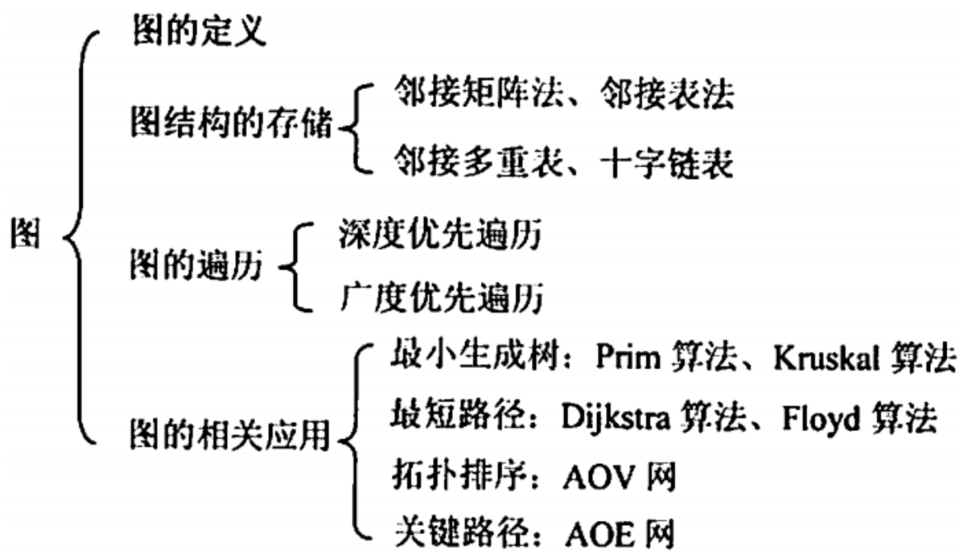
8-图

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

1、目标

> 📌 掌握图的各种概念
>
> 掌握图的存储结构
>
> 掌握图的遍历
>
> 掌握图的应用



2、图的25个相关概念

- 图的概念

> 📌 图是非线性结构，由**顶点**和**边**组成
>
> **[注意]：图不能是空图，至少要存在顶点**

- 顶点

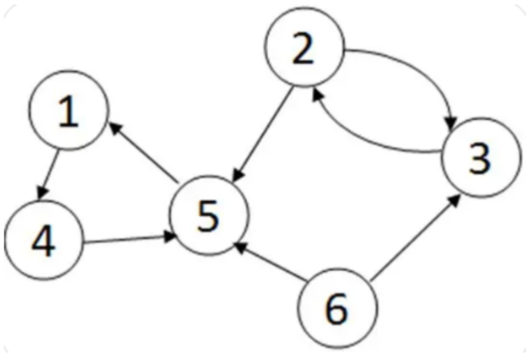
> 📌 图中的结点称为顶点

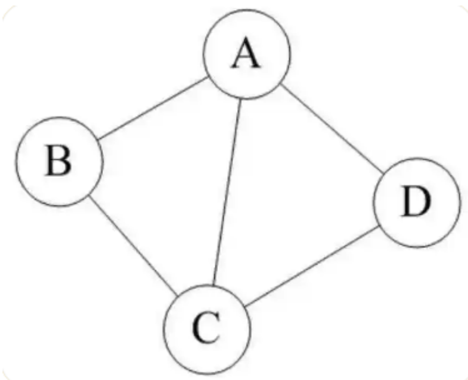
- 边

> 📌 连接顶点之间的线叫做边，无向边简称**边**，有向边称为**弧**

- 顶点的度、入度、出度

> 📌 连接顶点的边就称为顶点的度
>
> 对有向图来说，度还分入度和出度





- 有向图、无向图

> 📌 有向图的边使用尖括号<>表示，无向图的边使用圆括号()表示

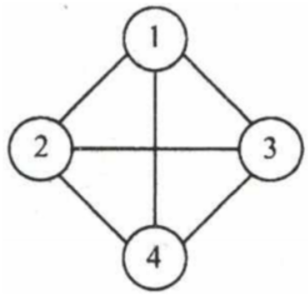
$$G_1 = (V_1, E_1)$$

$$V_1 = \{1, 2, 3\}$$

$$E_1 = \{<1, 2>, <2, 1>, <2, 3>\}$$

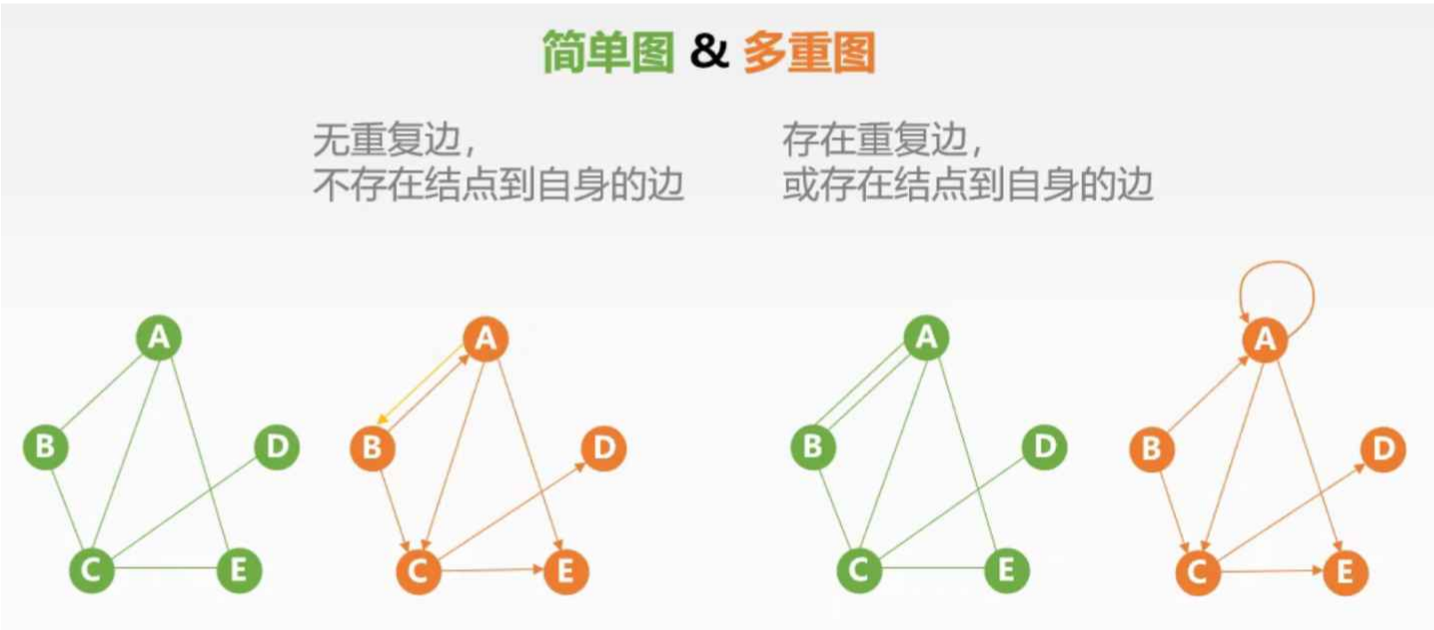
有向图

$$G_2 = (V_2, E_2)$$
$$V_2 = \{1, 2, 3, 4\}$$
$$E_2 = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$



无向图

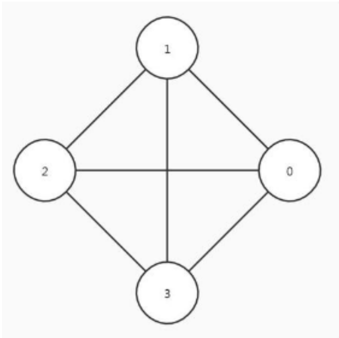
- 简单图、多重图

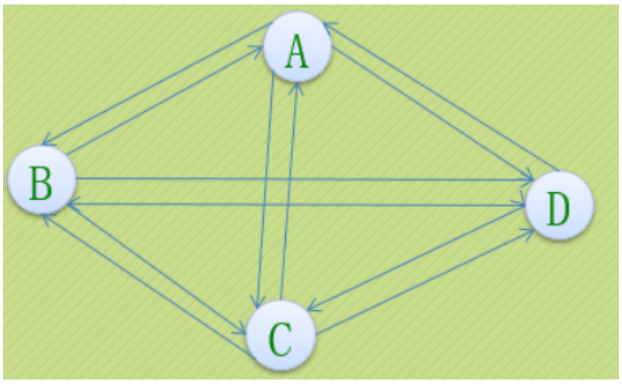


- 完全图

> 📌 对于**无向图**，边的取值范围为0到n(n-1)/2, 如果图有n(n-1)/2条边，则无向称为图称无向完全图，即在完
>
> 全图中任意两个顶点之间都存在边。
>
> 对于**有向图**，边的取值范围为0到n(n-1), 如果图有n(n-1)条边，则称向图称为有向完全图，在有向完全图中任意两个顶点之间都存在方向相反的两条弧

