

圖



图的应用

无向网的应用-----最小生成树

Prim算法

Kruskal算法

有向网的应用

单源点最短路径-----Dijkstra算法

每对顶点间的最短路径-----Floyd算法

有向无环图的应用

AOV网的应用-----拓扑排序

AOE网的应用-----关键路径

图的应用

最小生成树



最小生成树的定义



Prim算法



Kruskal算法

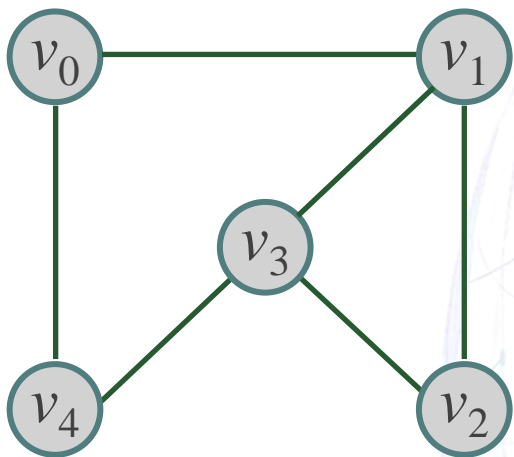
生成树的定义

📌 生成树：连通图的生成树是包含全部顶点的一个极小连通子图

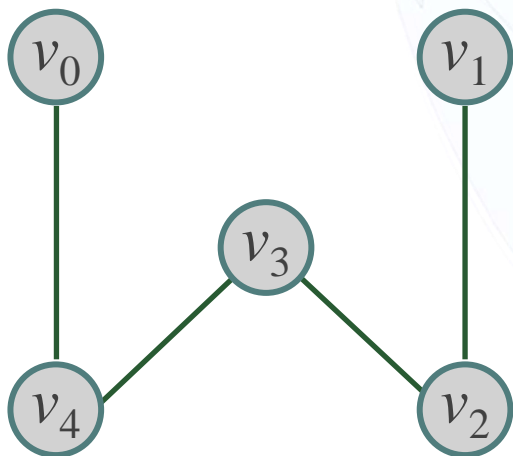
多——构成回路
少——不连通

看一个生成树实例

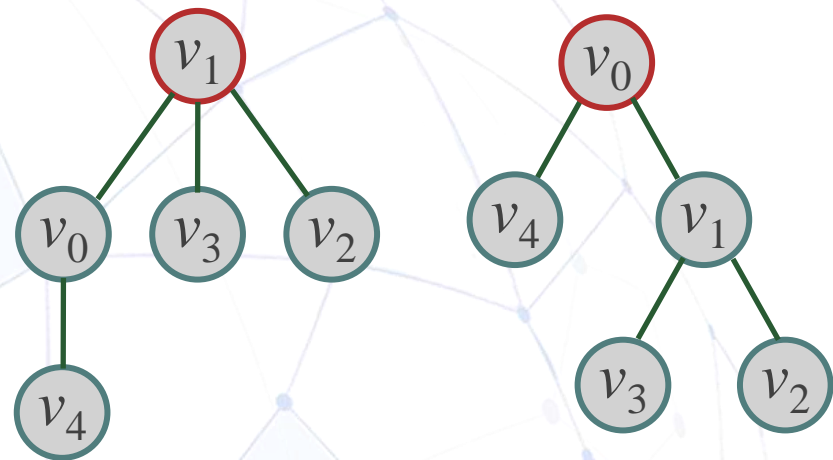
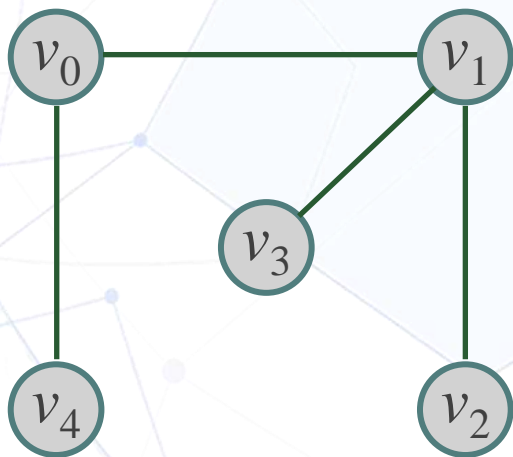
生成树特点



深度遍历获得的生成树 ↓



广度遍历获得的生成树



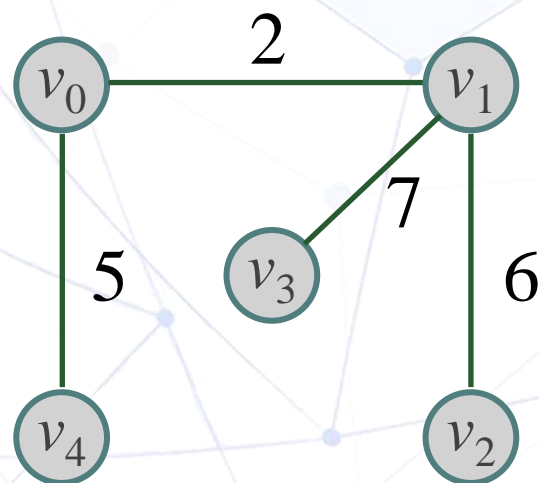
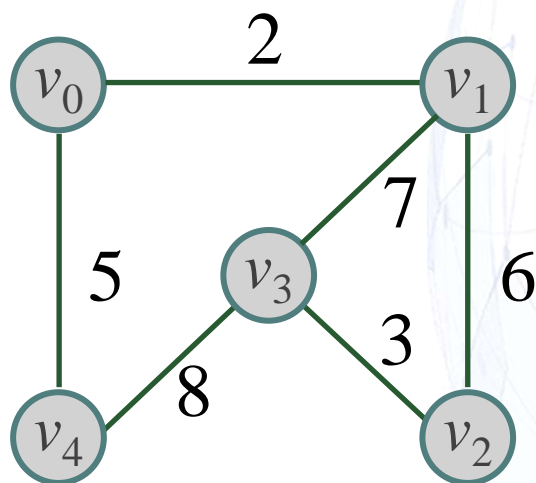
➤ 一个图可以有許多棵不同的生成树

- 生成树的顶点个数与图的顶点个数相同;
- 一个有 n 个顶点的连通图的生成树有 $n-1$ 条边;
- 在生成树中再加一条边必然形成回路。

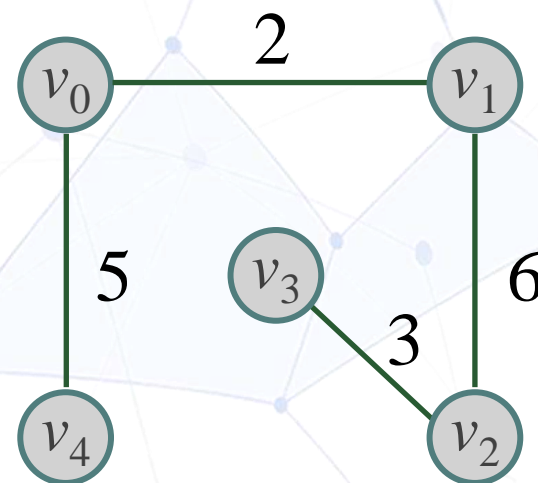
最小生成树的定义

✦ 生成树的代价：在无向**连通网**中，生成树上各边的权值之和

✦ 最小生成树：在无向**连通网**中，代价最小的生成树



生成树的代价： 20

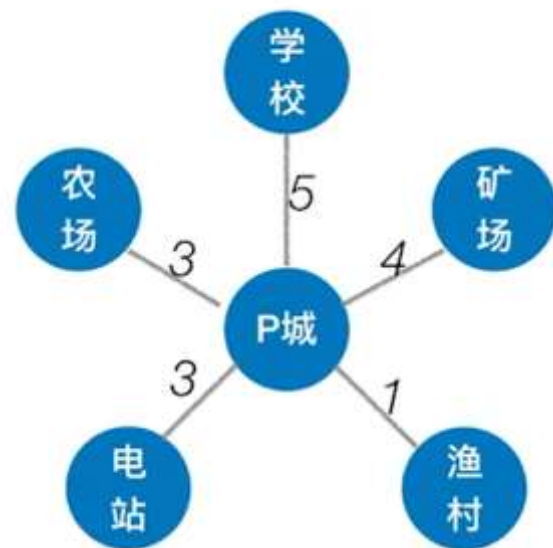


生成树的代价： 16

- 最小生成树可能有多个，但边的权值之和总是唯一且最小的
- 最小生成树的边数 = 顶点数 - 1。砍掉一条则不连通，增加一条边则会出现回路

最小生成树

- 如果一个连通图本身就是一棵树，则其最小生成树就是它本身
- 只有连通图才有生成树，非连通图只有生成森林



最小生成树的应用

在 n 个城市之间建造通信网络，至少要架设 $n-1$ 条通信线路，而每两个城市之间架设通信线路的造价是不一样的，那么如何设计才能使得总造价最小？

求最小生成树


Prim 算法

Kruskal 算法

最小生成树-Prim（普利姆）算法

 Prim算法——基本思想

 Prim算法——存储结构

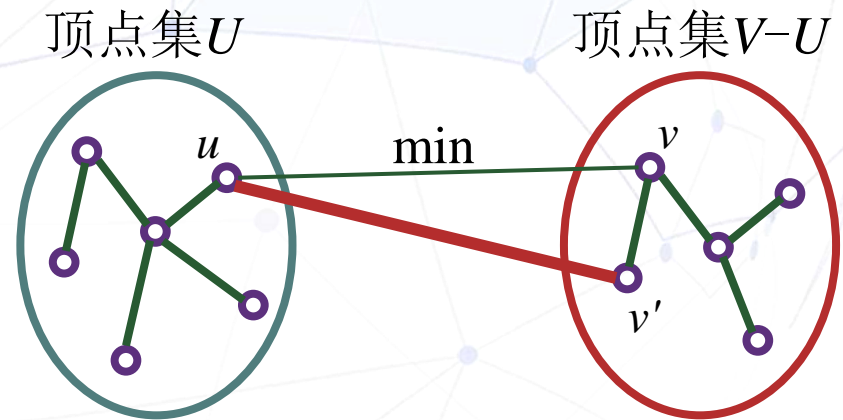
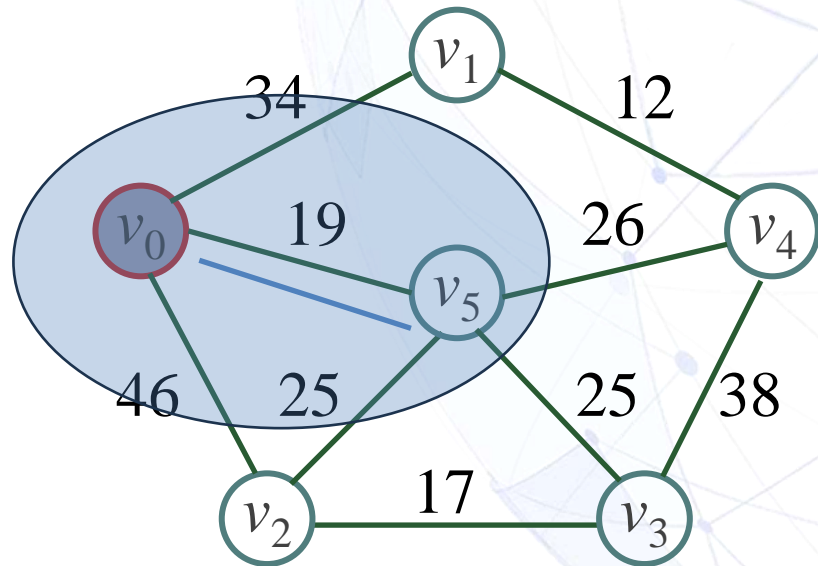
 Prim算法——算法描述

