数据结构——图

主讲：张昱
yuzhang@ustc.edu
0551-3603804

第七章 图

重点：图在邻接矩阵和邻接表上的遍历算法(DFS, BFS)及其应用

难点：求图的最小生成树、最短路径、拓扑排序、关键路径等算法及其应用、性能分析

第七章 图

7.1 图的定义和术语(1)

- 图的定义
  - 图是由顶点集合(vertex)及顶点之间的关系集合组成的一种数据结构。
    记为$G = (V, \{E\})$ V-顶点集，E-关系的集合
  - 顶点之间的关系是多对多的$(m : n)$
  - 顶点之间的关系是否有方向性:
    有向图、无向图

7.1 图的定义和术语(2)

- 无向图
  - 边$(v,w) \in E$ $(v,w \in V)$，v与w互为邻接点，边(v, w)依附于顶点v和w，或者说边(v, w)和顶点v, w相关联。
  - 顶点v 的度是和v相关联的边的数目,记为$TD(v)$ 。
- 有向图
  - 弧$<v,w> \in E$ $(v,w \in V)$，w为弧头, v为弧尾; 顶点v 邻接到顶点w，顶点w 邻接自顶点v，弧$< v, w >$和顶点v、w相关联。

7.1 图的定义和术语(2)

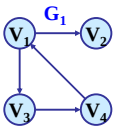
- 有向图
  - 顶点v 的入度是以v 为弧头的弧的数目,记为$ID(v)$;
    v 的出度是以v为弧尾的弧的数目,记为$OD(v)$;
    v 的度是$TD(v) = ID(v) + OD(v)$。
- 路径与连通性

  路径、简单路径、回路(环)、简单回路

  顶点之间的连通性、无向连通图、有向强连通图

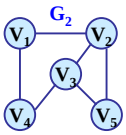
7.1 图的定义和术语(3)

- 对于有向图G_1
  - $V_1V_3V_4V_1V_2$ 是从V_1 到V_2 的路径，不是简单路径；
  - V_1V_2是简单路径；
  - $V_1V_3V_4V_1V_3V_4V_1$是环，不是简单环；
  - $V_1V_3V_4V_1$是简单环。
  - $ID(V_1)=1$, $OD(V_1)=2$, $TD(V_1)=3$
  - V_1和V_4是连通的；从V_1到V_2是连通的，但从V_2到V_1是不连通的。
  - G_1不是强连通图



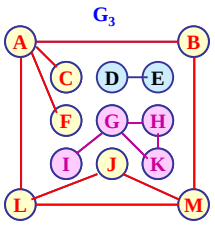
7.1 图的定义和术语(4)

- 对于无向图G_2
  - $V_1V_2V_3V_5V_2$ 是V_1和V_2间的路径，但不是简单路径；
  - V_1V_2是简单路径；
  - $V_2V_3V_5V_2V_3V_5V_2$是环，但不是简单环
  - $V_2V_3V_5V_2$是简单环。
  - $TD(V_1)=2$
  - V_1和V_4是连通的
  - G_2是连通图



7.1 图的定义和术语(5)

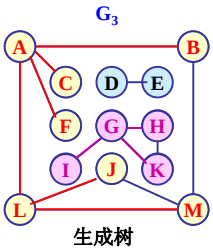
- 子图、连通性
  - 对于G=(V, {E})，它的子图G'满足:
    1) V'⊆ V;
    2) E'⊆ E;
    3) G'=(V', {E'})
  - 连通分量—极大连通子图
  - 强连通分量—有向图中的极大强连通子图



连通分量

7.1 图的定义和术语(6)

- 子图、连通性
  - 生成树—极小连通子图
  - 极大、极小是就边数的多少而言的



生成树

7.1 图的定义和术语(7)

- 边/弧数目

用n表示图中顶点数目，用e表示图中边或弧的数目

- 无向图：$0 \leqslant e \leqslant \frac{1}{2} n(n-1)$
  完全图 $e = \frac{1}{2} n(n-1)$
- 有向图：$0 \leqslant e \leqslant n(n-1)$
  有向完全图 $e = n(n-1)$
- 稀疏图： $e < n\log n$
  稠密图
- 若图中边或弧上有权，则该图称为网
  有向图、有向网、无向图、无向网

7.1 图的定义和术语(8)

- 边/弧数目

思考：若无向图中有21条边，则：

1) 保持该图不连通至少应具有多少个顶点？
2) 保持该图连通至少有多少个顶点，至多有多少个顶点？

1) m个顶点形成完全图，另一个顶点与这m个顶点不连通
   ½ m(m-1)=21 ➔ m=7 ➔ n=m+1=8
2) 至少有 7 个顶点(完全图)，至多有22个顶点(生成树)

7.1 图的定义-ADT Graph

- **ADT Graph**
  - 查找：LocateVex(G, u)
    GetVex(G, v)
    FirstAdjVex(G, v)
    NextAdjVex(G, v, w)
  - 插入：InsertVex(&G, v)
    InsertArc(&G, v, w)
  - 删除：DeleteVex(&G, v)
    DeleteArc(&G, v, w)
  - 遍历：DFSTraverse(G, v, Visit())
    BFSTraverse(G, v, Visit())

ADT Graph{
数据对象 V：V 是具有相同特性的数据元素的集合，称为顶点集。
数据关系 R：
R = {VR}
VR = {< v, w >|v, w∈V 且 P(v, w)，<v,w>表示从 v 到 w 的弧，谓词 P(v,w) 定义了弧<v,w>的意义或信息}
基本操作：
CreateGraph(&G, V, VR)
初始条件：V 是图的顶点集，VR 是图中弧的集合
操作结果：按 V 和 VR 的定义构造图 G
DestroyGraph(&G)
初始条件：图 G 存在
操作结果：销毁图 G
LocateVex(G, u)
初始条件：图 G 已存在，u 和 G 中顶点有相同特征
操作结果：若 G 中存在顶点 u，则返回该顶点在图中位置，否则返回其它信息
GetVex(G, v)
初始条件：图 G 存在，v 是 G 中某个顶点
操作结果：返回 v 的值

