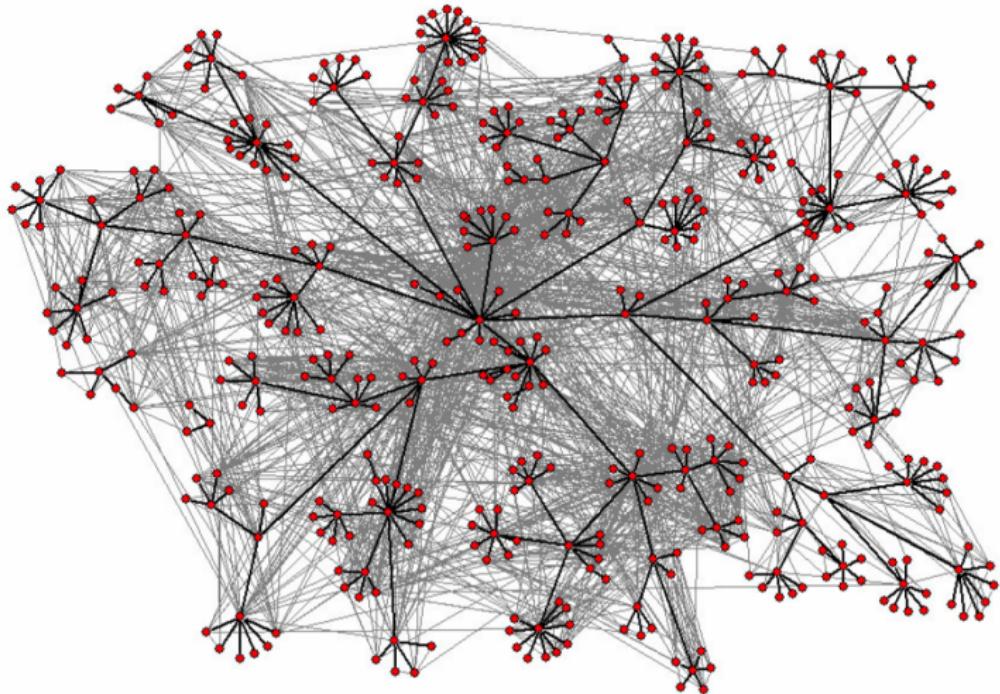
November 21, 2022

目录

- ① 认识图
- ② 图的实现
- ③ 图的搜索与遍历
- ④ 最小生成树
- ⑤ 有向无环图
- ⑥ 最短路径

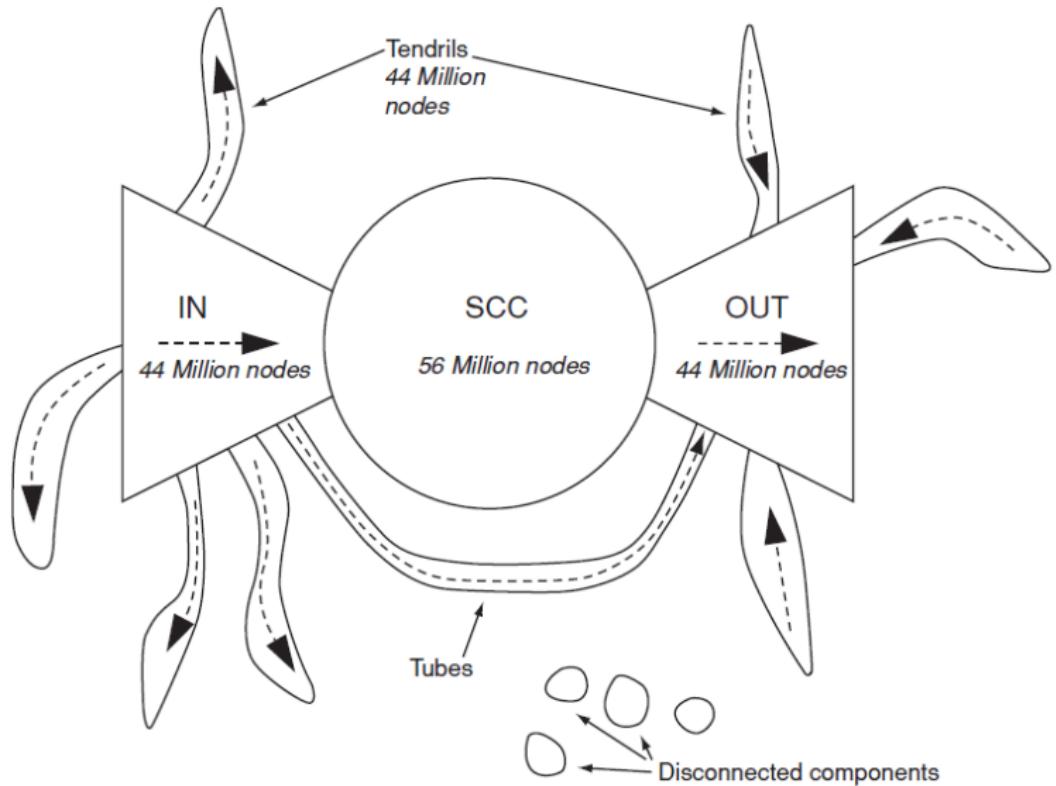
认识图

一个组织内部，不同部门员工之间的电子邮件网络



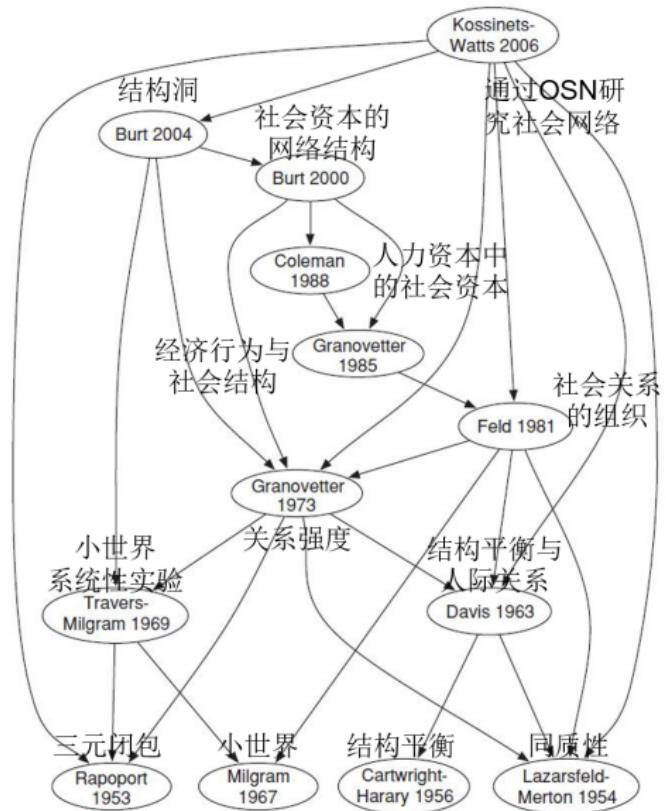
认识图

互联网长什么样子？全球 WWW 网页构成的图



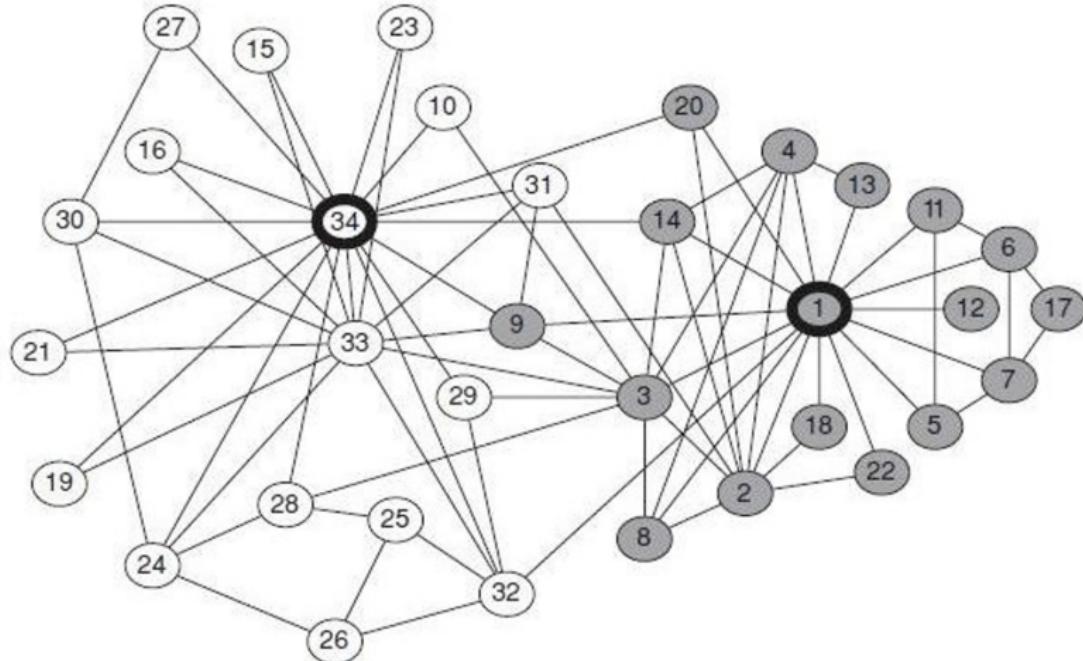
认识图

社会网络研究进展图



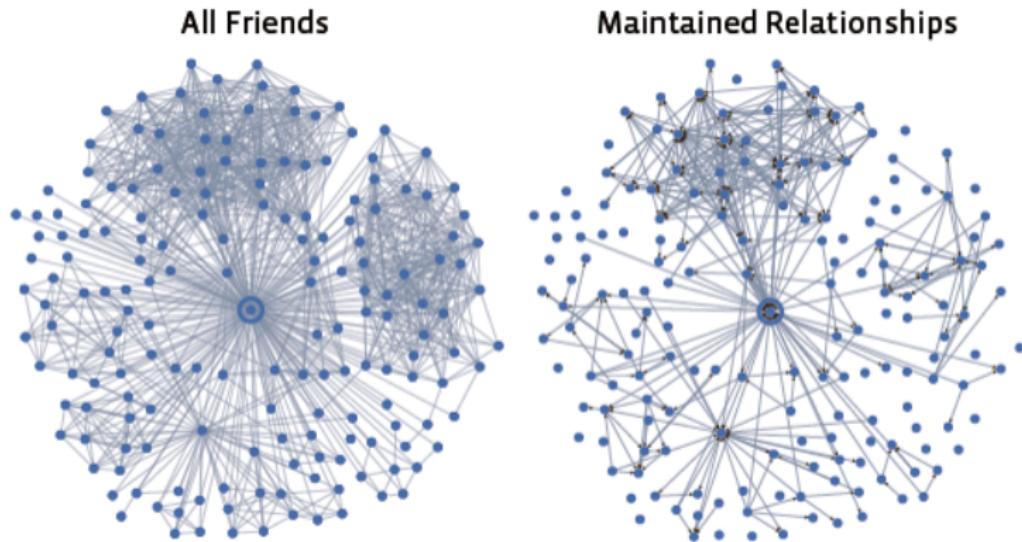
认识图

一个跆拳道俱乐部内部成员的社交关系



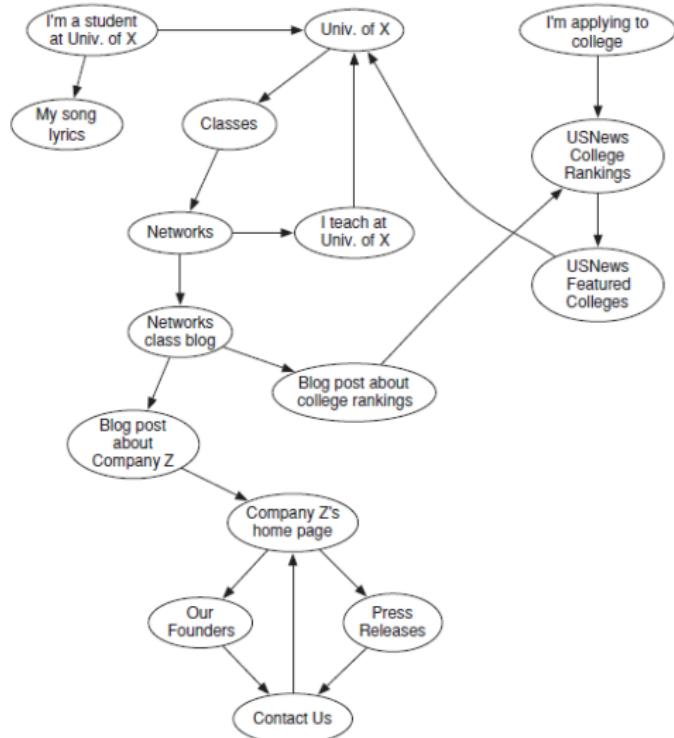
认识图

facebook 上宣称的好友关系和实质上常维护着的关系（片段）



认识图

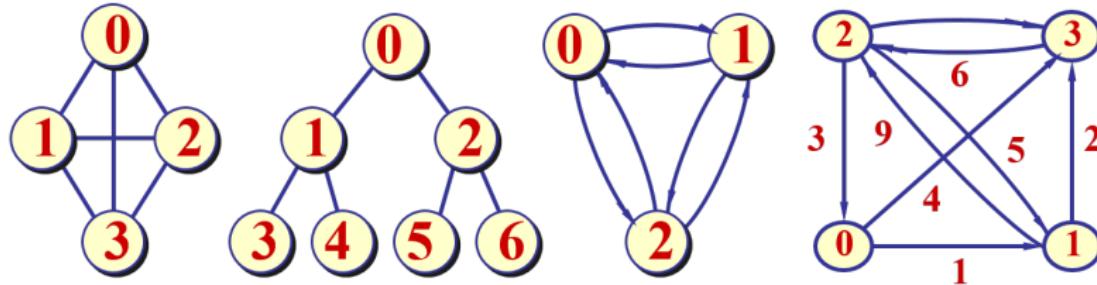
网页间相互链接的关系图（片段）



认识图

图是什么

- 我们用计算机来解决现实问题，需要将各种数据对象抽象成“节点”，然后讨论节点间的关系；
- 如人物关系、社交关系、路网、电网、分子结构、网页关系、知识概念关系等等
- 集合、线性表、树等，都是图的特例
- 由节点和节点间的连边构成的对象称为“图”，如下所示



认识图

图的各种基本术语

- 图 (Graph), 又称为网络 (Network), 由一个节点集合和边集合构成, 记作 $G = (V, E)$, 其中 V 是顶点集, E 是边集
- 节点/结点 (node), 又称顶点 (vertex)
- 边 (edge), 又称链接 (link), 联系 (tie), 弧 (arc)
- 在不同学科中, 对图的术语的使用不完全统一, 因此, 本课程中所有的这些用语都可能会被等同使用 (**两个例外: 网络指边带权的图, 弧指有向边**)

认识图

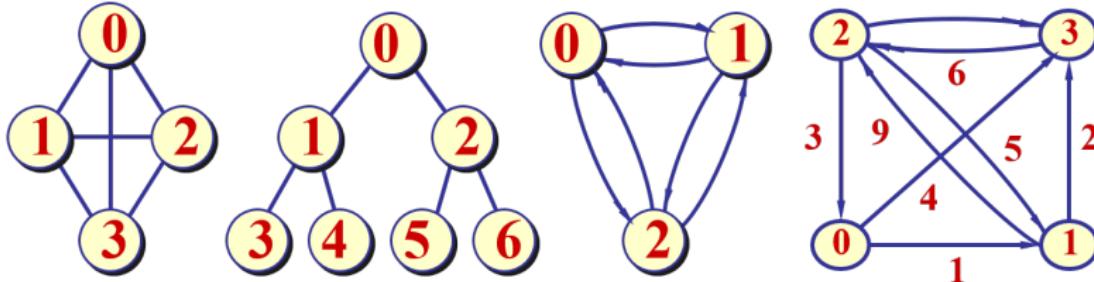
图的 ADT

```
1  ADT Graph{  
2      数据对象：具有相同特性的数据元素的集合，称为顶点集  $V$ 。  
3      数据关系：  $R = \{< v, w > | v, w \in V \& P(v, w)\}$ ,  $< v, w >$  表示从  $v$  到  $w$   
4          的一条弧， $v$  为弧尾， $w$  为弧头；  
5           $P(v, w)$  定义了弧  $< v, w >$  的意义或信息 }  
6  基本操作：  
7      Create_Graph() : 图的创建操作，生成一个没有顶点的空图  $G$   
8      GetVex( $G$ ,  $v$ ) : 求图中的顶点  $v$  的值  
9      ...  
10     DFStraver( $G$ ,  $v$ ) : 从  $v$  出发对图  $G$  深度优先遍历，  
11         每个顶点访问且只访问一次  
12     BFStraver( $G$ ,  $v$ ) : 从  $v$  出发对图  $G$  广度优先遍历，  
13         每个顶点访问且只访问一次  
14 }ADT Graph
```

