图

▼ 基本概念

完全图：n个顶点的无向图有 $\frac{n(n-1)}{2}$ 条边，n个顶点的有向图有 $n(n-1)$ 条边

邻接（顶）点：（u，v）or <u，v>是一条边，则u与v互为邻接顶点

顶点的度：顶点v相关联的边的条数

入度：以v为终点的有向边条数

出度：以v为起点的有向边条数

有向图中，入度=出度

子图

权：边相关的数。带权图称为“网”

路径：顶点序列 $(v_i v_{p1} v_{p2} ... v_{pm} v_j)$ 为 v_i 到 v_j 的路径

路径长度：路径上各边权之和

简单路径：路径上各顶点不重复

回路（环）：第一个顶点与最后一个顶点重合

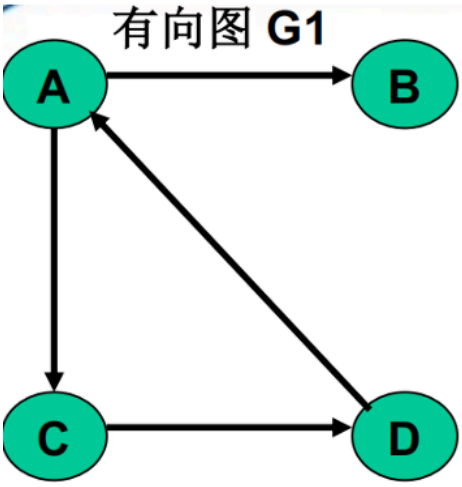
连通图：无向图中， v_1 到 v_2 有路径，则 v_1 和 v_2 连通。所有顶点都连通则为**连通图**

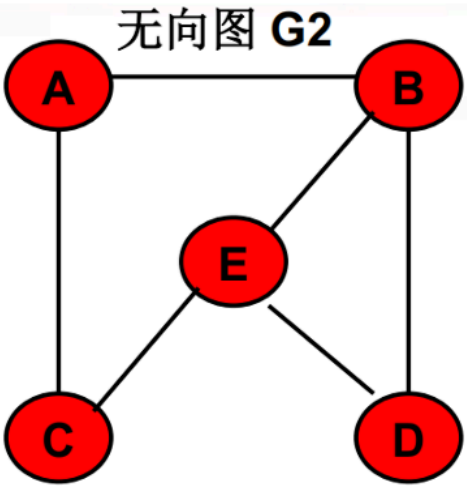
连通分量：极大连通子图

生成树：一个连通图的生成树是其极小连通子图。n个顶点，n-1条边。

生成森林：非连通图的每个连通分量构造一棵生成树，合起来是生成森林

强连通图与强连通分量：有向图中，对于每对 v_i 和 v_j，都存在 $v_i \to v_j$ 和 $v_j \to v_i$ 的通路。非强连通图的极大强连通子图叫强连通分量





G1 = (V1, {A1})
V1 = {A,B,C,D}
A1 = {<A,B>, <A,C>, <C,D>, <D,A>}

G2 = (V2, {A2})
V2 = {A,B,C,D,E}
A2 = {(A,B), (A,C),(B,D), (B,E), (C,E),(D,E)}

▼ 例题

10. 设无向图 G=(V,E)和 G'=(V',E')，如果 G'是 G 的生成树，则下列说法中错误的是[B]。

A. G'是 G 的连通分量 极大连通子图，可能含有环
B. G'是 G 的一个无环子图
C. G'是 G 的子图
D. G'是 G 的极小连通子图且 V=V'

11. 设有 5 个结点的无向图，该图至少应有[C]条边才能确保是一个连通图。

A. 5　　B. 6　　C. 7　　D. 8

4个结点的完全图+1条边 $\frac{n(n-1)}{2}=6$

在一个有向图中，所有顶点的度数之和等于所有弧数的__倍。

A. 3

B. 2

C. 1

D. 1/2

解析：本题考点是有向图顶点度数与弧数的关系。有向图的某个顶点v，把以v为终点的边的数目，称为v的入度；以v为始点的边的数目，称为v的出度；v的度则定义为该顶点的入度和出度之和。显然，在一个有向图中，所有顶点的度数之和等于所有弧数的2倍。因此，本题参考答案是B。

▼ 存储结构

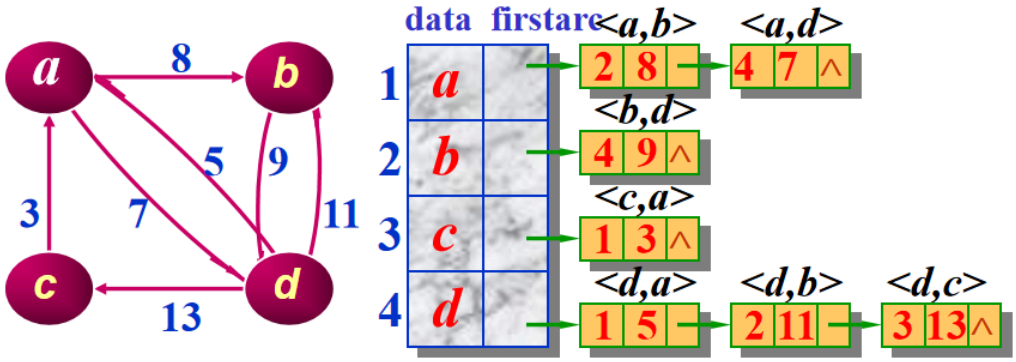
▼ 邻接矩阵

```
typedef struct{
    char Vex[MaxVertexNum]; //顶点表
    int Edge[MaxVertexNum][MaxVertexNum]; //边表（0，1）
    int vexnum,arcnum; //图当前顶点数和边数
}MGraph;

//带权图
typedef char VertexType;
typedef int EdgeType;
typedef struct{
    VertexType Vex[MaxVertexNum];
    EdgeType Edge[MaxVertexNum][MaxVertexNum];//边的权
    int vexnum,arcnum;
}MGraph1;
```

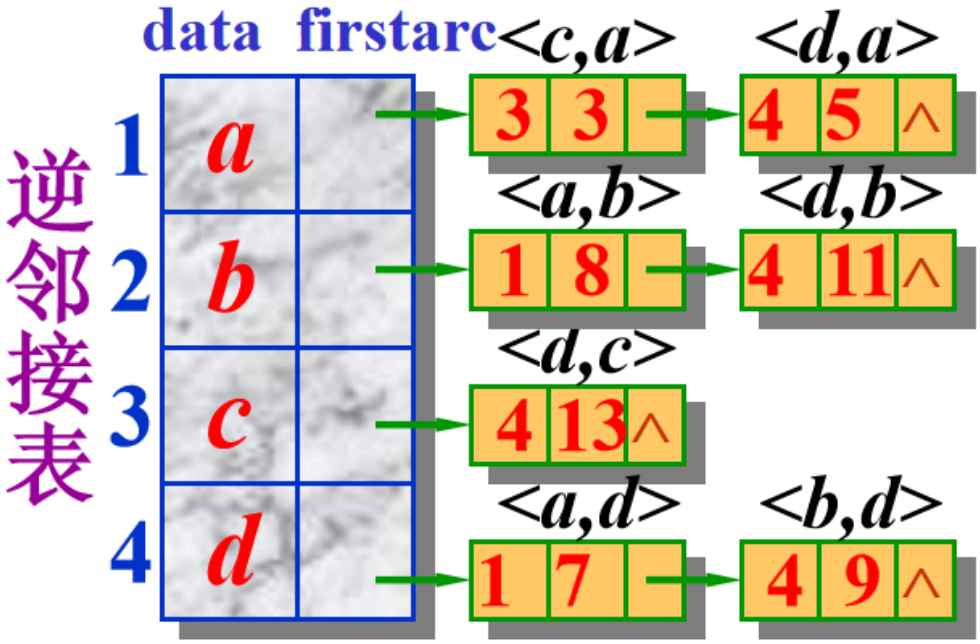
- **无向图的邻接矩阵是对称的**
- 无向图：度为行或列中非零元个数
- 有向图：出度为行中非零元个数，入度为列中非零元个数
- 边的插入和删除不影响存储空间大小
- 适合存稠密图
- 空间复杂度O(|V|^2)
- 邻接矩阵插入或删除一条边 $O(1)$，删除所有边 $O(n^2)$