PutVex(&G, v, value)
初始条件：图 G 存在，v 是 G 中某个顶点
操作结果：对 v 赋值 value
FirstAdjVex(G, v)
初始条件：图 G 存在，v 是 G 中某个顶点
操作结果：返回 v 的第一个邻接顶点。若顶点在 G 中没有邻接顶点，则返回“空”
NextAdjVex(G, v, w)
初始条件：图 G 存在，v 是 G 中某个顶点，w 是 v 的邻接顶点
操作结果：返回 v 的(相对于 w 的)下一个邻接顶点。若 w 是 v 的最后一个邻接点，则返回“空”
InsertVex(&G, v)
初始条件：图 G 存在，v 和 G 中顶点有相同特征
操作结果：在图中增添新顶点 v
DeleteVex(&G, v)
初始条件：图 G 存在，v 是 G 中某个顶点
操作结果：删除 G 中顶点 v 及其相关的弧

InsertArc(&G, v, w)
初始条件：图 G 存在，v 和 w 是 G 中两个顶点
操作结果：在图 G 中增添弧<v,w>，若 G 是无向的，则还应增添对称弧<w,v>
DeleteArc(&G, v, w)
初始条件：图 G 存在，v 和 w 是 G 中两个顶点
操作结果：删除 G 中的弧<v,w>，若 G 是无向的，则还应删除对称弧
DFSTraverse(G, v, visit( ))
初始条件：图 G 存在，v 是 G 中某个顶点，visit 是对顶点的应用函数
操作结果：从顶点 v 起深度优先遍历图 G，并对每个顶点调用函数 visit( )一次且至多一次。一旦 visit( )失败，则操作失败
BFSTraverse(G, v, visit( ))
初始条件：图 G 存在，v 是 G 中某个顶点，visit 是对顶点的应用函数
操作结果：从顶点 v 起广度优先遍历图 G，并对每个顶点调用函数 visit( )一次且至多一次。一旦 visit( )失败，则操作失败
}ADT Graph

7.2 图的存储结构(1)

- 图的存储表示分析
  - **特点**：顶点之间的关系是多对多($m:n$)
    ∵m 和 n 都是不定的，无法进行非线性结构的线性化
    ∴图中的关系不能通过顺序映像(即通过顶点之间的存储位置反映顶点之间的逻辑关系)反映；必须另外引入存储空间反映顶点之间的邻接关系。
  - **图的存储信息**
    顶点信息、边/弧信息
    整体信息：顶点数、边/弧数、图的种类(有向图/有向网/无向图/无向网)

7.2 图的存储结构(2)

- 图的存储表示分析
  - **顶点集的存储**
    用顺序表存储，不是顺序映像！
  - **关系集的存储**
    邻接矩阵、邻接表、多重邻接表、十字链表

7.2 图的存储结构-邻接矩阵

- 数组表示法——邻接矩阵

```
#define  INFINITY        INT_MAX
typedef  enum{DG, DN, AG, AN} GraphKind;
typedef struct ArcCell{
   // 顶点间的关系，无权图：0-不相邻，1-相邻
   // 有权图：权值，INFINITY-不相邻
   int          adj;
   InfoType     *info;  // 该弧相关信息的指针
}ArcCell,
AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
```


7.2 图的存储结构-邻接矩阵

- 数组表示法——邻接矩阵

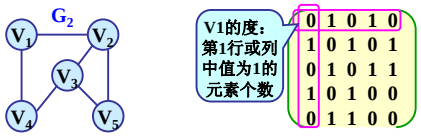
```
typedef struct {
   // 顶点向量
   VertexType  vexs[MAX_VERTEX_NUM];
   // 关系集
   AdjMatrix      arcs;
   int       vexnum, arcnum; // 顶点数、边/弧数
   GraphKind   kind;          // 图的种类
}MGraph;
```


7.2 图的存储结构-邻接矩阵

- 无向图—对称矩阵

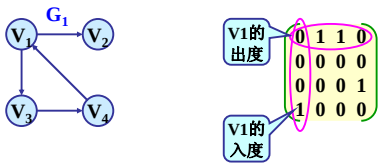
  无权图：邻接矩阵中的元素为0-不邻接；

  邻接矩阵中的元素为1-邻接



7.2 图的存储结构-邻接矩阵

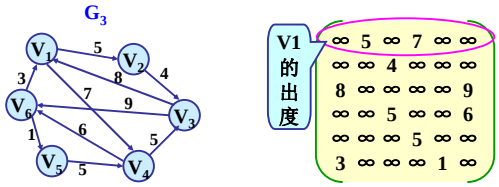
- 有向图—未必是对称矩阵



7.2 图的存储结构-邻接矩阵

- 有向网—未必是对称矩阵 (图7.9, P162)

  网：权值-邻接；极限值-不邻接



7.2 图的存储结构-邻接表

- 邻接表

  无向图/网：边表，表结点的个数为边数的两倍

  有向图/网：出边表，表结点的个数为弧数

```
typedef  struct ArcNode{     // 表结点
   int        adjvex;     // 弧所指向的顶点的位置
   struct ArcNode  *nextarc;// 指向下一条弧的指针
   InfoType       *info;
}ArcNode;
```

7.2 图的存储结构-邻接表

- 邻接表

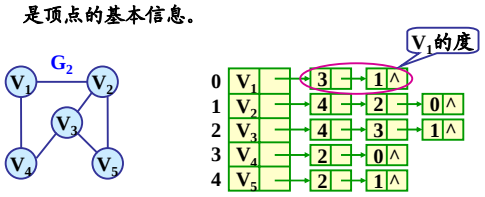
```
typedef struct VNode{        // 头结点
   VertexType   data;        // 顶点信息
   ArcNode      *firstarc;   // 邻接表指针
}VNode, AdjList[MAX_VERTEX_NUM];

typedef struct {
   AdjList      vertices;
   int      vexnum, arcnum;  // 顶点数、边/弧数
   GraphKind   kind;         // 图的种类
}ALGraph;
```


7.2 图的存储结构-邻接表

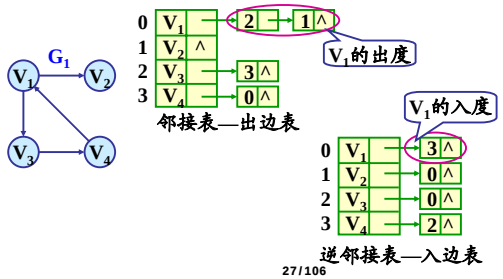
- 无向图/网—边表

注意：表结点存放的是顶点在顶点顺序表中的编号，不是顶点的基本信息。



7.2 图的存储结构-邻接表

- 有向图/网—出边表



7.2 图的存储结构—图的创建

- 图的创建算法
  - 输入图的类型，根据类型选择相应的创建算法
  - 输入图的顶点数，边/弧数
  - 输入并存储顶点信息
  - 输入边/弧所关联的顶点对，将边或弧的信息存储到邻接矩阵/邻接表中