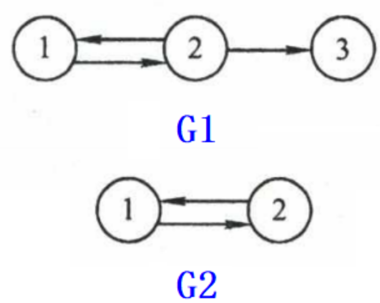
- 连通图、连通分量、强连通图，强连通分量





- 子图

> 📌 有两个图G1,G2,其中G2是G1的子集，就称G2是G1的子图

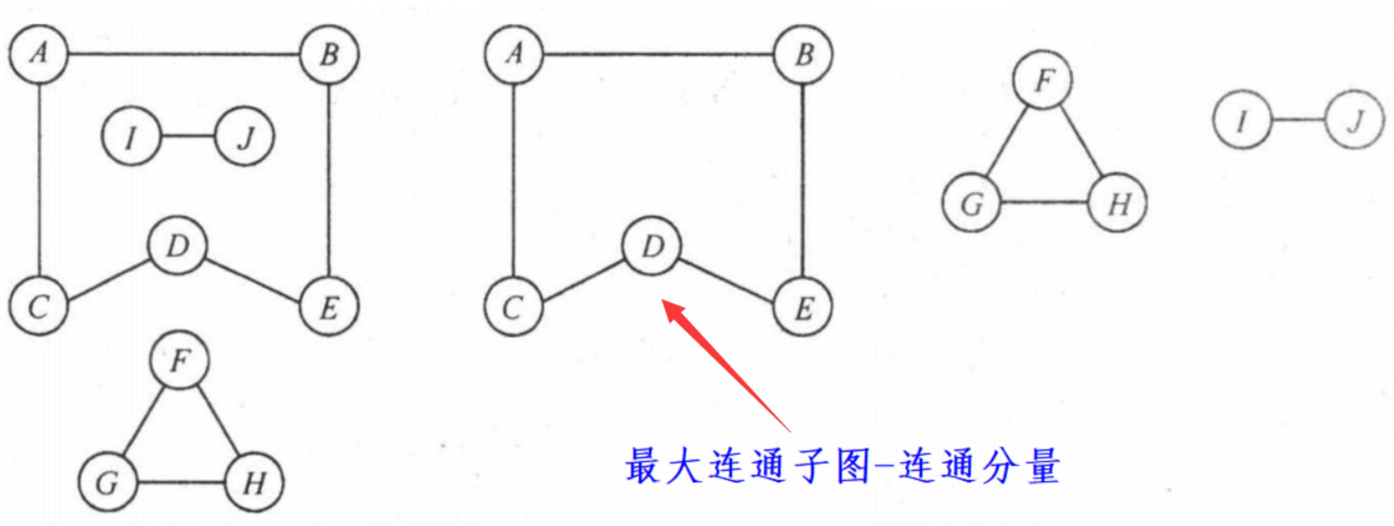


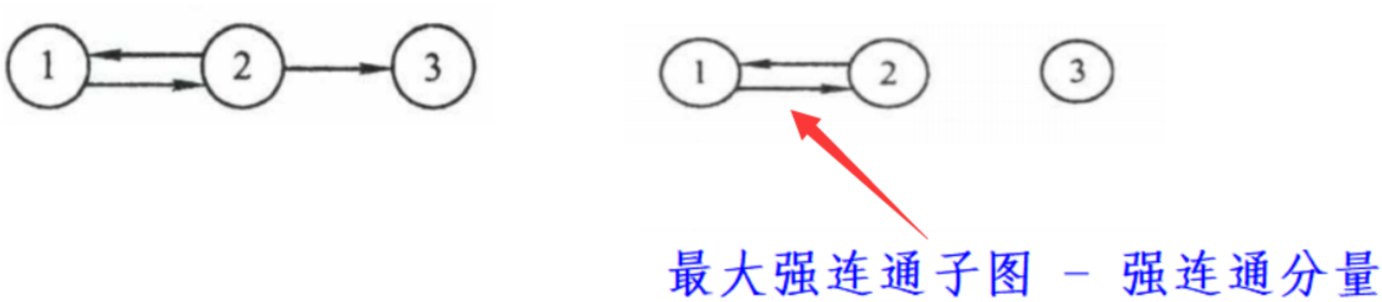
G2是G1的子图

- 连通

> 📌 两个顶点v1, v2之间有路径（不一定是直接的边），则称v1和v2是连通的

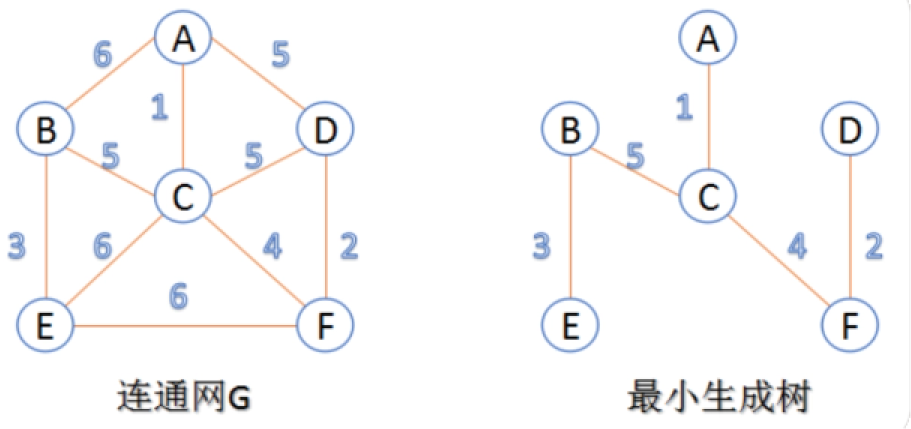
- 连通图、连通分量、强连通图，强连通分量





- 生成树

> 📌 连通图的生成树是包含图中全部顶点的一个极小连通子图。若图中顶点数为n，则它的生成树含有n-1条边
>
> 如果边带有权值，且生成树的边的权值之和最小，此时的树即为最小生成树