▼ 邻接表（逆邻接表）

```
typedef struct ArcNode{
    int adjvex; //边指向哪个结点
    struct ArcNode *next;   //指向下一条边
    int info;   //边权值
}ArcNode;
//顶点
typedef struct VNode{
    VertexType data;    //顶点信息
    ArcNode *first;     //第一条边
}VNode,AdjList[MaxVertexNum];
//用邻接表存储的图
typedef struct{
    AdjList vertices;
    int vexnum,arcnum;
}ALGraph;
```



- 适合存稀疏图
- 空间复杂度 无向图$O(V+2E)$ 有向图$O(V+E)$
- 不方便找有向图的所有入度



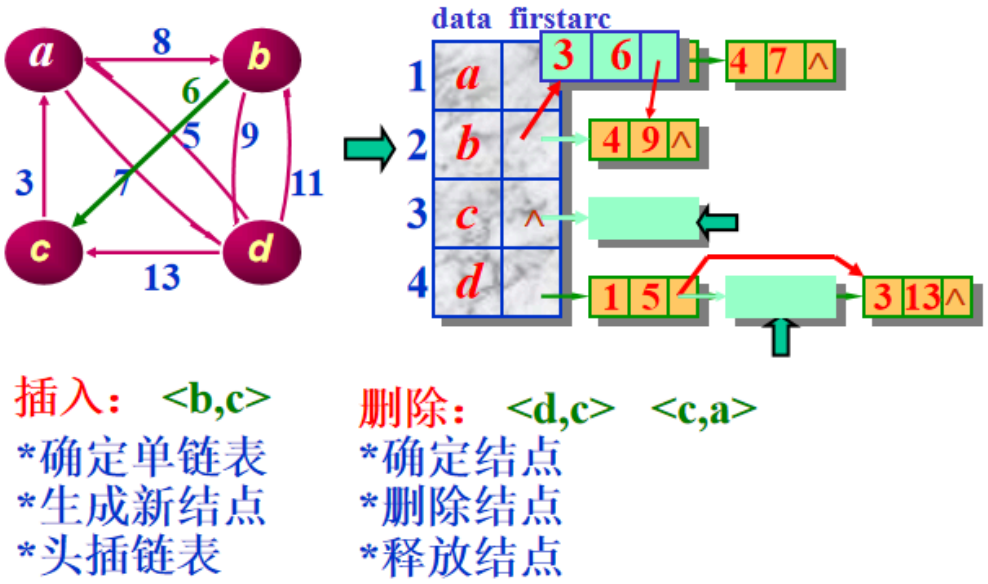
邻接表插入和删除

建表时间复杂度 $O(V+E)$

插入一条边 $O(1)$，删除一条边最坏 $O(n)$，删除所有边、删除一个节点 $O(V+E)$

有向图只插入（删除）一个结点

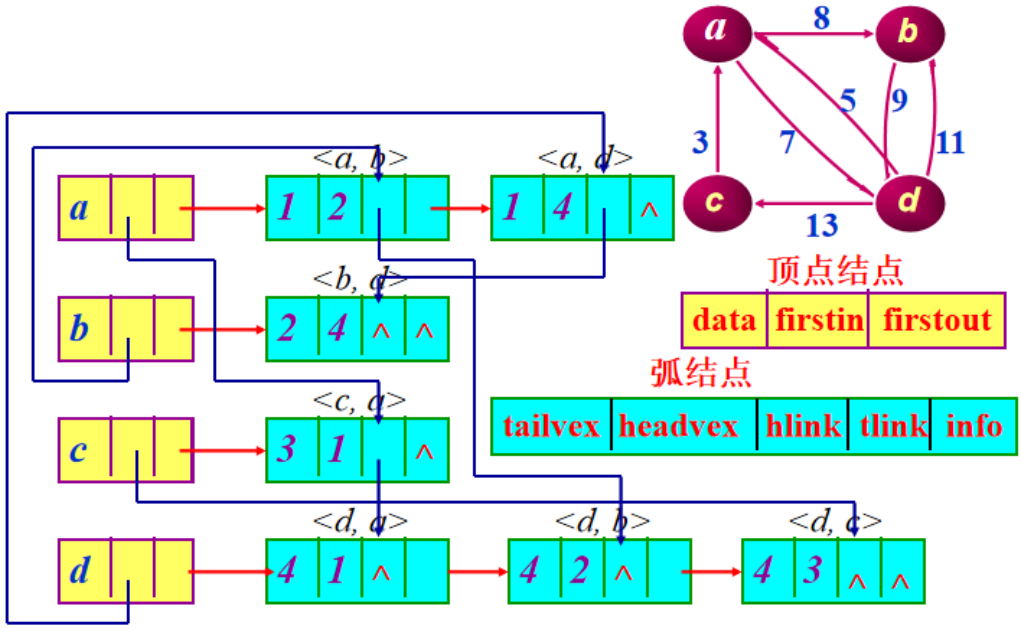
无向图插入（删除）两个结点



▼ 十字链表（有向图）

```
// 弧结点结构定义
typedef struct ArcBox {
    int tailvex, headvex; // 该弧的尾和头顶点的位置
    struct ArcBox *hlink, *tlink; // 分别为弧头相同和弧尾相同的弧的链域
    VertexType weight; // 与弧相关的权值，无权则为0
} ArcBox;
//顶点结构定义
typedef struct VexNode {
    VertexType data;
    ArcBox *firstin, *firstout; // 分别指向该顶点第一条入弧和出弧
} VexNode;
typedef struct { // 十字链表结构定义
    VexNode xlist[MaxVertexNum]; // 表头向量
    int vexnum, arcnum; // 有向图的当前顶点数和弧数
} OLGraph;
```