教材中的算法7.1~7.3

图的存储结构不同、图的类型不同，都会影响创建算法的实现细节；但是，图的总体创建流程是一致的！

- 邻接矩阵与邻接表的对比

假设图为 G，顶点数为 n，边/弧数为 e。

	邻接矩阵	邻接表
存储空间	$O(n+n^2)$	$O(n+e)$
图的创建算法	$T1(n)=O(e+n^2)$或 $T2(n)=O(e^*n+n^2)$	$T1(n)=O(n+e)$或 $T2(n)=O(e^*n)$
	T1(n)是指在输入边/弧时，输入的顶点信息为顶点的编号；而 T2(n)则指在输入边/弧时，输入的为顶点本身的信息，此时需要查找顶点在图中的位置	
无向图中求第 i 顶点的度	$\sum_{j=0}^{n-1} G.arcs[i][j].adj$ (第 i 行之和)或 $\sum_{j=0}^{n-1} G.arcs[j][i].adj$ (第 i 列之和)	G.vertices[i].firstarc 所指向的邻接表包含的结点个数
无向网中求第 i 顶点的度	第 i 行/列中 adj 值不为 INFINITY 的元素个数	

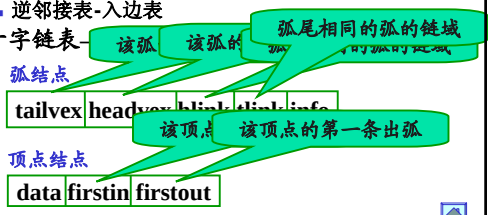
	邻接矩阵	邻接表
有向图中求第 i 顶点的入/出度	入度：$\sum_{j=0}^{n-1} G.arcs[j][i].adj$ (第 i 列) 出度：$\sum_{j=0}^{n-1} G.arcs[i][j].adj$ (第 i 行)	入度：扫描各顶点的邻接表，统计表结点的 adjvex 为 i 的表结点个数 T(n)=O(n+e) 出度：G.vertices[i].firstarc 所指向的邻接表包含的结点个数
有向网中求第 i 顶点的入/出度	入度：第 i 列中 adj 值不为 INFINITY 的元素个数 出度：第 i 行中 adj 值不为 INFINITY 的元素个数	

	邻接矩阵	邻接表
统计边/弧数	无向图：$\frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}G.arcs[i][j].adj$ 无向网：Garcs 中 adj 值不为 INFINITY 的元素个数的一半 有向图：$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}G.arcs[i][j].adj$ 有向网：Garcs 中 adj 值不为 INFINITY 的元素个数	无向图/网：图中表结点数目的一半 有向图/网：图中表结点的数目

结论：邻接矩阵适于稠密图的存储，邻接表适于稀疏图的存储；
邻接表求有向图的顶点的入度不方便，要遍历各个顶点的邻接表。

7.2 图的存储结构-十字链表

- 十字链表—有向图 (P165)
  - 有向图
    - 邻接表-出边表,求入度不方便
    - 逆邻接表-入边表
  - 十字链表-



7.2 图的存储结构-十字链表

- 十字链表—有向图

```
#define MAX_VERTEX_NUM  20
// 弧结点
typedef struct ArcBox{
   int  tailvex, headvex;
   struct ArcBox  *hlink, *tlink;
   InfoType  *info;
}ArcBox;
```

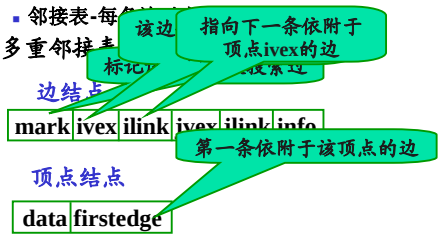

7.2 图的存储结构-十字链表

- 十字链表—有向图

```
//顶点结点
typedef struct VexNode{
   VertexType  data;
   ArcBox  *firstin, *firstout;
}VexNode;
// 十字链表
typedef struct{
   VexNode  xlist[MAX_VERTEX_NUM];
   int  vexnum, arcnum;
}OLGraph;
```

7.2 图的存储结构-邻接多重表

- 邻接多重表—无向图 (P166)
  - 无向图
    - 邻接表-每
  - 多重邻接表



7.2 图的存储结构-邻接多重表

- 邻接多重表—无向图

```
#define MAX_VERTEX_NUM  20
typedef enum { unvisited, visited}  VisitIf;
// 边结点
typedef struct EBox{
   VisitIf  mark;
   int  ivex, jvex;
   struct EBox  *ilink, *jlink;
   InfoType  *info;
}EBox;
```

7.2 图的存储结构-邻接多重表

- 邻接多重表—无向图

```
// 顶点结点
typedef struct VexBox{
   VertexType  data;
   EBox  *firstedge;
}VexBox;
//邻接多重表
typedef struct{
   VexBox  adjmulist[MAX_VERTEX_NUM];
   int  vexnum, edgenum;
}AMLGraph;
```

7.3 图的遍历(1)

- 图的遍历

  顶点之间的邻接关系 m : n

  引入访问标志数组visited[0..n-1]，防止顶点多次被访问

- 图的遍历种类
  - 深度优先遍历：→ 树的先序遍历

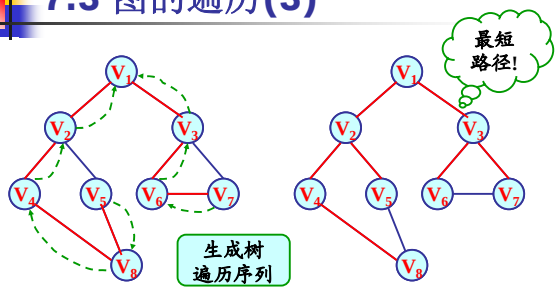
    从某顶点*v* 出发，访问该顶点，然后依次从*v* 的未被访问的邻接点出发深度优先遍历图，直至图中所有和*v* 有路径相通的顶点都被访问到；

7.3 图的遍历(2)

- 图的遍历种类
  - 广度优先遍历：→ 树的层次遍历

    从某顶点*v* 出发，访问该顶点，然后依次访问*v* 的所有未曾访问过的邻接点，再分别从这些邻接点出发依次访问它们的邻接点，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，直至图中所有已被访问的顶点的邻接点都被访问到；
  - 若图中还存在尚未访问过的顶点，则另选图中一个未曾被访问的顶点作起始点，继续重复上述过程