- 边的权，网

> 📌 在一个图中，每条边都可以标上具有某种含义的数值，该数值称为该边的权值。这种边上带有权值的图称为带权图，也称**网**。

- 稠密图、稀疏图

> 📌 边数很少的图称为稀疏图，反之称为稠密图，一般当图G满足 |E|<|V|log|V| 时，可以将G视为稀疏图

- 简单回路、简单路径

> 📌 在路径序列中，顶点不重复出现的路径称为**简单路径**。除第一个顶点和最后一个顶点外，其余顶点不重复出现的回路称为**简单回路**

- 有向树

> 📌 一个顶点的入度为0、其余顶点的入度均为1的有向图，称为有向树

3、图的存储结构

1、邻接矩阵

> 📌 使用二维数组存储，能看懂图的关键在于，**要把数组的行列想象成对应顶点的值，**剩下的就能看懂

其次，邻接矩阵的代码实现，就是一个一维数组顶点，和二维数组边的操作过程

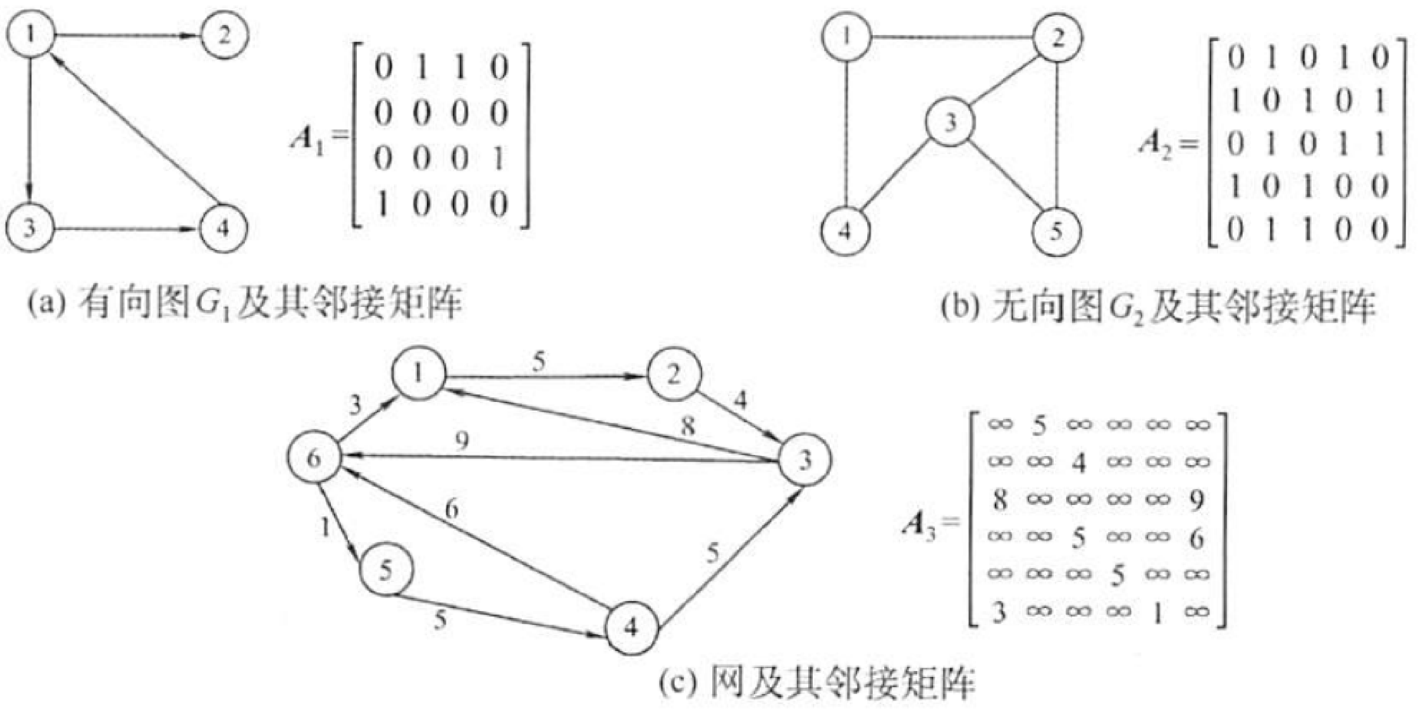


图 6.5　有向图、无向图及网的邻接矩阵

- **图的ADT**

📌 邻接矩阵静态表示法

```
#define MAX_VERTEX_SIZE 10
#define T char

typedef struct GraphMtx
{
    int NumVertices;
    int NumEdges;

    T   VerticesList[MAX_VERTEX_SIZE];   //静态开辟顶点空间
    int Edge[MAX_VERTEX_SIZE][MAX_VERTEX_SIZE]; //静态开辟边空间
}GraphMtx;
```

📌 邻接矩阵动态表示法

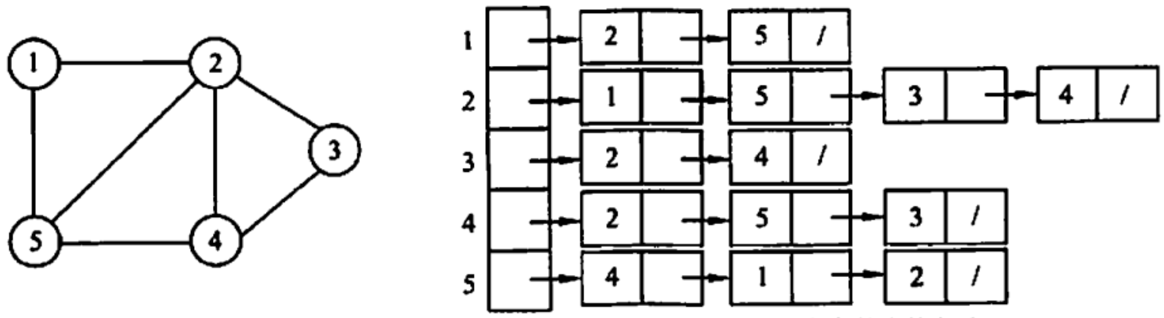
```
#define T char
```

```c

typedef struct GraphMtx
{
    int MaxVertices;
    int NumVertices;
    int NumEdges;

    T   *VerticesList;  //动态开辟顶点空间
    int **Edge;         //动态开辟边空间
}GraphMtx;
```

```c
//初始化图
void InitGraph(GraphMtx *g);
//获取顶点位置
int  GetVertexPos(GraphMtx *g, T v);
//显示图
void ShowGraph(GraphMtx *g);
//插入顶点
void InsertVertex(GraphMtx *g, T v);
//在顶点v1和v2之间插入边
void InsertEdge(GraphMtx *g, T v1, T v2);
//删除顶点
void RemoveVertex(GraphMtx *g, T v);
//删除顶点v1和v2之间的边
void RemoveEdge(GraphMtx *g, T v1, T v2);
//释放图
void DestroyGraph(GraphMtx *g);
//获取顶点v的邻接顶点
int  GetFirstNeighbor(GraphMtx *g, T v);
//获取顶点v的邻接顶点w的下一个邻接顶点
int  GetNextNeighbor(GraphMtx *g, T v, T w);
```

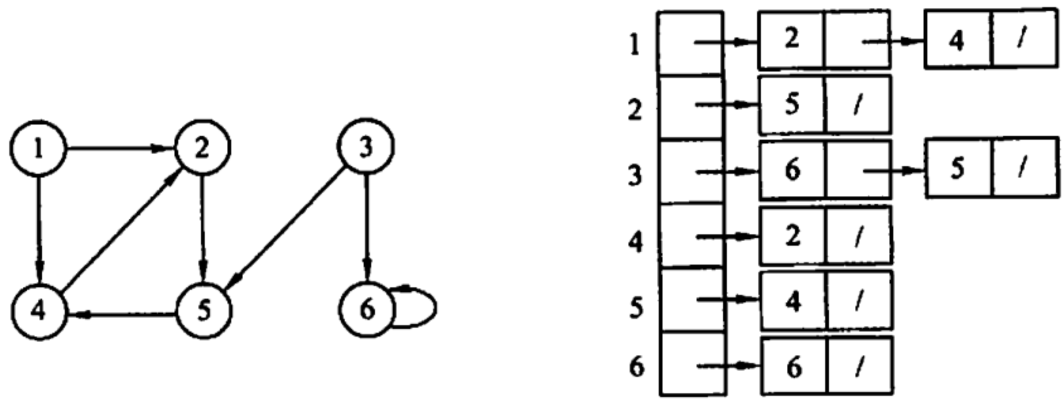
2、邻接表

📌 邻接表就是数组+链表的存储操作，看懂图的关键在于，**数组存放的是顶点，链表的节点表示边**



(a)无向图*G*　(b)无向图*G*的邻接表的表示

图 6.7　无向图邻接表表示法实例



(a)有向图 *G*　(b)有向图 *G* 的邻接表的表示

图 6.8　有向图邻接表表示法实例

- **邻接表ADT**

> 📌 邻接表静态表示法

```
#define MAX_VERTEX_SIZE 10
#define T  char

typedef struct Edge
{
    int dest;
    struct Edge *next;
}Edge;

typedef struct Vertex
{
    T data;
    Edge *first;
}Vertex;

typedef struct GraphLnk
{
```

```
    int NumVertices;
    int NumEdges;
    Vertex NodeTable[MAX_VERTEX_SIZE]; //静态开辟
}GraphLnk;
```

📌 邻接表动态表示法

```
#define T  char

typedef struct Edge
{
    int dest;
    struct Edge *next;
}Edge;

typedef struct Vertex
{
    T data;
    Edge *first;
}Vertex;

typedef struct GraphLnk
{
    int MaxVertices;
    int NumVertices;
    int NumEdges;
    Vertex *NodeTable;   //动态开辟
}GraphLnk;
```

```
void InitGraph(GraphLnk *g);
int  GetVertexPos(GraphLnk *g, T v);
void ShowGraph(GraphLnk *g);
void InsertVertex(GraphLnk *g, T v);
void InsertEdge(GraphLnk *g, T vertex1, T vertex2);

void RemoveEdge(GraphLnk *g, T vertex1, T vertex2);
void RemoveVertex(GraphLnk *g, T vertex);
void DestroyGraph(GraphLnk *g);
int  GetFirstNeighbor(GraphLnk *g, T vertex);
int  GetNextNeighbor(GraphLnk *g,  T vertex1, T vertex2);
```