 Prim算法——性能分析

Prim算法:

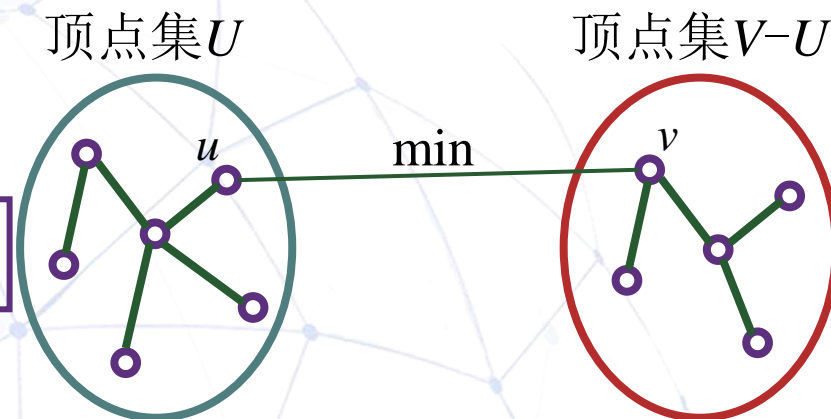
从某一个顶点开始构建生成树，每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。

从顶点集合角度解决问题



基本思想

关键：是如何找到连接 U 和 $V-U$ 的最短边



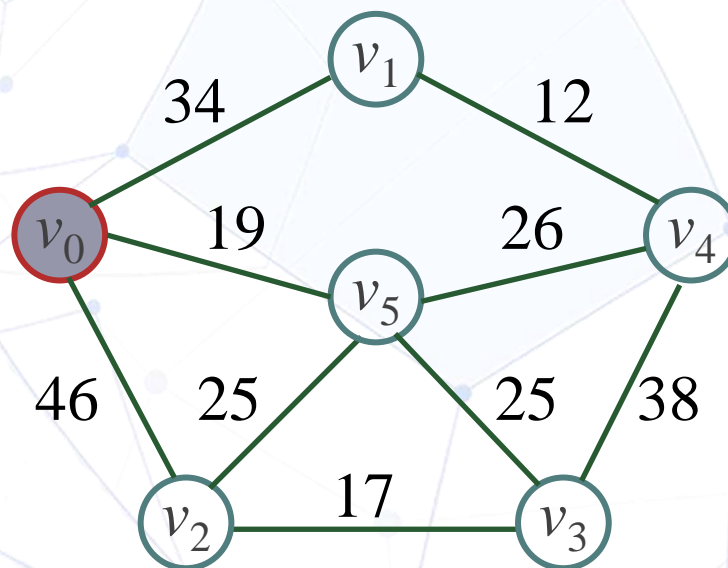
算法：Prim

输入：无向连通网 $G=(V, E)$

输出：最小生成树 $T=(U, TE)$

1. 初始化： $U = \{v\}$; $TE = \{ \}$;
2. 重复下述操作直到 $U = V$:
 - 2.1 在 E 中寻找最短边 (i, j) ，且满足 $j \in U, i \in V-U$;
 - 2.2 $U = U + \{i\}$;
 - 2.3 $TE = TE + \{(i, j)\}$;

基本思想



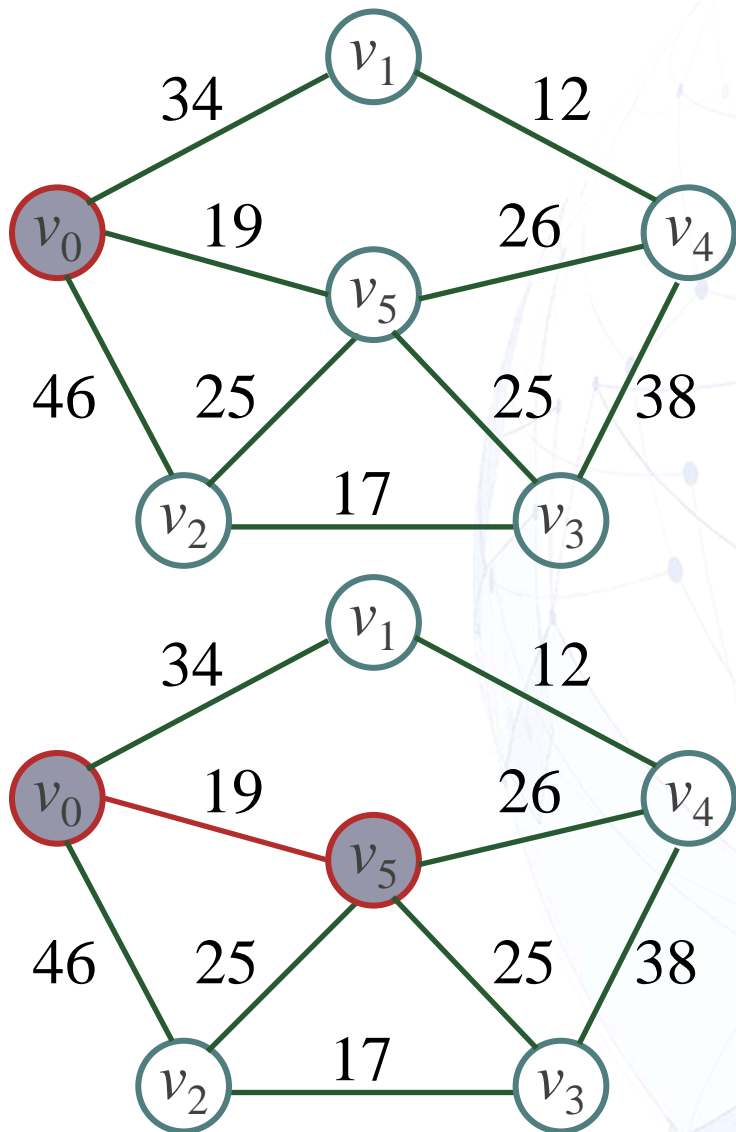
关键：是如何找到连接 U 和 $V-U$ 的最短边？

U ：涂色

$V-U$ ：尚未涂色

方法：一个顶点涂色、另一个顶点尚未涂色的最短边

运行实例



初始化:

$$U = \{v_0\}$$

$$V - U = \{v_1, v_2, v_3, v_4, v_5\}$$

$$\text{cost} = \{(v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, (v_0, v_5)19\}$$

第一次迭代:

$$U = \{v_0, v_5\}$$

$$V - U = \{v_1, v_2, v_3, v_4\}$$

$$\text{cost} = \{(v_0, v_1)34, (v_5, v_2)25, (v_5, v_3)25, (v_5, v_4)26\}$$

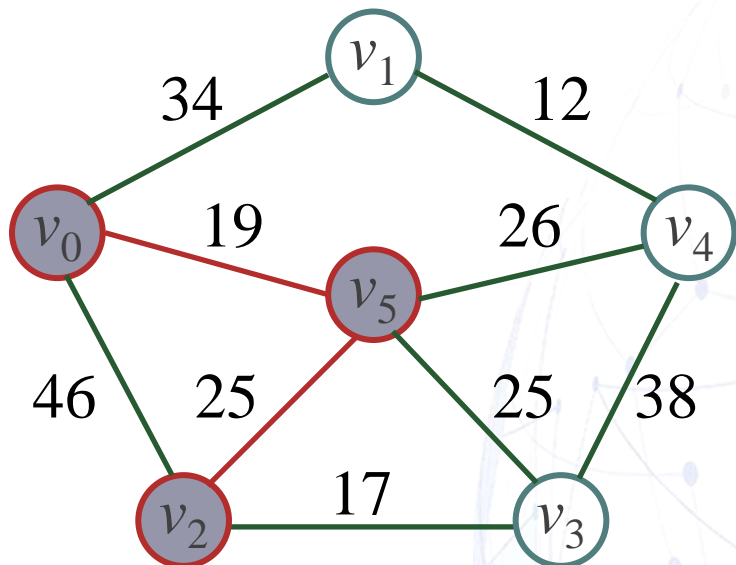
注意: v_5 加入 U 后, 要考虑更新 $V-U$ 中的点与生成树相连的代价

(1)有可能不影响, 比如 v_1 ;

(2)也可能影响与 v_0 连接的点, 比如 v_2 ;

(3)也有可能影响与 v_0 无连接的点, 比如 v_3 和 v_4

运行实例



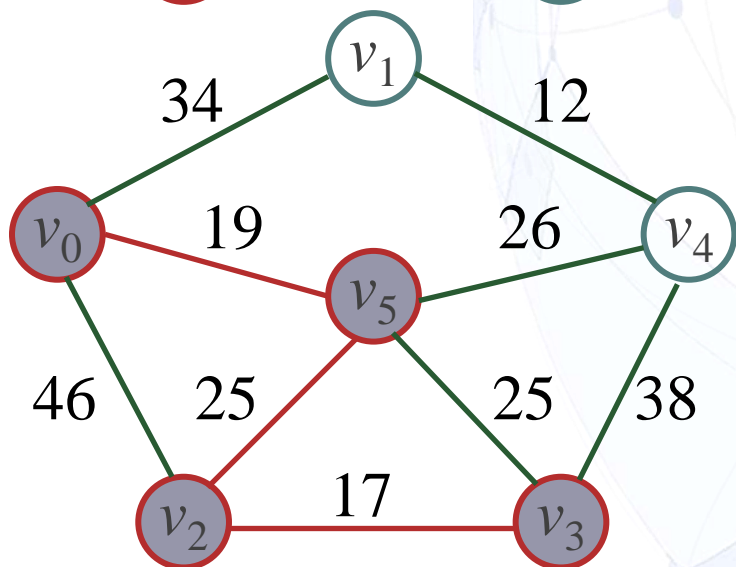
第二次迭代:

$$U = \{v_0, v_5, v_2\}$$

$$V - U = \{v_1, v_3, v_4\}$$

$$\text{cost} = \{(v_0, v_1)34, (v_2, v_3)17, (v_5, v_4)26\}$$

v_2 加入到 U 后, v_3 与 U 中 v_5 连的代价比与 v_2 连的代价大, 所以要更新 v_3 的代价



第三次迭代:

$$U = \{v_0, v_5, v_2, v_3\}$$

$$V - U = \{v_1, v_4\}$$

$$\text{cost} = \{(v_0, v_1)34, (v_5, v_4)26\}$$

v_3 加入到 U 后, v_1 与 v_4 与 U 中结点的代价没有发生变化, 直接选最小的 cost 的顶点 v_4 加入到 U

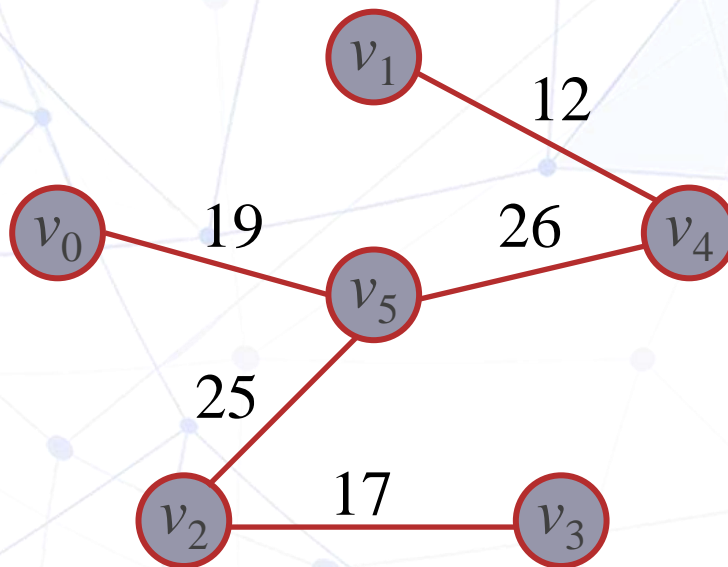
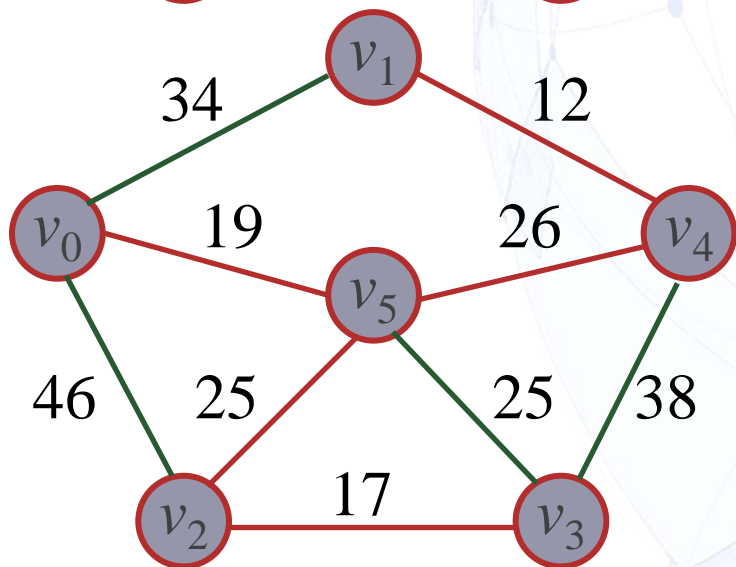
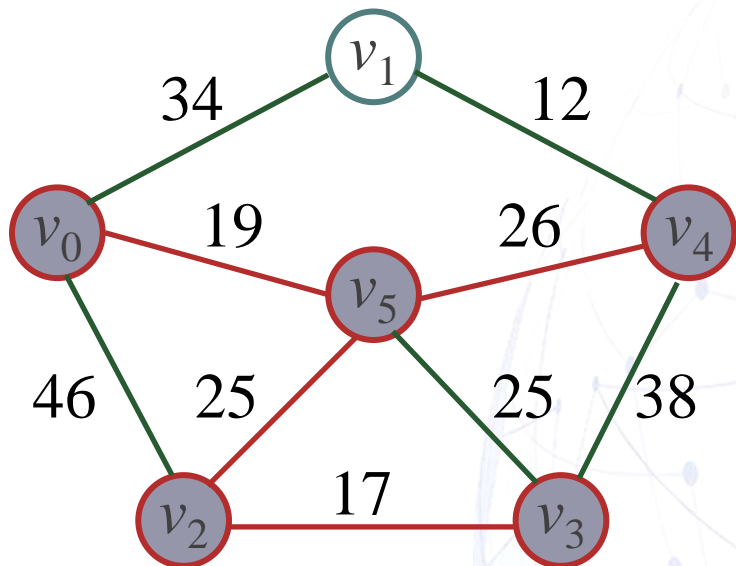
运行实例

第四次迭代:

$$U = \{v_0, v_5, v_2, v_3, v_4\}$$

$$V - U = \{v_1\}$$

$$\text{cost} = \{(v_4, v_1)12\}$$



存储结构



图采用什么存储结构呢？

需要不断读取任意两个顶点之间边的权值



图采用邻接矩阵存储



如何存储候选最短边集（连接 U 和 $V-U$ 的候选最短边）？

存储结构

🕒 如何存储候选最短边集（连接 U 和 $V-U$ 的候选最短边）？

U 代表的是已处于生成树里的顶点集合（即已连通的顶点集合），
 $U-V$ 代表还没有纳入生成树的顶点集合

数组adjvex[n]：表示候选最短边的邻接点

数组lowcost[n]：表示候选最短边的权值

$$\begin{cases} \text{adjvex}[i] = j \\ \text{lowcost}[i] = w \end{cases}$$

此处 $j \in U$, $i \in U-V$, 表示从未纳入生成树的顶点 i 去找与已在生成树中的结点存在最短边的顶点 j 。候选最短边 (i, j) 的权值为 w 。

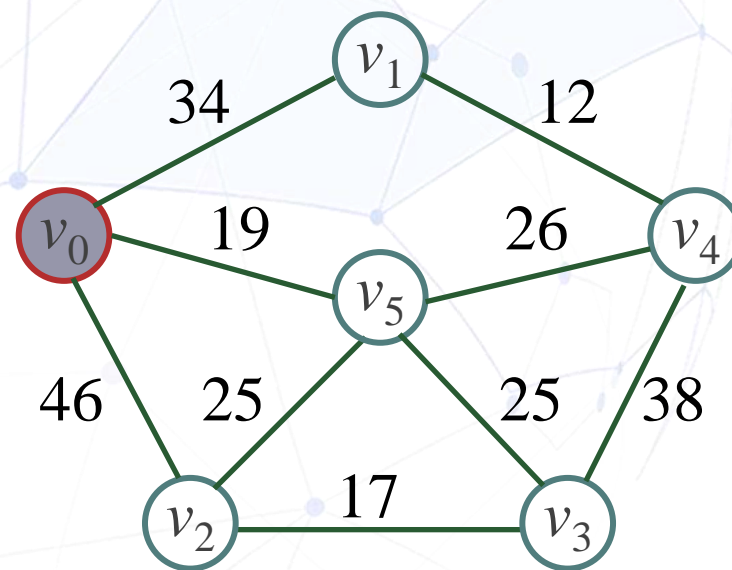
存储结构

初始时, $\text{lowcost}[v] = 0$, 表示将顶点 v 加入集合 U 中;

$\text{adjvex}[i] = v$, $\text{lowcost}[i] = \text{edge}[v][i]$ ($0 \leq i \leq n-1$)

例如: $\{(v_0, v_0)0, (v_0, v_1)34, (v_0, v_2)46, (v_0, v_3)\infty, (v_0, v_4)\infty, (v_0, v_5)19\}$

	0	1	2	3	4	5
$\text{adjvex}[n] =$	0	0	0	0	0	0
$\text{lowcost}[n] =$	0	34	46	∞	∞	19



存储结构

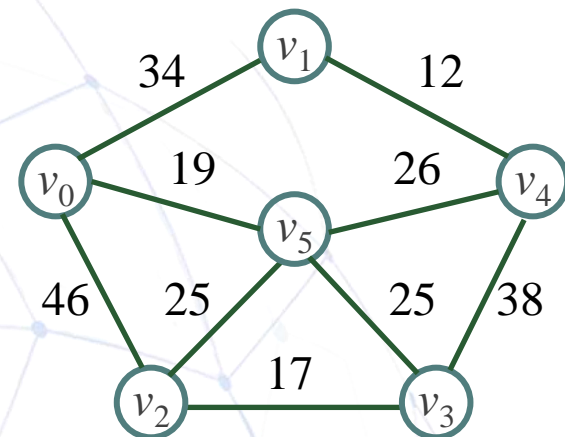
每一次迭代，假设数组 $\text{lowcost}[n]$ 中的最小权值是 $\text{lowcost}[j]$ ，则
令 $\text{lowcost}[j] = 0$ ，表示将顶点 j 加入集合 U 中；

由于顶点 j 从集合 $V-U$ 进入集合 U ，候选最短边集发生变化，需要更新：

$$\begin{cases} \text{lowcost}[i] = \min\{\text{lowcost}[i], \text{edge}[i][j]\} \\ \text{adjvex}[i] = j \text{ (如果 } \text{edge}[i][j] < \text{lowcost}[i] \text{)} \end{cases} \quad (0 \leq i \leq n-1)$$

算法描述

下标 顶点V	0	1	2	3	4	5		
	v_0	v_1	v_2	v_3	v_4	v_5	U	
adjvex	0	0	0	0	0	0		
lowcost	0	34	46	∞	∞	19	v_0	



```
void MGraph<DataType> :: Prim(int v)
```

/*假设从顶点v出发*/

```
{
```

```
    int i, j, k, adjvex[MaxSize], lowcost[MaxSize];
```

```
    for (i = 0; i < vertexNum; i++)
```

/*初始化辅助数组shortEdge*/

```
    {
```

```
        lowcost[i] = edge[v][i]; adjvex[i] = v;
```

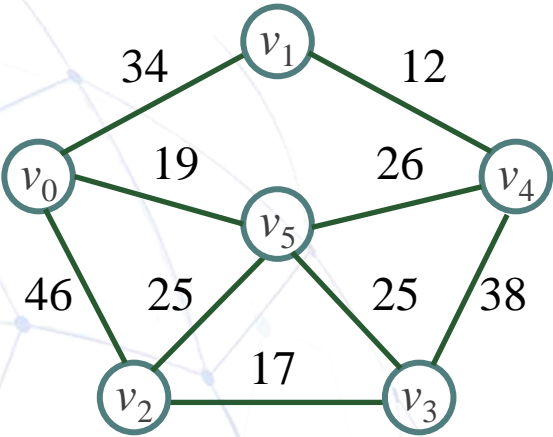
```
    }
```

```
    lowcost[v] = 0;
```

/*将顶点v加入集合U*/

算法描述

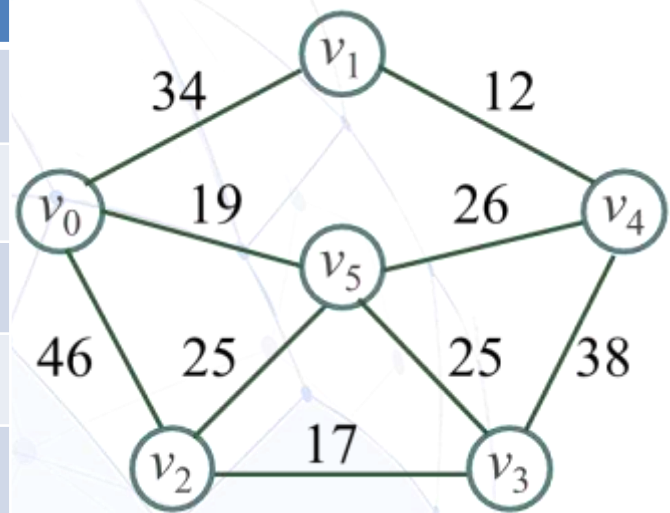
下标 顶点V	0	1	2	3	4	5	U	最小边及权值
adjvex	0	0	0	0	0	0		
lowcost	0	34	46	∞	∞	19	v_0	$(v_0, v_5)19$
adjvex		0	5	5	5	0		
lowcost		34	25	25	26	0	v_0, v_5	



```
for (k = 1; k < vertexNum; i++) /*迭代n-1次*/
{
    j = MinEdge(lowcost, vertexNum) /*寻找最短边的邻接点j*/
    cout << j << adjvex[j] << lowcost[j];
    lowcost[j] = 0;
    for (i = 0; i < vertexNum; i++) /*调整数组*/
        if (edge[i][j] < lowcost[i]) {
            lowcost[i] = edge[i][j]; adjvex[i] = j;
        }
}
```