7.3 图的遍历(3)



深度优先搜索(DFS)

$V_1\ V_2\ V_4\ V_8\ V_5\ V_3\ V_6\ V_7$

广度优先搜索(BFS)

$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$

7.3 图的遍历(4)

- DFSTraverse(G)
  - 基于ADT Graph
  - 基于某种存储结构：邻接矩阵/邻接表/十字链表/邻接多重表
- BFSTraverse(G)
  - 基于ADT Graph
  - 基于某种存储结构：邻接矩阵/邻接表/十字链表/邻接多重表

基于ADT Graph的DFS算法(算法7.4和7.5)

```
Boolean  visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph G, Status ( *Visit ) (Graph G, int v) )
{
   for ( v = 0; v < G.vexnum; ++v )  visited[v] = FALSE;
   for ( v = 0; v < G.vexnum; ++v )
      if ( !visited[v] )
         DFS(G, v, Visit);
}

void DFS(Graph G, int v, Status ( *Visit ) (Graph G, int v) )
{
   visited[v] = TRUE;  Visit(G, v);
   for ( w = FirstAdjVex(G, v); w; w = NextAdjVex(G, v, w) )
      if ( !visited[w] )
         DFS(G, w, Visit);
}
```

基于某种存储结构的DFS算法

根据选择的存储结构，决定 FirstAdjVex( )和NextAdjVex( )的实现，重新整合算法。

如采用邻接矩阵表示法表示的图，则DFS算法如下：

```
void DFS (MGraph G, int v,
          Status ( *Visit ) (MGraph G, int v)){
   visited[v] = TRUE;  Visit(G, v);
   for ( w = 0; w < G.vexnum; w ++)
      if ( G.arcs[v][w].adj && !visited[w] )
         DFS(G, w, Visit );
}
```

$T(n)=O(n^2)$

基于某种存储结构的DFS算法

根据选择的存储结构，决定 FirstAdjVex( )和NextAdjVex( )的实现，重新整合算法。

如采用邻接表表示法表示的图，则DFS算法如下：

```
void DFS(ALGraph G, int v,
         Status ( *Visit )(ALGraph G, int v) ){
   visited[v] = TRUE;  Visit(G, v);
   for ( p = G.vertices[v].firstarc; p!=NULL ;
         p = p->nextarc)
      if (!visited[p->adjvex] )
         DFS(G, p->adjvex, Visit);
}
```

$T(n)=O(n+e)$

基于ADT Graph的BFS算法(算法7.6)

```
void BFSTraverse (Graph G,
                  Status ( *Visit ) (Graph G, int v) )
{
   for ( v = 0; v < G.vexnum; ++v )
      visited[v] = FALSE;
   InitQueue (Q);
   for ( v = 0; v < G.vexnum; ++v )
      if ( !visited[v] ){
         /* 访问某连通分量的起始顶点，起点入队 */
         visited[v] = TRUE;  Visit(G, v);
         EnQueue(Q, v);
         while( !QueueEmpty(Q) ){
```

基于ADT Graph的BFS算法(算法7.6)

```
            /* 出队，访问出队元素的邻接点 */
            DeQueue(Q,u);
            for ( w = FirstAdjVex( G, u) ; w ;
                  w = NextAdjVex(G, u, w) )
               if ( !visited[w] ){
                  /* 访问u的未访问的邻接点并入队 */
                  visited[w] = TRUE; Visit(G, w);
                  EnQueue(Q, w);
               } //end of if
            } //end of while
         } //end of if
} //end of BFSTraverse
```


BFS算法--采用邻接矩阵表示法表示的网

```
void BFSTraverse (MGraph G,
                  Status ( *Visit ) (MGraph G, int v) )
{
   for ( v = 0; v < G.vexnum; ++v )
      visited[v] = FALSE;
   InitQueue (Q);
   for ( v = 0; v < G.vexnum; ++v )
      if ( !visited[v] ){
         /* 访问某连通分量的起始顶点，起点入队 */
         visited[v] = TRUE;  Visit(G, v);
         EnQueue(Q, v);
         while( !QueueEmpty(Q) ){
```

BFS算法--采用邻接矩阵表示法表示的网

```
            /* 出队，访问出队元素的邻接点 */
            DeQueue(Q,u);
            for ( w = 0 ; w < G.vexnum; w++ )
               if ( G.arcs[u][w].adj != INFINITY
                  && !visited[w] ){
                  /* 访问u的未访问的邻接点并入队 */
                  visited[w] = TRUE; Visit(G, w);
                  EnQueue(Q, w);
               } //end of if
            } //end of while
         } //end of if
} //end of BFSTraverse
```

$T(n)=O(n^2)$

BFS算法--采用邻接表表示法表示的网

```
void BFSTraverse (ALGraph G,
                  Status ( *Visit ) (ALGraph G, int v) )
{
   for ( v = 0; v < G.vexnum; ++v )
      visited[v] = FALSE;
   InitQueue (Q);
   for ( v = 0; v < G.vexnum; ++v )
      if ( !visited[v] ){
         /* 访问某连通分量的起始顶点，起点入队 */
         visited[v] = TRUE;  Visit(G, v);
         EnQueue(Q, v);
         while( !QueueEmpty(Q) ){
```


BFS算法--采用邻接表表示法表示的网

```
            /* 出队，访问出队元素的邻接点 */
            DeQueue(Q,u);
            for (p = G.vertices[u].firstarc ; p!=NULL;
                 p = p->nextarc )
               if (!visited[p->adjvex] ){
                  /* 访问u的未访问的邻接点并入队 */
                  visited[p->adjvex ] = TRUE;
                  Visit(G, p->adjvex );
                  EnQueue(Q, p->adjvex );
               } //end of if
         } //end of while
      } //end of if
   } //end of BFSTraverse
```

T(n)=O(n+e)

7.3 图的遍历—非递归的DFS算法

- 思路：DFS(G, v, Visit)
  - G为要遍历的图，v为遍历的起始顶点；
  - 在访问v后，需要顺着v的各个未访问过的邻接点深度优先遍历——形成深度优先生成树，v可能存在多棵子树。
  - 问题：由v找到一个未访问过的邻接点w后，顺着w深度优先遍历后，如何回到原先的v，找v的下一个未访问的邻接点？