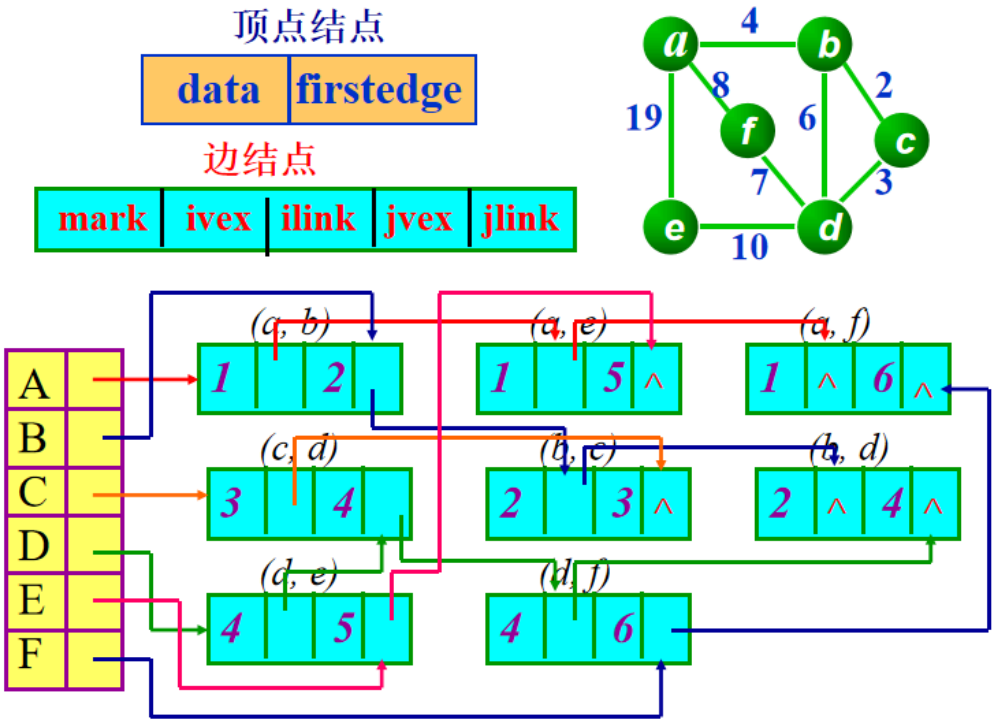
- 空间复杂度 $O(V+E)$
- 十字链表=邻接表+逆邻接表



▼ 邻接多重表（无向图）

```
// 边结点结构定义
typedef struct Ebox {
    VisitIf mark; // 访问标记
    int ivex, jvex; // 该边依附的两个顶点的位置
    struct EBox *ilink, *jlink; // 分别指向依附这两个顶点的下一条边
    VertexType weight; // 与弧相关的权值，无权则为0
} Ebox;
// 顶点结点结构定义
typedef struct VexBox {
    VertexType data;
    Ebox *firstedge; // 指向第一条依附该顶点的边
} VexBox;
// 多重链表结构定义
typedef struct {
    VexBox adjmulist[MaxVertexNum];
    int vexnum, edgenum; // 无向图的当前顶点数和边数
} AMLGraph;
```

- 空间复杂度 $O(V+E)$



▼ 遍历

- **深度优先&广度优先时间复杂度一样，取决于存储图的结构**
- **时间复杂度=访问各结点所需时间+探索各条边所需时间**
- DFS可以求两个顶点间的一条简单路径，BFS可以求两个顶点间的最短路径

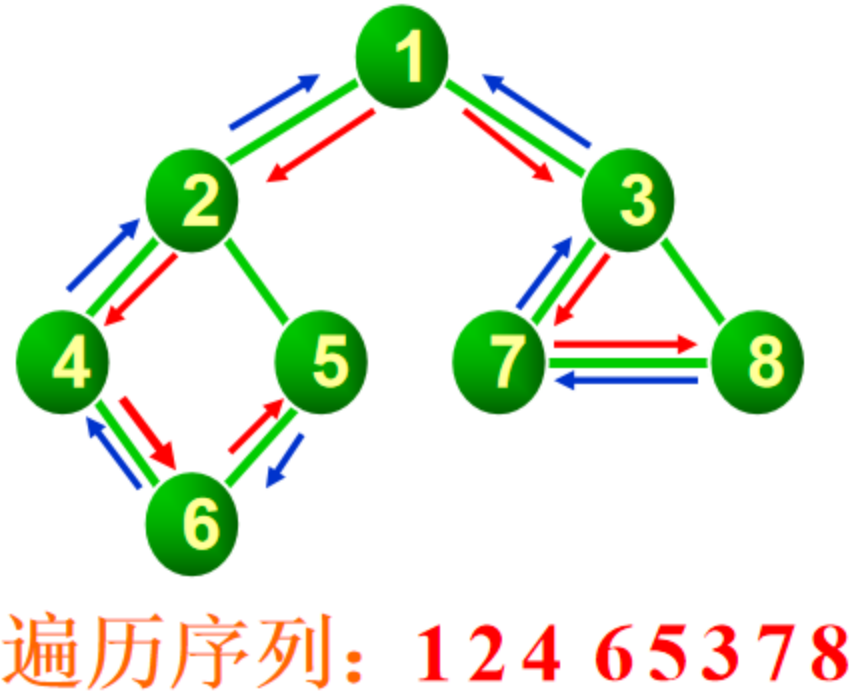
▼ DFS（深度优先）

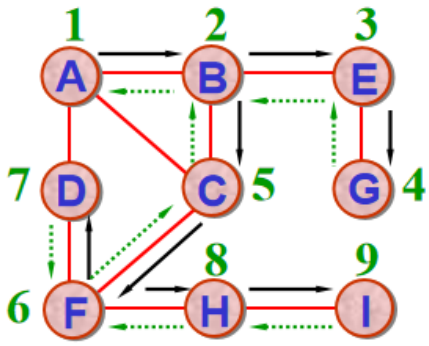
```
void DFS(Graph G,int v){
   visit(v);
   visited[v]=TRUE;
   for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){
      if(!visited[w])  //w为u尚未访问的邻接顶点
         DFS(G,w);
   }
}
//遍历图中的所有连通图
void DFSTraverse(Graph G){
   for(v=0;v<G.vexnem;++v){
```

```
        visited[v]=FALSE;
    }
    for(v=0;v<G.vexnum;++v){
        if(!visited[v])
            DFS(G,v);
    }
}
```