3、十字链表

📌 图的十字链表表示只要求能够看懂图即可

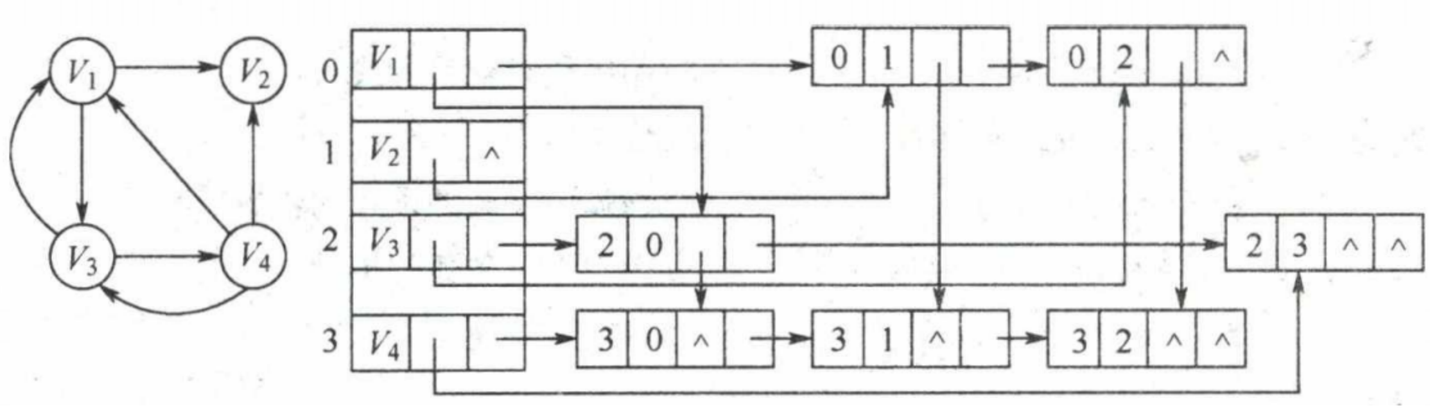
十字链表是针对**有向图**的一种链式存储结构

弧结点

tailvex	headvex	hlink	tlink	(info)

顶点结点

data	firstin	firstout



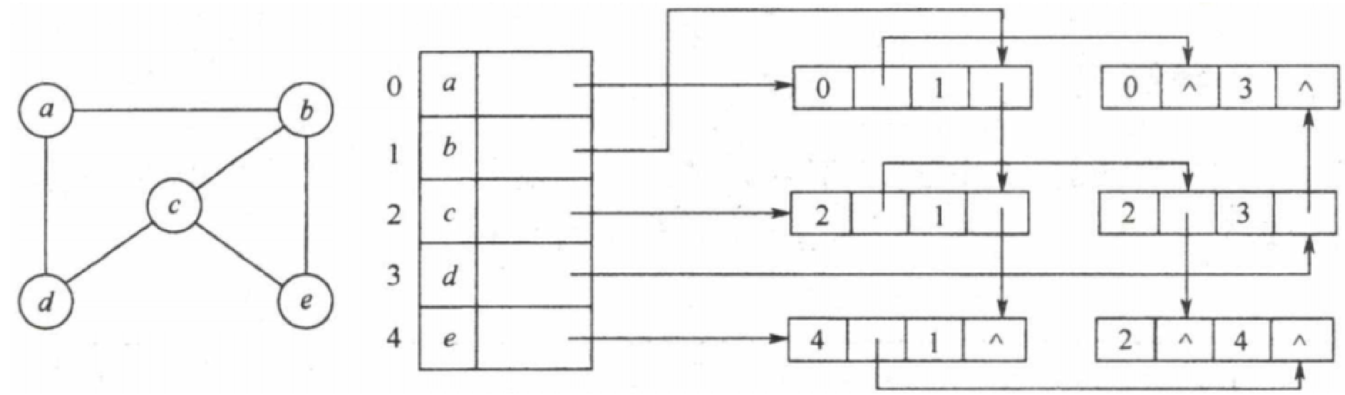
4、邻接多重表

📌 图的邻接多重表表示只要求能够看懂图即可

图的邻接多重表是针对**无向图**的一种链式存储结构

ivex	ilink	jvex	jlink	(info)

data	firstedge

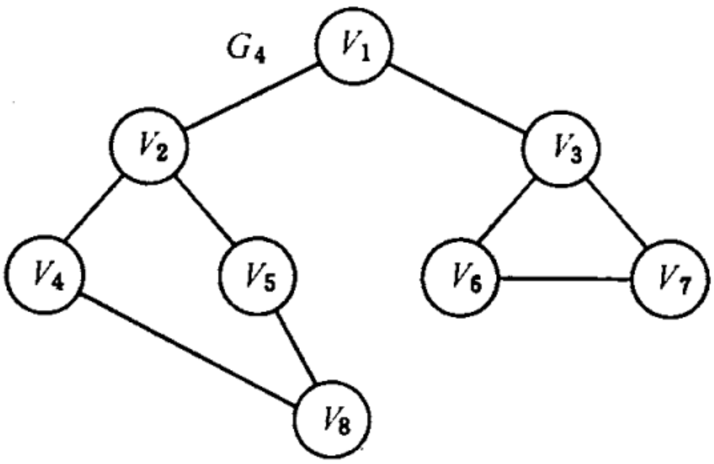


4、图的遍历

📌 深度优先遍历 DFS

广度优先遍历 BFS

注意：**图的遍历结果顺序是不唯一的，跟选择的起始结点和所求邻接结点的顺序有关**



1、深度优先遍历

📌 遍历思想:

1、借助n个临时空间来标记节点是否被访问过

2、首先访问图中的某一顶点v, 接着访问v的邻接顶点w,在访问w的下一邻接顶点，依次类推，重复上述过程

3、当不能在继续向下访问顶点时，依次退回到最近被访问的顶点，如果还有邻接顶点没有被访问，则继续从该节点出发开始上述的遍历过程，直到图的所有节点被访问完为止。

4、深度优先遍历相当于二叉树中的前序遍历

```
void _DFS(GraphLnk *g, T vertex, bool vistied[])
{
    printf("%c-->", vertex);
    int v = GetVertexPos(g, vertex);
    vistied[v] = true;

    int w = GetFirstNeighbor(g, vertex);
    while(w != -1)
    {
        if(!vistied[w])
        {
            _DFS(g, GetVertexValue(g, w), vistied);
        }
        w = GetNextNeighbor(g, vertex, GetVertexValue(g, w));
    }
}
```

```
void DFS(GraphLnk *g, T vertex)
{
    bool *vistied = (bool*)malloc(sizeof(bool) * g->NumVertexs);
    for(int i=0; i<g->NumVertexs; ++i)
        vistied[i] = false;

    _DFS(g, vertex, vistied);
    free(vistied);
}
```

2、广度优先遍历

遍历思想：

1、借助n个临时空间来标记节点是否被访问过

2、广度优先遍历相当于二叉树的层次遍历，因此需要借助一个队列

3、从顶点v开始访问，接着访问v顶点的各个未访问的邻接顶点w1,w2,....wn,然后依次访问w1,w2,....wn的所有未被访问过的邻接顶点，直到所有节点被访问完为止。

```
void BFS(GraphLnk *g, T vertex)
{
    int n = g->NumVertexs;
    bool *visited = (bool*)malloc(sizeof(bool) * n);
    assert(visited != NULL);
    for(int i=0; i<n; ++i)
        visited[i] = false;

    int v = GetVertexPos(g,vertex);
    printf("%c-->",vertex);
    visited[v] = true;

    std::queue<int> Q;
    Q.push(v);

    int w;
    while(!Q.empty())
    {
        v = Q.front();
        Q.pop();

        w = GetFirstNeighbor(g,GetVertexValue(g,v));
        while(w != -1)
```

```
        {
            if(!visited[w])
            {
                printf("%c-->",GetVertexValue(g,w));
                visited[w] = true;
                Q.push(w);
            }
            w = GetNextNeighbor(g,GetVertexValue(g,v),GetVertexValue(g,w));
        }
    }
    free(visited);
}
```

3、非连通图的遍历

```
void Components(GraphLnk *g)
{
    int n = g->NumVertices;
    bool *visited = (bool*)malloc(sizeof(bool) * n);
    assert(visited != NULL);
    for(int i=0; i<n; ++i)
    {
        visited[i] = false;
    }
    for(i=0; i<n; ++i)
    {
        if(!visited[i])
            DFS(g,i,visited);
    }
    free(visited);
}
```

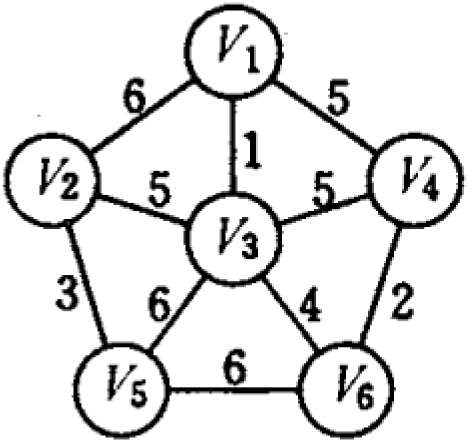
5、图的应用

> 📌 图的应用是历年考研考查的重点。图的应用主要包括：最小生成树、最短路径、拓扑排序和关键路径。一般而言，这部分内容直接以算法设计题形式考查的可能性很小，而更多的是结合图的实例来考查算法的具体操作过程，**所以图的应用要求同学至少掌握手工模拟给定图的各个算法的执行过程**。此外，还需掌握对给定模型建立相应的图去解决问题的方法。