7.3 图的遍历—非递归的DFS算法

- 设置栈保存v和w的相关信息
  - 在顺着w深度优先遍历之前，进行现场保护；
  - 在顺着w深度优先遍历结束后，进行现场恢复
- 栈帧的设计
  - 邻接矩阵：保存顶点v和w的编号
  - 邻接表：保存顶点v的编号和邻接点w在v的邻接表中对应的表结点(或其后继)的指针。


```
void   DFS( Graph G, int v,
           Status ( *Visit ) (ALGraph G, int v) ){
   InitStack(S);
   Visit(G, v); visited[v]= TRUE;
   push(v, -1, S);  // 假设顶点编号是从0开始
   while ( !StackEmpty(S)) {
      pop(S, v, w);
      if (w==-1)   w = FirstAdjvex(G, v);
      else     w = NextAdjvex(G, v, w);
      if ( w!=-1){     push(v, w, S);
         if ( !visited[w] ){
            Visit(G, w); visited[w]= TRUE;
            push(w, -1, S);
      } }
   }
}
```


7.3 图的遍历—非递归的DFS算法

- 对上页算法的评价
  - 针对每个顶点v和其邻接点w,如果w已经访问过,则也会入栈
    ➔一些入栈和出栈操作是不必要的，这会引起时间的开销
- 思考
  - 如何修改算法，使得只将v和未访问过的邻接点w入栈？

7.3 图的遍历—遍历算法的应用(1)

- 图的深度优先遍历
  - 求一条包含图中所有顶点的简单路径(简单回路)
  - 判断图中是否存在环
  - 求图中通过给定顶点vk的简单回路
  - 判断是否存在从顶点vi到顶点vj的路径
  - 判别v0和v1之间是否存在一条长度为k的路径

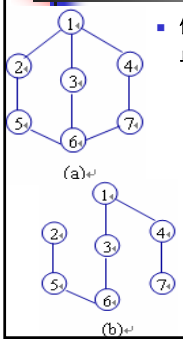
7.3 图的遍历—遍历算法的应用(2)

- 图的广度优先遍历
  - 判断是否存在从顶点vi到顶点vj的路径
  - 求距v0的各顶点中最短路径长度最长的一个顶点
  - 求v0和v1之间的最短路径.
  - 在顶点子集U中找出距离顶点v0最近的顶点
  - 求顶点v0到其余每个顶点的最短路径
  - 求距离顶点v0的最短路径长度为k的所有顶点

7.3 图的遍历—遍历算法的应用(3)

- 例1：求一条包含图中所有顶点的简单路径(简单回路)
  - 思路
    - 对于任意的有向图或无向图G，并不一定都能找到符合题意的简单路径。它的查找可以基于深度优先遍历。
    - 在一个存在包含全部顶点的简单路径的图中，起点的选择和顶点的邻接点次序会影响该简单路径是否能顺利地查到。

7.3 图的遍历—遍历算法的应用(4)



- 例1：求一条包含图中所有顶点的简单路径(简单回路)
  - 1) 起点的选择：如图(a)，其符合题意的一条简单路径如图(b)。若起点为1，则不能找到符合题意的简单路径；
    2) 顶点的邻接点次序：进一步考察图(a)，即使以2为起点，但是2的邻接点选择的是1，而不是5，此时也不能找到符合题意的解。

7.3 图的遍历—遍历算法的应用(5)

- 在基于DFS的查找算法中，由于起点和邻接点的选取与顶点和邻接点的存储次序以及算法的搜索次序有关，不可能依据特定的图给出特定的解决算法。因此，在整个搜索中应允许有查找失败，此时可采取回溯到上一层的方法，继续查找其他路径。
- 这样，引入数组Path用来保存当前已搜索的简单路径上的顶点，引入计数器n用来记录当前该路径上的顶点数。

7.3 图的遍历—遍历算法的应用(6)

- 例1：求一条包含图中所有顶点的简单路径(简单回路)
  - 对DFS算法的修改
    - 初始化计数器n，放在visited的初始化前或后；
    - 访问顶点时，增加将该顶点序号加入到数组Path中，计数器n++；
      判断是否已获得所求路径，是则输出结束，否则继续遍历邻接点；
    - 某顶点的全部邻接点都访问后，仍未得到简单路径，则回溯，将该顶点置为未访问，计数器n--。

```
// 邻接矩阵表示法
// 红色字部分为在DFS上增加或修改的步骤
void  Hamilton(MGraph G)
{
  for ( i=0; i<G.vexnum; i++ )
    visited[i] = FALSE;
  n = 0;
  for ( i=0; i<G.vexnum; i++ )
    if ( !visited[i] ) DFS (G, i);
}
```



```
void  DFS(MGraph G, int i)
{
  visited[i] = TRUE;
  Path[n] = i;
  n++;
  if ( n == G.vexnum )  Print(Path); //输出简单路径
  for( j=0; j<G.vexnum; j++ )
    if ( G.arcs[i][j].adj && !visited[j])
      DFS( G, j );
  visited[i] = FALSE;
  n--;
}
```


7.3 图的遍历—遍历算法的应用(7)

- 思考
  - 若图中存在多条符合条件的路径，本算法是输出一条，还是输出全部？
  - 如何修改算法，变成判断是否有包含全部顶点的简单路径？
  - 如何修改算法，输出所有包含全部顶点的简单路径的条数？
  - 如何修改算法，使得能通过变参或全局变量将所有包含全部顶点的简单路径带回给调用者？

7.3 图的遍历—遍历算法的应用(8)

- 思考(续)
  - 如何修改算法，输出所有的包含全部顶点的简单回路？
  - 如何修改算法，输出所有包含全部顶点的简单回路的数目？
  - 考虑其他图的存储方法和图的种类对算法的影响；
  - 考虑在非递归的深度优先遍历算法上，如何修改使之适应本问题的求解。

7.3 图的遍历—遍历算法的应用(9)

- 例2：求距v0的各顶点中最短路径长度最长的一个顶点
  - 思路
    - 由于题意强调为最短路径，因此应当考虑BFS的算法应用
    - 本问题的求解转变成：从v0出发进行BFS，最后一层的顶点距离v0的最短路径长度最长。
    - BFS遍历中最后一层的顶点一定是最后出队的若干顶点，队列中最后一个出队的顶点必定是符合题意的顶点。
      只需调用BFS的算法，将最后出队的元素返回即可。


```
int  MaxDistance (MGraph G, int v0 )
{
  /* 初始化各顶点的访问标志，设置为未访问 */
  for ( i=0; i<G.vexnum; i++ )
    visited[i] = FALSE;
  InitQueue(Q);
  // 不需要考虑其他的连通分量
  // 因为所求的顶点必定与v0在同一个连通分量中
  EnQueue (Q, v0 );
  visited[v0] = TRUE;
```

```
    while( !QueueEmpty(Q) ){
      DeQueue(Q, v);
      for( w=0; w<G.vexnum; w++ )
        if ( G.arcs[v][w].adj && !visited[w] ){
          visited[w] = TRUE;
          EnQueue(Q, w);
        }
    }
    return(v);
}
```