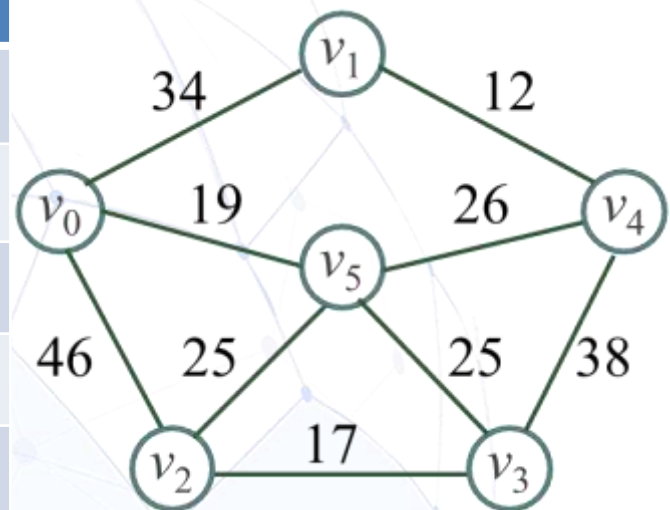
求解过程

迭代次数	加入TE的边	边长	加入U的顶点{0}	i	0	1	2	3	4	5
				lowcost	0	34	46	∞	∞	19
				adjvex	0	0	0	0	0	0
1				lowcost						
				adjvex						
2				lowcost						
				adjvex						
3				lowcost						
				adjvex						
4				lowcost						
				adjvex						
5				lowcost						
				adjvex						



求解过程


迭代次数	加入TE的边	边长	加入U的顶点{0}	i	0	1	2	3	4	5
				lowcost	0	34	46	∞	∞	19
				adjvex	0	0	0	0	0	0
1	(0,5)	19	5	lowcost	0	34	25	25	26	0
				adjvex	0	0	5	5	5	0
2	(5,2)	25	2	lowcost	0	34	0	17	26	0
				adjvex	0	0	5	2	5	0
3	(2,3)	17	3	lowcost	0	34	0	0	26	0
				adjvex	0	0	5	2	5	0
4	(5,4)	26	4	lowcost	0	12	0	0	0	0
				adjvex	0	4	5	2	5	0
5	(4,1)	12	1	lowcost	0	0	0	0	0	0
				adjvex	0	4	5	2	5	0



算法描述

```
void Prim(MGraph *G, int v)
{
    int i, j, k, adjvex[MaxSize], lowcost[MaxSize];
    for (i = 0; i < G->vertexNum; i++)
    {
        lowcost[i] = G->edge[v][i]; adjvex[i] = v; }  $O(n)$ 
    }
    lowcost[v] = 0;
    for (k = 1; k < G->vertexNum; k++)
    {
        j = MinEdge(lowcost, G->vertexNum)
        printf("(%d, %d)%d ", j, adjvex[j], lowcost[j]); lowcost[j] = 0; }  $O(n)$ 
        for (i = 0; i < G->vertexNum; i++) }  $O(n)$ 
            if G->edge[i][j] < lowcost[i] {
                lowcost[i] = G->edge[i][j]; adjvex[i] = j;
            }
    }
}
```

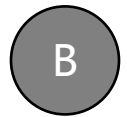
🕒 时间复杂度? $\Rightarrow O(n^2)$



1. 无向图的生成树是该图的一个极小连通子图。



正确



错误

提交

2. Prim算法采用（ ）作为图的存储结构。

- ☐ A 顺序存储
- ☐ B 链接存储
- ☒ C 邻接矩阵
- ☐ D 邻接表

提交

3. 对于如图6-8所示无向连通图，最小生成树的代价是（ ）。

- ☐ A 15
- ☒ B 17
- ☐ C 19
- ☐ D 20

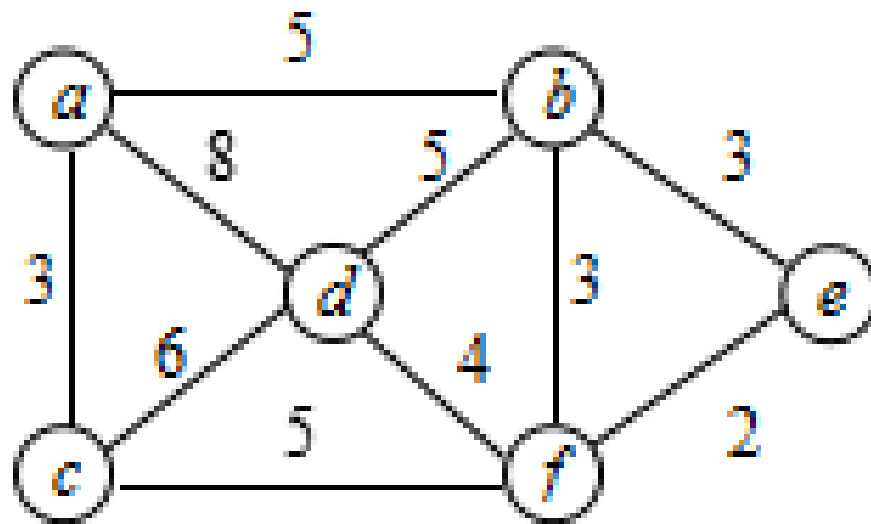


图 6-8 一个无向连通网

提交

4. 对于如图6-8所示无向连通图，从顶点d出发用Prim算法构造最小生成树，加入最小生成树的第4条边是（ ）。

- ☐ A (f, e)2
- ☐ B (b, e)3
- ☐ C (f, b)3
- ☒ D (b, a)5

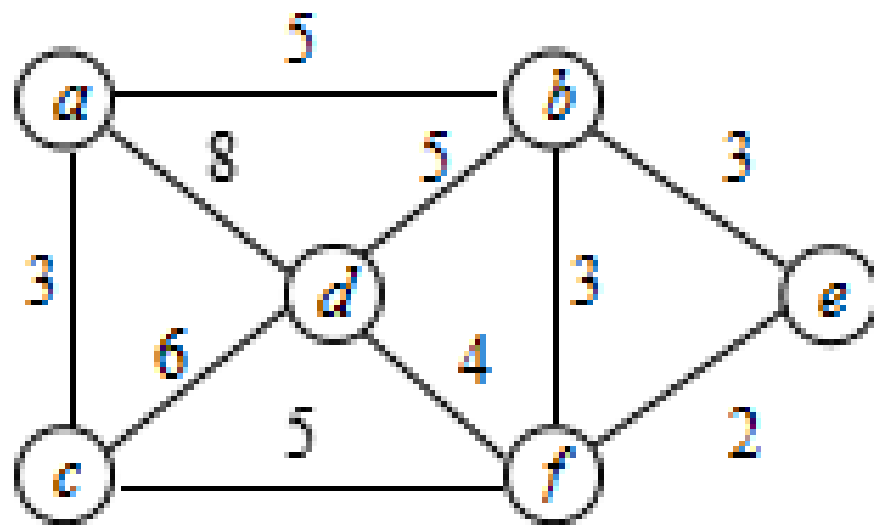


图 6-8 一个无向连通网

5. Prim算法如何存储候选最短边集？

正常使用主观题需2.0以上版本雨课堂

作答

最小生成树-Kruskal（克鲁斯卡尔）算法



Kruskal算法——基本思想



Kruskal算法——存储结构



Kruskal算法——算法描述



Kruskal算法——性能分析

基本思想



Prim算法的关键是什么？

找到连接 U 和 $V-U$ 的最短边

最短边

顶点分别位于 U 和 $V-U$ 中



Prim算法：先构造满足条件的候选最短边集，再查找最短边



Kruskal算法：先查找最短边，再判断是否满足条件



Prim算法和Kruskal算法都是采用贪心法，区别是贪心策略

基本思想

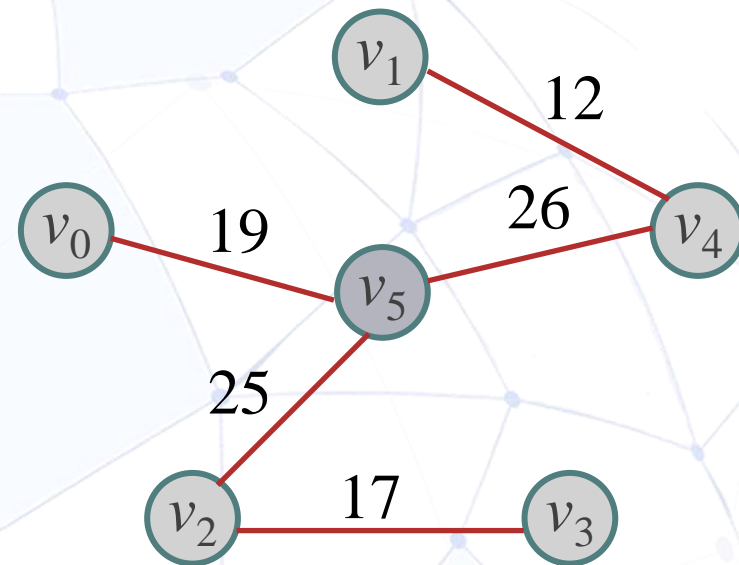
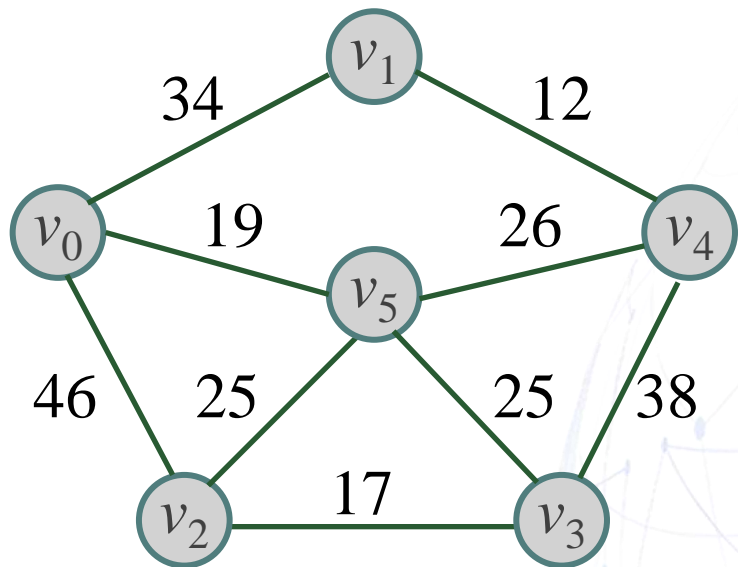
算法: Kruskal算法

输入: 无向连通网 $G=(V, E)$

输出: 最小生成树 $T=(U, TE)$

1. 初始化: $U=V$; $TE=\{ \}$;
2. 重复下述操作直到所有顶点位于一个连通分量:
 - 2.1 在 E 中选取最短边 (u, v) ;
 - 2.2 如果顶点 u 、 v 位于两个连通分量, 则
 - 2.2.1 将边 (u, v) 并入 TE ;
 - 2.2.2 将这两个连通分量合成一个连通分量;
 - 2.3 在 E 中标记边 (u, v) , 使得 (u, v) 不参加后续最短边的选取;

运行实例



(1,4)	12
(2,3)	17
(0,5)	19
(2,5)	25
(3,5)	25
(4,5)	26
(0,1)	34
(0,2)	46

初始化：连通分量 = $\{v_0\}, \{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}$

第一次迭代：连通分量 = $\{v_0\}, \{v_1, v_4\}, \{v_2\}, \{v_3\}, \{v_5\}$

第二次迭代：连通分量 = $\{v_0\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_5\}$

第三次迭代：连通分量 = $\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

第四次迭代：连通分量 = $\{v_0, v_2, v_3, v_5\}, \{v_1, v_4\}$

第五次迭代：连通分量 = $\{v_0, v_2, v_3, v_5, v_1, v_4\}$

存储结构



图采用什么存储结构呢?