1、最小生成树 - MinSpanTree

📌 Prim算法

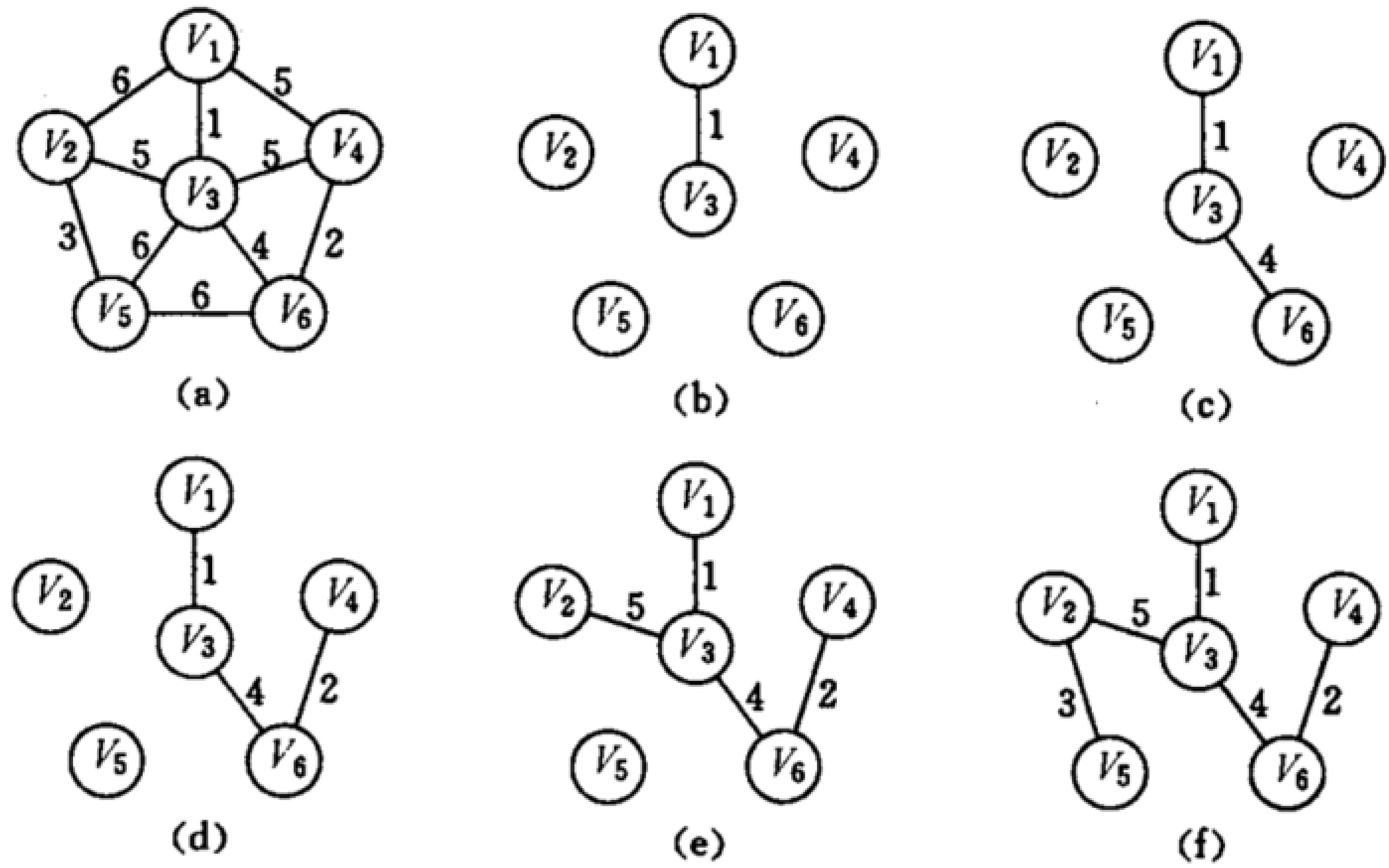
Kruskal算法



- Prim算法

📌
- Prim算法是从**顶点**的方面考虑构建一颗MST
- 构建思想为：设图G顶点集合为U，首先**任意**选择图G中的一点作为起始点a，将该点加入集合V，再从集合U-V中找到另一点b使得点b到V中任意一点的权值最小，此时将b点也加入集合V；以此类推，现在的集合V={a，b}，再从集合U-V中找到另一点c使得点c到V中任意一点的权值最小，此时将c点加入集合V，直至所有顶点全部被加入V，此时就构建出了一颗MST。
- 因为有N个顶点，所以该MST就有N-1条边，每一次向集合V中加入一个点，就意味着找到一条MST的边。

- Prim算法 - 最小生成树的构造过程

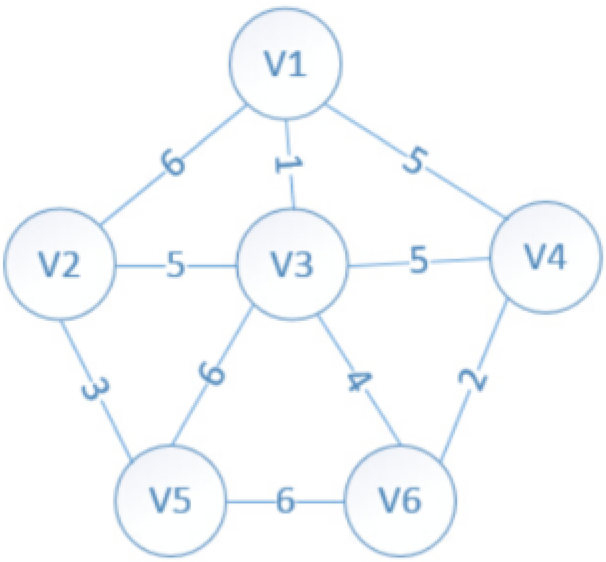


- Prim算法 - 具体视频讲解

https://www.bilibili.com/video/BV1bM411u7Ki?p=54&vd_source=e1686c82128572ae175af1318c920cde