7.3 图的遍历-遍历算法的应用(10)

- 例2：求距v0的各顶点中最短路径长度最长的一个顶点
  - 思考
    - 若要求输出满足条件的全部顶点，则如何修改上述算法；
    - 如何修改算法，变成输出最长的最短路径的长度？
    - 如何修改算法，输出一条满足条件的路径？
    - 如何修改算法，输出所有满足条件的路径？
    - 如何修改算法，考虑其他图的存储方法对算法的影响.

7.4 图的连通性问题

7.4.1 无向图的连通分量和生成树(1)

- 基本概念
  - 连通分量的顶点集：即从该连通分量的某一顶点出发进行搜索所得到的顶点访问序列；
  - 生成树：某连通分量的极小连通子图
  - 深度优先搜索生成树、广度优先搜索生成树：由该连通分量的顶点集和搜索所经过的边组成
  - 生成森林：非连通图的各个连通分量的极小连通子图构成的集合

7.4.1 无向图的连通分量和生成树(2)

- 深度/广度优先生成树/森林的创建
  - 算法思路
    - 基于图的深度/广度优先遍历算法
    - 将访问顶点变成创建生成树的根结点或增加一个结点到生成树中的操作
    - 生成树/森林的存储表示采用孩子-兄弟表示法

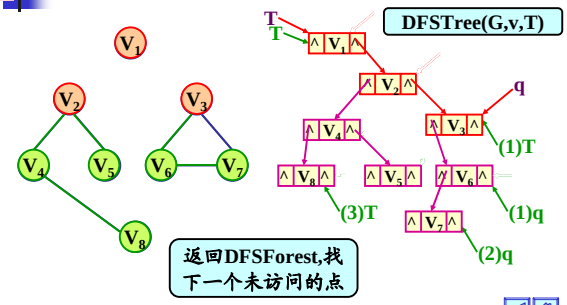
基于图的抽象数据类型的算法(算法7.7)

```
void DFSForest(Graph G, CSTree &T){
  T = NULL;
  for ( v=0; v<G.vexnum; ++v )  visited[v] = FALSE;
  for ( v=0; v<G.vexnum; ++v )
    if ( !visited[v] ){
      p = ( CSTree ) malloc(sizeof(CSNode));
      *p = {GetVex(G, v), NULL, NULL};
      if ( T == NULL ) T = p; // 第一棵生成树的根
      else  q->nextsibling = p; // 其它生成树的根
      q = p; // q指示当前生成树的根
      DFSTree(G, v, p); // 建立以*p为根结点的生成树
    }
}
```

基于图的抽象数据类型的算法(算法7.8)

```
void DFSTree(Graph G, int v, CSTree & T ){
  visited[v] = TRUE;
  first = TRUE;
  for ( w = FirstAdjVex( G, v); w; w = NextAdjVex(G, v, w) )
    if ( !visited[w] ){
      p = ( CSTree ) malloc(sizeof(CSNode));// 分配孩子结点
      *p = {GetVex(G, w), NULL, NULL};
      if ( first ){     // w是v的第一个未访问的邻接点
        T->firstchild = p; first = FALSE; //是根的第一个孩子
      } else
        q->nextsibling = p;          // 是上一邻接点的右兄弟结点
      q = p; // 修改上一邻接点的指针
      DFSTree(G, w, q);
    }
}
```


7.4 图的连通性-DFS生成森林



基于图的抽象数据类型的算法

```
void BFSForest(Graph G, CSTree &T){
  T = NULL;
  for ( v=0; v<G.vexnum; ++v)  visited[v] = FALSE;
  InitQueue(Q);
  for ( v=0; v<G.vexnum; ++v)
    if ( !visited[v] ){
      // 访问某连通分量的起始顶点，起点入队
      visited[v] = TRUE;
      p = ( CSTree ) malloc(sizeof(CSNode));
      *p = {GetVex(G, v), NULL, NULL};
      if ( T==NULL ) T = p;// 第一棵生成树的根(T的根)
      else q->nextsibling = p;// 其它生成树的根(前一棵的“兄弟”)
      q = p;                  // q指示当前生成树的根
      EnQueue(Q, v, p);
      while( !QueueEmpty(Q) ){
        DeQueue(Q,u, q); // 出队，访问出队元素的邻接点
```


基于图的抽象数据类型的算法

```
        first = TRUE;
        for ( w = FirstAdjVex( G, u) ; w ; w = NextAdjVex(G, u, w))
          if ( !visited[w] ){
            /* 访问顶点u的尚未访问的邻接点并入队 */
            visited[w] = TRUE;
            p = ( CSTree ) malloc(sizeof(CSNode));//分配孩子结点
            *p = {GetVex(G, w), NULL, NULL};
            if ( first ){     // w是v的第一个未访问的邻接点
              q->firstchild = p; first = FALSE;//根的第一个孩子
            } else {
              q->nextsibling = p;          // 是上一邻接点的右兄弟结
            }
            q = p;          // 修改上一邻接点的指针
            EnQueue(Q, w, p);
          }
      }
    }
}
```

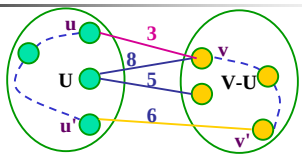

7.4.2 有向图的强连通分量

- 有向图的强连通分量
  - 从某顶点出发沿以该顶点为尾的弧进行DFS，按其所有邻接点的搜索都完成(退出DFS)的顺序将顶点依次存储到数组finished中
  - 从最后存储到数组finished中的顶点出发，沿着以该顶点为头的弧作逆向DFS，能访问到的顶点在一个强连通分量中
  - 利用DFS遍历求强连通分量的时间复杂度和DFS遍历相同

7.4.3 最小生成树(1)

- 最小生成树(Minimum Cost Spanning Tree)
  - 问题：如何构造无向连通网的最小代价生成树？
    一棵生成树的代价就是树上各边的代价之和.
  - MST性质：设N=(V,{E})为一连通网，U⊆V(U≠Ø), 若(u,v)是一条具有最小权值的边，u∈U,v∈V-U,则必存在一棵包含边(u,v)的最小生成树。

7.4.3 最小生成树(2)



如上图所示，跨越U和V-U的边有4条，假设(u,v)不在最小生成树中，则是另外3条边之一，如(u',v')在最小生成树中.