图的基本类型和相关概念

图的类型

- 无向图：若弧 $\langle v, w \rangle \in R$ 则必有 $\langle w, v \rangle \in R$, 则用边 (v, w) 记之，此图称为无向图，顶点对 (v, w) 是无序的
- 有向图：图中连接顶点对 $\langle v, w \rangle$ 的弧是有序的，表示时带“指向箭头”。注意表示用尖括号 $\langle \rangle$ 和圆括号 $()$ 以示区别
- 完全图：假设不允许重边出现，那么有 n 个顶点的无向图若有 $n(n - 1)/2$ 条边，则此图为无向完全图。有 n 个顶点的有向图若有 $n(n - 1)$ 条边，则此图为有向完全图。
- 如下图所示，四个子图分别为：无向完全图，无向图，有向图，边带权图/网络



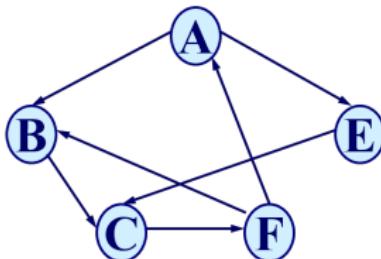
图的重要概念

边相关的概念

- 邻接点：用边连接的两个顶点互为邻接点，通常指无向图
- 边权值：描述边的属性或特点；若边上无权，可假设所有边上的权值为某个固定的常数

顶点的度

- 度：顶点连接的边的条数；
- 有向图的度为“出度”和“入度”二者之和，出度是顶点发出“箭头”的数目；入度是顶点被箭头指向的数目



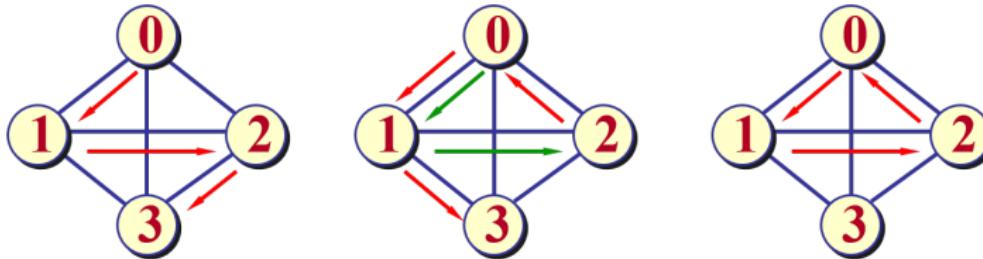
有向图

$$\begin{aligned}\text{顶点的度(TD)} &= \text{出度(OD)} + \text{入度(ID)} \\ \text{TD(B)} &= \text{OD(B)} + \text{ID(B)} = 3\end{aligned}$$

图的重要概念

路径相关

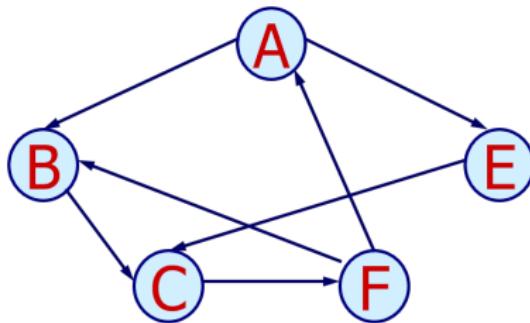
- 在图 $G(V, E)$ 中, 若从顶点 v_i 出发, 沿 E 中的一些边经过一些 V 中的顶点 v_1, v_2, \dots, v_m , 到达顶点 v_j , 则称顶点序列 $(v_i, v_1, v_2, \dots, v_m, v_j)$ 为从顶点 v_i 到顶点 v_j 的路径
- 路径长度: 非带权图的路径长度是指此路径上边的条数。带权图的路径长度是指路径上各边的权之和
- 简单路径: 若路径上各顶点 v_1, v_2, \dots, v_m 均不互相重复, 则称这样的路径为简单路径, 如左子图; 中子图先沿绿色色再沿红色的路径不是简单路径
- 简单回路: 若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合, 则称这样的路径为回路或环 (Cycle); 如右子图所示



路径长度

例子如图所示

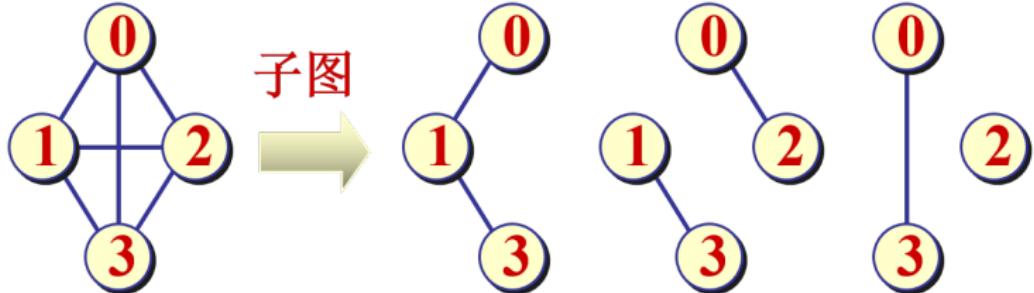
- 从 A 到 F, 路径长度为 3 的路径为 $(A, B, C, F), (A, E, C, F)$
- 从 A 到 B 的路径长度是多少 ?



图的重要概念

子图相关

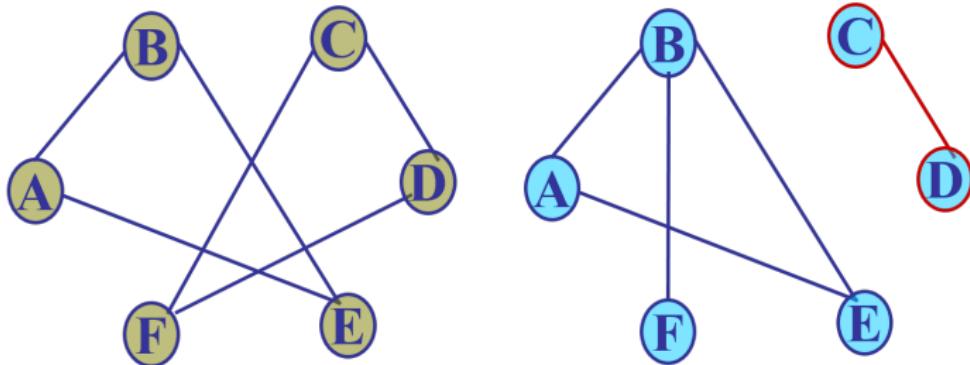
- 设有两个图 $G = (V, E)$ 和 $G' = (V', E')$ 。若 $V' \subseteq V$ 且 $E' \subseteq E$, 则称图 G' 是图 G 的子图。如图所示
- 无向连通图与连通分量：在无向图中，若从顶点 v_1 到顶点 v_2 有路径，则称顶点 v_1 与 v_2 是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图。非连通图的极大连通子图叫做连通分量
- 强连通图与强连通分量在有向图中，若对于每一对顶点 v_i 和 v_j ，都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径，则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量



连通图

结合例子理解连通图

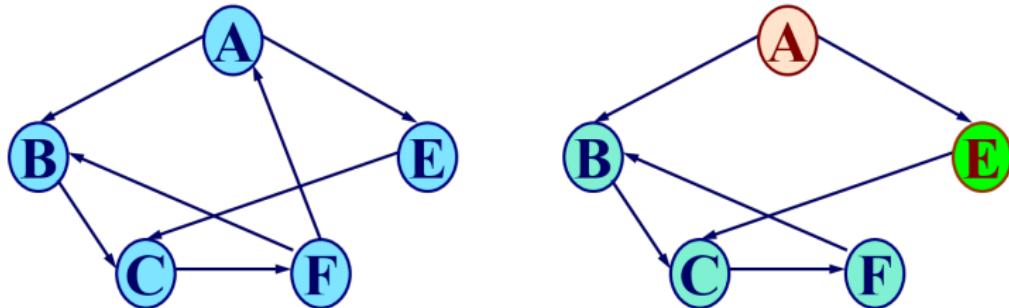
- 无向图中任意两个顶点之间都有路径相通，则称此图为连通图
- 若无向图为非连通图，则图中各个极大连通子图称作此图的连通分量



强连通图

结合例子理解强连通图

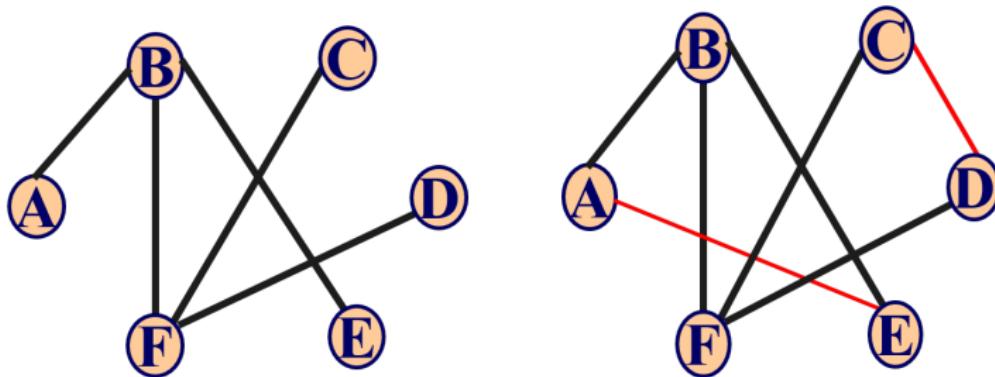
- 有向图中任意两个顶点之间都存在一条有向路径，则称此有向图为强连通图
- 否则，其各个极大强连通子图称作它的强连通分量



图的重要概念

树相关的概念（无向图）

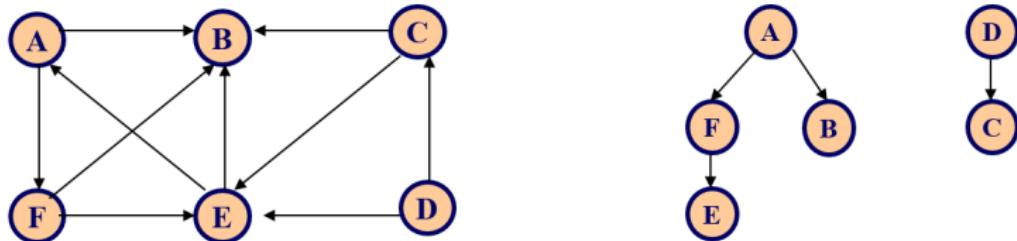
- 生成树：有 n 个顶点和 e 条边的连通图，其 $n - 1$ 条边和 n 个顶点构成一个极小连通子图为此连通图的生成树
- 如左下图为一棵生成树，任意添加一条边，都会形成环（如右子图）
- 删除生成树的任何一条边，则变成非连通图



图的重要概念

树相关的概念（有向图）

- 有向树：恰有一个顶点入度为 0，其余顶点入度均为 1 的有向图
- 生成森林：一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，且仅包含足以构成若干棵不相交的有向树的弧



图的存储结构

图存储时要考虑的问题

- 任意顶点之间可能存在联系，无法以数据元素在存储区中的物理位置来表示元素之间的关系
- 图中顶点的度不一样，有的可能相差很大，若按度数最大的顶点设计结构，则会浪费很多存储单元，反之按每个顶点自己的度设计不同的结构，又会影响操作

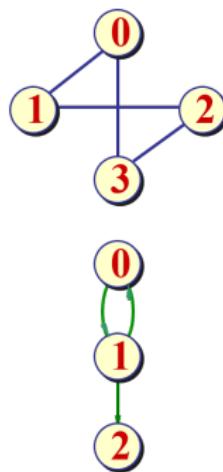
常见的图存储结构

- 邻接矩阵/数组表示法
- 邻接链表
- 十字链表
- 邻接多重表
- 边表

邻接矩阵/数组表示法

顶点表 + 邻接矩阵

- 包括：一个记录各个顶点信息的顶点表 + 一个表示各个顶点之间关系的邻接矩阵。或者更准确地说是“数组表示法”，一个一维顶点数组 + 一个二维边数组
- 设图 $A = (V, E)$ 是一个有 n 个顶点的图，图的邻接矩阵是一个二维数组 $A.adj[n][n]$, $A.adj_{ij} = 1[< i, j > \in E]$



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

邻接矩阵

有向图和无向图的不同 1: 对称性

- 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。

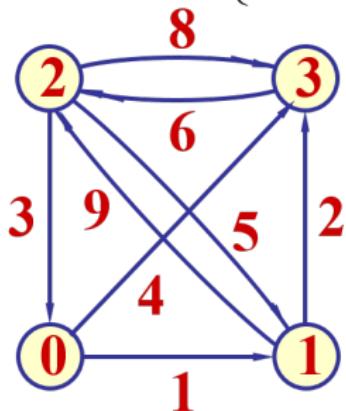
有向图和无向图的不同 2: 求度的差异

- 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 i 列 1 的个数可得顶点 i 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。

网的邻接矩阵

网即边带权的图

$$A.\text{arcs}[i][j] = \begin{cases} W(i, j), & \text{若 } i \neq j \text{ 且 } \langle i, j \rangle \in E \text{ 或 } (i, j) \in E \\ \infty, & \text{若 } i \neq j \text{ 且 } \langle i, j \rangle \notin E \text{ 或 } (i, j) \notin E \\ 0, & \text{若 } i == j \end{cases}$$



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

邻接矩阵的实现

静态结构部分

```
1 #define INFINITY INT_MAX //最大值
2 #define MAX_VERTEX_NUM 20 //最大顶点个数
3 typedef enum{DG,DN,UDG,UDN} GraphKind; //图的类型
4
5 typedef struct ArcCell{
6     WType w; // WType为边权值类型
7     InfoType *info; //该弧相关信息指针
8 }ArcCell,AdjMatrix[MAX_VERTEX_NUM] [MAX_VERTEX_NUM];
9
10 typedef struct {
11     VexType vexts[MAX_VERTEX_NUM]; //顶点向量
12     AdjMatrix adj; //邻接矩阵
13     int vexnum,arcnum; //顶点数和弧数
14     GraphKind kind; //图的种类标志
15 } MGraph;
```

基于邻接矩阵的基本操作实现

图的构造

```
1 Status CreateUDN(Mgraph &G){  
2     //无向网的构造  
3     scanf(&G.vexnum,&G.arcnum,&IncInfo);  
4     for(i=0;i<G.vexnum;++i) scanf(&G.vexs[i]);  
5     for (i=0 ;i<G.vexnum;++i) //初始化邻接矩阵  
6         for (j=0;j<G.vexnum;++j)  
7             G.adj[i][j]={INFINITY,NULL};  
8         for(k=0 ;k<G.arcnum;++k){  
9             scanf(&v1,&v2,&w); //读入边依附的顶点和权值  
10            i=LocateVex(G,v1);  
11            j=LocateVex(G,v2); //顶点定位  
12            G.adj[i][j].w=w; //弧<v1,v2>的权值  
13            if (IncInfo)  
14                Input(*G.adj[i][j].info); //输入弧的相关信息  
15                G.adj[j][i]=G.adj[i][j]; //置对称弧<v2,v1>  
16        }  
17        return OK;  
18 }
```

图的构造

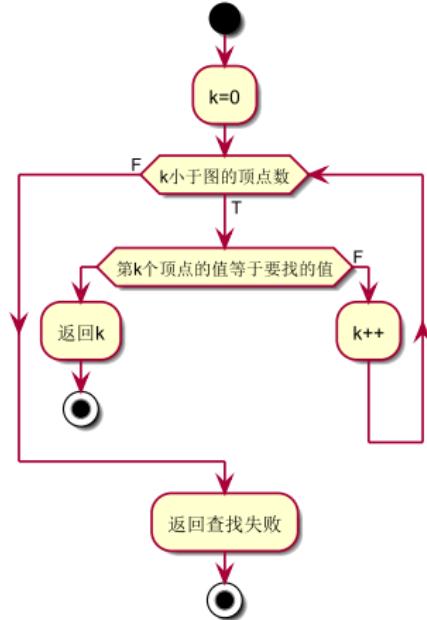


基于邻接矩阵的基本操作实现

图的顶点定位

- 确定一个顶点在 vexs 数组中的位置 (下标)
其过程完全等同于在顺序存储的线性表中查找一个数据元素

由值找顶点下标



```
1 int LocateVex(MGraph *G , VexType *vp){  
2     int k;  
3     for (k=0;k< G->vexnum; k++)  
4         if (G->vexs [k]==*vp)  
5             return(k);  
6     return(-1); //图中无此顶点  
7 }
```