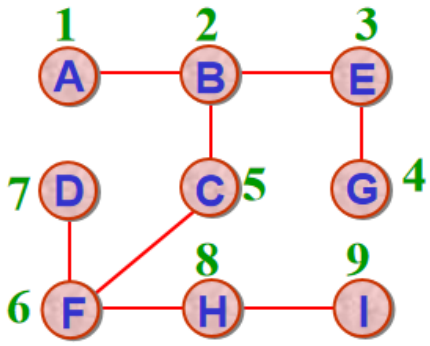
邻接矩阵DFS：空间复杂度 $O(V)$、时间复杂度 $O(V^2)$

邻接表DFS：时间复杂度 $O(V+E)$





深度优先搜索过程

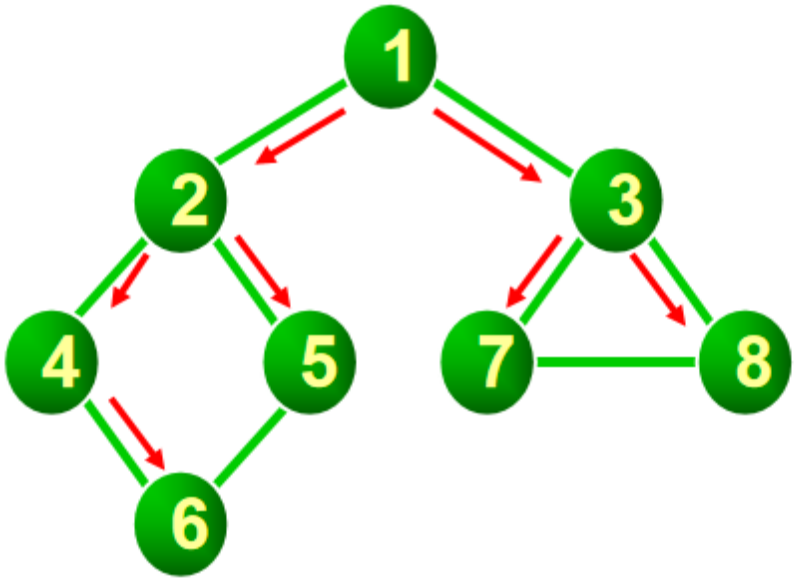


深度优先生成树

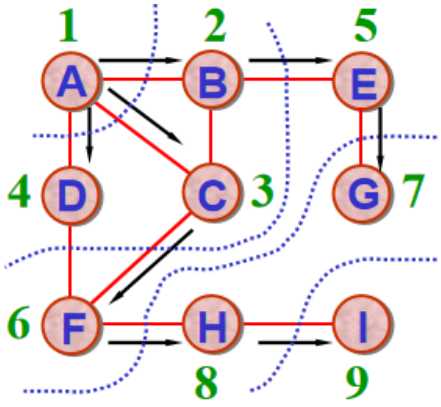
▼ BFS（广度优先）

```
bool visited[Maxsize];
void BFS(Graph G,int v){
   visit(v);   //访问
   visited[v]=TRUE;   //访问标记
   Enqueue(Q,v);       //入队
   while(!isEmpty(Q)){
      DeQueue(Q,v);   //出队
      //检测v的所有邻接点
      for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){
         if(!visited[w]){    //w未被访问
            visit(w);
            visited[w]=TRUE;
            EnQueue(Q,w);   //w入队
         }
      }
   }
}
//遍历表内所有的连通图
void BFSTraverse(Graph G){
   //初始化标记
   for(int i=0;i<G.vexnum;++i){
      visited[i]=FALSE;
   }
   InitQueue(Q);
```

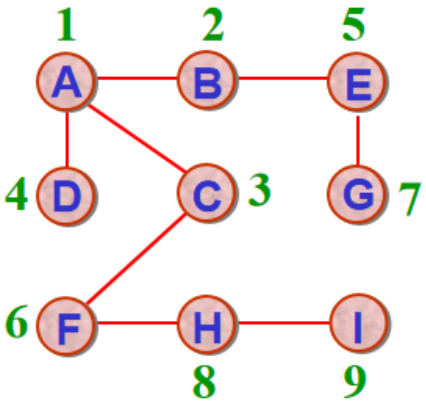
```
    for(int i=0;i<G.vexnum;++i){
        if(!visited[i])
            //有几个连通图就执行几次
            BFS(G,i);
    }
}
```



遍历序列：1 2 3 4 5 7 8 6



广度优先搜索过程



广度优先生成树

▼ 最小生成树

有 n 个顶点的连通图的生成树有 $n-1$ 个顶点

Prim和Kruskal都只适用无向图

▼ Prim算法

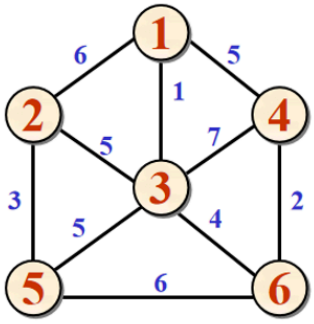
每次找代价最小的新顶点加入生成树

适用于边稠密图

时间复杂度 $O(V^2)$

```
// 普里姆算法构造最小生成树
void MiniSpanTree_PRIM(int u) {
    // 从顶点u出发构造最小生成树
    int i, j, k;

    // 初始化lowcost数组和closest数组
    for (j = 0; j < NumVertex; j++) {
        lowcost[j] = cost[u][j]; // lowcost[u] = 0，因为u是起始点
        closest[j] = u;
    }

    // 循环直到所有顶点都被加入最小生成树
    for (i = 1; i < NumVertex; i++) {
        // 寻找lowcost中不为0的且最小的顶点
        k = -1;
        for (j = 0; j < NumVertex; j++) {
            if (!visited[j] && lowcost[j] != 0 && (k == -1 || lowcost[j] < lowcost[k]
                k = j;
            }
        }

        // 如果没有找到这样的顶点，说明图不连通
        if (lowcost[k] == INFINITY) {
            printf("ERROR: 图不连通\n");
```

```
            break;
        }

        // 将(closest[k], k)这条边加入到最小生成树
        printf("边(%d, %d)加入最小生成树，权重为%d\n", closest[k], k, lowcos
        visited[k] = 1; // 标记顶点k已加入最小生成树
        lowcost[k] = 0; // 将顶点k从lowcost中移除

        // 更新lowcost和closest数组
        for (j = 0; j < NumVertex; j++) {
            if (!visited[j] && cost[k][j] < lowcost[j]) {
                lowcost[j] = cost[k][j];
                closest[j] = k;
            }
        }
    }
}
```