边集数组表示法

算法: Kruskal算法

输入: 无向连通网 $G=(V, E)$

输出: 最小生成树 $T=(U, TE)$

1. 初始化: $U=V$; $TE=\{ \}$;

2. 重复下述操作直到所有顶点位于一个连通分量:

2.1 在 E 中选取最短边 (u, v) ;

2.2 如果顶点 u 、 v 位于两个连通分量, 则

2.2.1 将边 (u, v) 并入 TE ;

2.2.2 将这两个连通分量合成一个连通分量;

2.3 在 E 中标记边 (u, v) , 使得 (u, v) 不参加后续最短边的选取;

难点

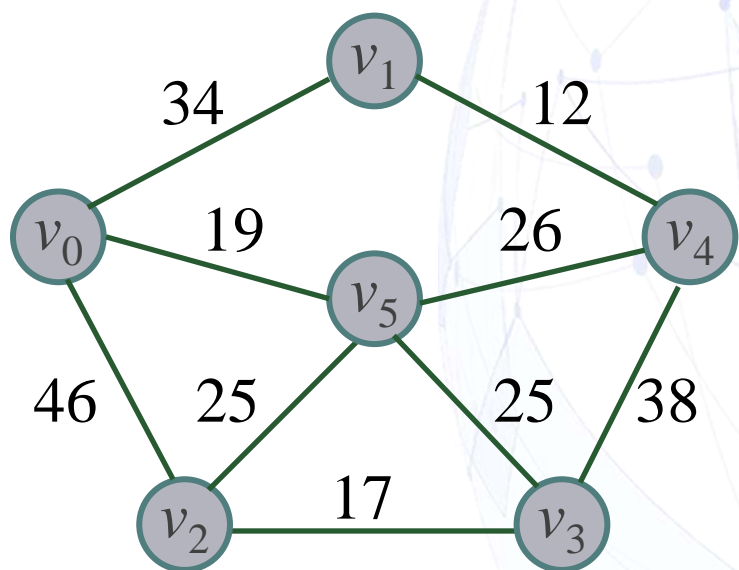
存储结构



图采用什么存储结构呢?



边集数组表示法



下标: 0 1 2 3 4 5

v_0	v_1	v_2	v_3	v_4	v_5
-------	-------	-------	-------	-------	-------

from
to
weight

1	2	0	2	3	4	0	3	0
4	3	5	5	5	5	1	4	2
12	17	19	25	25	26	34	38	46

排好序

存储结构



如何定义存储结构?



边集数组类的定义

```
const int MaxVertex = 10;
const int MaxEdge = 100;
template <typename DataType>
class EdgeGraph
{
public:
    EdgeGraph(DataType a[ ], int n, int e);
    ~EdgeGraph( );
    void Kruskal( );
private:
    int FindRoot(int parent[ ], int v)
    DataType vertex[MaxVertex];
    EdgeType edge[MaxEdge];
    int vertexNum, edgeNum;
};
```

下标: 0 1 2 3 4 5

v_0	v_1	v_2	v_3	v_4	v_5
-------	-------	-------	-------	-------	-------

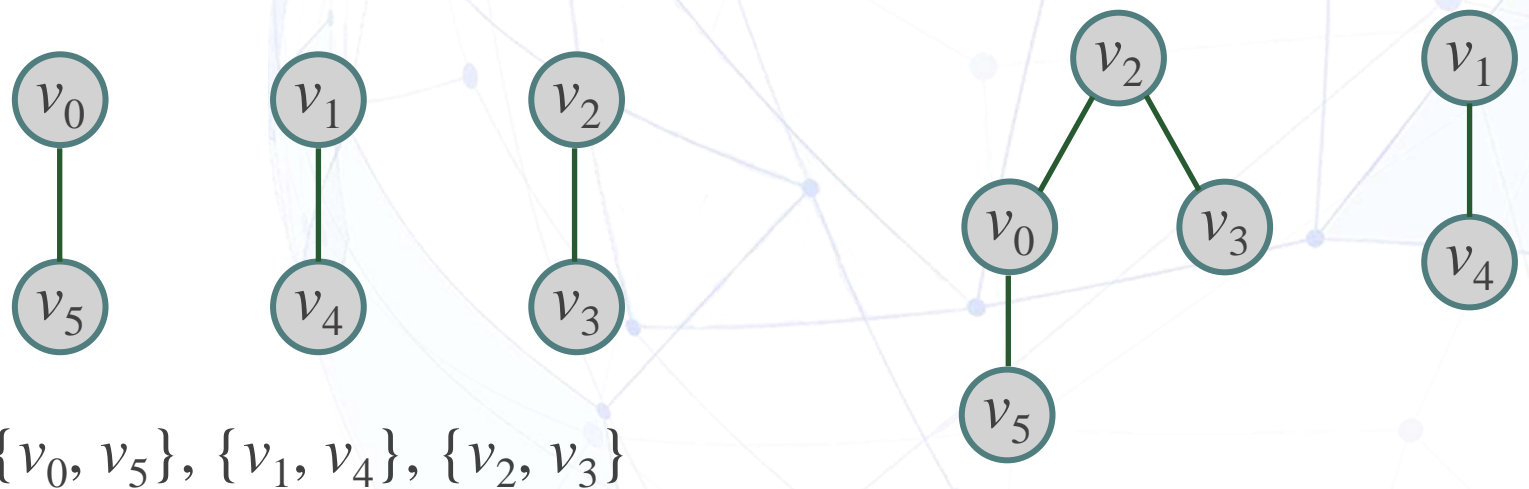
from	1	2	0	2	3	4	3	0	0
to	4	3	5	5	5	5	4	1	2
weight	12	17	19	25	25	26	38	34	46

存储结构

🕒 如何存储连通分量呢? ➡ 并查集

📌 并查集：集合中的元素组织成树的形式：

- (1) 查找两个元素是否属于同一集合：所在树的根结点是否相同
- (2) 合并两个集合——将一个集合的根结点作为另一个集合根结点的孩子



$\{v_0, v_5, v_2, v_3\}, \{v_1, v_4\}$

存储结构

🕒 如何存储并查集呢? \Rightarrow 双亲表示法 \Rightarrow parent[n]

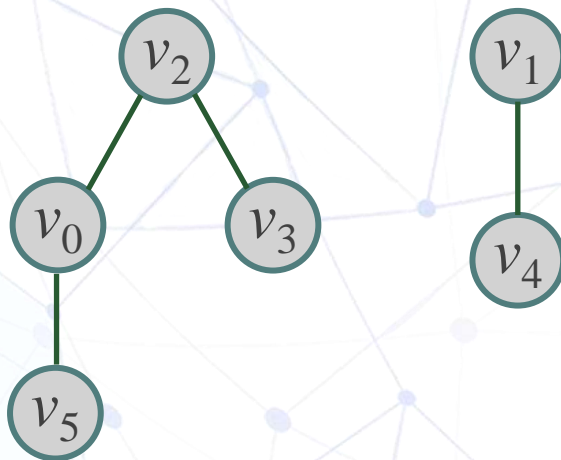
📌 并查集：集合中的元素组织成树的形式：

(1) 查找：FindRoot(v_5) = v_0 , FindRoot(v_2) = v_2

(2) 合并：parent[v_0] = v_2



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$



$\{v_0, v_5, v_2, v_3\}, \{v_1, v_4\}$

下标:	0	1	2	3	4	5
vertex	v_0	v_1	v_2	v_3	v_4	v_5
parent	-1	-1	-1	2	1	0

算法描述

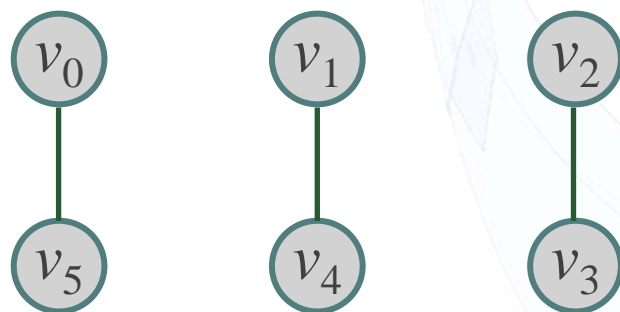


如何判断两个顶点是否位于同一个连通分量呢？

例如，边 (v_2, v_5) ？

```
vex1 = FindRoot(parent, i);  
vex2 = FindRoot(parent, j);  
if (vex1 != vex2) {  
    }  
}
```

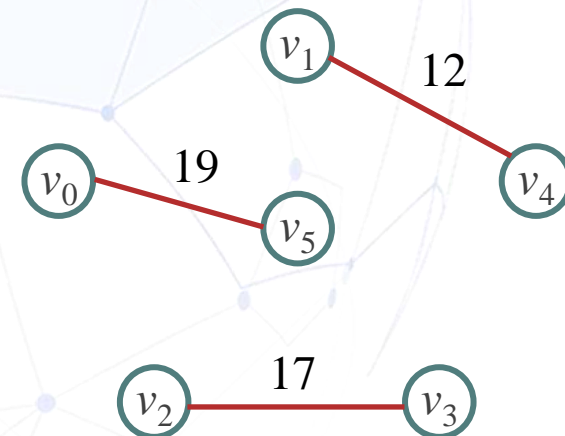
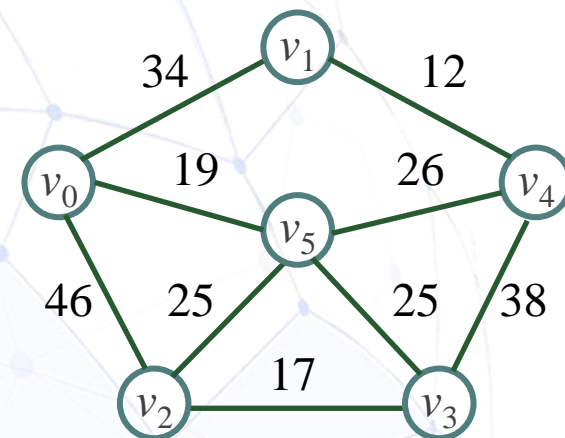
```
int FindRoot(int parent[ ], int v)  
{  
    int t = v;  
    while (parent[t] > -1)  
        t = parent[t];  
    return t;  
}
```