基于邻接矩阵的基本操作实现

向图中增加顶点：不增加边

- 向图中增加一个顶点的操作，在顶点数组 vexs 的末尾增加一个数据元素。

```
1 int AddVertex(MGraph *G , VexType *vp){  
2     int k, j;  
3     if (G->vexnum>=MAX_VEX){  
4         printf( "Vertex Overflow !\n" );  
5         return(-1);}  
6     if (LocateVex(G, vp)!=-1){  
7         printf( "Vertex has existed !\n" );  
8         return(-1);}  
9     k=G->vexnum; G->vexs[G->vexnum++]=*vp;  
10    if (G->kind==DG || G->kind==UDG)  
11        for (j=0; j<G->vexnum; j++)  
12            G->adj[j][k].w=G->adj[k][j].w=0;  
13        /*是不带权的有向图或无向图*/  
14    else  
15        for (j=0; j<G->vexnum; j++){  
16            G->adj[j][k].w=INFINITY;  
17            G->adj[k][j].w=INFINITY;  
18            /*是带权的有向图或无向图*/  
19        }  
20    return(k);  
21 }
```

增加图的顶点



基于邻接矩阵的基本操作实现

向图中增加一条弧

- 根据给定的弧或边所依附的顶点，修改邻接矩阵中所对应的数组元素

```
1 int AddArc(MGraph *G, ArcType *arc){//尝试给出ArcType的定义
2     int k , j;
3     k=LocateVex(G,&arc->vex1);
4     j=LocateVex(G,&arc->vex2);
5     if (k== -1 || j== -1){
6         printf( "Vertex of arc do not existed!\n" );
7         return(-1);
8     if (G->kind==DG||G->kind==DN){
9         //是有向图或带权的有向图
10        G->adj [k] [j] .w=arc->weight;
11        G->adj [k] [j] .info=arc->info;
12    } else{
13        G->adj [k] [j] .w=arc->weight;
14        G->adj [j] [k] .w=arc->weight;
15        G->adj [k] [j] .info=arc->info;
16        G->adj [j] [k] .info=arc->info; //是无向图或带权的无向图，需对称赋值
17    }
18    return(1);
19 }
```

增加图的一条弧

查找弧两端点的下标k和j L3-L4

检查k和j是否合法 L5-L7

依据图的类型
修改邻接矩阵
L8-L16

邻接表

一种图的链式存储方式：广义表的链式存储结构

- 顶点结构：结构体数组，结构体包括数据域 data 和指向第一个依附该顶点的弧节点的指针 firstarc
- 弧节点结构：每个顶点依附的弧构成一个单链表；节点结构包括：弧的另一个顶点 adjvex, 指向下一个弧节点的指针 nextarc 和弧信息 info
- 如下图所示两类节点的结构，顶点结构体称为弧节点构成链表的头节点。（即树的孩子链表）

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

顶点/头结点

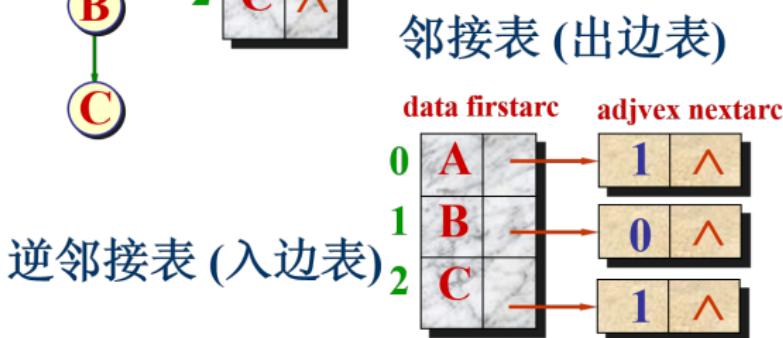
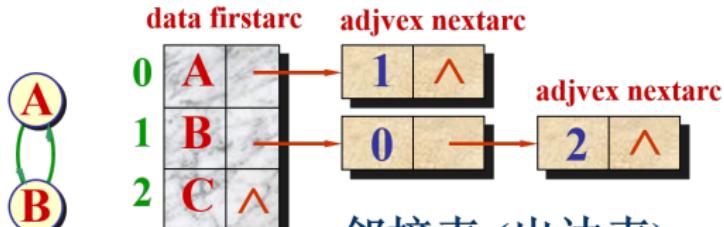
adjvex	nextarc	info
--------	---------	------

弧节点/边结点

有向图的邻接表和逆邻接表

特点

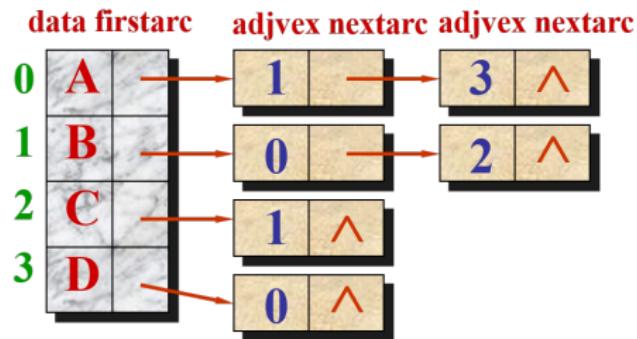
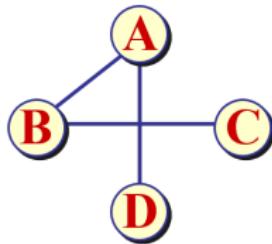
- 头节点 = 顶点节点的数目，弧数目 = 弧节点数目
- 邻接表：头节点代表的顶点是箭头的起点，弧节点中的顶点是箭头的指向；逆邻接表反之；
- 邻接表求顶点的出度，就是求顶点后接的链表的长度；
- 逆邻接表求顶点的入度，就是求顶点后接的链表的长度；



无向图的邻接表

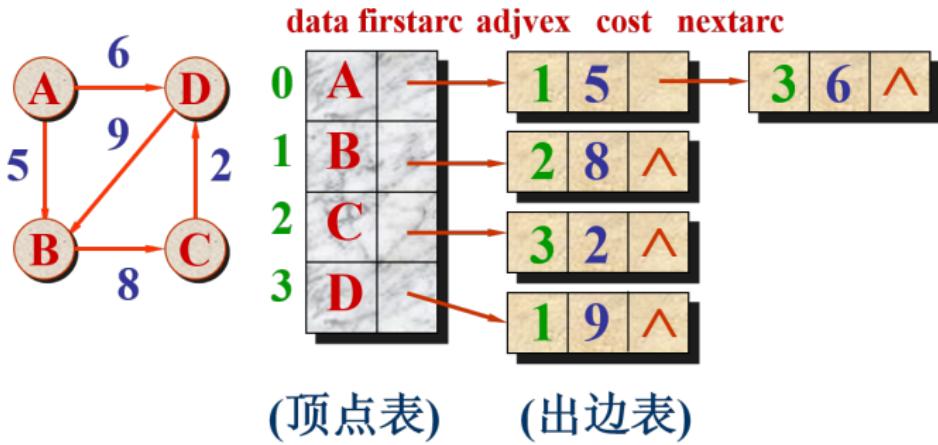
特点

- 头节点 = 顶点节点的数目，弧节点数目 = 2 倍边数目
- 同一个顶点发出的边链接在同一个边链表中
- 每条边被存在其两个顶点为头节点的边链表中
- 邻接表求顶点度，就是求顶点后接的链表的长度；



网（边带权图）的邻接表

例子如图所示



图的邻接表特点总结

邻接表：链表构成的数组

- 在边或弧稀疏的条件下，用邻接表表示比用邻接矩阵表示节省存储空间
- 在无向图，顶点 v_i 的度是第 i 个链表的结点数
- 有向图可以建立邻接表或逆邻接表。邻接表是以顶点 v_i 为出度（即为弧的起点）而建立的邻接表；逆邻接表是以顶点 v_i 为入度（即为弧的终点）而建立的邻接表
- 在有向图中，第 i 个链表中的结点数是顶点 v_i 的出（或入）度；求入（或出）度，须遍历整个邻接表
- 在邻接表上易找出任一顶点的第一个邻接点和下一个邻接点

邻接表的实现

邻接表在内存中的表示

```
1 #define MAX_VERTEX_NUM 20
2 typedef struct ArcNode {
3     int adjvex; // 该弧所指向的顶点的位置
4     struct ArcNode *nextarc; // 指向下一条弧指针
5     InfoType *info; // 该弧相关信息的指针
6 } ArcNode;
7
8 typedef struct VNode {
9     VexType data; // 顶点信息
10    ArcNode *firstarc; // 指向第一条依附该顶点的弧
11 } VNode, AdjList[MAX_VERTEX_NUM];
12
13 typedef struct {
14     AdjList vertices;
15     int vexnum,arcnum;//图的当前顶点数和弧数
16     int kind;//图的种类标志
17 }ALGraph;
```

基于邻接表的基本操作的实现

图的构建

```
1 ALGraph *Create_Graph(ALGraph * G){  
2     printf(“请输入图的种类标志：”);  
3     scanf(“%d”, &G->kind);  
4     G->vexnum=0;      //初始化顶点个数  
5     return(G);  
6 } //生成一个空图
```

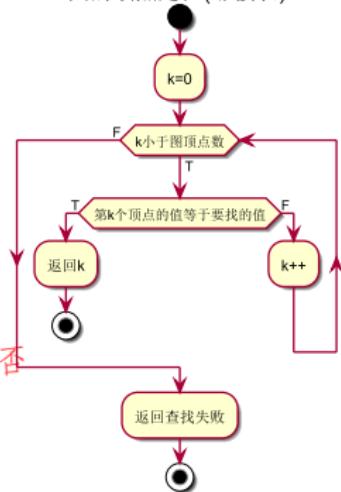
图的顶点定位：顶点在头指针数组中的位置

```
1 int LocateVex(ALGraph *G , VexType *vp){  
2     int k;  
3     for (k=0; k<G->vexnum; k++)  
4         if (G->vertices[k].data==*vp) return(k);  
5     return(-1); //图中无此顶点  
6 } //给一个顶点数据域指针，判断其在顶点结构体数组中否
```

图的构建(邻接表)



图的顶点定位(邻接表)



基于邻接表的基本操作的实现

向图中增加一个顶点

```
1 int AddVertex(ALGraph *G ,VexType *vp){  
2     int k, j;  
3     if (G->vexnum>=MAX_VEX){  
4         printf( "Vertex Overflow !\n" );  
5         return(-1);  
6     }  
7     if (LocateVex(G , vp)!=-1){  
8         printf( "Vertex has existed !\n" );  
9         return(-1);  
10    }  
11    G->vertices[G->vexnum].data=*vp;  
12    G->vertices[G->vexnum].degree=0;  
13    G->vertices[G->vexnum].firstarc=NULL;  
14    k=++G->vexnum;  
15    return(k);  
16 } //在顶点数组vertices末尾增加一个数据元素
```

图中增加顶点(邻接表)



基于邻接表的基本操作的实现

向图中增加一条弧

```
1 int AddArc(ALGraph *G , ArcType *arc){  
2     int k , j ; ArcNode *p ,*q ;  
3     k=LocateVex(G , &arc->vex1) ;  
4     j=LocateVex(G , &arc->vex2) ;  
5     if (k== -1||j== -1) {  
6         printf(``弧依附的顶点不存在\n''); return(-1) ;}  
7     p=(ArcNode *)malloc(sizeof(ArcNode)) ;  
8     p->adjvex=arc->vex1 ; p->info=arc->info ;  
9     p->nextarc=NULL ; //边的起始表结点赋值  
10    q=(ArcNode *)malloc(sizeof(ArcNode)) ;  
11    q->adjvex=arc->vex2 ; q->info=arc->info ;  
12    q->nextarc=NULL ; //边的末尾表结点赋值  
13    if (G->kind==UDG||G->kind==UDN) {  
14        q->nextarc=G->vertices[k].firstarc ;  
15        G->vertices[k].firstarc=q ;  
16        p->nextarc=G->vertices[j].firstarc ;  
17        G->vertices[j].firstarc=p ;  
18    } //以上, 是无向图, 用头插入法插入到两个单链表  
19    else{ //以下, 建立有向图的邻接链表, 用头插入法  
20        q->nextarc=G->vertices[k].firstarc ;  
21        G->vertices[k].firstarc=q ; //建立正邻接链表用  
22        //q->nextarc=G->vertices[j].firstarc ;  
23        //G->vertices[j].firstarc=q ;} //建立逆邻接链表用  
24    return(1) ;  
25 } //根据给定的弧或边所依附的顶点, 修改链表。
```

图中增加一条弧(邻接表)



有向图的十字链表

有向图的链式存储结构：融合（拼接）邻接表和逆邻接表

- 节点结构如下图所示，结构体成员的含义如下：

- data 域：存储和顶点相关的信息；
- 指针域 firstin：指向以该顶点为弧头的第一条弧所对应的弧结点；
- 指针域 firstout：指向以该顶点为弧尾的第一条弧所对应的弧结点；
- 尾域 tailvex：指示弧尾顶点在图中的位置；
- 头域 headvex：指示弧头顶点在图中的位置；
- 指针域 hlink：指向弧头相同的下一条弧；
- 指针域 tlink：指向弧尾相同的下一条弧；
- Info 域：指向该弧的相关信息；



弧结点

顶点结点

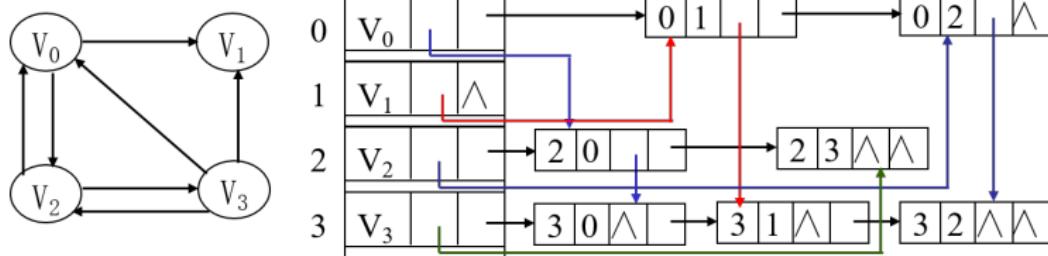
有向图十字链表的实现

```
1 #define INFINITY MAX_VAL /*最大值*/
2 #define MAX_VEX 30 //最大顶点数
3 typedef struct ArcNode{
4     int tailvex, headvex; //尾结点和头结点在图中的位置
5     InfoType *info; //与弧相关的信息，如权值
6     struct ArcNode *hlink, *tlink;
7 }ArcNode; //弧结点类型定义
8
9 typedef struct VexNode{
10    VexType data; // 顶点信息
11    ArcNode *firstin, *firstout;
12 }VexNode; //顶点结点类型定义
13
14 typedef struct{
15     int vexnum;
16     VexNode xlist[MAX_VEX];
17 }OLGraph; //图的类型定义
```

有向图十字链表的例子

图解十字链表

- 下图所示是一个有向图及其十字链表（略去了表结点的 info 域）
- 从这种存储结构图可以看出，从一个顶点结点的 firstout 出发，沿表结点的 tlink 指针构成了正邻接表的链表结构，而从一个顶点结点的 firstin 出发，沿表结点的 hlink 指针构成了逆邻接表的链表结构。



无向图的邻接多重表

无向图的链式表示

- 仿造有向图的十字链表 (融合邻接表和逆邻接表); 将无向图的邻接表中的重复边节点去掉
- 节点结构如下图所示, 说明如下
 - Data 域: 存储和顶点相关的信息;
 - 指针域 firstedge: 指向依附于该顶点的第一条边所对应的表结点;
 - 标志域 mark: 用以标识该条边是否被访问过;
 - ivex 和 jvex 域: 分别保存该边所依附的两个顶点在图中的位置;
 - info 域: 保存该边的相关信息;
 - 指针域 ilink: 指向下一条依附于顶点 ivex 的边;
 - 指针域 jlink: 指向下一条依附于顶点 jvex 的边;

边结点

mark	ivex	alink	jvex	jlink	info
data		firstedge			

顶点结点

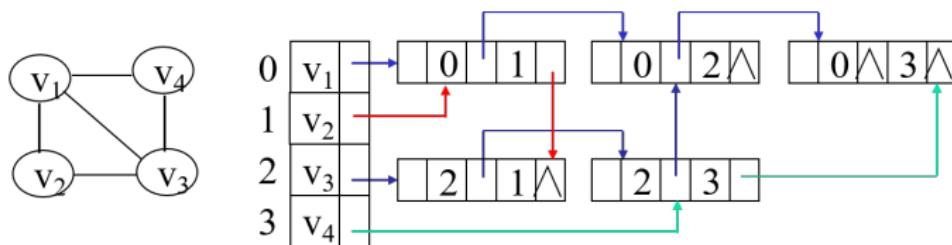
无向图的邻接多重表的实现

```
1 #define INFINITY MAX_VAL //最大值
2 #define MAX_VEX 30 //最大顶点数
3 typedef enum {unvisited, visited} Visitting;
4 typedef struct EdgeNode{
5     Visitting mark; //访问标记
6     int ivex, jvex; //该边依附的两个结点在图中的位置
7     InfoType *info; //与边相关的信息，如权值
8     struct EdgeNode *ilink, *jlink;
9     // 分别指向依附于这两个顶点的下一条边
10 }EdgeNode; //弧边结点类型定义
11
12 typedef struct VexNode{
13     VexType data; //顶点信息
14     ArcNode *firstedge; //指向依附于该顶点的第一条边
15 }VexNode; //顶点结点类型定义
16
17 typedef struct{
18     int vexnum;
19     VexNode mullist[MAX_VEX];
20 }AMGraph;
```