加入1结点

	1	2	3	4	5	6
lowcost	0	6	1	5	∞	∞
closest	1	1	1	1	1	1

加入3结点

	1	2	3	4	5	6
lowcost	0	5	0	5	5	4
closest	1	3	1	1	3	3

加入6结点

	1	2	3	4	5	6
lowcost	0	5	0	2	5	0
closest	1	3	1	6	3	3

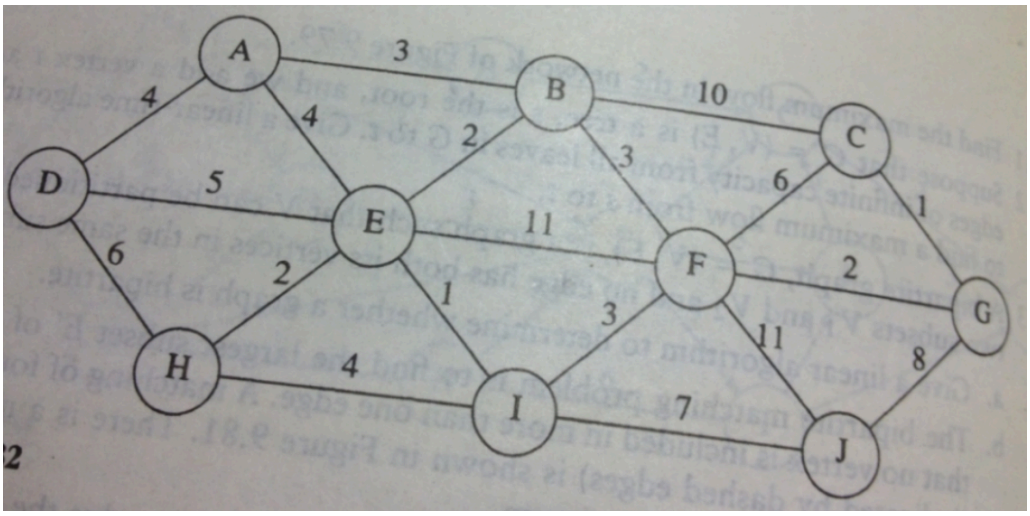
加入4结点

	1	2	3	4	5	6
lowcost	0	5	0	0	5	0
closest	1	3	1	6	3	3

加入5结点

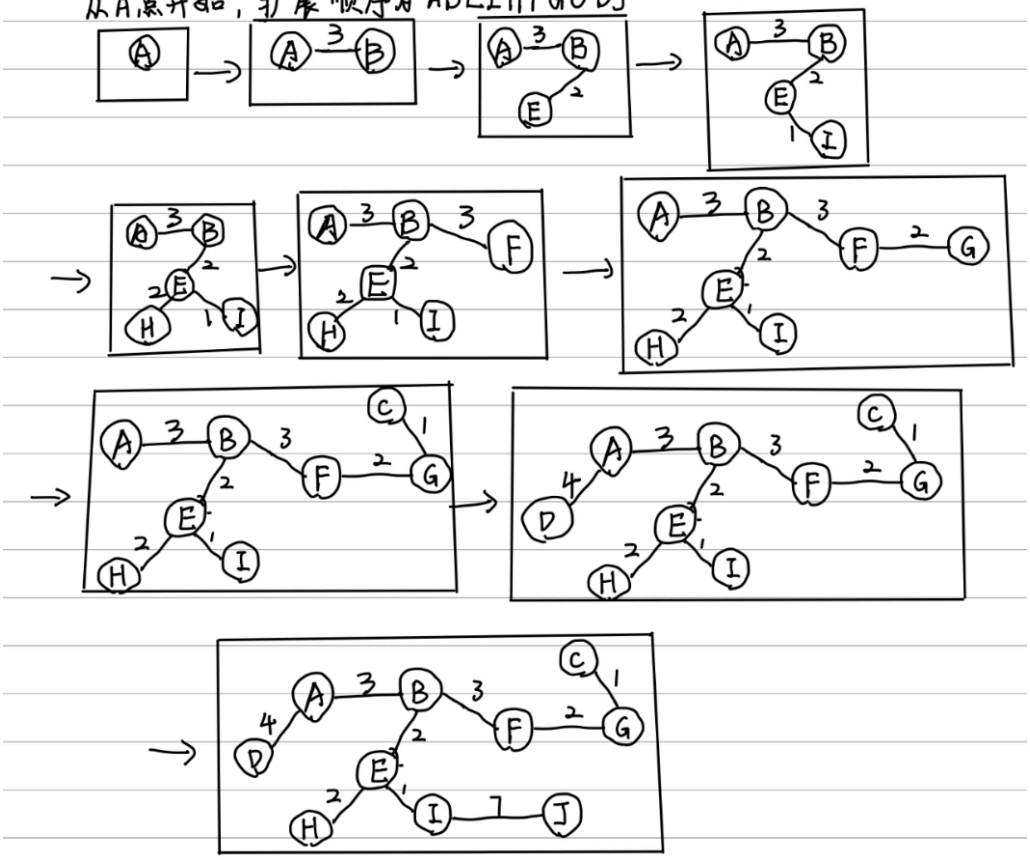
	1	2	3	4	5	6
lowcost	0	3	0	0	0	0
closest	1	5	1	6	3	3

加入2结点

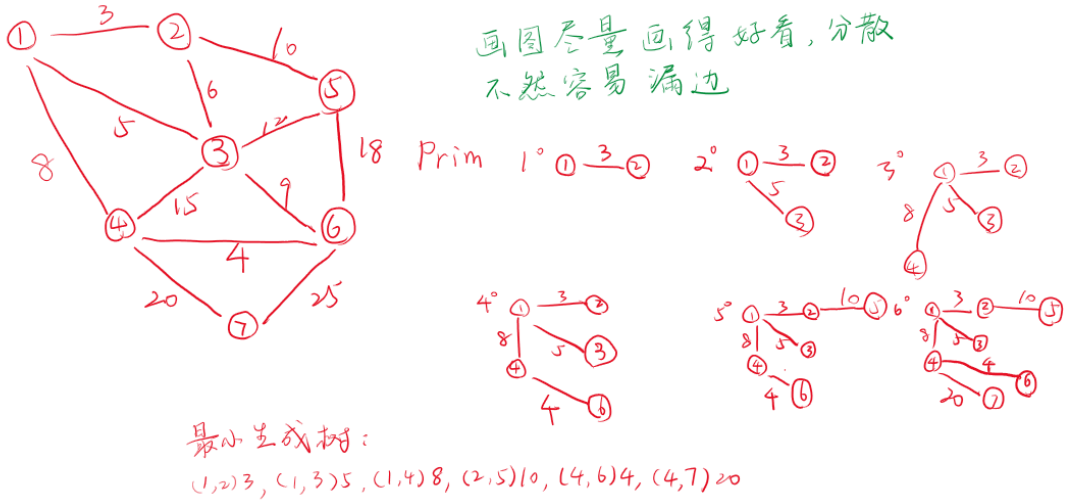


Prim算法：

从A点开始，扩展顺序为 ABEIHFGCDJ



已知一个图的顶点集 V 和边集 E 分别为：V={1, 2, 3, 4, 5, 6, 7}；
E={(1, 2)3, (1, 3)5, (1, 4)8, (2, 5)10, (2, 3)6, (3, 4)15,
(3, 5)12, (3, 6)9, (4, 6)4, (4, 7)20, (5, 6)18, (6, 7)25}；
用克鲁斯卡尔算法得到最小生成树，试写出在最小生成树中依次得到的各条边。