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

下标: 0 1 2 3 4 5

parent	-1	-1	-1	2	1	0
--------	----	----	----	---	---	---

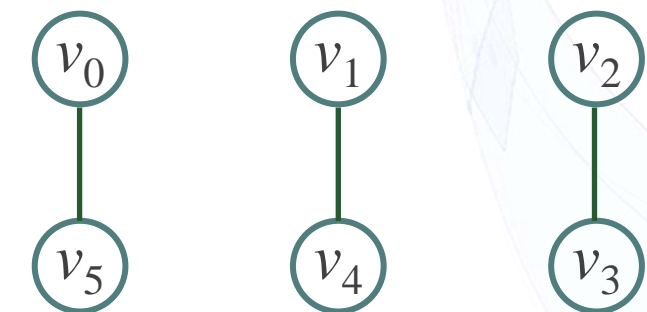


算法描述

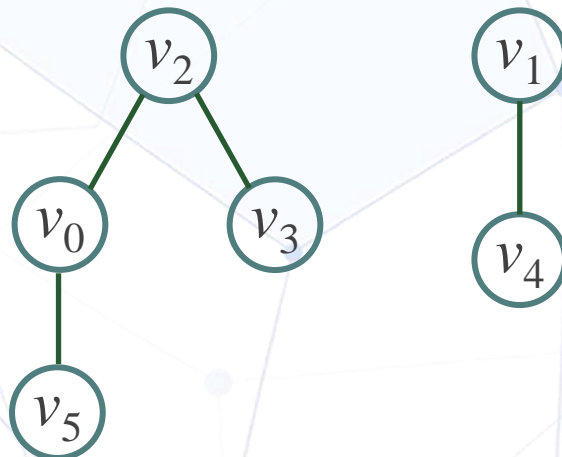


如何合并两个连通分量呢？

```
vex1 = FindRoot(parent, i);  
vex2 = FindRoot(parent, j);  
if (vex1 != vex2) {  
    parent[vex2] = vex1;  
}
```



$\{v_0, v_5\}, \{v_1, v_4\}, \{v_2, v_3\}$

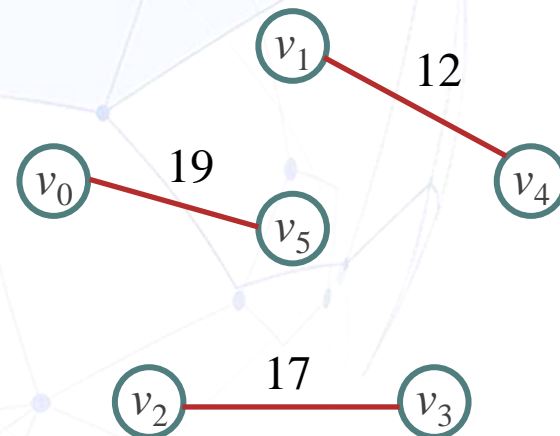
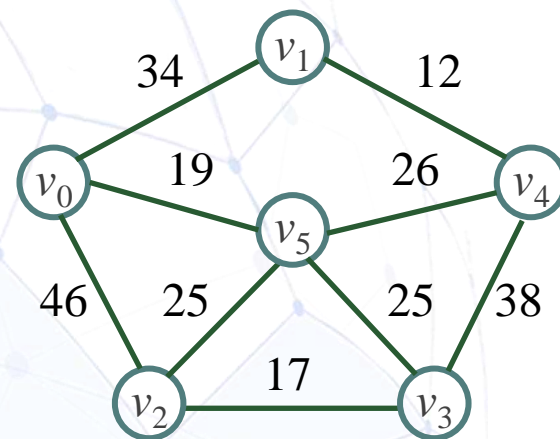


$\{v_0, v_5, v_2, v_3\}, \{v_1, v_4\}$

下标: 0 1 2 3 4 5

parent	-1	-1	-1	2	1	0
	2					

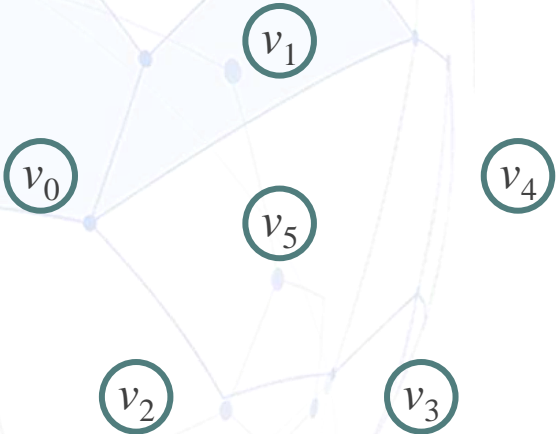
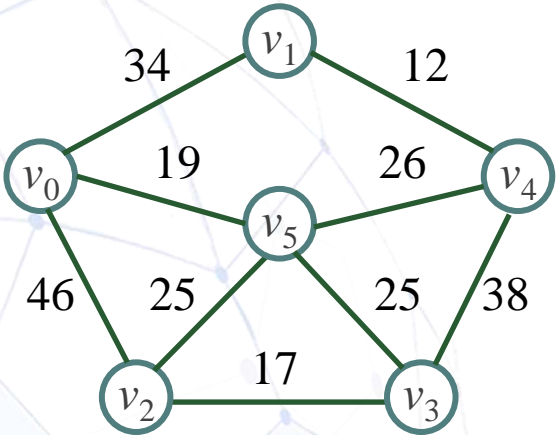
例如, 边 (v_2, v_5) ?



算法描述

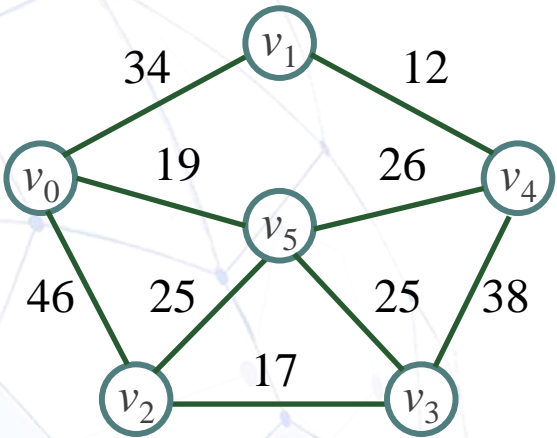
下标	0	1	2	3	4	5		
parent	v_0	v_1	v_2	v_3	v_4	v_5	最短边	说明
parent	-1	-1	-1	-1	-1	-1		初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$

```
void EdgeGraph :: Kruskal()  
{  
    int i, num = 0, vex1, vex2;  
    int parent[vertexNum];  
    for (i = 0; i < vertexNum; i++)  
        parent[i] = -1;  
}
```

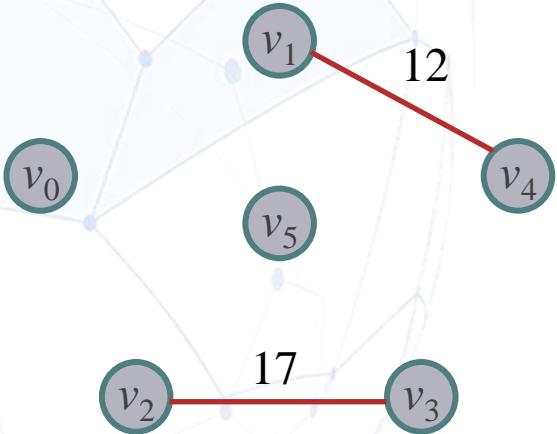


算法描述

下标 parent	0 v_0	1 v_1	2 v_2	3 v_3	4 v_4	5 v_5	最短边	说明
parent	-1	-1	-1	-1	-1	-1	$(v_1, v_4)12$	初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$
parent	-1	-1	-1	-1	1	-1	$(v_2, v_3)17$	$vex1=1, vex2=4, parent[4]=1$ $\{v_0\}\{v_1, v_4\}\{v_2\}\{v_3\}\{v_5\}$
parent	-1	-1	-1	2	1	-1		$vex1=2, vex2=3, parent[3]=2$ $\{v_0\}\{v_1, v_4\}\{v_2, v_3\}\{v_5\}$

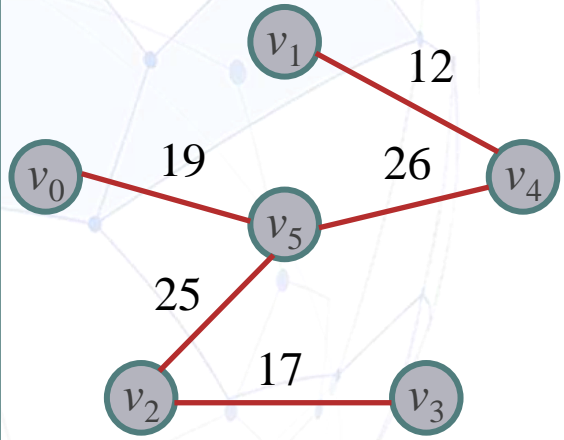
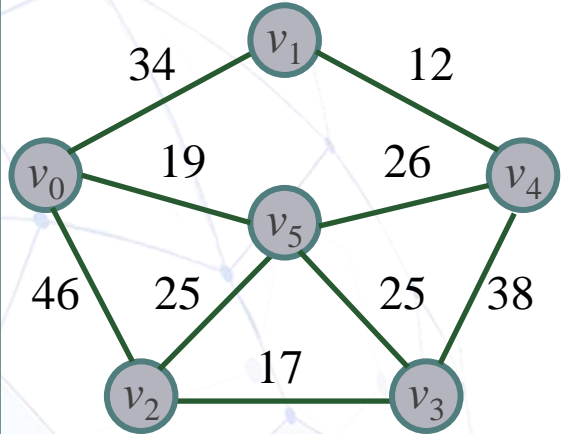


```
for (num = 0, i = 0; num < vertexNum; i++)
{
    vex1 = FindRoot(parent, edge[i].from);
    vex2 = FindRoot(parent, edge[i].to);
    if (vex1 != vex2) {
        cout << edge[i].from << edge[i].to << edge[i].weight;
        parent[vex2] = vex1; num++;
    }
}
```



验证算法

下标 parent	0 v_0	1 v_1	2 v_2	3 v_3	4 v_4	5 v_5	最短边	说明
parent	-1	-1	-1	-1	-1	-1	$(v_1, v_4)12$	初始化 $\{v_0\}\{v_1\}\{v_2\}\{v_3\}\{v_4\}\{v_5\}$
parent	-1	-1	-1	-1	1	-1	$(v_2, v_3)17$	$vex1=1, vex2=4, parent[4]=1$ $\{v_0\}\{v_1, v_4\}\{v_2\}\{v_3\}\{v_5\}$
parent	-1	-1	-1	2	1	-1	$(v_0, v_5)19$	$vex1=2, vex2=3, parent[3]=2$ $\{v_0\}\{v_1, v_4\}\{v_2, v_3\}\{v_5\}$
parent	-1	-1	-1	2	1	0	$(v_2, v_5)25$	$vex1=0, vex2=5, parent[5]=0$ $\{v_0, v_5\}\{v_1, v_4\}\{v_2, v_3\}$
parent	2	-1	-1	2	1	0	$(v_3, v_5)25$	$vex1=2, vex2=0, parent[0]=2$ $\{v_0, v_5\}\{v_1, v_4\}\{v_2, v_3\}$
parent	2	-1	1	2	1	0	$(v_4, v_5)26$	$vex1=2, vex2=2$ 在一个连通分量中
parent								$vex1=1, vex2=2, parent[2]=1$ $\{v_0, v_5, v_1, v_4, v_2, v_3\}$



算法描述

```
void EdgeGraph :: Kruskal( )
```

```
{
```

```
    int i, num = 0, vex1, vex2;
```

```
    int parent[vertexNum];
```

```
    for (i = 0; i < G->vertexNum; i++)
```

```
        parent[i] = -1;
```

```
    for (num = 0, i = 0; num < G->vertexNum; i++)
```

```
    {
```

```
        vex1 = FindRoot(parent, G->edge[i].from);
```

```
        vex2 = FindRoot(parent, G->edge[i].to);
```

```
        if (vex1 != vex2) {
```

```
            printf("(%d, %d)%d ", G->edge[i].from, G->edge[i].to, G->edge[i].weight);
```

```
            parent[vex2] = vex1;
```

```
            num++;
```

```
        }
```

```
    }
```

```
}
```

```
int FindRoot(int parent[ ], int v)
```

```
{
```

```
    int t = v;
```

```
    while (parent[t] > -1)
```

```
        t = parent[t];
```

```
    return t;
```

```
}
```

} $O(n)$

} $O(\log_2 n)$

} $O(e)$

如果合并时总是把较矮的树合并为较高的树的子树



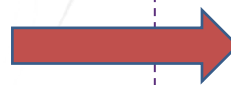
时间复杂度?