在最小生成树中，u和u'是连通的，v和v'是连通的，则若将(u,v)加入生成树中，会形成环。针对该环去掉(u',v'),会得到权值之和更小的生成树!

7.4.3 最小生成树(3)

最小生成树不断壮大的过程

- 普里姆算法(Prim)算法7.9,P175

初始：最小生成树仅包括一个顶点 $U=\{u_0\}$，无边TE = {}；

循环处理：从V-U中选择一个顶点v_0

选择条件：

$(u, v)\in$

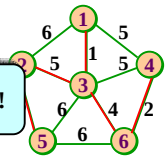
边(u', v_0

v_0加入到U，(u', v_0)加入到TE

循环结束条件：V = U

$T(n,e)=O(n^2)$ 适用于稠密图

最小生成树不一定唯一!



普里姆算法

7.4.3 最小生成树(4)

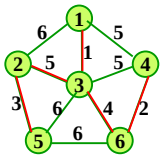
连通分量不断合并的过程

- 克鲁斯卡尔算法(Kruskal)
  - 初始：n个顶点组成n个连通分量

    循环处理：在两连通分量中选择一条边(u', v')

    选择条件：权值最小，连接两连通分量

    循环结束条件：一个连通分量
  - T(n,e)=O(elog(e)) 适用于稀疏图



克鲁斯卡尔算法

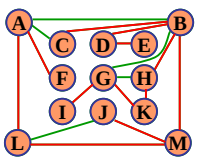
7.4.4 关节点和重连通分量(1)

- 关节点(割点)v：删去顶点v及v相关联的各个边→将图的一个连通分量分割成两个或两个以上的连通分量。
- 重连通图：没有关节点的连通图。

  任意一对顶点之间至少存在两条路径。
- 连通度：若在连通图上至少删去k个顶点才能破坏图的连通性，则称此图的连通度为k。

7.4.4 关节点和重连通分量(2)

- 利用深度优先搜索可求得图的关节点，判别图是否是重连通图。

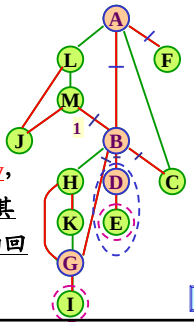


7.4.4 关节点和重连通分量(3)

- 关节点的特性

若生成树的根有两棵或两棵以上的子树，则此根顶点必是关节点。

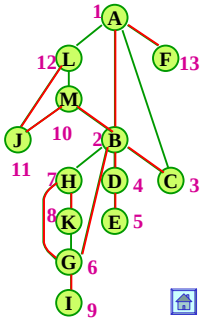
若生成树中某个非叶子顶点v，其某棵子树的根和子树中的其它顶点均没有指向v的祖先的回边，则v为关节点。



7.4.4 关节点和重连通分量(4)

- 改造DFS算法求关节点
  - visited[v]:
    DFS遍历连通图时访问顶点v的次序号;

visited[v]=TRUE
→ visited[v]=++count;

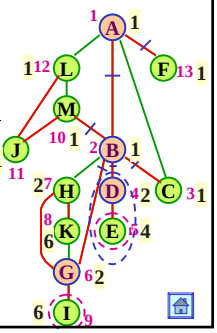


7.4.4 关节点和重连通分量(5)

- 改造DFS算法求关节点
  low[v] = min(visited[v],
  low[w], visited[k])
  w是顶点v在深度优先生成树上的孩子

对生成树后序遍历求low[v]，
v在后序序列中的次序和遍历时退出DFS函数的次序相同

结点到v的祖先的边所关联的祖先的最小次序号（当这样的边存在时）



7.4.4 关节点和重连通分量(6)

- 改造DFS算法求关节点(算法7.10和7.11)
  - visited[v]: DFS遍历连通图时访问顶点v的次序号;
    visited[v]=TRUE → visited[v]=++count;
  - low[v] = min(visited[v], low[w], visited[k])
    low[v]: 生成树中以v为根的子树中的结点到v的祖先的边所关联的祖先的最小次序号（当这样的边存在时）

对生成树后序遍历求low[v],
v在后序序列中的次序和遍历时退出DFS函数的次序相同

  - 若对于某顶点v，存在孩子结点w且
    low[w]≥visited[v]，则顶点v必为关节点

7.5 有向无环图及其应用

- 有向无环图，简称DAG图
  - DAG是描述含有公共子式的表达式的有效工具
    ((a+b)*(b*(c+d))+(c+d)*e)*((c+d)*e)
    二叉树→DAG：共享相同子式，节省存储空间；相同子式的计算次数变为1。
  - 如何检查一个无向图是否存在环？
    若深度优先搜索中遇到回边(指向已访问过的顶点的边)，则有环。

7.5 有向无环图及其应用

- 有向无环图，简称DAG图
  - 如何检查一个有向图是否存在环？
    从某顶点v出发，DFS(v)结束前出现一条从顶点u到顶点v的回边，则有环。
  - DAG也是描述一项工程或系统的进行过程的有效工具：
    顶点-子工程/活动，弧-子工程间的约束

7.5 有向无环图及其应用

7.5.1 拓扑排序

7.5.2 关键路径

7.5.1 拓扑排序(1)

- **问题描述**：由集合上的一个偏序得到该集合上的一个全序
  - **偏序**：集合X上的关系R是自反的、反对称的和传递的

部分成员具有可比性

自反的：$\forall x \in X, <x, x> \in R$

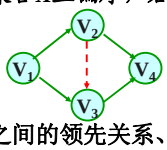
反对称的：$\forall x, y \in X$，若 $<x, y> \in R$，则 $<y, x> \notin R$

传递的：$\forall x, y, z \in X$，若 $<x, y> \in R$且 $<y, z> \in R$，则 $<x, z> \in R$

7.5.1 拓扑排序(2)