- Prim算法 - 实现过程推演

1. **初始状态**



设置2个数据结构：

lowcost[i]:表示以i为终点的边的最小权值,当lowcost[i]=0，说明以i为终点的边的最小权值=0,表示i点加入了MST

mst[i]:表示对应lowcost[i]的起点，即说明边<mst[i],i>是MST的一条边，当mst[i]=0表示起点i加入MST

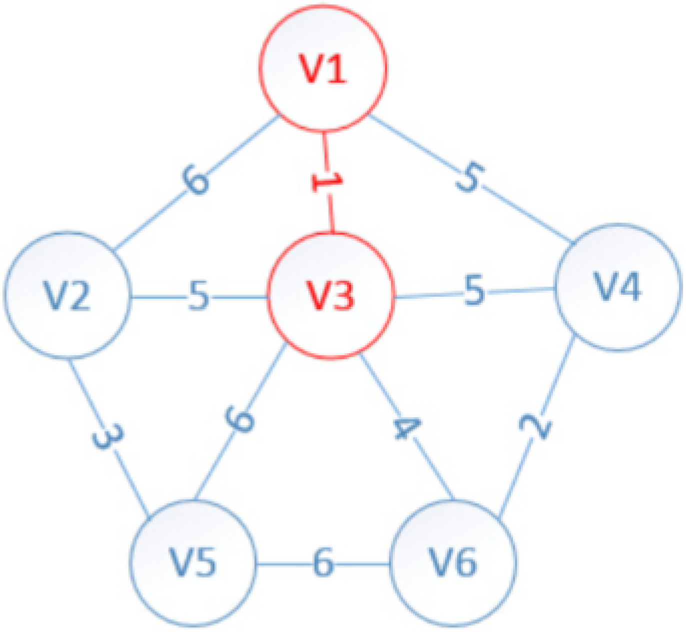
2. **假设V1作为起始点，进行初始化（*代表无限大）：**

lowcost[2]=6，lowcost[3]=1，lowcost[4]=5，lowcost[5]=*，lowcost[6]=*

mst[2]=1，　mst[3]=1，　mst[4]=1，　mst[5]=1，　mst[6]=1，（所有点默认起点是V1）

3. **合并v3顶点和(v1, v3)的边**

明显看出，以V3为终点的边的权值最小=1，所以边<mst[3],3>=1加入MST



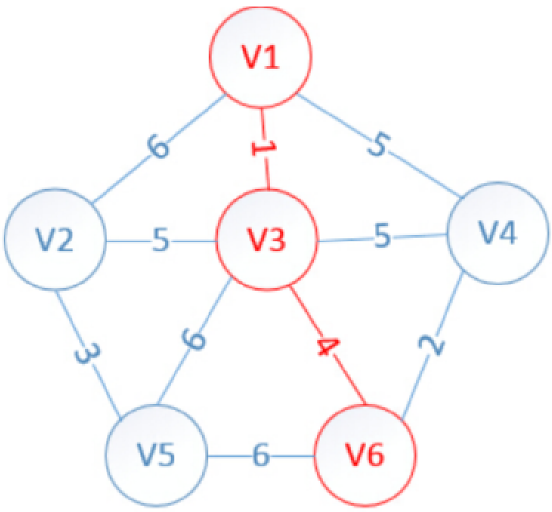
此时，因为点V3的加入，需要更新lowcost数组和mst数组：

lowcost[2]=5，lowcost[3]=0，lowcost[4]=5，lowcost[5]=6，lowcost[6]=4

mst[2]=3，　mst[3]=0，　mst[4]=1，　mst[5]=3，　mst[6]=3

4. **合并v6顶点和(v3, v6)的边**

明显看出，以V6为终点的边的权值最小=4，所以边<mst[6],6>=4加入MST

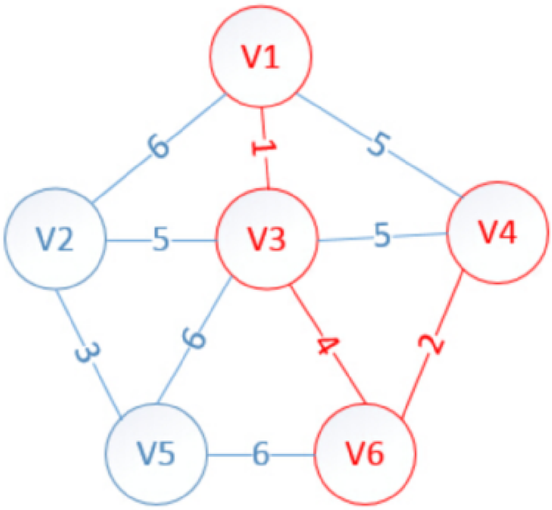


此时，因为点V6的加入，需要更新lowcost数组和mst数组：

lowcost[2]=5，lowcost[3]=0，lowcost[4]=2，lowcost[5]=6，lowcost[6]=0

mst[2]=3，　mst[3]=0，　mst[4]=6，　mst[5]=3，　mst[6]=0

5. **合并v4顶点和(v6, v4)的边**

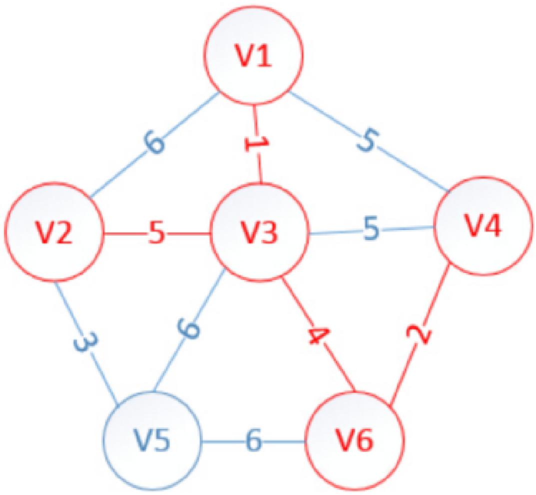


此时，因为点V4的加入，需要更新lowcost数组和mst数组：

lowcost[2]=5，lowcost[3]=0，lowcost[4]=0，lowcost[5]=6，lowcost[6]=0

mst[2]=3，　mst[3]=0，　mst[4]=0，　mst[5]=3，　mst[6]=0

6. **合并v2顶点和(v3, v2)的边**

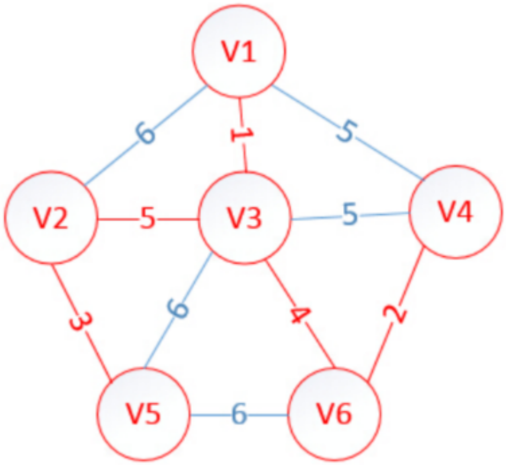


此时，因为点V2的加入，需要更新lowcost数组和mst数组：

lowcost[2]=0，lowcost[3]=0，lowcost[4]=0，lowcost[5]=3，lowcost[6]=0

mst[2]=0，　mst[3]=0，　mst[4]=0，　mst[5]=2，　mst[6]=0

7. **合并v5顶点和(v2, v5)的边**



很明显，以V5为终点的边的权值最小=3，所以边<mst[5],5>=3加入MST

lowcost[2]=0，lowcost[3]=0，lowcost[4]=0，lowcost[5]=0，lowcost[6]=0

mst[2]=0， mst[3]=0， mst[4]=0， mst[5]=0， mst[6]=0

至此，MST构建成功.

- Prim算法 - 代码实现（最低要求能够理解代码求解MST的过程）

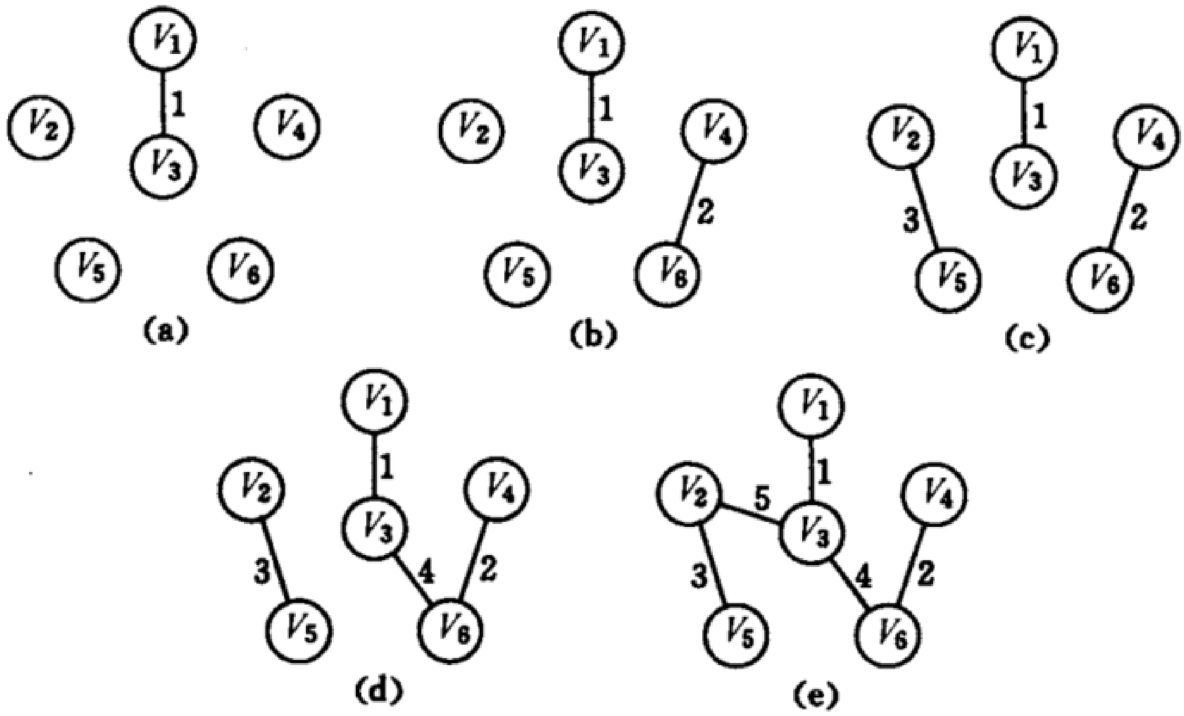
```
void MinSpanTree_Prim(GraphMtx *g, T vertex)
{
    int n = g->NumVertices;
    E *lowcost = (E*)malloc(sizeof(E)*n); //lowcost[n]
    int *mst = (int *)malloc(sizeof(int)*n);//mst[n]
    assert(lowcost!=NULL && mst!=NULL);

    int k = GetVertexPos(g,vertex);

    for(int i=0; i<n; ++i)
    {
        if(i != k)
        {
            lowcost[i] = GetWeight(g,k,i);
            mst[i] = k;
        }
        else
        {
            lowcost[i] = 0;
        }
    }

    int min,min_index;
    int begin,end;
    E cost;
```

```
    for(i=0; i<n-1; ++i)
    {
        min = MAX_COST;
        min_index = -1;
        for(int j=0; j<n; ++j)
        {
            if(lowcost[j]!=0 && lowcost[j]<min)
            {
                min = lowcost[j];
                min_index = j;
            }
        }
        begin = mst[min_index];
        end = min_index;
        printf("%c-->%c : %d\n",g->VerticesList[begin],g->VerticesList[end],min)

        lowcost[min_index] = 0;

        for(j=0; j<n; ++j)
        {
            cost = GetWeight(g,min_index,j);
            if(cost < lowcost[j])
            {
                lowcost[j] = cost;
                mst[j] = min_index;
            }
        }
    }
}
```