无向图的邻接多重表的例子

图解邻接多重表

- 邻接表的同一条边用两个表结点表示，而临界多重表只用一个表结点表示；
- 除标志域外，邻接多重表与邻接表表达的信息是相同的，因此，操作的实现也基本相似



思考题

结合实际问题，设计存储结构

- 写程序展示中国高铁地图，每个站点定义为一个节点，用哪种存储结构？
- 校内电子邮件网络中，每个账号是一个节点，发过电子邮件的账号间连一条弧；给定账户，要查找该账户联系最多的 10 个账户。采样什么数据结构来存储数据？
- 在微博上，每个账号是一个节点，给定某两个账号，计算它们之间有多少条长度 ≤ 6 的路径。采用什么样的数据结构存储图？描述算法思想，时间复杂度？
- 假设顶点用一个长度为 n 的向量来描述，其中向量的每个分量取值 0 或 1，共有多少个不同的顶点？任何两个顶点/向量之间，若海明距离为 1，则连边，有多少条边？这种图能用什么方式存储吗？（我们暂时称其为：**指数图**，其存储和访问代价超出了可接受范围）
- 计算机应用—人工智能：将一切事物、环境用长度为 n 的向量来描述，给定一个向量（事物或环境），如何处理和分析这个向量；每个向量用函数 $y = f(x)$ 来变换， y 代表机器人遇到环境 x 时的最佳应对或最佳行动，找到函数 f 就是人工智能的核心任务之一：机器学习

指数图的存储实现

无法存下所有的顶点和所有的边

- 我们只能考虑特殊情况的指数图：用一个向量变量 x 来描述一个顶点，用一个后继函数 $\text{successors}(x)$ 来描述顶点之间的连边关系
- 给定任何一个顶点，我们能用后继函数得到它所有的邻接点，这种方式“近似”完美地描述了一大类指数图
- 这种方式也完美解决了人工智能中面临的“数据结构/存储”方面的问题：人工智能倾向于将一切描述为向量，向量间转换用边来描述，人工智能通常面临的都是指数图问题，这些问题可大致归结为两类基本问题：
 - 搜索问题：给定图的描述/定义，求从起点到终点的路径。几乎所有的计算机求解的问题，都可以建模为搜索问题，通用问题求解器。若面对指数图：访问的时间复杂性如何？
 - 学习问题/数据分析：给定图的部分顶点和边及其性质，得到完整的图的描述或定义。可以转化为搜索问题。若面对指数图：存储和访问时的时空复杂性如何？

图上的搜索

问题定义

- 输入：图 $G = (V, E)$ ，初始顶点 v_0 ，目标测试函数 $Goal(s)$ 判断顶点 s 是否满足给定条件，满足给定条件返回 true 否则返回 false
- 输出：从给定初始顶点 v_0 出发沿着边到达目标顶点的路径
 - 一条路径或多条路径；最优（最短）路径
 - 退化时，不需要路径，只要目标状态

分析和理解：典型的不同目标测试函数 $Goal()$

- ① 找到指定编号的顶点（返回路径，应用案例：迷宫寻路）
- ② 找到一个或所有红色（某个指定属性）的顶点（返回目标顶点或路径，应用案例： n 皇后或华容道）
- ③ 输出连通图每个顶点的着色信息恰好一次（应用案例：遍历图）
 - 此时，测试函数的输入不是顶点 s ，与前两个有区别；
 - 除初始顶点 v_0 外，其它顶点输出之前，要保证其至少一个邻接点已被输出，这就是连通图的遍历

图上的搜索

问题定义

- 输入：图 $G = (V, E)$ ，初始顶点 v_0 ，目标测试函数 $Goal(s)$ 判断顶点 s 是否满足给定条件，满足给定条件返回 true 否则返回 false
- 输出：从给定初始顶点 v_0 出发沿着边到达目标顶点的路径

分析和理解：沿着边前进

- 思考：什么时候（实际应用问题或场景）可以不沿着边走，直接访问邻接矩阵或邻接表的顶点数组，完成搜索任务？
- 指数图上的搜索任务通常需要沿着边前进
- 非指数图上，要求搜索返回路径的，也需要沿着边前进
- 非指数图上，只要求返回顶点的，可以考虑依次访问顶点数组
- 例子： n 元实值函数的优化问题（求最小值），每个解是一个顶点，问题建模成指数图上的搜索问题，从一个随机解开始，采用爬山法搜索最小值，仅要求返回顶点，不要求路径。（指数图）

图的搜索算法框架

关键概念

- 搜索过程中图 G 的节点分为三类：访问过的节点，访问过节点的直接邻接点（称为搜索的边界），其它未被访问过的节点

Input: 图 G ; 初始节点 s_0 ;

Output: $path$: 代表解的路径

$path \leftarrow (s_0)$, $FRINGE \leftarrow \phi$ // 初始化，其中 $FRINGE$ 是保存搜索边界的数组;

if ($GOAL(s_0) = T$) then

| return $path = (s_0)$;

end

INSERT(s_0 , $FRINGE$);

while T do

| if $isEmpty(FRINGE) == T$ then

| | return failure //返回 failure, 表示无解;

end

$s \leftarrow REMOVE(FRINGE)$ //将搜索边界 $FRINGE$ 中第一个节点移除，并放到 s ;

if s 未访问过 then

| visit(s); //思考如何判定;

| update $path$; //思考如何更新路径;

| foreach s 的邻接点 s' do

| | if $GOAL(s') = T$ then

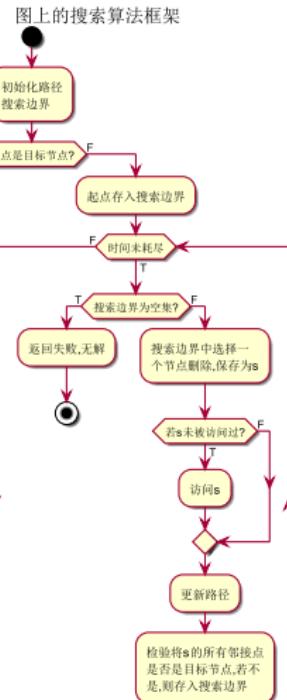
| | | return $(path, s')$ //找到目标节点，返回解;

| | end

| | INSERT($s', FRINGE$);

| end

end



图的搜索的各种不同变型

搜索边界的次序

- 各种搜索算法，不同之处在于对搜索边界中节点的次序确定方法
- 更具体地说，就是 $\text{INSERT}(s', \text{FRINGE})$ 函数的实现方法不同
- 广度优先搜索：每次插入 s' 时，放在 FRINGE 的末尾
- 深度优先搜索：每次插入 s' 时，放在 FRINGE 的开头
- 启发式搜索算法：依据问题的特点，设计一个节点排序的策略或方法，将 s' 插入在 FRINGE 的某个特殊位置
 - 贪婪/贪心算法：只考虑从当前节点到目标节点的最短路径
 - A* 算法：考虑从出发点开始，到目标点的最短路径
- 搜索策略：本质上就是决定哪一个节点放在 FRINGE 开头，不排序，只选择最有“希望的”节点

图搜索中的技术问题 1

如果一个节点被多次访问会出现什么问题？

- 算法不会停止，比如三个节点构成三角形，若搜索时，允许重复访问节点，访问会沿三角形永远停不下来；
- 允许节点重复访问的好处？（可以帮助找到最短路径！思考为什么）

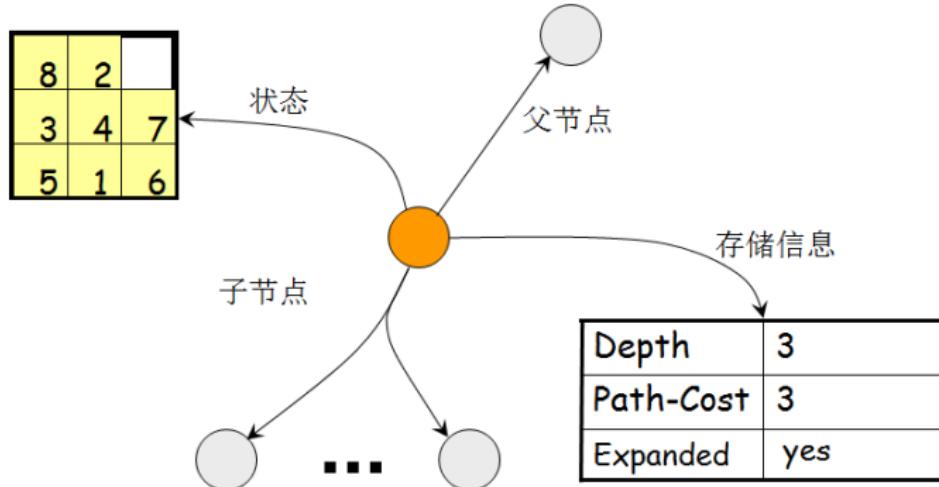
防止节点被多次访问

- 将访问过的节点保存在一个数组中，两种做法：
 - 节点数目少时：用一个标志数组 `bool visited[]` 来记录图的每个节点是否被访问过，假设图节点的编号是数组下标 $0, 1, 2, \dots$
 - 节点数目较多时，比如是 $O(b^d)$, $b > 1$ ，当 d 较大时，无法在内存开辟 `visited[]` 数组；这时内存中仅保存已经访问过的节点和 FRINGE 中的节点

图搜索中的技术问题 2

如何保存和更新路径

- 设计数据结构保存图的节点时，需要保存信息：经过的边数，经过边上权值之和，节点是否已经扩展过，指向父节点的指针
- 算法输出一棵树，每个子节点指向其双亲节点（扩展该子节点的节点）；根的出度为0；树的双亲表示法
- 输出路径时，从找到的目标节点，沿双亲节点“上行”，直至根节点
- 找最短路径时，检查第二次或者后来的对同一子节点的多次扩展时（重复访问），是否有更小的边权值之和或边数，有则更新该子节点的双亲指针



图搜索的附带输出

搜索树

- 如果在图搜索的过程中，不考虑重复，每次都将当前节点的直接邻居存入 Fringe 集合，以及该顶点的“来边/双亲节点”，每个节点会多次进入 Fringe 集合；最后输出一棵“搜索树”
- 此时，搜索树的不同节点可能对应原图的同一个节点，这是因为“允许重复访问”
- 若不允许重复访问，合并搜索树中属于相同原图节点的树节点，得到部分“原图”；若只保留从根到每个节点的“最短路径”，则得到一棵树
- 识别一个节点的“来边”，即节点加入 FRINGE 的原因：当前节点扩展时，生成其邻居节点，同时保存指向当前节点的指针

生成树

- 对连通图而言，假设搜索目标是访问所有的节点一次（遍历），那么搜索树就是生成树
- 对非连通图而言，假设搜索目标是访问所有节点一次（遍历），那么需要每个连通分量都启动一次搜索算法，得到生成森林

图搜索的时空复杂性

普通图/非指数图：图中所有的顶点和边都可以在内存中同时保存下来

- 若搜索的目标是必须返回路径，那么额外的存储空间需求为 `visited` 数组，故空间需求是 $O(n)$, n 是图的顶点数目，而时间复杂度是 $O(n + e)$, e 是图的边数
- 若搜索的目标仅需返回与路径无关的目标节点，那么依次检查顶点数组即可，平均查找时间为 $(n + 1)/2$ ，空间需求是 $O(1)$

图搜索的时空复杂性

指数图：节点无法全部同时存储在内存中

- “沿边前进” 访问节点成为必需；不管是找目标节点还是找路径
- 假设图中顶点的最大度为 b , 那么图的广度优先搜索算法的时空开销都是指数的 $O(b^d)$, d 是搜索树中埋藏最浅的目标节点的深度
- 而图的深度优先搜索算法的时间开销也是指数的 $O(b^m)$, 空间开销 $O(bm)$, m 是搜索树叶子的最大深度 (深度优先搜索过程中搜索树每一层最多只在 *FRINGE* 集合中保存 b 个邻居节点)
- 回溯算法：深度优先的改进，搜索树的每一层只保留 1 个邻居节点到 *FRINGE*, 所有空间开销是 $O(m)$
- 深度优先搜索和回溯法带来了空间开销的降低，然而损失了发现“最短路径”的能力

在连通图和非连通图上的图搜索算法

连通图

- 初始节点和目标节点（假设存在）位于同一个连通片，此时有解（存在从初始节点到目标节点的路径）
- 图为边无权或等权时，广度优先搜索一定能找到最短路径；深度优先搜索不一定能找到最短路径

非连通图

- 可能不存在从初始节点到目标节点的路径，此时无解；比如如下的15-数码问题：每个棋盘是一个顶点

8	2	
3	4	7
5	1	6



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	



1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

= \$1000

图的遍历：仅适用于非指教图

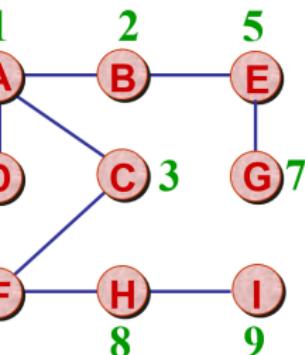
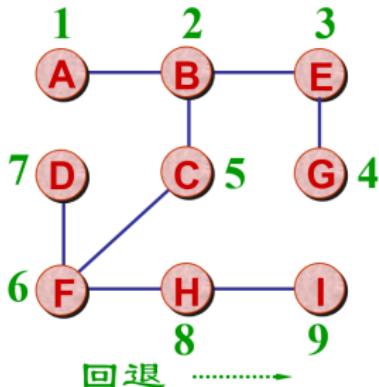
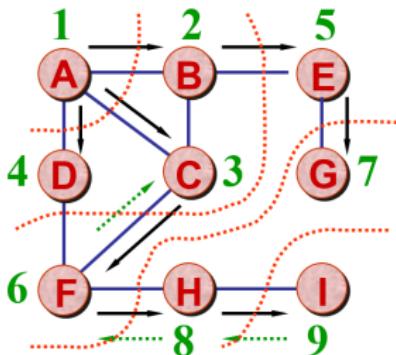
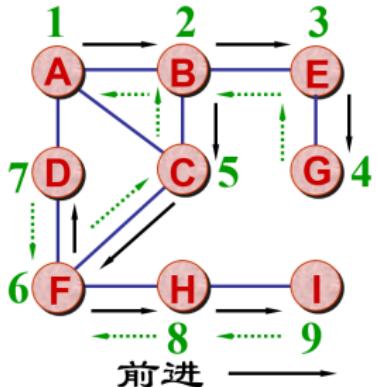
基本思想：访问图的每个节点仅一次（来自教材的描述）

- 从图中某一顶点出发，沿着图的边，访遍图中所有的顶点，且使每个顶点仅被访问一次，这一过程就叫做图的遍历 (Traversing Graph)
- 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辅助数组 visited []

从图的搜索来理解图的遍历

- 搜索的目标设置为：访问所有的节点一次
- 非连通图的每个连通分量需要从某个初始节点开始启动一次搜索
- 遍历时，搜索的目标测试被改成：visited 数组的元素都被置为了 true；（指教图，怎么测试该目标？没意义！）
- 遍历仅仅针对非指教图或节点数目较少的图才有意义！

连通图遍历的方法及例子



深度优先生生成树

广度优先生生成树

深度优先搜索遍历（来自教材，非指数图适用）

算法思想

- 在访问图中某一起始顶点 v 后, 由 v 出发, 访问它任一邻接顶点 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, 接着从此顶点出发, 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。

算法分析：关键操作是查找给定节点的邻居节点

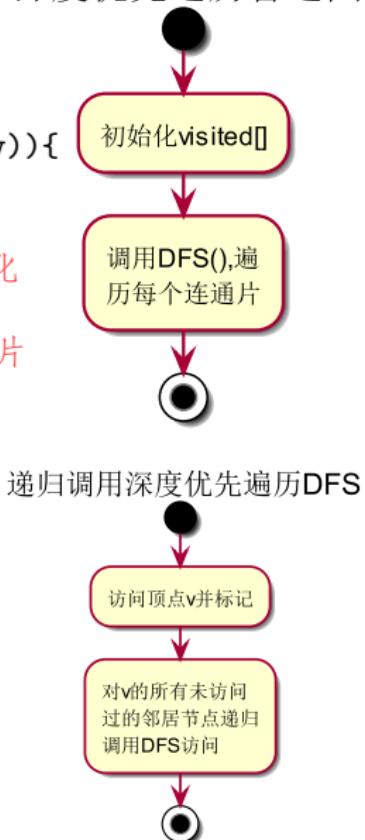
- 采用邻接矩阵实现算法: $O(n^2)$
- 采用邻接表实现算法: $O(n + e)$
- 哪一个更好？