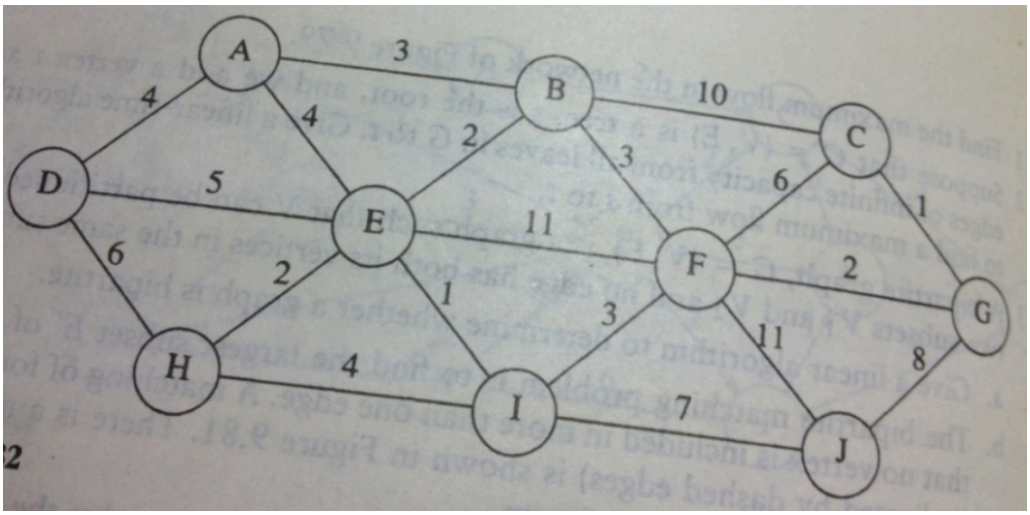


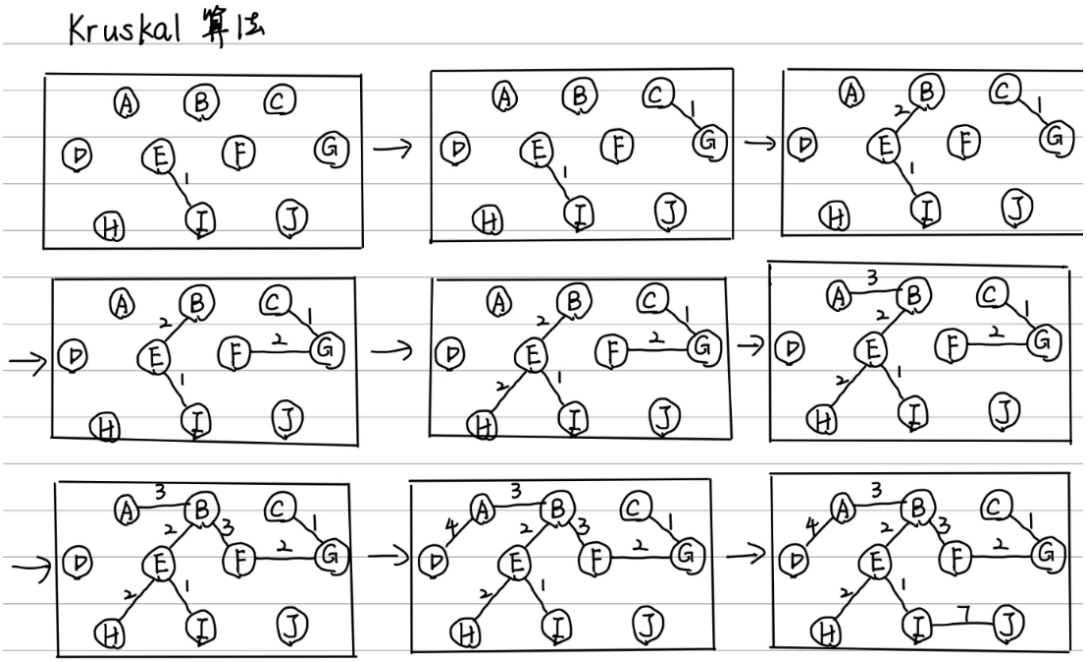
▼ Kruskal算法

每次选一条权值最小的边（不构成环）加入

适用于边稀疏图

时间复杂度 $O(E\log_2 E)$

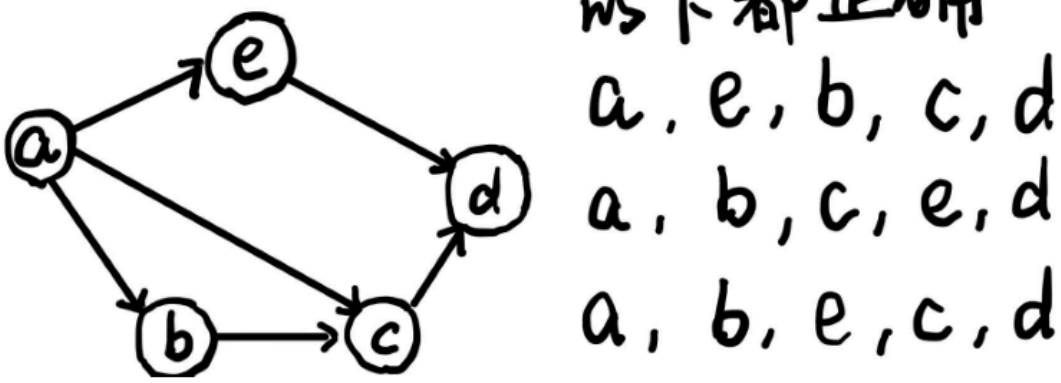




▼ 拓补排序

- 不唯一

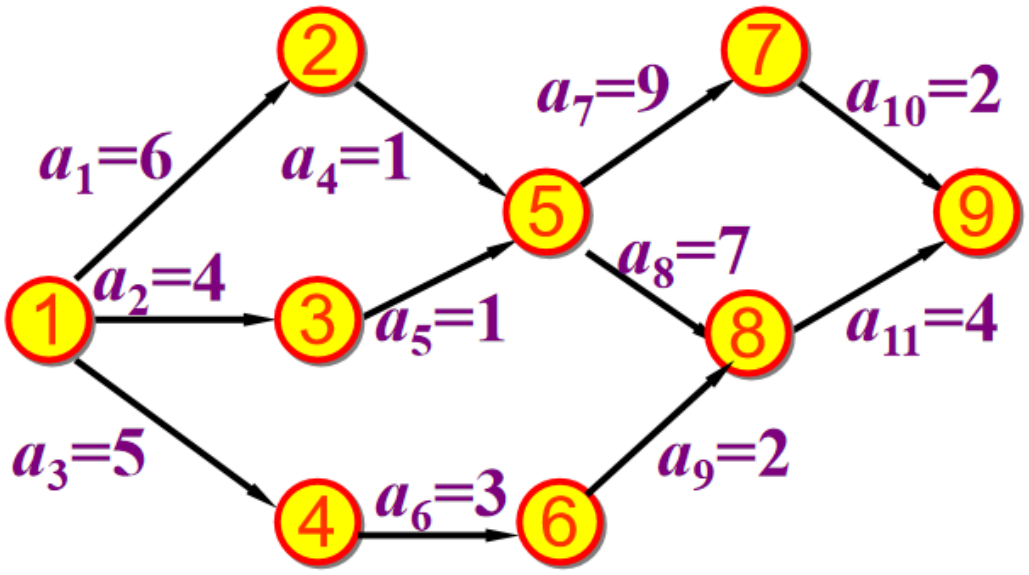
每次选入度为0的点，删除这个点和它的出边。



逆拓补排序

每次选出度为0的点，删除这个点和它的入边

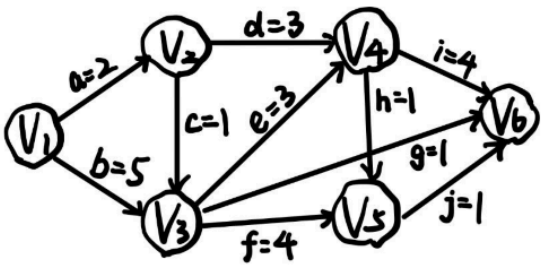
关键路径（AOE网络）



顶点表示事件，边表示活动。解决问题：工程全部完成至少需要多少时间？

取决于最长的一条路径，即关键路径。

- 事件 V_i最早可能发生时间 $V_e(i)$
- 事件 V_i最迟允许发生时间 $V_l(i)$
- 活动 a_k最早可能开始时间 $e[k]$
- 活动 a_k最迟允许开始时间 $l[k]$
- 时间余量 $l[k]-e[k]$