- Kruskal算法

> 📌 Kruskal算法是从**边**的方面考虑构建一颗MST
>
> 每次挑选权值最小的边（不能形成回路），最后让所有顶点形成一个连通分量，即为最小生成树
>
> 挑选权值最小的边的关键是：1、对边进行排序 2、通过并查集进行选择不在同一联通分量的边

- Kruskal算法 - 最小生成树的构造过程



- Kruskal算法 - 具体视频讲解

https://www.bilibili.com/video/BV1bM411u7Ki?p=55&vd_source=e1686c82128572ae175af1318c920cde

- Kruskal算法 - 代码实现

```
typedef struct Edge
{
    int x; // start
    int y; // end
    E   cost;
}Edge;
/////////////////////////////////////////////////////////////////////////
int cmp(const void*a, const void *b)
{
    return (*(Edge*)a).cost - (*(Edge*)b).cost;
}
bool Is_same(int *father, int i, int j)
{
    while(father[i] != i)
    {
        i = father[i];
    }
    while(father[j] != j)
    {
        j = father[j];
    }
    return i==j;
}
```

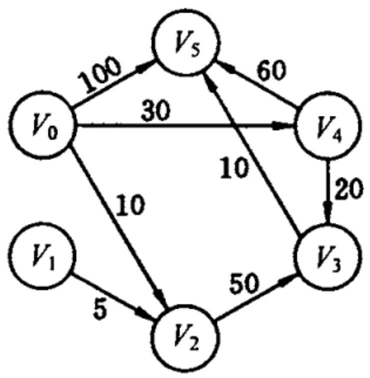
```c
void Mark_same(int *father, int i, int j)
{
    while(father[i] != i)
    {
        i = father[i];
    }
    while(father[j] != j)
    {
        j = father[j];
    }
    father[j] = i;
}

void MinSpanTree_Kruskal(GraphMtx *g)
{
    int n = g->NumVertices;
    Edge *edge = (Edge *)malloc(sizeof(Edge) * (n*(n-1)/2));
    assert(edge != NULL);

    int k = 0;
    for(int i=0; i<n; ++i)
    {
        for(int j=i; j<n; ++j)
        {
            if(g->Edge[i][j]!=0 && g->Edge[i][j]!=MAX_COST)
            {
                edge[k].x = i;
                edge[k].y = j;
                edge[k].cost = g->Edge[i][j];
                k++;
            }
        }
    }

    int v1,v2;

    qsort(edge,k,sizeof(Edge),cmp);

    int *father = (int*)malloc(sizeof(int) * n);
    assert(father != NULL);
    for(i=0; i<n; ++i)
    {
        father[i] = i;
    }

    for(i=0; i<n; ++i)
    {
```

```
        if(!Is_same(father,edge[i].x,edge[i].y))
        {
            v1 = edge[i].x;
            v2 = edge[i].y;
            printf("%c-->%c : %d\n",g->VerticesList[v1],g->VerticesList[v2],edge
            Mark_same(father,edge[i].x,edge[i].y);
        }
    }
}
```

2、最短路径

📌 Dijkstra算法

Floyd算法

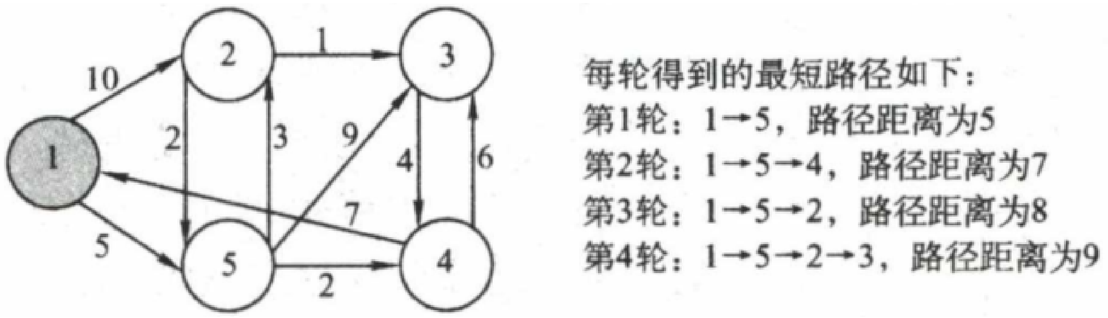


- Dijkstra算法

📌 用来求从一个顶点到其他所有顶点的最短路径

终点	从 v_0 到各终点的 D 值和最短路径的求解过程				
	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
v_1	∞	∞	∞	∞	∞ 无
v_2	10 (v_0, v_2)				
v_3	∞	60 (v_0, v_2, v_3)	50 (v_0, v_4, v_3)		
v_4	30 (v_0, v_4)	30 (v_0, v_4)			
v_5	100 (v_0, v_5)	100 (v_0, v_5)	90 (v_0, v_4, v_5)	60 (v_0, v_4, v_3, v_5)	
v_j	v_2	v_4	v_3	v_5	
S	$\{v_0, v_2\}$	$\{v_0, v_2, v_4\}$	$\{v_0, v_2, v_3, v_4\}$	$\{v_0, v_2, v_3, v_4, v_5\}$	v_0, v_2, v_3, v_4, v_5 v_1

- Dijkstra算法 - 最短距离求解练习



- Dijkstra算法 - 具体视频讲解

https://www.bilibili.com/video/BV1bM411u7Ki?p=58&vd_source=e1686c82128572ae175af1318c920cde

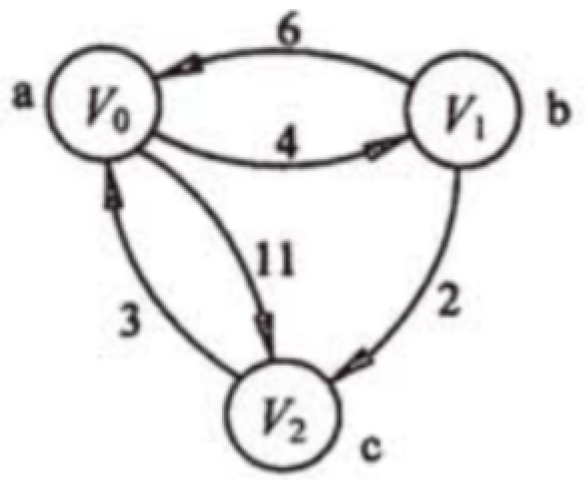
- Dijkstra算法 - 代码实现

```
void ShortestPath(GraphMtx *g, T vertex, E dist[], int path[])
{
    int n = g->NumVertices;
    bool *S = (bool*)malloc(sizeof(bool) * n);
    assert( S != NULL);
    int v = GetVertexPos(g,vertex);

    for(int i=0; i<n; ++i)
    {
        dist[i] = GetWeight(g,v,i);
        S[i] = false;
        if(i!=v && dist[i]<MAX_COST)
        {
```

```
                path[i] = v;
            }
            else
            {
                path[i] = -1;
            }
        }

        S[v] = true;
        int min;
        int w;

        for(i=0; i<n-1; ++i)
        {
            min = MAX_COST;
            int u = v;
            for(int j=0; j<n; ++j)
            {
                if(!S[j] && dist[j]<min)
                {
                    u = j;
                    min = dist[j];
                }
            }

            S[u] = true;
            for(int k=0; k<n; ++k)
            {
                w = GetWeight(g,u,k);
                if(!S[k] && w<MAX_COST && dist[u]+w<dist[k])
                {
                    dist[k] = dist[u]+w;
                    path[k] = u;
                }
            }
        }
    }
```

- Floyd算法

> 📌 用来求解每一对顶点之间的最短路径



D	$D^{(-1)}$			$D^{(0)}$			$D^{(1)}$			$D^{(2)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0	0	4	11	0	4	11	0	4	6	0	4	6
1	6	0	2	6	0	2	6	0	2	5	0	2
2	3	∞	0	3	7	0	3	7	0	3	7	0
P	$P^{(-1)}$			$P^{(0)}$			$P^{(1)}$			$P^{(2)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0		AB	AC		AB	AC		AB	ABC		AB	ABC
1	BA		BC	BA		BC	BA		BC	BCA		BC
2	CA			CA	CAB		CA	CAB		CA	CAB	