深度优先搜索遍历（来自教材，非指数图适用）

算法实现：递归算法

```
1  bool visited[MAX];
2  Status (* VisitFunc)(int v);
3  void DFSTraverse (Graph G, Status (* Visit)(int v)){
4      VisitFunc = Visit;
5      for ( v= 0; v < G.vexnum; v++ )
6          visited [v] = FALSE; //访问数组 visited 初始化
7      for ( int v = 0; v < G.vexnum; v++ )
8          if (!visited[v]) DFS(G, v); //遍历每个连通片
9  }
10
11 void DFS (Graph G, int v) { //递归函数
12     visited[v] = TRUE; //顶点 v 作访问标记
13     VisitFunc(v); //访问顶点 v
14     for (w=FirstAdjVex(G,v);
15          w>=0;w=NextAdjVex(G,v,w))
16         if ( !visited[w] ) DFS (G, w);
17         //若顶点 w 未访问过，递归访问顶点 w
18 }
```

深度优先遍历普通图



宽度/广度优先搜索遍历（来自教材，非指数图适用）

算法思想

- 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。

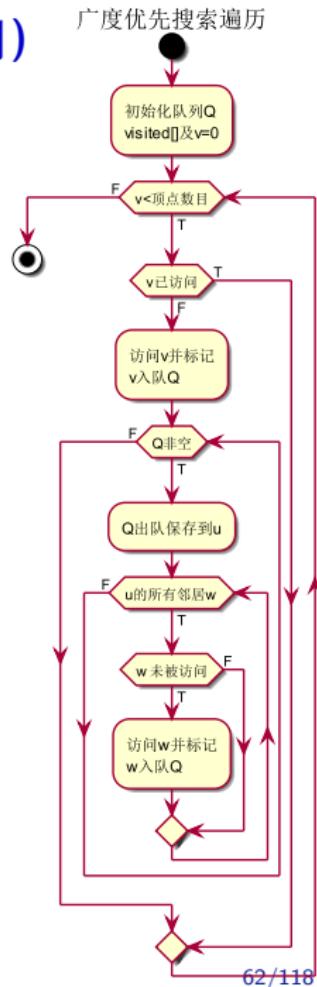
算法分析：关键操作是查找给定节点的邻居节点

- 为了实现逐层访问，算法中使用了一个队列 Q ，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
- 用广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是邻接点搜索次序不同，因此，广度优先搜索算法遍历图的时间复杂度与深度优先搜索一样，也和具体实现的存储结构相关

广度优先搜索遍历（来自教材，非指数图适用）

算法实现

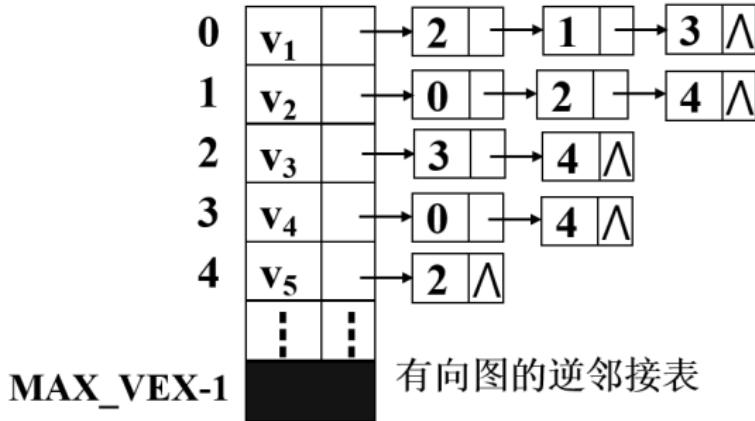
```
1 void BFSTraverse(Graph G,
2     Status (* Visit)(int v)){
3     for (v=0;v<=G.vexnum;++v)
4         visited[v]=FALSE; //初始化数组
5     InitQueue(Q); //初始化队列
6     for (v=0;v<=G.vexnum;++v)
7         if (!visited[v]){
8             visited[v]=TRUE;
9             Visit(v); //标记顶点并访问
10            EnQueue(Q,v); //入队列
11            while (!QueueEmpty(Q)){
12                DeQueue(Q,u); //出队列
13                //遍访u的所有邻接点
14                for (w=FirstAdjVex(G,u); w>=0;
15                    w=NextAdjVex(G,u,w))
16                    if (!Visited[w]) {
17                        Visited[w]=TRUE; Visit(w);
18                        EnQueue(Q,W); //入队列
19                    }
20            }}}
```



图的练习题

图的存储结构和遍历题

- 画出下图逆邻接表所示的图，并给出邻接矩阵
- 给出从顶点 v_1 开始的深度和广度优先遍历序列
- 画出深度和广度优先生成树



图遍历的输出

生成树和生成森林

- 图的搜索，输出是目标节点或到达目标节点的路径；
- 图的遍历，我们定义其输出为图搜索时输出访问的节点以及访问该节点的“原因”，这样就获得了：
 - 连通图：生成树
 - 非连通图：生成森林

算法实现：修改图的搜索算法

- 确定生成树或生成森林的存储结构
- 修改搜索算法中对每个节点的处理函数 `visit()` 的位置和功能（调整到 `foreach` 语句中，功能改成构造孩子节点）

深度优先搜索遍历图时，在内存中构造生成树 1

生成树存储结构的确定

- 不能确定输出的生成树有多少个孩子，因此采用“孩子-兄弟”链表来存储生成树

算法思想：递归算法

- 首先从某个顶点 V 出发，建立一个树结点 T
- 然后再以 V 的一个邻接点（第一个未被访问者）为起始点，建立子生成树（无右孩子，why？）
- 将子生成树作为 T 结点的子树链接为 T 结点的左孩子
- 第 2, 3, 4, ..., 个未被访问邻接点为起始点，建立子生成树，并链接为 T 或前一棵子生成树的右孩子

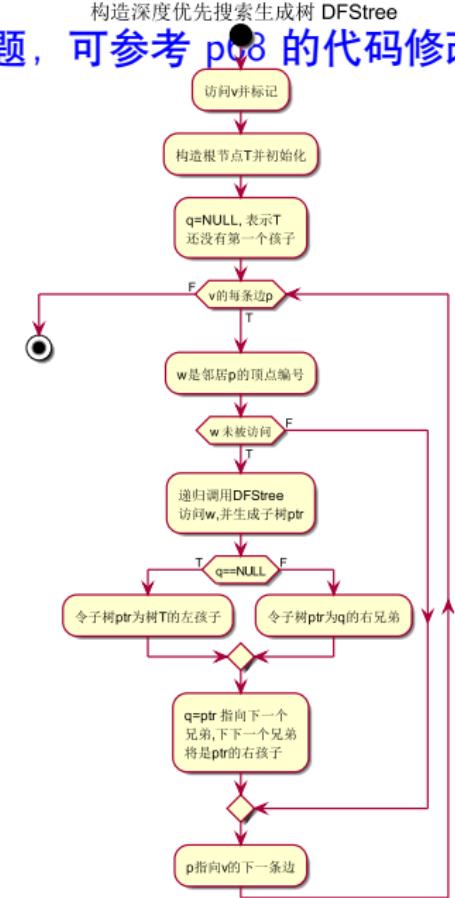
树节点的定义

```
1  typedef struct CSNode
2  {
3      ELEMTYPE data ;
4      struct CSNode *firstchild , *nextsibling ;
5  }CSNode ;
```

深度优先搜索遍历图时，在内存中构造生成树 2

实现代码：图用邻接表存储（代码细节有问题，可参考 pc8 的代码修改）

```
1 CSNode *DFStree(ALGraph *G, int v){  
2     CSNode *T, *ptr, *q; ArcNode *p; int w;  
3     visited(v); Visited[v]=TRUE;  
4     T=(CSNode *)malloc(sizeof(CSNode));  
5     T->data=G->vertices[v].data; //建立根结点  
6     T->firstchild=T->nexstsibling=NULL;  
7     q=NULL; p=G->vertices[v].firstarc;  
8     while (p!=NULL){  
9         w=p->adjvex;  
10        if (!Visited[w]) {  
11            ptr=DFStree(G,w); //子树根结点  
12            if (q==NULL) T->firstchild=ptr;  
13            else q->nexstsibling=ptr;  
14            q=ptr;  
15        }  
16        p=p->nextarc;  
17    }  
18    return(T);  
19 }
```



广度优先搜索遍历图时，在内存中构造生成树 1

生成树存储结构的确定

- 不能确定输出的生成树有多少个孩子，与深度遍历是一样采用“孩子-兄弟”链表来存储生成树

```
1  typedef struct Queue
2  {
3      int elem[MAX_VEX];
4      int front, rear;
5  }Queue;    //定义一个队列保存将要访问顶点
```

广度优先搜索遍历图时，在内存中构造生成树 2

生成树存储结构的确定 (感谢 PB19000078 汪震同学对该代码的贡献)

- 不能确定输出的生成树有多少个孩子，与深度遍历是一样采用“孩子-兄弟”链表来存储生成树

```
1 CSNode *BFSTree(ALGraph *G ,int v){  
2     CSNode *T, *ptr, *q, *now;  
3     CSNode *vertices[MAX_VERTEX_NUM] = {NULL};  
4     ArcNode *p; Queue Q; int w, k;  
5     //将G的所有顶点建立相应的树结点，并存入vertices[]中  
6     for (int i = 0; i < G->vexnum; i++) {  
7         vertices[i]=(CSNode *)malloc(sizeof(CSNode));  
8         if (vertices[i] == NULL) {exit();}  
9         vertices[i]->data = G->vertices[i].data;  
10        vertices[i]->firstchild = NULL;  
11        vertices[i]->nexstsibling = NULL; }  
12    //初始化visited[]  
13    bool visited[MAX_VERTEX_NUM] = {false};  
14    Q->front=Q->rear=0; //初始化空队列  
15    Visited[v]=TRUE;  
16    T=(CSNode *)malloc(sizeof(CSNode));  
17    if (!T) {....;exit();}  
18    T->data=G->vertices[v].data;  
19    T->firstchild=T->nexstsibling=NULL; //建立根结点  
20    Q->elem[Q.rear++]=v; // v入队
```

构造广度优先搜索生成树 BFSTree

```
21     while (Q.front!=Q.rear){  
22         w=Q.elem[Q.front++];  
23         q=NULL; //孩子-兄弟  
24         //p的所有邻点为兄弟  
25         p=G->vertices[w].firstarc;  
26         now = vertices[p].  
27         //p表示出队节点w关联的各条边/节点  
28         //p的所有未访问邻居入队  
29         while (p!=NULL){  
30             k=p->adjvex;  
31             if (!visited[k])  
32                 Visited[k]=TRUE, //添加树的根  
33                 ptr=vertices[k];  
34                 if (q==NULL) now=ptr;  
35                 else q->nexstsibling=ptr;  
36                 q=ptr;  
37                 Q->elem[Q.rear]=q;  
38             } //end if  
39             p=p->nextarc;  
40         } //endwhile p,Q  
41     return(T); /*求图G广度优先生成树算法BFSTree*/  
42     68/118*/
```

广度优先搜索遍历图时，在内存中构造生成树 2

用队列Q广度优先遍历图

生成树存储结构的确定 (感谢 PB19000078 汪震同学对该代码的贡献)

- 不能确定队列Q非空的生成树有多少个孩子，与深度遍历是一样采用“孩子-兄弟”链表来存储生成树

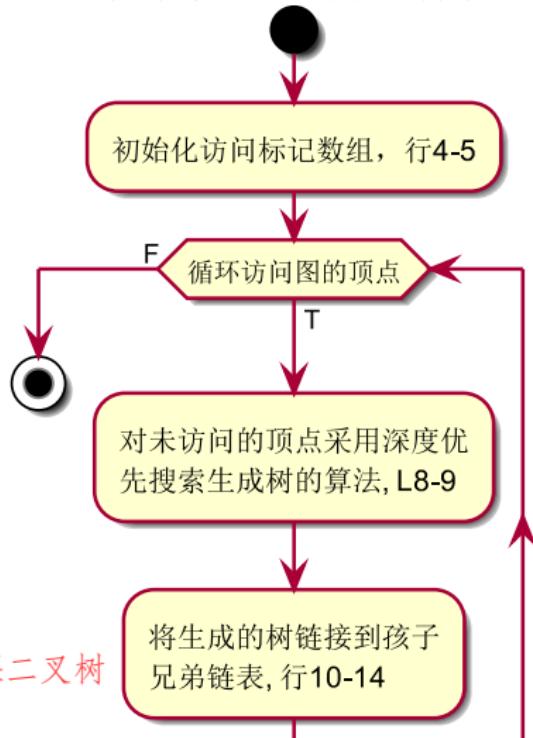
```
1 CSNode *BFSTree(ALGraph *G, int v){  
2     CSNode *T, *P;  
3     CSNode *vertices[Max_VERTEX_NUM] = {NULL};  
4     ArcNode *p; Queue k;  
5     //将G的所有顶点建立相应的树结点，并存入vertices[]中  
6     for (int i = 0; i < exnum; i++) {  
7         vertices[i]=(CSNode *)malloc(sizeof(CSNode));  
8         if (vertices[i] == NULL) {exit();}  
9         vertices[i]->data=G->vertices[i].data;  
10        vertices[i]->nextarc=NULL;  
11        vertices[i]->firstchild=NULL;  
12        //初始化visited[]  
13        bool visited[Max_VERTEX_NUM] = {false};  
14        Q->front=Q->rear=q=vertices[v];  
15        Visited[v]=TRUE;  
16        T=(CSNode *)malloc(sizeof(CSNode));  
17        if (!T) {....;exit();}  
18        T->data=G->vertices[v].data;  
19        T->firstchild=T->nextarc=NULL;  
20        Q->elem[Q->rear++]=v; //入队  
    }  
    Data Structure
```

```
21     while (Q.front!=Q.rear){  
22         w=Q.elem[Q.front++];  
23         q=NULL; //孩子-兄弟表示，理解标记q非常重要  
24         //p的所有邻点为兄弟，均为w的孩子  
25         p=G->vertices[w].firstarc;  
26         now = vertices[p.adjvex]; //now为当前顶点  
27         //p表示出队节点w关联的各条连边/节点  
28         //p的所有未访问邻居入队  
29         while (p!=NULL){  
30             k=p->adjvex;  
31             if (!visited[k]){  
32                 Visited[k]=TRUE; //添加访问代码  
33                 ptr=vertices[k];  
34                 if (q==NULL) now->firstchild=ptr;  
35                 else q->nexstsibling=ptr;  
36                 q=ptr;  
37                 Q->elem[Q.rear++]=k; //k入队  
38             } //end if  
39             p=p->nextarc;  
40         } //endwhile p,Q  
    } //endfor w,Q  
    return(T); } /*求图G广度优先生成树算法BFSTree*/  
68/118*
```

深度优先搜索遍历图时，在内存中构造生成森林

```
1 CSNode *DFSForest(ALGraph *G){  
2     CSNode *T, *ptr, *q;  
3     int w;  
4     for (w=0; w<G->vexnum; w++)  
5         Visited[w]=FALSE;  
6     T=NULL;  
7     for (w=0; w<G->vexnum; w++)  
8         if (!Visited[w]) {  
9             ptr=DFStree(G, w);  
10            if (T==NULL)  
11                T=ptr;  
12            else  
13                q->nexstsibling=ptr;  
14                q=ptr;  
15            }  
16        return(T);  
17 } //孩子兄弟表示法，在内存中表示成为一棵二叉树
```

深度优先遍历构造森林



尝试将各种生成树/生成森林构造算法改用 FRINGE 数组来实现

将图的遍历应用于图的连通性

求无向图的各个连通分量

- 输入：图 $G = (V, E)$
- 输出： $V = V_1 \cup V_2 \cup \dots \cup V_k$, $V_i \cap V_j = \emptyset, 1 \leq i \neq j \leq k$, V_i 内的节点属于同一个连通片，不同子集内的点属于不同连通片
- 算法思想：从任意节点开始，视其为初始节点，执行图搜索算法，直至算法结束，得到 V_1 ；选择任意未访问过的节点为初始节点，再次执行图搜索算法，得到 V_2, \dots ，不断重复，直到所有节点都被访问过

讨论

- 采用深度优先搜索遍历还是广度优先搜索遍历？
- 时空复杂度是多少？
- 问题变型：求连通片数量

图的遍历应用于图的连通性

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
- 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 ，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 ，则包含 V 的强连通分量的顶点集合是 T_1 和 T_2 的交集