$O(e \log_2 e + e \log_2 n) = O(e \log_2 e)$



边集数组排序



1. Kruskal算法采用（ ）作为存储结构。

- ☐ A 邻接矩阵
- ☐ B 邻接表
- ☒ C 边集数组
- ☐ D 多重链表

提交

2. 并查集将集合中的元素组织成树的形式，并采用（ ）存储。

- ☒ A 双亲表示法
- ☐ B 孩子表示法
- ☐ C 二叉链表
- ☐ D 列举法

提交

4. 对于如图6-9所示无向连通图，用Kruskal算法构造最小生成树，加入最小生成树的第4条边是（ ）。

- ☐ A (a, c)3
- ☐ B (b, e)3
- ☐ C (f, b)3
- ☒ D (d, f)4

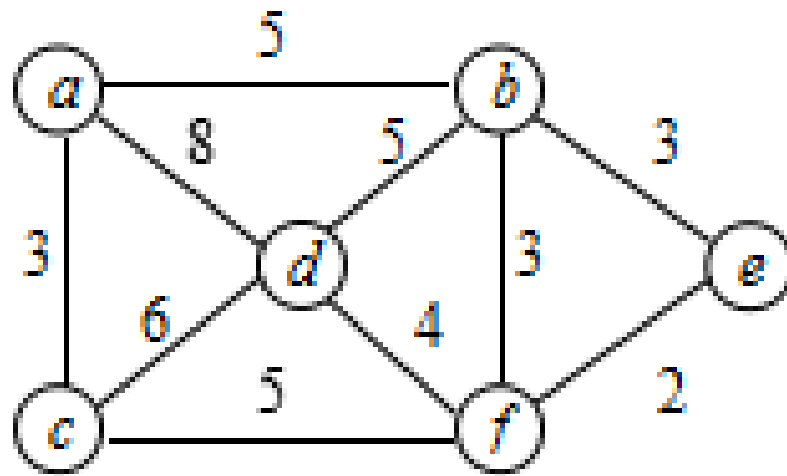


图 6-9 一个无向连通网

5. 对于如图6-9所示无向连通图，用Kruskal算法构造最小生成树，假设加入最小生成树的第2条边是 (a, c) ，则当前的连通分量是（ ）。

- ☒ A $\{a, c\}\{f, e\}\{d\}\{b\}$
- ☐ B $\{a, c\}\{f, e\}\{d, b\}$
- ☐ C $\{a, c, f, e\}\{d\}\{b\}$
- ☐ D $\{a, c, f, e\}\{d, b\}$

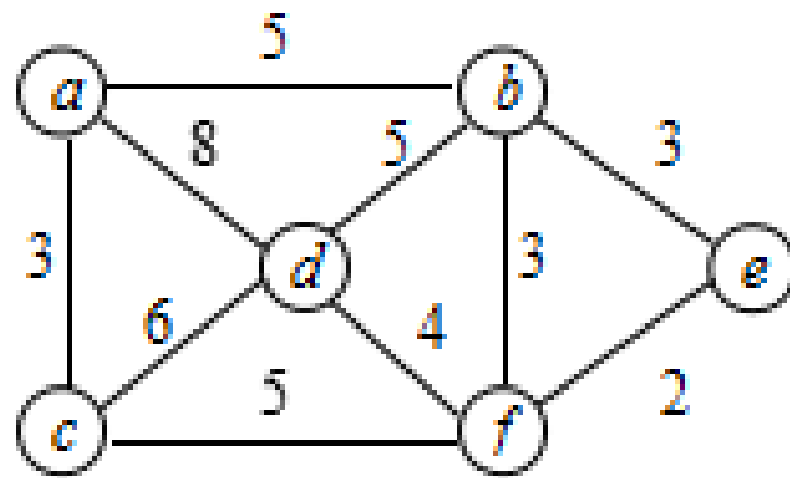


图 6-9 一个无向连通网

最小生成树算法小结

Prim 算法（普里姆）：

从某一个顶点开始构建生成树；
每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。

⇒ 图的邻接矩阵表示法

{ $\text{adjvex}[n]$
 $\text{lowcost}[n]$

时间复杂度： $O(|V|^2)$

适合用于边稠密图

Kruskal 算法（克鲁斯卡尔）：

每次选择一条权值最小的边，使这条边的两头连通（原本已经连通的就不选）
直到所有结点都连通

⇒ 边集数组表示法

并查集 $\text{parent}[n]$

时间复杂度： $O(|E|\log_2|E|)$

适合用于边稀疏图

图的应用2



最短路径算法



最短路径的定义



单源点最短路径 (BFS、Dijkstra算法)



各顶点间的最短路径算法 (Floyd算法)

最短路径问题

单源最短路径

BFS 算法 (无权图)

Dijkstra 算法 (带权图、无权图)

各顶点间的最短路径

Floyd 算法 (带权图、无权图)

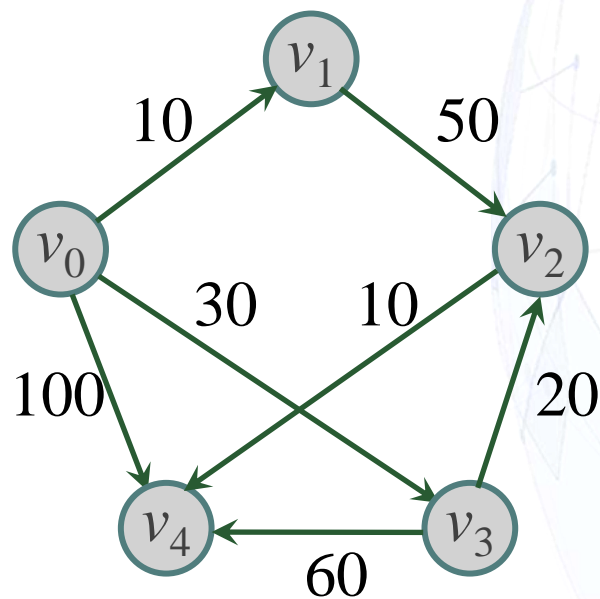
最短路径的定义



最短路径: **非带权图**——边数最少的路径

带权图——边上的权值之和最小的路径

单源点最短路径



(无权图) v_0 到 v_4 的最短路径:

$v_0 v_4$: 1

$v_0 v_3 v_4$: 2

$v_0 v_1 v_2 v_4$: 3

$v_0 v_3 v_2 v_4$: 3

(带权图) v_0 到 v_4 的最短路径:

$v_0 v_4$: 100

$v_0 v_3 v_4$: 90

$v_0 v_1 v_2 v_4$: 70

$v_0 v_3 v_2 v_4$: 60

任意两点的最短路径

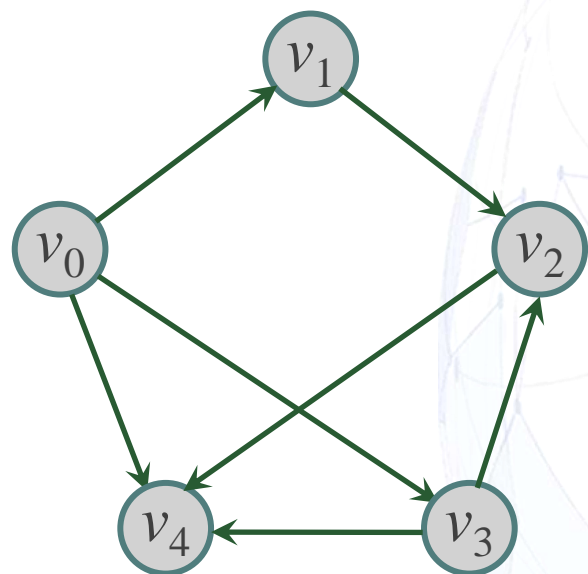
$v_3 v_4$

60?

$v_3-v_2-v_4$? 30?

单源点最短路径问题 (无权图)-BFS算法

🕒 对于非带权图，如何求最短路径？ \Rightarrow 广度优先遍历



也可看作权值为1

level = ~~0~~ 1

$v_0 \ v_1 v_3 v_4$

广度优先遍历序列: $v_0 \ v_1 \ v_3 \ v_4 \ v_2$

1

2

//求顶点 u 到其他顶点的最短路径

```
void BFS_MIN_Distance(Graph G,int u){
```

```
    //d[i]表示从u到i结点的最短路径
```

```
    for(i=0;i<G.vexnum;++i){
```

```
        d[i]= $\infty$ ;    //初始化路径长度
```

```
        path[i]=-1; //最短路径从哪个顶点过来
```

```
    }
```

```
    d[u]=0;
```

```
    visited[u]=TRUE;
```

```
    EnQueue(Q,u);
```

```
    while(!isEmpty(Q)){
```

//BFS算法主过程

```
        DeQueue(Q,u);
```

//队头元素 u 出队

```
        for(w=FirstNeighbor(G,u);w<G.vexnum;w=NextNeighbor(G,u,w)){
```

```
            if(!visited[w]){
```