- **全序**：设R是集合X上偏序，若$\forall x, y \in X$必有 $x\ R\ y$ 或 $y\ R\ x$

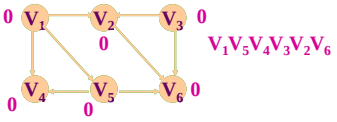
全体成员均可比较



- 应用: 子工程之间的领先关系、课程之间的优先关系
- **AOV-网(Activity on vertex network)** :不应该有环

若从顶点i到顶点j有一条有向路径，则i是j的前驱，j是i的后继。

7.5.1 拓扑排序(3)



$V_1V_5V_4V_3V_2V_6$

- 如何进行拓扑排序？
  - 在有向图中选一个没有前驱的顶点且输出之；
  - 从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出，或者图中不存在无前驱的顶点为止

7.5.1 拓扑排序(4)

- 算法7.12，P182

indegree[0..G.vexnum-1]

表/栈/队列：保存入度已为0且未输出的顶点

T(n,e)=O(n+e) //采用邻接表存储

可判断有向图中是否存在环

7.5.1 拓扑排序(5)

- 若有向图中无环，则进行深度优先遍历，最先退出DFS函数的顶点即出度为零的顶点，是拓扑有序序列中最后一个顶点。

——逆拓扑有序序列

- 若有向图的邻接矩阵为三角矩阵，则该有向图一定存在有拓扑有序序列（一个或多个）。

7.5.2 关键路径(1)

- **AOE-网(Activity on edge)**

—带权的有向无环图

顶点表示事件，弧表示活动，权表示活动持续的时间

——估算工程的完成时间

正常情况下，网中只有一个入度为零的点(源点)和一个出度为零的点(汇点)

7.5.2 关键路径(2)

- 问题：完成整项工程至少需要多少时间？哪些活动是影响工程进度的关键？

  最短时间：从源点到汇点的最长路径的长度。

  （此处的路径长度是指路径上各活动持续时间之和）

  关键路径：路径长度最长的路径。

7.5.2 关键路径(3)

- 设源点为v_1，从v_1到v_i的最长路径长度叫做事件v_i的最早发生时间。

  这个时间决定了所有以v_i为尾的弧表示的活动的最早开始时间。

- 活动a_i的最早开始时间为$e(i)$，最迟开始时间为$l(i)$
- 关键活动a_i：$e(i)=l(i)$

7.5.2 关键路径(4)

- 若事件j的最早发生时间为$ve(j)$，最迟发生时间为$vl(j)$

  活动a_i由弧$<j, k>$表示，其持续时间记为$dut(<j, k>)$，则有：

  $e(i)=ve(j)$

  $l(i)=vl(k)-dut(<j, k>)$

  $ve(0)=0, ve(k)=\max_j\{ve(j)+dut(<j, k>)\}$

  —拓扑有序

  $vl(n-1)=ve(n-1), vl(j)=\min_k\{vl(k)-dut(<j, k>)\}$

  —逆拓扑有序

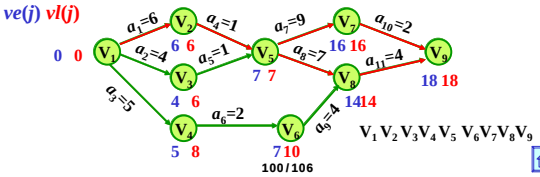
7.5.2 关键路径(5)

- $e(i)=ve(j)$

  $l(i)=vl(k)-dut(<j, k>)$

  $ve(0)=0, ve(k)=\max_j\{ve(j)+dut(<j, k>)\}$--拓扑有序

  $vl(n-1)=ve(n-1), vl(j)=\min_k\{vl(k)-dut(<j, k>)\}$ –逆拓扑有序



7.5.2 关键路径(6)

- P185，算法7.13—修改拓扑排序算法

  1)保存拓扑有序序列T;

  2)计算ve[j]

  栈

  T(n,e)=O(n+e)—图用邻接表表示

  算法7.14—求关键路径算法

  1)调用算法7.13;

  2)依次从T中取元素(逆拓扑有序序列)，计算vl[j]

  3)计算每一活动的最早发生时间和最迟发生时间，输出关键活动

  T(n,e)=O(n+e) —图用邻接表表示

7.6 最短路径(1)

- 无权图的最短路径—广度优先搜索
- 带权有向图的最短路径
  - 一些定义
    - 路径长度：路径上边的权值之和
    - 源点：路径上的第一个顶点
    - 终点：路径上的最后一个顶点
  - 单源最短路径：从某个源点到其余各顶点的最短路径——Dijkstra算法

7.6 最短路径(2)

- Dijkstra算法：按路径长度递增的次序产生最短路径(假设图用数组表示法表示)
  - D[i]：当前所找到的从起始点v (设v为顶点编号)到每个终点v_i的最短路径的长度
  - S：已求得最短路径的终点的集合

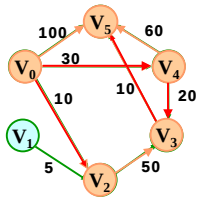
  初态：D[i] = arcs[v][i]

  选择v_j： D[j]=Min{D[i] | $v_i \in$ V-S }, S = S∪{j}

  修改： D[k]=D[j]+arcs[j][k] if D[j]+arcs[j][k] < D[k]
- P189, 算法7.15　　T(n,e) = O(n^2)

7.6 最短路径(3)

- Dijkstra算法：按路径长度递增的次序产生最短路径



始点	终点	D[i]	
V_0	V_1	∞	
	V_2	10	(V_0, V_2)
	V_3	50	(V_0, V_4, V_3)
	V_4	30	(V_0, V_4)
	V_5	60	(V_0, V_4, V_3, V_5)

7.6 最短路径(4)

- 任意一对顶点之间的最短路径
  - 依次以每个顶点作为源点，重复执行Dijkstra算法n次

    T(n,e) = O(n^3)
  - Floyd算法：

    欲求v_i和v_j之间的最短路径，依次使得中间顶点序号不大于k (k=0, 1, …, n-1) 的最短路径

7.6 最短路径(5)

- 任意一对顶点之间的最短路径
  - 定义n阶方阵序列：

    $D^{(-1)}, D^{(0)}, D^{(1)}, \dots D^{(k)}, \dots, D^{(n-1)}$

    $D^{(-1)}[i][j]=arcs[i][j]$

    $D^{(k)}[i][j]=Min\{ D^{(k-1)}[i][j],$

    $D^{(k-1)}[i][k]+ D^{(k-1)}[k][j]\}$ $0 \leqslant k \leqslant n-1$

    算法 7.16　　T(n, e)=O(n^3) 形式上简单些