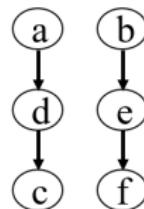
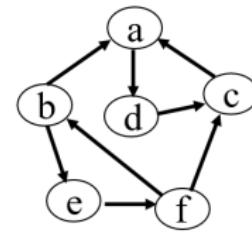
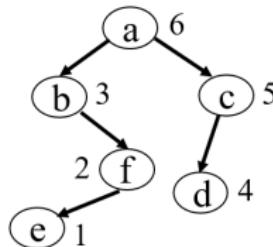
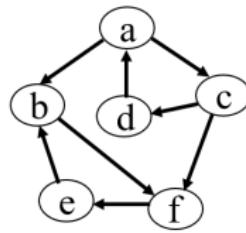
基本步骤

- 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 的顶点按后根遍历顺序进行编号
- 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

求强连通分量的例子

图解过程

- 图 (a) 表示的原图, 经过步骤 1, 2, 得到节点编号如 (b)
- (c) 将原图的边逆转, (d) 是执行步骤 (4),(5) 得到的强连通分量



(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)

图 利用深度优先搜索求有向图的强连通分量

生成树与最小生成树

生成树

- 使用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树
- 按照生成树的定义， n 个顶点的连通网络的生成树有 n 个顶点、 $n - 1$ 条边

最小生成树：构造准则

- 必须使用且仅使用该网络中的 $n - 1$ 条边来联结网络中的 n 个顶点
- 不能使用产生回路的边
- 各边上的权值的总和达到最小

最小生成树

应用场景

- 在 n 个城市间构建高铁网络，使得所有城市通过高铁可以互联，求建设成本最低的设计方案（假定所有路线的单位长度成本一样）

n 个城市之间，彼此距离不一样，构成一个网络/有边权的图

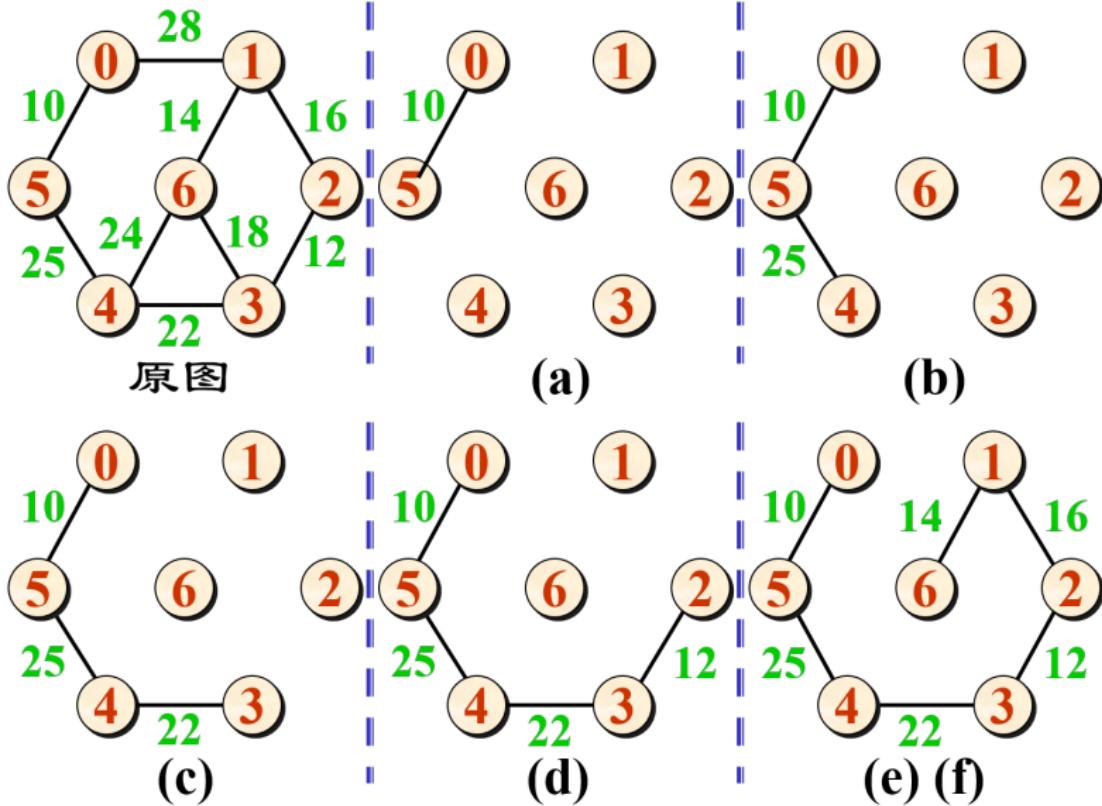
- 首先确认建设成本最低的方案，不会存在环
- 设计方案中，城市为节点的图是连通的
- 求建设成本最低，即求最小生成树
- 距离最近的两个城市间，一定会修高铁（若两对或多对城市间是相等的最短距离，则至少有一对最近的城市间会修高铁）
 - 证明：反证法。假设最小生成树不包含最短边，连接最短边的两个顶点，此时必然生成环，删除环上的任何一条边，得到新的生成树，新的生成树边权值之和相对于添加边之前是“不增的”，而此时最短边已经加入了生成树。此时要保证最小生成树依然是最小的，那么必然要求添加的边与删除的边等权值。

求最小生成树：普里姆算法

思想：

- 采用邻接矩阵作为图/网络的存储表示
- 从连通网络 $N = (V, E)$ 中的某一顶点 u_0 出发, $U = \{u_0\}$
- 选择与 u_0 关联的具有最小权值的边 (u_0, v_0) , 将其顶点 v_0 加入到生成树顶点集合 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) , 把顶点加入 v 到集合 U 中
- 如此继续下去, 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

求最小生成树：普里姆算法例子

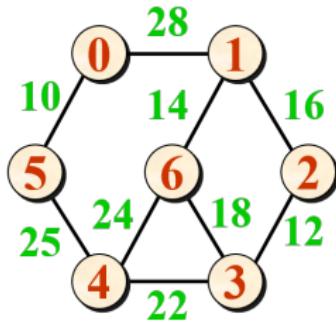


求最小生成树：普里姆算法实现 1

实现的关键点

- 设置一个辅助数组：closedge[0..vexnum-1]: 记录 $V - U$ 中点到 U 具有最小代价的边（为什么不是 U 中点到 $V - U$ 的最小边？）
- 对每个顶点 $v_i \in V - U$
 - ① closedge[i-1].lowcost = $\text{Min}\{cost(u, v_i) | u \in U\}$
 - ② closedge[i-1].adjvex: 对应这条边依附的 U 中的顶点

例子

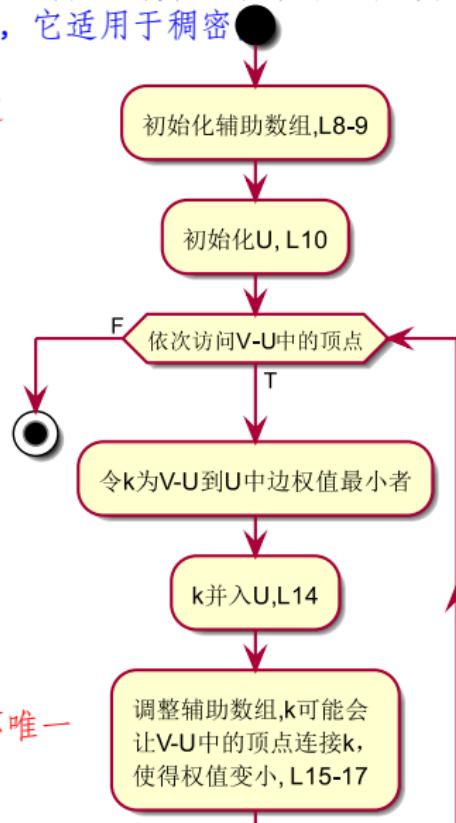


∞	28	∞	∞	∞	10	∞
28	∞	16	∞	∞	∞	14
∞	16	∞	12	∞	∞	∞
∞	∞	12	∞	22	∞	18
∞	∞	∞	22	∞	25	24
10	∞	∞	∞	25	∞	∞
∞	14	∞	18	24	∞	∞

求最小生成树：普里姆算法实现 2 —伪代码

```
1  typedef struct{ 时间复杂度为  $O(n^2)$ , n 是节点数目, 它适用于稠密图  
2      VexType adjvex;  
3      WType lowcost;//0:已经加入U; 无穷大:无直接连边  
4  } closedge[MAX_VERTEX_NUM];  
5  
6  void MiniSpanTree_PRIM(MGraph G, VexType u){  
7      f=LocateVex(G,u);  
8      for (j=0;j<G.vexnum;++j)  
9          if (j!=f) closedge[j]={u,G.adj[f][j].w};  
10     closedge[f].lowcost =0; //初始, U={u}  
11     for (i=0;i<G.vexnum&&i!=f;++i){  
12         k=minimum(closedge); //非0最小权值的下标  
13         printf(closedge[k].adjvex,G.vexs[k]);  
14         closedge[k].lowcost = 0; //k并入U集  
15         for (j=0;j<G.vexnum;++j)//调整辅助数组  
16             if (G.adj[k][j].w<closedge[j].lowcost)  
17                 closedge[j]={G.vexs[k],G.adj[k][j].w};  
18     }各边有相同权值时, 因选择的随意性, 生成树可能不唯一  
19 }各边的权值不相同时, 产生的生成树是唯一的
```

普里姆算法求最小生成树

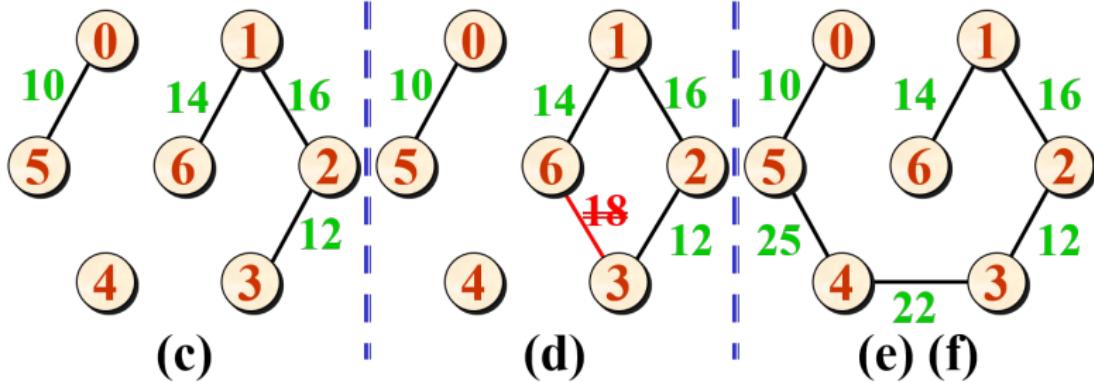
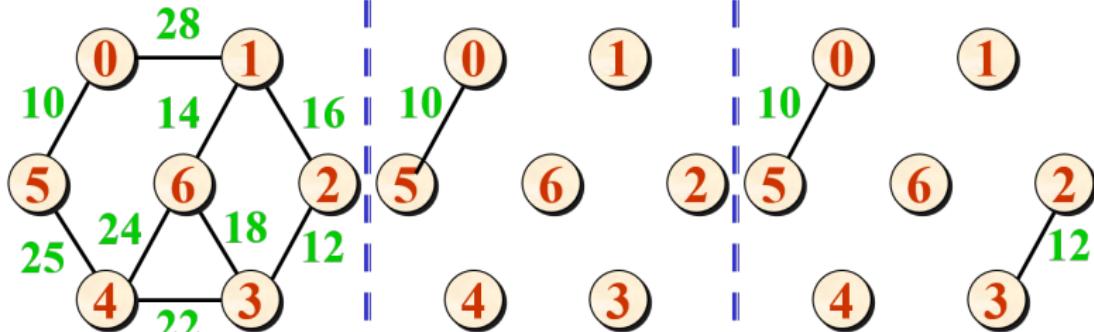


求最小生成树：克鲁斯卡尔算法

思想

- 从 n 个孤立顶点出发，顺次加入 $n - 1$ 条边
- 假设连通网 $N = (V, E)$ ，令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T = (V, \{\})$ ，图中每个顶点自成一个连通分量
- 在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上，则将此边加入，否则舍去该边而选择下一条代价最小的边
- 依次类推，直至 T 中所有顶点都在同一连通分量上为止

求最小生成树：克鲁斯卡尔算法例子



求最小生成树：克鲁斯卡尔算法分析

时间复杂度分析

- 对每条边最多访问一次；时间开销 $O(e)$
- 每次要选择最小的边，花费的时间为 $\log(e)$
- 故时间复杂度为 $O(e \log e)$
- 另一种分析方法：花费 $O(e \log e)$ 的时间将边排序，然后依次对最小边处理
- 当网的边比较稀疏时，比如边数是 $O(n)$ ，那么克鲁斯卡尔算法性能比克里姆算法优

最小生成树练习题

求最小生成树

- 如图，求最小生成树
- 分别采用普里母算法和克鲁斯卡尔算法，给出求解过程

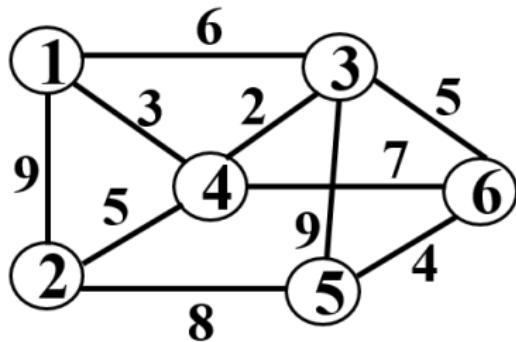


图 带权无向图

有向无环图

定义

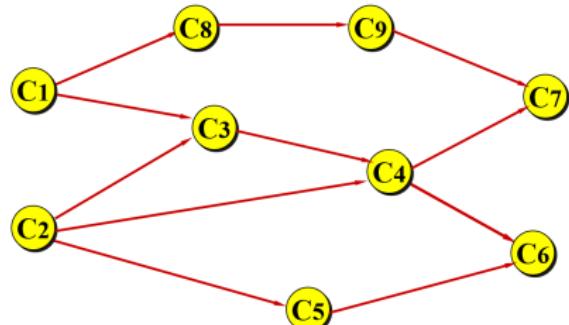
- 无环的有向图

应用场景

- 描述含有公共子式的表达式以及一项工程或系统进行过程的工具
- 用顶点表示活动的网络 (AOV 网络)
 - 计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了
 - 例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习

有向无环图的应用例子

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 ₁ , C ₂
C ₄	数据结构	C ₃ , C ₂
C ₅	高级语言程序设计	C ₂
C ₆	编译方法	C ₅ , C ₄
C ₇	操作系统	C ₄ , C ₉
C ₈	普通物理	C ₁
C ₉	计算机原理	C ₈



学生课程学习工程图

AOV 网络的应用中通常不允许出现环

- 可以用有向图表示一个工程。在这种有向图中，用顶点表示活动，用有向边 $< V_i, V_j >$ 表示活动 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)
- 在 AOV 网络中不能出现有向回路，即有向环。如果出现了有向环，则意味着某项活动应以自己作为先决条件
- 因此，对给定的 AOV 网络，必须先判断它是否存在有向环

检测 AOV 网络中是否有环

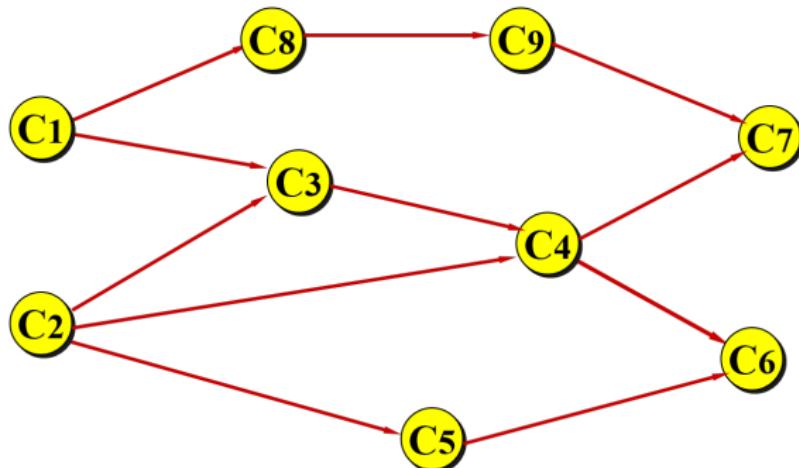
拓扑排序

- 检测有向环的一种方法是对 AOV 网络构造它的拓扑有序序列。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
AOV 网络中所有应存在的前驱和后继关系都能得到满足
- 通常图中的有向边只直接给出了部分顶点的“先后”次序，能否给
出任意两个顶点的“先后”次序？（拓扑偏序 \Rightarrow 拓扑全序）
- 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
- 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出现有向环

拓扑排序的例子

不能给出拓扑全序的 AOV 网络表示了一项不可行的工程

- 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
- C1 , C2 , C3 , C4 , C5 , C6 , C8 , C9 , C7
- 或 C1 , C8 , C9 , C2 , C5 , C3 , C4 , C7 , C6
- 上述线性序列定义了任意两个顶点的先后次序, 同时保证了原图中的先后次序都成立