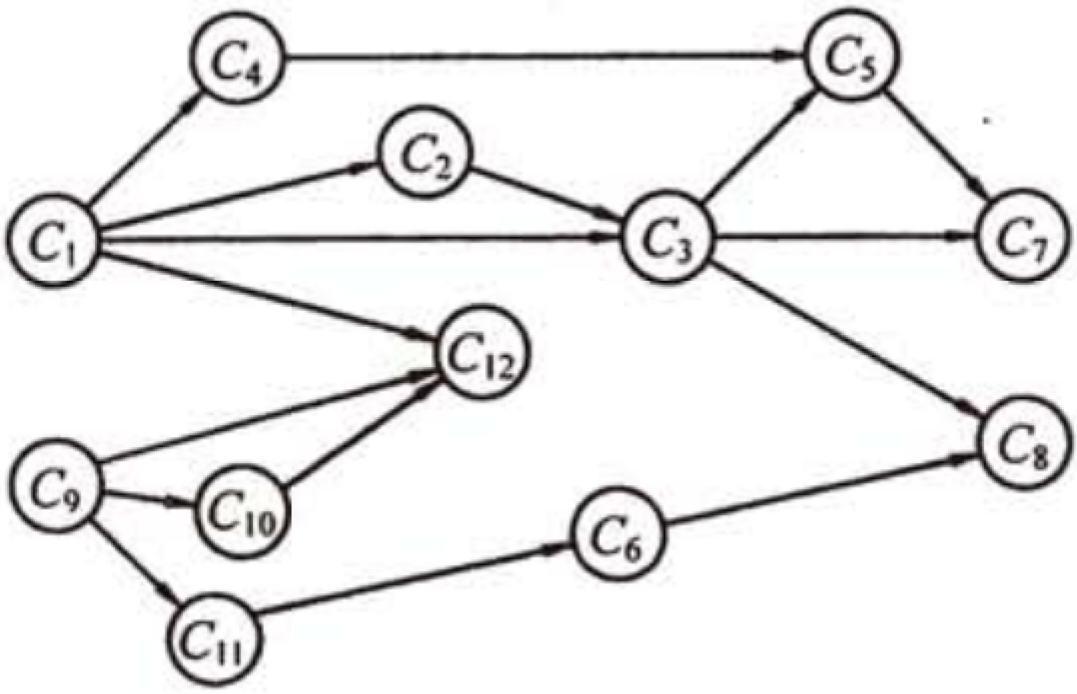
3、拓扑排序

📌 拓扑排序有两个用途：

1、判断图是否有回路

2、用来规定顶点活动的开展顺序

【注意】：拓扑排序的结果可能不唯一



- 拓扑排序 - 具体视频讲解

https://www.bilibili.com/video/BV1bM411u7Ki?p=56&vd_source=e1686c82128572ae175af1318c920cde

- 拓扑排序 - 代码实现

```
void TopologicalSort(GraphLnk *g)
{
    int n = g->NumVertices;
    int *count = (int *)malloc(sizeof(int)*n);
    assert(count != NULL);
    for(int i=0; i<n; ++i)
    {
        count[i] = 0;
    }

    Edge *p;
    for(i=0; i<n; ++i)
    {
        p = g->NodeTable[i].adj;
        while(p != NULL)
        {
            count[p->dest]++;
            p = p->link;
        }
    }

    int top = -1;
    for(i=0; i<n; ++i)
```

```
    {
        if(count[i] == 0)
        {
            count[i] = top;    //Push
            top = i;
        }
    }

    int v,w;
    for(i=0; i<n; ++i)
    {
        if(top == -1)
        {
            printf("网络中有回路.\n");
            return;
        }
        else
        {
            v = top;             //Pop
            top = count[top];
            printf("%c-->",g->NodeTable[v]);
            w = GetFirstNeighbor(g,g->NodeTable[v].data);
            while(w != -1)
            {
                if(--count[w] == 0)
                {
                    count[w] = top;
                    top = w;
                }
                w = GetNextNeighbor(g,g->NodeTable[v].data,g->NodeTable[w].data)
            }
        }
    }
    free(count);
}
```

4、关键路径

📌 关键路径的求解步骤：

1、**正向**求解每个顶点的**最早**开始时间 - 找**最大**时间

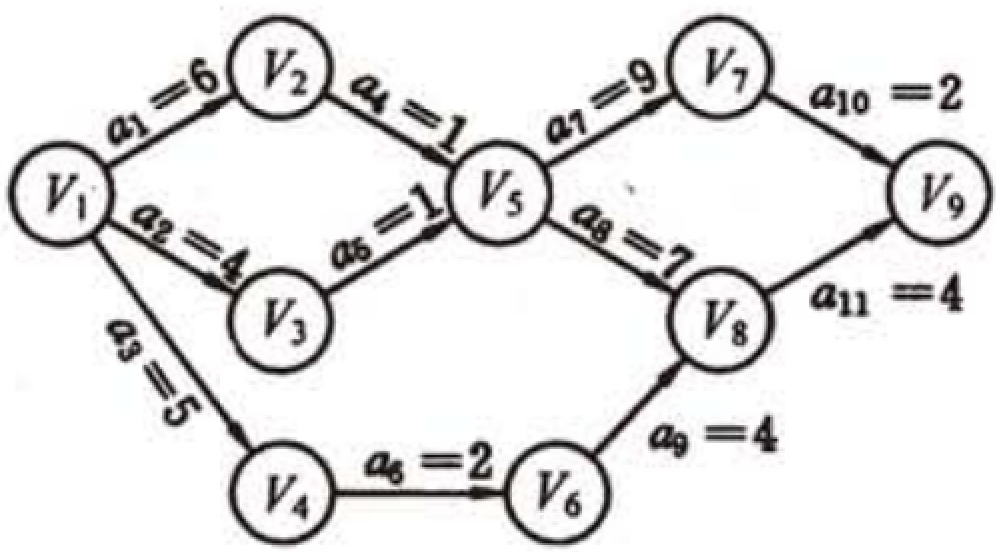
2、**逆向**求解每个顶点的**最晚**开始时间 - 找**最小**时间

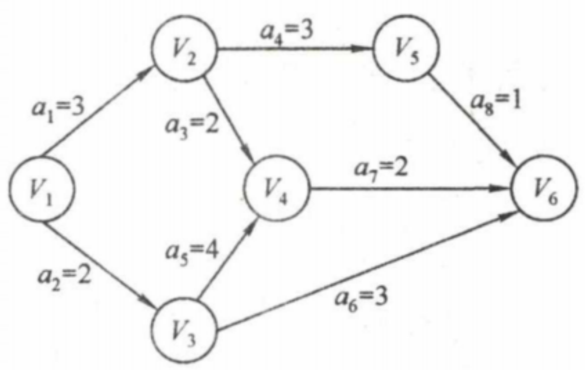
3、判断顶点的最早开始时间和最晚开始时间**是否相等**，相等的顶点即为关键路径

📌 关键路径的求解可以分为两种方式：

1、以**顶点**的方式求解（也称为通过**事件**求解）

2、以**边**的方式求解 （也称为通过**活动**求解）





	v_1	v_2	v_3	v_4	v_5	v_6
$ve(i)$	0	3	2	6	6	8
$vl(i)$	0	4	2	6	7	8

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(i)$	0	0	3	3	2	2	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	3	0	1

图 6.23　求解关键路径的过程

- 关键路径 - 具体视频讲解

https://www.bilibili.com/video/BV1bM411u7Ki?p=57&vd_source=e1686c82128572ae175af1318c920cde

- 关键路径 - 代码实现

```
void CriticalPath(GraphMtx *g)
{
    int n = g->NumVertices;
    int *ve = (int*)malloc(sizeof(int) * n);
    int *vl = (int*)malloc(sizeof(int) * n);
    assert(ve!=NULL && vl!=NULL);
```

```

    for(int i=0; i<n; ++i)
    {
        ve[i] = 0;
        vl[i] = MAX_COST;
    }

    int j,w;
    //ve
    for(i=0; i<n; ++i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            w = GetWeight(g,i,j);
            if(ve[i]+w > ve[j])
            {
                ve[j] = ve[i]+w;
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }

    //vl
    vl[n-1] = ve[n-1];
    for(i=n-2; i>0; --i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            w = GetWeight(g,i,j);
            if(vl[j]-w < vl[i])
            {
                vl[i] = vl[j]-w;
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }

    int Ae, Al;
    for(i=0; i<n; ++i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            Ae = ve[i];
            Al = vl[j] - GetWeight(g,i,j);
```

```
            if(Ae == Al)
            {
                printf("<%c,%c>是关键路劲.\n",g->VerticesList[i],g->VerticesList[
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }

    free(ve);
    free(vl);
}
```