	Ve	Vl
V_1	0	0
V_2	2	4
V_3	5	5
V_4	8	8
V_5	9	11
V_6	12	12

	e	l
a	0	2
b	0	0
c	2	4
d	2	5
e	5	5
f	5	7
g	5	11
h	8	10
i	8	8
j	9	11

拓扑排序确定Ve(i) 取大

初始 $Ve_1=0$

删V_1，得$Ve_2=2$，$Ve_3=5$

删V_2，得$Ve_3=3<5$，保留$Ve_3=5$，$Ve_4=5$

删V_3，得$Ve_4=8>5$，更新，$Ve_5=9$，$Ve_6=6$

删V_4，得$Ve_5=9$，$Ve_6=12>6$，更新

删V_5，得$Ve_6=10<12$，保留$Ve_6=12$

逆拓扑排序确定Vl(i) 取小

初始 $Vl_6=12$

删V_6，$Vl_4=8$，$Vl_5=11$

删V_5，$Vl_4=10>8$，保留$Vl_4=8$

删V_4，$Vl_2=5$，$Vl_3=5$

删V_3，$Vl_2=4$，$Vl_1=0$

删V_2，$Vl_1=2>0$，保留$Vl_1=0$

e[K]为发出活动的顶点的Ve(i)

l[K]为接收活动的顶点的Vl(i)减活动时间

e[K]=l[K]的活动是关键活动

关键路径为：V_1, V_3, V_4, V_6

▼ 最短路径

单源点路径：

▼ BFS（无权图）

```
bool visited[Maxsize];
void BFS_Distance(Graph G,int v){
   //初始化距离和前驱
   for(int i=0;i<G.vexnum;++i){
      dist[i]=INF;
      path[i]=-1;
   }
   d[u]=0;
   visited[v]=TRUE;    //访问标记
   Enqueue(Q,v);       //入队
```

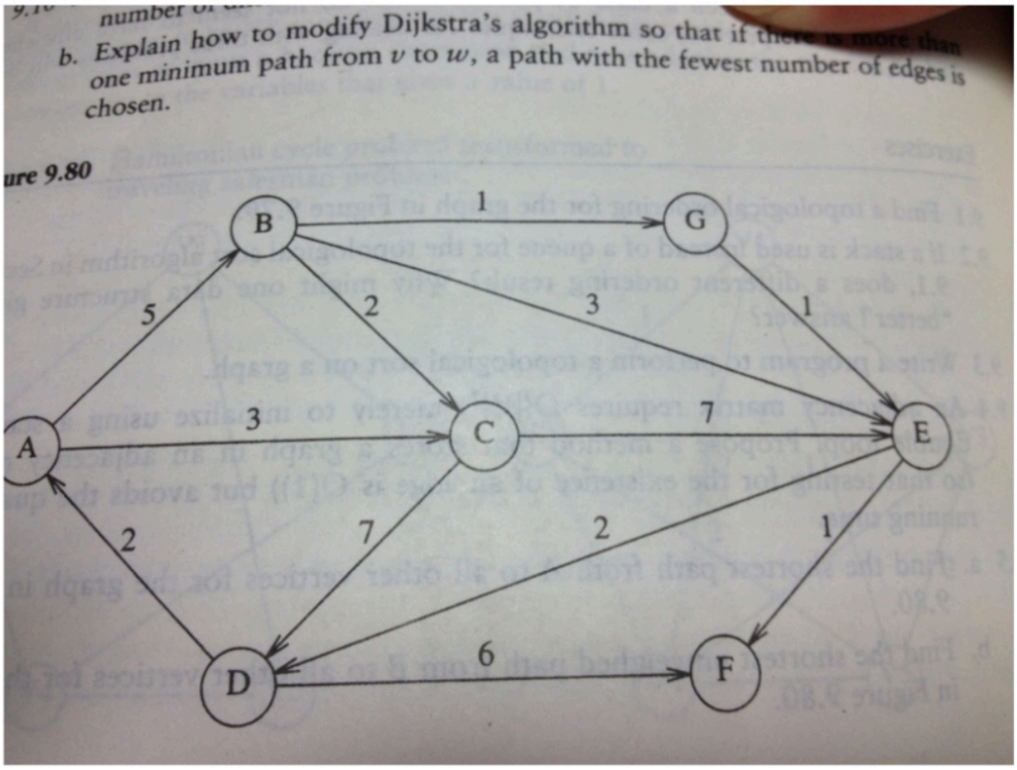
```
    while(!isEmpty(Q)){
        DeQueue(Q,v);   //出队
        //检测v的所有邻接点
        for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){
            if(!visited[w]){    //w未被访问
                dist[w]=dist[v]+1; //路径长度+1
                path[w]=v;
                visited[w]=TRUE;
                EnQueue(Q,w);   //w入队
            }
        }
    }
}
```

▼ Dijkstra算法（有权图和无权图，不带负值的）

时间复杂度 $O(V^2)$

- $dist[i]$存起始点至 i 的最短路径长度， $path[i]$存放前驱结点， $final[i]$存放该点是否已经找出最短路径。

2. 使用 Dijkstra 算法求下图中顶点 A 到其他顶点的最短路径，写出计算的过程



Dijkstra 算法

Graph

	1	2	3	4	5	6	7
1	∞	5	3	∞	∞	∞	∞
2	∞	∞	2	∞	3	∞	1
3	∞	∞	∞	7	7	∞	∞
4	2	∞	∞	∞	∞	6	∞
5	∞	∞	∞	2	∞	1	∞
6	∞	∞	∞	∞	∞	∞	∞
7	∞	∞	∞	∞	1	∞	∞

从A开始

	1	2	3	4	5	6	7
dist[]	0	5	3	∞	∞	∞	∞
path[]	-1	1	1	-1	-1	-1	-1
visited[]	1	0	0	0	0	0	0

$dist_{min}$=C

	1	2	3	4	5	6	7
dist[]	0	5	3	10	10	∞	∞
path[]	-1	1	1	3	3	-1	-1
visited[]	1	0	1	0	0	0	0

$dist_{min}$=B

	1	2	3	4	5	6	7
dist[]	0	5	3	10	8	∞	6
path[]	-1	1	1	3	2	-1	2
visited[]	1	1	1	0	0	0	0

$dist_{min}$=G

	1	2	3	4	5	6	7
dist[]	0	5	3	10	7	∞	6
path[]	-1	1	1	3	7	-1	2
visited[]	1	1	1	0	0	0	1

$dist_{min}$=E

	1	2	3	4	5	6	7
dist[]	0	5	3	9	7	8	6
path[]	-1	1	1	5	7	5	2
visited[]	1	1	1	0	1	0	1

$dist_{min}$=F

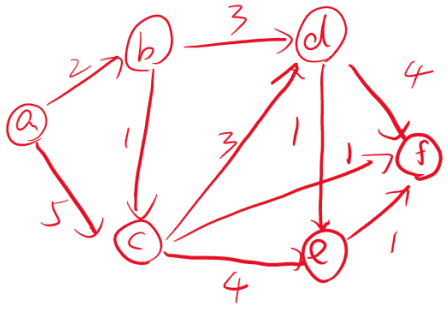
	1	2	3	4	5	6	7
dist[]	0	5	3	9	7	8	6
path[]	-1	1	1	5	7	5	2
visited[]	1	1	1	0	1	1	1