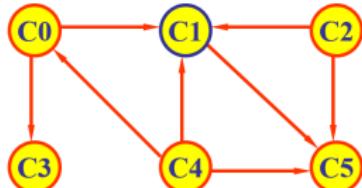
学生课程学习工程图

拓扑排序

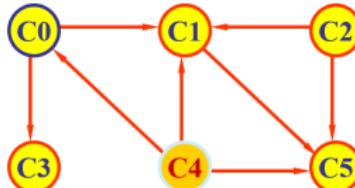
算法思想

- 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

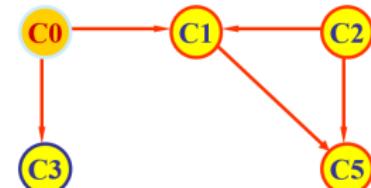
拓扑排序的例子



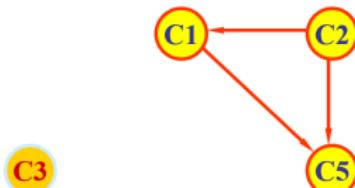
(a) 有向无环图



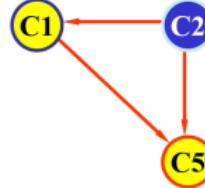
(b) 输出顶点C4



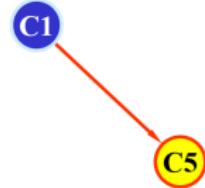
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

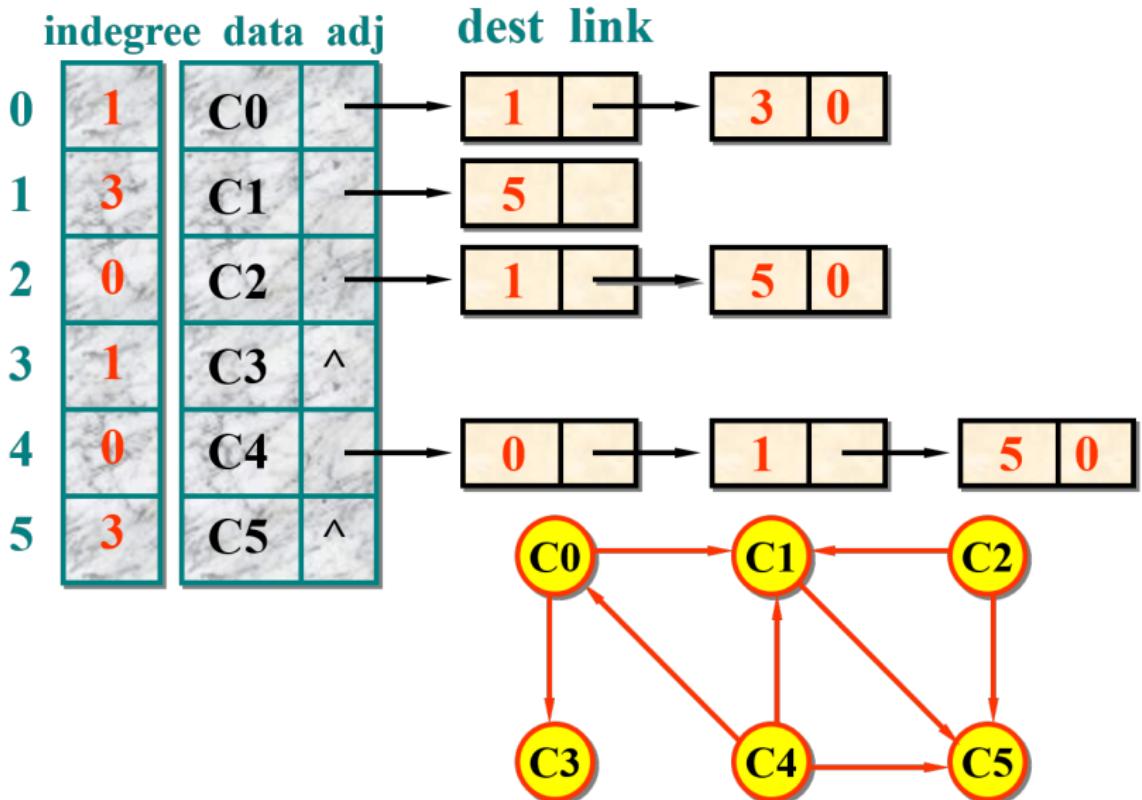
C5

(g) 输出顶点C5

拓扑排序完成

拓扑有序序列为 $C_4, C_0, C_3, C_2, C_1, C_5$

AOV 网络的邻接表表示



拓扑排序的实现

方法一：AOV 网络存储及改动

- 在邻接表中增设一个数组 `indegree[]`, 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 `VexList[]` 和入度数组 `indegree[]` 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:

```
1 EdgeNode * p=new EdgeNode;
2 p->dest=k; //建立边结点
3 p->link=G.VexList[j].firstAdj;
4 VexList[j].firstAdj=p;
5 //链入顶点 j 的边链表的前端
6 indegree[k]++; //顶点 k 入度加一
```

拓扑排序的实现

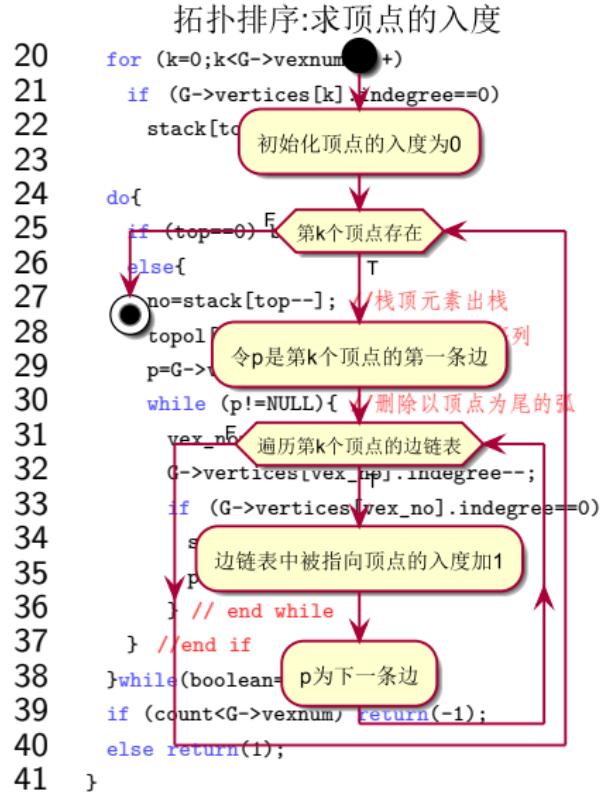
结合存储结构的实现描述

- 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：
 - ① 建立入度为零的顶点栈
 - ② 当入度为零的顶点栈不空时，重复执行
 - ③ 从顶点栈中退出一个顶点，并输出之
 - ④ 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
 - ⑤ 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

拓扑排序的实现

实现代码

```
1 void count_indegree(ALGraph *G){//方法二
2     int k; ArcNode *p;
3     for (k=0;k<G->vexnum;k++)
4         G->vertices[k].indegree=0; //顶点入度初始化
5     for(k=0;k<G->vexnum;k++){
6         p=G->vertices[k].firstarc;
7         while (p!=NULL){ //顶点入度统计
8             G->vertices[p->adjvex].indegree++;
9             p=p->nextarc;
10        }
11    }
12 }
13
14 int Topologic_Sort(ALGraph *G, int topol[]){
15 //顶点的拓扑序列保存在一维数组topol中
16 int k, no, vex_no, top=0, count=0, boolean=1;
17 int stack[MAX_VEX]; //用作堆栈
18 ArcNode *p;
19 count_indegree(G); //统计各顶点的入度
```



拓扑排序的实现

实现代码

```
1 void count_indegree(ALGraph *G){ //方法二
2     int k; ArcNode *p;
3     for (k=0;k<G->vexnum;k++)
4         G->vertices[k].indegree=0;
5     for(k=0;k<G->vexnum;k++)
6         p=G->vertices[k].firstarc;
7         while (p!=NULL){ //顶点no的边p非空
8             G->vertices[p->adjvex].indegree++;
9             p=p->nextarc;
10        }
11    }
12 }
13
14 int Topologic_Sort(ALGraph *G, int topol[]){
15     //顶点的拓扑序列保存在一维数组topol中
16     int k, no, vex_no, top=0, count=0, boolean=1;
17     int stack[MAX_VEX]; //用作堆栈
18     ArcNode *p;
19     count_indegree(G); //如果拓扑序列中顶点数小于图中顶点数,那么图中一定有环
20 }
```

初始化顶点的入度,L19

入度为0的顶点入栈,L20-22

出栈,输出栈顶no到拓扑序列top1,L27-28

将被指向顶点的入度减1,L31-32

被指向顶点的入度为0
被指向顶点入栈,L33-34

如果拓扑序列中顶点数小于图中的顶点数,那么图中一定有环

度初始化

F 案非空

T

顶点no的边p非空

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

for (k=0;k<G->vexnum;k++)

if (G->vertices[k].indegree==0)

stack[top++]=k;

do{

if (top==0) boolean=0;

else{

no=stack[top--]; //栈顶元素出栈

topol[count++]=no; //记录顶点序列

p=G->vertices[no].firstarc;

while (p!=NULL){ //删除以顶点为尾的弧

vex_no=p->adjvex;

G->vertices[vex_no].indegree--;

if (G->vertices[vex_no].indegree==0)

stack[top++]=vex_no;

p=p->nextarc;

} // end while

} //end if

}while(boolean==1);

if (count<G->vexnum) return(-1);

else return(1);

}

拓扑排序算法分析

时间复杂度

- 设 AOV 网有 n 个顶点, e 条边, 则算法的主要执行是:
- 统计各顶点的入度: 时间复杂度是 $O(n + e)$
- 入度为 0 的顶点入栈: 时间复杂度是 $O(n)$
- 排序过程: 顶点入栈和出栈操作执行 n 次, 入度减 1 的操作共执行 e 次, 时间复杂度是 $O(n + e)$;
- 因此, 整个算法的时间复杂度是 $O(n + e)$ 。

拓扑排序练习题

题目说明

- 给出图 1 表示邻接矩阵的 AOV 网络的拓扑排序过程
- 给出图 2 所示有向图的所有可能的拓扑序列

$$\begin{matrix} V_0 & \left(\begin{array}{ccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ V_1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ V_2 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ V_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ V_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ V_5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ V_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ V_7 & \end{matrix}$$

图1 一个AOV网的邻接矩阵

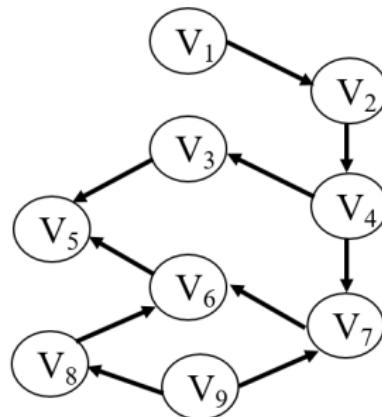
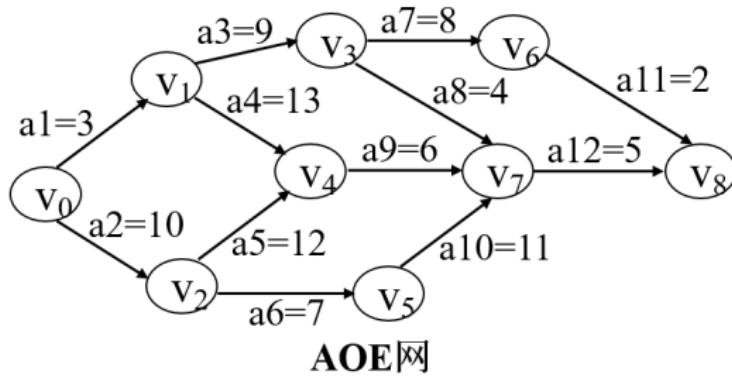


图2 有向图

AOE 网络

定义

- 与 AOV 网相对应的是 AOE (Activity On Edge)，是边表示活动的有向无环图，如图所示
- 图中顶点表示事件 (Event)，每个事件表示在其前的所有活动已经完成，其后的活动可以开始
- 弧表示活动，弧上的权值表示相应活动所需的时间或费用



AOE 网络的应用场景

完成工程至少需要多少时间? 哪些活动是影响工程进度(费用)的关键?

- 工程完成最短时间: 从起点到终点的最长路径长度 (路径上各活动持续时间之和)
- 长度最长的路径称为关键路径, 关键路径上的活动称为关键活动。关键活动是影响整个工程的关键
- 设 v_0 是起点, 从 v_0 到 v_i 的最长路径长度称为事件 v_i 的最早发生时间, 即是以 v_i 为尾的所有活动的最早发生时间

AOE 网络的术语与符号

术语与符号说明

- 若活动 a_i 是弧 $\langle j, k \rangle$, 持续时间是 $dut(\langle j, k \rangle)$, 设
- $e(i)$: 表示活动 a_i 的最早开始时间
- $l(i)$: 在不影响进度的前提下, 表示活动 a_i 的最晚开始时间; 则
 $l(i) - e(i)$ 表示活动 a_i 的时间余量, 若 $l(i) - e(i) = 0$, 表示活动 a_i 是关键活动
- $ve(i)$: 表示事件 v_i 的最早发生时间, 即从起点到顶点 v_i 的最长路径长度
- $vl(i)$: 表示事件 v_i 的最晚发生时间。则有以下关系:
 $e(i) = ve(j)$
 $l(i) = vl(k) - dut(\langle j, k \rangle)$

AOE 网络的术语与符号

理解 $ve(i)$

- $ve(j) = \begin{cases} 0 & j = 0, 表示 v_j 是起点 \\ \max\{ve(i) + dut(<i, j>) | < v_i, v_j > 是网中的弧\} & 其它 \end{cases}$
- 含义：源点事件的最早发生时间设为 0；除源点外，只有进入顶点 v_j 的所有弧所代表的活动全部结束后，事件 v_j 才能发生。即只有 v_j 的所有前驱事件 v_i 的最早发生时间 $ve(i)$ 计算出来后，才能计算 $ve(j)$
- 方法：对所有事件进行拓扑排序，再按拓扑顺序计算每个事件的最早发生时间

理解 $vl(j)$

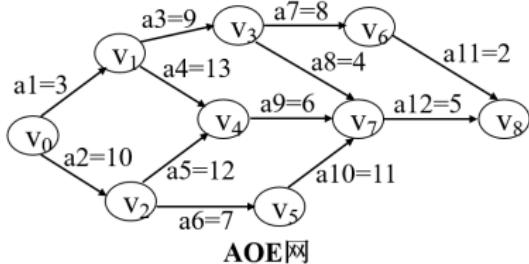
- $vl(j) = \begin{cases} ve(n - 1) & j = n - 1, 表示 v_j 是终点 \\ \min\{vl(k) - dut(<j, k>) | < v_j, v_k > 是网中的弧\} & 其它 \end{cases}$
- 含义：当 v_j 的所有后继事件 v_k 的最晚发生时间 $vl(k)$ 计算出来后，才能计算 $vl(j)$
- 方法：按拓扑排序的逆顺序，依次计算每个事件的最晚发生时间。

求 AOE 网络中的关键路径和关键活动

算法思想

- ① 利用拓扑排序求出 AOE 网的一个拓扑序列
- ② 从拓扑排序的序列的第一个顶点（源点）开始，按拓扑顺序依次计算每个事件的最早发生时间 $ve(i)$
- ③ 从拓扑排序的序列的最后一个顶点（汇点）开始，按逆拓扑顺序依次计算每个事件的最晚发生时间 $vl(i)$

求 AOE 网络中的关键路径和关键活动：例子



AOE网

ve(i)和vl(i)的值

顶点	v ₀	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆	v ₇	v ₈
ve(i)	0	3	10	12	22	17	20	28	33
vl(i)	0	9	10	23	22	17	31	28	33

给定 AOE 图，算法执行过程

- 拓扑排序的序列是: (v₀, v₁, v₂, v₃, v₄, v₅, v₆, v₇, v₈)
- 根据计算 $ve(i)$ 的公式和计算 $vl(i)$ 的公式，计算各个事件的 $ve(i)$ 和 $vl(i)$ 值，如表所示
- 根据关键路径的定义，知该 AOE 网的关键路径是: (v₀, v₂, v₄, v₇, v₈) 和 (v₀, v₂, v₅, v₇, v₈)
- 关键路径活动是: <v₀, v₂>, <v₂, v₄>, <v₂, v₅>, <v₄, v₇>, <v₅, v₇>, <v₇, v₈>

活动余量时间为 $0 = l(i) - e(i) = (vl(k) - dut(<j, k>)) - ve(j)$

关键路径算法

实现代码

```
1 void critical_path(ALGraph *G){  
2     int j,k,m; ArcNode *p;  
3     int topool[MAX_VER],ve[MAX_VER],vl[MAX_VER];  
4     if (Topologic_Sort(G, topol)==-1)  
5         printf(``\nAOE网中存在回路，错误!!\n\n'');  
6     else{  
7         for(j=0;j<G->vexnum;j++)  
8             ve[j]=0; //事件最早发生时间初始化  
9         for (m=0;m<G->vexnum;m++){  
10            j=topol[m];//存放了拓扑有序序列  
11            p=G->vertices[j].firstarc;  
12            for (;p!=NULL;p=p->nextarc){  
13                k=p->adjvex;  
14                if (ve[j]+p->weight>ve[k])  
15                    ve[k]=ve[j]+p->weight;  
16            } //遍历拓扑序列中第m个顶点的边链表  
17            //更新被指向节点的ve(k)，求最大值  
18        } //从拓扑有序序列的第一个开始，依次  
19        //计算每个事件的最早发生时间ve值  
20        for(j=0;j<G->vexnum;j++)  
21            vl[j]=ve[topol[g->vexnum-1]]; //事件最晚发生时42初始化  
22        for (m=G->vexnum-1;m>=0;m--){  
23            j=topol[m];  
24            p=G->vertices[j].firstRarc;//逆邻接表  
25            for (; p!=NULL; p=p->nextRarc ){  
26                k=p->adjvex;  
27                if (vl[k]-p->weight<vl[j])  
28                    vl[j]=vl[k]-p->weight;  
29            } //遍历拓扑序列中第m个顶点的边链表  
30            //更新被指向节点的vl(k)，求最小值  
31        } //从拓扑有序序列的最后一个开始，  
32            //依次计算每个事件的最早发生时间vl值  
33        for (m=0;m<G->vexnum;m++){  
34            p=G->vertices[m].firstarc;  
35            for (;p!=NULL;p=p->nextarc){  
36                k=p->adjvex;  
37                if ((ve[m]+p->weight)==vl[k])  
38                    printf("<%d, %d>", m, k);  
39            } //检查每个顶点的每条边，是否是关键活动  
40        } //输出所有的关键活动  
41    } // end of else
```

关键路径算法

算法时间复杂度分析

- 设 AOE 网有 n 个事件, e 个活动, 则算法的主要执行是:
- 进行拓扑排序: 时间复杂度是 $O(n + e)$
- 求每个事件的 ve 值和 vl 值: 时间复杂度是 $O(n + e)$
- 根据 ve 值和 vl 值找关键活动: 时间复杂度是 $O(n + e)$
- 因此, 整个算法的时间复杂度是 $O(n + e)$

关键路径练习题

题目说明

- 如图所示 AOE 网络，求出每个事件发生的最早和最晚时间
- 给出工程完工的最少时间
- 求出所有关键路径和关键活动

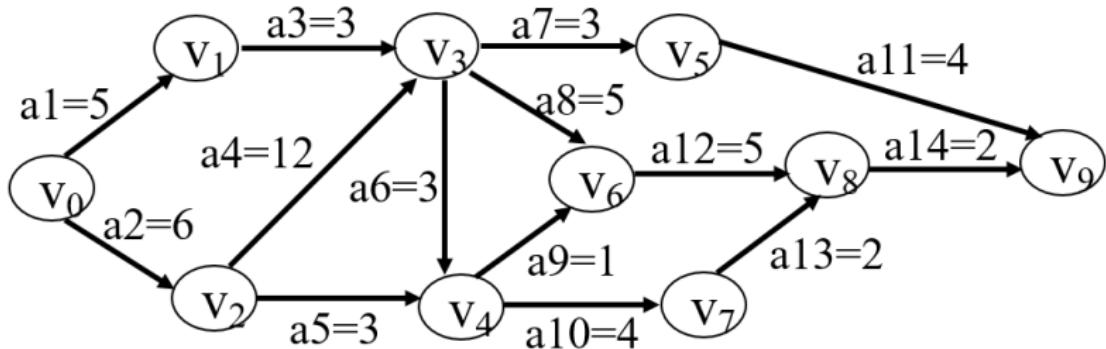


图 一个AOE网

最短路径

问题描述

- 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？

问题分析与理解

- 最短路径即图的搜索问题添加一个约束条件“从源点到终点”的路径“耗散”最小
- 当图的边无权时，可认为每条边上的权值是 1

主要解决办法

- 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

Dijkstra 算法

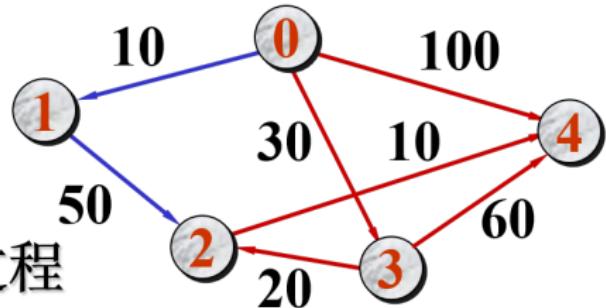
边上权值非负情形的单源最短路径问题

- 给定一个带权有向图 G 与源点 v , 求从 v 到 G 中其它顶点的最短路径。限定各边上的权值大于或等于 0

Dijkstra 算法思想

- 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
- 源点到其他 $n - 1$ 个顶点的最短距离, 从小到大, 依次求出
- 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
- 算法求出了源点到所有点的最短路径, 而不仅是到终点的最短路径

Dijkstra 算法执行的例子



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_0	v_2	(v_0, v_3, v_2)	50
v_0	v_3	(v_0, v_3)	30
v_0	v_4	(v_0, v_3, v_2, v_4)	60

Dijkstra 算法的关键步骤

求下一条最短路径

- 两个顶点集合 $S, V - S$ 视为两个对立的军事集团， S 的人希望把 $V - S$ 里的人策反，每次策反一个，选谁为目标？ S 中的大 Boss 源点，把活动经费派下去，沿途都会被截留（边权值），沿最短路径到达 S 的不同点，他们然后联系 $V - S$ 中的顶点；找到下一条（ S 到 $V - S$ 的）最短路径
- 计算方法： S 中每个点 v_j 都记住源点到自己的最低代价，然后检查自己“一步”策反对方的最小代价（与对手直接相连的最短边）； S 中每个点得到的最短路径间再次选出最短路径

体会细微差别

- 两个顶点集合 A, B 之间连了若干边，求这些边的权值最小者
- 两个顶点集合 A, B 之间连了若干边，顶点集合 A 内部也有部分连边；从顶点集 A 的 v_0 出发，找到 B 的某一顶点，路径最短

Dijkstra 算法的实现

引入辅助变量 D : 存储已找到的各个最短路径长度

- 它的每一个分量 $D[i]$ 表示当前找到的从源点 v 到终点 v_i 的最短路径的长度。初始状态：
 - 若从源点 v 到顶点 v_i 有弧，则 $D[i]$ 为该边上的权值；
 - 若从源点 v 到顶点 v_i 无弧，则 $D[i]$ 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（设其终点为 x ）或者是弧 (v, x) ，或者是从 v 出发，中间只经过 S 中的顶点而最后到达 x 的路径
- 每次求得一条最短路径后，其终点 v_j 加入集合 S ，然后对所有的 $v_k \in V - S$ ，修改其 $D[i]$ 值

Dijkstra 算法的实现

算法流程

① 初始化：

$$S \leftarrow \{v\}$$

$$D[i] \leftarrow adj[LocateVex(G, v)][i]$$

② 选择 v_j 使得：

$$D[j] \leftarrow \min\{D[i]\}, v_i \in V - S$$

$$S \leftarrow S \cup \{j\}$$

③ 修改：

$$D[k] \leftarrow \min\{D[k], D[j] + adj[j][k], v_k \in V - S\}$$

④ 判断：

若 $S = V$, 则算法结束, 否则转步骤 2

Dijkstra 算法的实现

实现代码

- 参考教材算法 7.15

算法评述

- 时间复杂度 $O(n^2)$, 不管采用邻接矩阵还是邻接表来存储图
- 若仅仅求给定源点和一个终点的最短路径, 时间复杂度还是 $O(n^2)$, 结合算法实现代码思考为什么。
- 空间需求是多少 ?

Dijkstra 算法练习题

题目说明

- 如图，采用 Dijkstra 算法求出从顶点 V4 出发到其余顶点的最短路径，给出过程

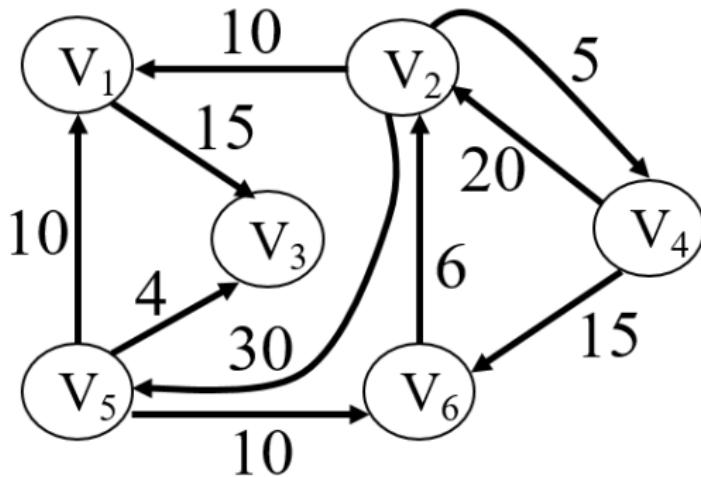


图 带权有向图

求所有点对之间的最短距离

解法一：基于 Dijkstra 算法

- 构建一个循环，每次循环用不同的源点调用一次 Dijkstra 算法
- 所以算法的时间复杂度是 $O(n^3)$ 。共有 $O(n^2)$ 条最短路径，每条最短路径的找寻时间至少需要线性的时间

解法二：Floyd 算法

- 其时间复杂度仍是 $O(n^3)$
- 但算法形式更为简明，步骤更为简单
- 数据结构仍然是基于图的邻接矩阵

Floyd 算法

算法思想：设顶点集 S （初值为空），用数组 A 的每个元素 $A[i][j]$ 保存从 v_i 只经过 S 中的顶点到达 v_j 的最短路径长度

- ① 初始时令 $S = \{\}$ ， $A[i][j]$ 的赋初值方式是

$$A[i][j] = \begin{cases} 0 & i = j \\ W_{ij} & i \neq j, \langle v_i, v_j \rangle \in E, w_{ij} : weight \\ \infty & i \neq j, \langle v_i, v_j \rangle \notin E \end{cases}$$

- ② 将图中一个顶点 v_k 加入到 S 中，修改 $A[i][j]$ 的值

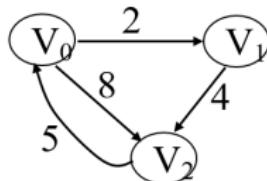
- 修改方法是： $A[i][j] = \min\{A[i][j], (A[i][k] + A[k][j])\}$
- 原因：从 v_j 只经过 S 中的顶点 (v_k) 到达 v_j 的路径长度可能比原来不经过 v_k 的路径更短

- ③ 重复步骤 2，直到 G 的所有顶点都加入到 S 中为止

Floyd 算法执行的例子

用Floyd算法求任意一对顶点间最短路径

步骤	初态	$k=0$	$K=1$	$K=2$
A	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ 9 & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$
Path	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ 2 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$
S	{}	{ 0 }	{ 0, 1 }	{ 0, 1, 2 }



$$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$$

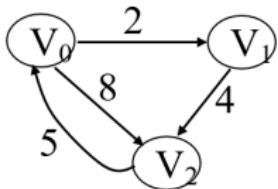
带权有向图及其邻接矩阵

初始化为 $\text{Path}[i][j]=-1$, 表示从 V_i 到 V_j 不经过任何(S 中的中间)顶点。当某个顶点 V_k 加入到 S 中后使 $A[i][j]$ 变小时, 令 $\text{Path}[i][j]=k$

Floyd 算法执行的例子

用Floyd算法求任意一对顶点间最短路径

步骤	初态	$k=0$	$K=1$	$K=2$
A	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ 9 & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$
Path	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ 2 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$
S	{ }	{ 0 }	{ 0, 1 }	{ 0, 1, 2 }



- V_0 到 V_1 ：最短路径是{ 0, 1 }，路径长度是2；
- V_0 到 V_2 ：最短路径是{ 0, 1, 2 }，路径长度是6；
- V_1 到 V_0 ：最短路径是{ 1, 2, 0 }，路径长度是9；
- V_1 到 V_2 ：最短路径是{ 1, 2 }，路径长度是4；
- V_2 到 V_0 ：最短路径是{ 2, 0 }，路径长度是5；
- V_2 到 V_1 ：最短路径是{ 2, 0, 1 }，路径长度是7；

Floyd 算法实现

实现代码

```
1 int A[MAX_VEX][MAX_VEX];
2 int Path[MAX_VEX][MAX_VEX];
3 void Floyd_path (MGraph *G){
4     int j, k, m;
5     for(j=0;j<G->vexnum;j++){
6         for(k=0;k<G->vexnum;k++){
7             A[j][k]=G->adj[j][k];
8             Path[j][k]=-1;
9         }//各数组的初始化
10    for(m=0;m<G->vexnum;m++)
11        for(j=0;j<G->vexnum;j++)
12            for(k=0;k<G->vexnum;k++)
13                if ((A[j][m]+A[m][k])<A[j][k]){
14                    A[j][k]=A[j][m]+A[m][k];
15                    Path[j][k]=m;
16                }//修改数组A和Path的元素值
17    for(j=0;j<G->vexnum;j++)
18        for(k=0;k<G->vexnum;k++)
19            if (j!=k){
20                printf(` `%d到%d的最短路径为:\n", j, k);
21                printf(` `%d",j) ; prn_pass(j, k);
22                printf(` `%d", k);
23                printf(` `最短路径长度为: %d\n",A[j][k]);
24            }
25 } // end of Floyd
26 void prn_pass(int j,int k){
27     if (Path[j][k]!=-1){
28         prn_pass(j,Path[j][k]);
29         printf(` `, %d" ,Path[j][k]);
30         prn_pass(Path[j][k],k);
31     }
32 }
```

解释说明

- 若 $\text{Path}[i][j]=k$: 从 v_i 到 v_j 经过 v_k , 最短路径序列是 $(v_i, \dots, v_k, \dots, v_j)$, 则路径子序列: (v_i, \dots, v_k) 和 (v_k, \dots, v_j) 一定是从 v_i 到 v_k 和从 v_k 到 v_j 的最短路径。
- 从而可以根据 $\text{Path}[i][k]$ 和 $\text{Path}[k][j]$ 的值再找到该路径上所经过的其它顶点, … 依此类推。
- 时间复杂度 $O(n^3)$, 空间复杂度 $O(n^2)$

Floyd 算法练习题

题目说明

- 如图，用 Floyd 算法给出每对顶点之间的最短路径。给出每一步的矩阵 A 和矩阵 Path

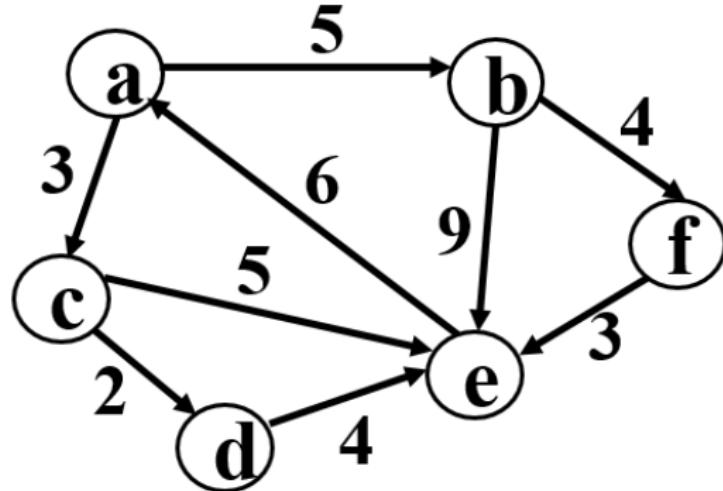


图 带权有向图

练习题

题目说明

- ① 给定一棵树及树上两个节点，求两节点间的距离
- ② 给定一个朋友网络，求两个人之间的（最短）距离
- ③ 给定一个中国象棋棋盘，其中随机散落 k 个棋子，初始时刻，“马”落子在 (x, y) ，求马所有能去的地方
- ④ 分布式死锁检测算法：一个有向图，每个节点只有自己“出边”邻居的位置信息，即节点只能沿出边发信息，如何判断有向图有环？（没有上帝知道图的拓扑结构）
- ⑤ 路由算法：网络中每台路由器是一个节点，节点间互联；假设节点间通信延迟固定不变（边权值），求任何两个路由器之间的通信链路。（没有上帝知道图的拓扑结构）