$dist_{min}$=D

	1	2	3	4	5	6	7
dist[]	0	5	3	9	7	8	6
path[]	-1	1	1	5	7	5	2
visited[]	1	1	1	1	1	1	1

A→B: (A,B)=5　　A→D: (A,B,G,E,D)=9　　A→F: (A,B,G,E,F)=8

A→C: (A,C)=3　　A→E: (A,B,G,E)=7　　A→G: (A,B,G)=6



从a出发

	初始化	加b	加c	加f	加d
	1	2	3	4	5
b	2 (a,b)	\	\	\	\
c	5 (a,c)	3 (a,b,c)	\	\	\
d	∞	5 (a,b,d)	5 (a,b,d)	5 (a,b,d)	\
e	∞	∞	7 (a,b,c,e)	7 (a,b,c,e)	6 (a,b,d,e)
f	∞	∞	4 (a,b,c,f)	\	\

a→b : (a,b)=2　　a→d : (a,b,d)=5　　a→f : (a,b,c,f)=4

a→c : (a,b,c)=3　　a→e : (a,b,d,e)=6

```
void init(int dist[V],int path[V]){
    for(int i=1;i<V;i++){
        dist[i]=INF;
        path[i]=-1;
    }
}
//打印路径
void printPath(int n,int st,int path[V]){
    if(n!=st){
        printPath(path[n],st,path);
    }
    printf("%d,",n);
}
```

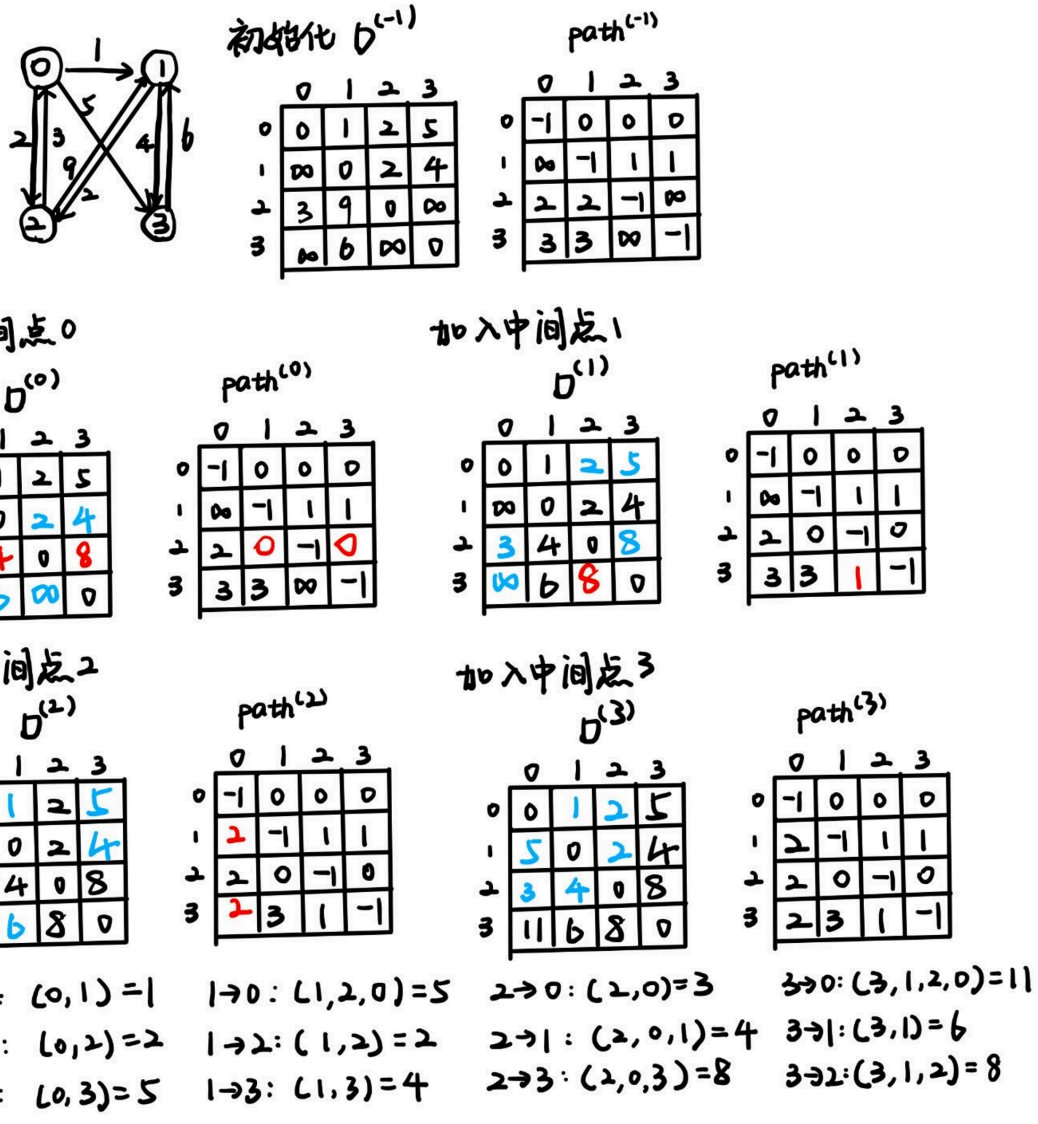
```
int main() {
    int graph[V][V] = {0},final[V]={0},dist[V]={0},path[V]={0};
    create(graph);
    init(dist,path);
    int st,end;
    scanf("%d,%d",&st,&end);
    final[st]=1;
    dist[st]=0;
    //初始化
    for(int i=1;i<V;i++){
        if(graph[st][i]!=INF){
            dist[i]=graph[st][i];
            path[i]=st;
        }
    }
    while(1){
        //找dist[]中最小的
        int min=-1,mindist=INF;
        for(int i=1;i<V;i++){
            if(final[i]==0&&dist[i]<mindist){
                mindist=dist[i];
                min=i;
            }
        }
        if(min==-1) break;
        final[min]=1;
        //更新path[]
        for(int i=1;i<V;i++){
            if(final[i]==0&&dist[min]+graph[min][i]<dist[i]){
                dist[i]=dist[min]+graph[min][i];
                path[i]=min;
            }
        }
    }
    //不可达
    if(dist[end]==INF){
```

```
        printf("-1\n");
        return 0;
    }
    printPath(end,st,path);
    return 0;
}
```

各对顶点间最短路径：

▼ Floyd算法（不能解决带负权回路的）

- 加入中间点k，D的第k行第k列和主对角线不变
- 对应行列值相加与现有比较，取小（例如增加中间点2，1→3与1→2→3比较）




```
//核心代码
for(int k=0;k<n;k++){   //以k为中间点
    for(int i=0;i<n;i++){   //遍历矩阵
        for(int j=0;j<n;j++){
            if(A[i][j]>A[i][k]+A[k][j]){   //以k为中间点的路径更短
                A[i][j]=A[i][k]+A[k][j];
                path[i][j]=k;
            }
        }
```

```
    }
}
```