//w为u的尚未访问的邻接顶点

```
                d[w]=d[u]+1;
```

//路径长度加1

```
                path[w]=u;
```

//最短路径应从u到w

```
                visited[w]=TRUE;
```

//设已访问标记

```
                EnQueue(Q,w);
```

//顶点w入队

```
            }//if
```

```
    }//while
```

```
}
```



对于带权图，如何求最短路径？

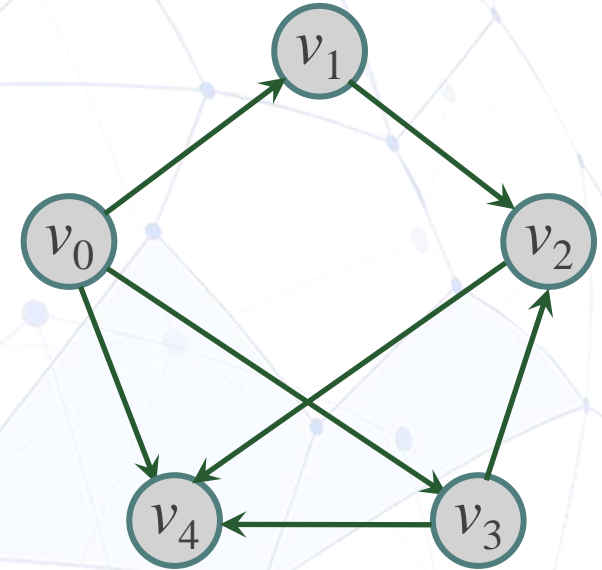


单源点最短路径问题（带权图）

在有向带权图中单源点最短路径问题是指求解指定的起点到其余各顶点之间的最短路径，并限定图中每条边的权值大于0。

指定的起点称为源点，对应的其余各顶点则称为终点。

如右图中，假设源点为0，则需要分别求出顶点0到1,2,3,4这四个顶点的最短路径及长度。



应用实例——计算机网络传输的问题：怎样找到一种最经济的方式，从一台计算机向网上所有其它计算机发送一条消息

单源点最短路径问题-Dijkstra算法



Dijkstra算法——基本思想



Dijkstra算法——存储结构



Dijkstra算法——算法描述



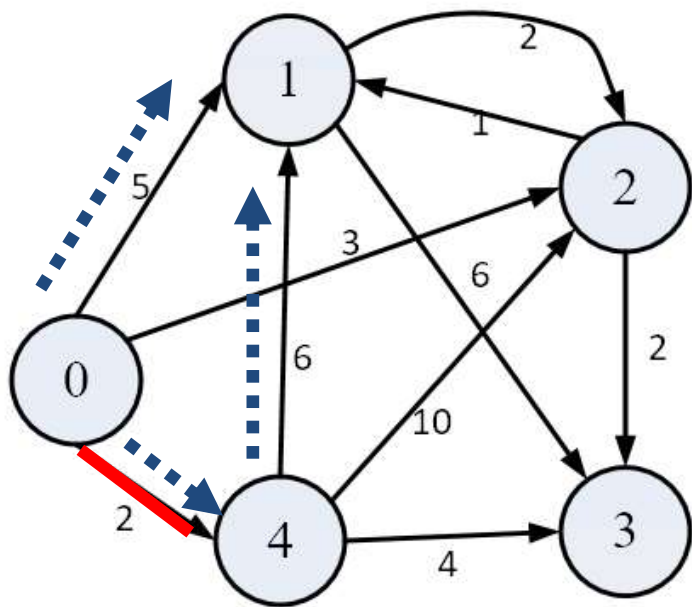
Dijkstra算法——性能分析

Dijkstra算法

Dijkstra算法与Prim算法一样是贪心算法；

按照**路径长度不减的次序**依次找到源点到各终点的最短路径；

第1条最短路径在即将找出的所有最短路径中长度最短，然后依次递增。



如图，第1条最短路径是从源点0出发不经过其他任何顶点直接到达顶点4的长度为2的路径；

对于顶点1，最短路径可能是从源点直接到达顶点1，或经过顶点4再到达顶点1，取决于两种情形下的路径哪条更短。

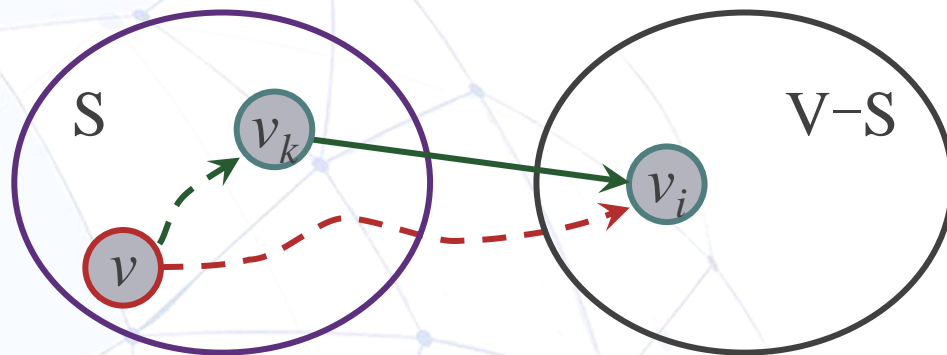
Dijkstra算法——基本思想

v : 源点

S : 已经确定了最短路径的顶点

$w\langle v, v_i \rangle$: 从顶点 v 到顶点 v_i 的权值

$\text{dist}(v, v_i)$: 表示从顶点 v 到顶点 v_i 的最短路径长度



算法: Dijkstra算法

输入: 有向网图 $G=(V, E)$, 源点 v

输出: 从 v 到其他所有顶点的最短路径

1. 初始化: 集合 $S = \{v\}$; $\text{dist}(v, v_i) = w\langle v, v_i \rangle, (i=1\dots n)$;
2. 重复下述操作直到 $S == V$
 - 2.1 $\text{dist}(v, v_k) = \min\{\text{dist}(v, v_j), (j=1\dots n)\}$;
 - 2.2 $S = S + \{v_k\}$;
 - 2.3 $\text{dist}(v, v_j) = \min\{\text{dist}(v, v_j), \text{dist}(v, v_k) + w\langle v_k, v_j \rangle\}$;

Dijkstra算法存储结构

主存储结构： 图的邻接矩阵表示

辅助结构：

(1) 集合S，存储**已明确最短路径的顶点**，初始时含有源点0。

(2) distance[n]数组

distance[i]中记录从源点出发，可以经过S集合中的顶点，最终到达顶点i的**最短路径长度**；

(3) pre[n]数组

pre[i]记录与distance[i]对应的顶点i的前驱顶点。

也可以设置path[n]数组，存放从源点到i的顶点序列

Dijkstra算法步骤

(1) 初始化distance和pre数组数组， $\text{distance}[i]$ 为边 $\langle 0, i \rangle$ 的权值（假设 v_0 为源点，如果该边不存在，即为 ∞ ），即对应于G的邻接矩阵中0行的元素。初始化所有的 $\text{pre}[i]$ 为起点0。

(2) 循环执行n-1次：

①从distance数组中找出非0的最小权值所对应的最短路径

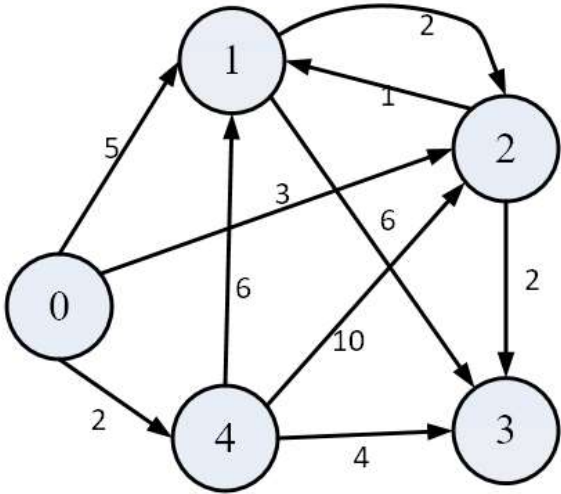
假设最小者为 $\text{distance}[k]$ ；即找到了源点到k的最短路径，将k加入到S集合；

②更新数组中其它非0的distance值和pre值

$\text{distance}[i] = \min(\text{distance}[i], \text{distance}[k] + \langle k, i \rangle \text{的权值})$ ，如果distance更新为后者，则同时更新 $\text{pre}[i]$ 为k。

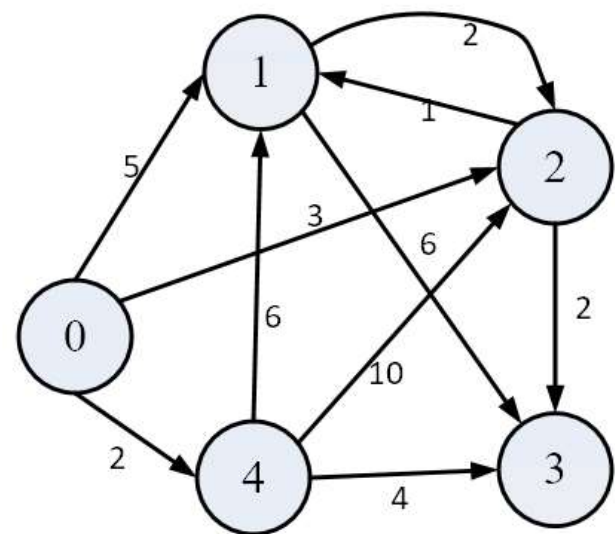
运行实例求解过程

迭代次数	最短路径	路径长度	加入S集合的顶点{0}	i	0	1	2	3	4
				distance[i]	0	5	3	∞	2
1				pre[i]	0	0	0	0	0
				distance[i]					
2				pre[i]					
				distance[i]					
3				pre[i]					
				distance[i]					
4				pre[i]					
				distance[i]					



求解过程

迭代次数	最短路径	路径长度	加入S集合的顶点{0}	i	0	1	2	3	4
				distance[i]	0	5	3	∞	2
				pre[i]	0	0	0	0	0
1	0->4	2	4	distance[i]		5	3	6	
				pre[i]		0	0	4	
2	0->2	3	2	distance[i]		4		5	
				pre[i]		2		2	
3	0->2->1	4	1	distance[i]				5	
				pre[i]				2	
4	0->2->3	5	3	distance[i]					
				pre[i]					



算法描述

```
void MGraph<DataType> :: Dijkstra(int v)
```

/*从源点v出发*/

```
{  
    int i, k, num, dist[MaxSize]; string path[MaxSize];  
    for (i = 0; i < vertexNum; i++)  
    {  
        dist[i] = edge[v][i]; path[i] = vertex[v] + vertex[i];
```

} $O(n)$

```
    for (num = 1; num < vertexNum; num++)
```

```
    {  
        for (k = 0, i = 0; i < vertexNum; i++)  
            if ((dist[i] != 0) && (dist[i] < dist[k])) k = i;  
        cout << path[k] << dist[k];
```

} $O(n)$

```
        for (i = 0; i < vertexNum; i++)
```

```
            if (dist[i] > dist[k] + edge[k][i]) {
```

```
                dist[i] = dist[k] + edge[k][i]; path[i] = path[k] + vertex[i];
```

} $O(n)$

```
            }  
            dist[k] = 0; 最好不要让dist置0, 再次访问会缺信息
```



时间复杂度?



$O(n^2)$



1. Dijkstra算法采用（ ）作为存储结构。

- ☐ A 边集数组
- ☐ B 多重链表
- ☒ C 邻接矩阵
- ☐ D 邻接表

提交

2. 对于如图6-10所示有向图，用Dijkstra算法求最短路径，从 v_0 到 v_2 的最短路径长度是（ ）。

- ☐ A 30
- ☐ B 25
- ☐ C 26
- ☒ D 22

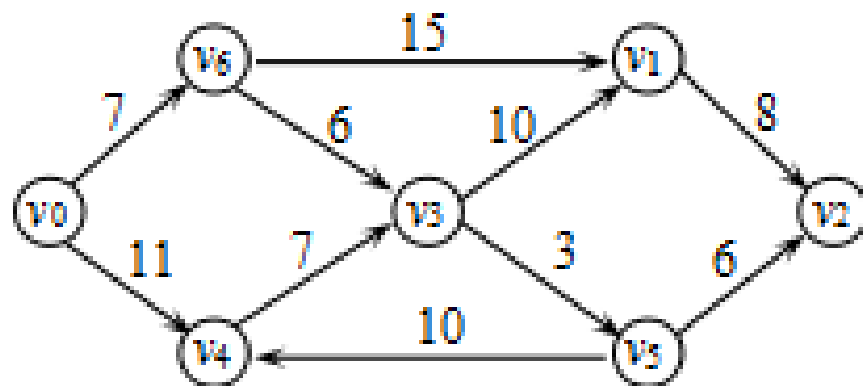


图 6-10 有向网图

提交

3. 对于如图6-10所示有向图，用Dijkstra算法求最短路径，求得的第3条最短路径是（ ）。

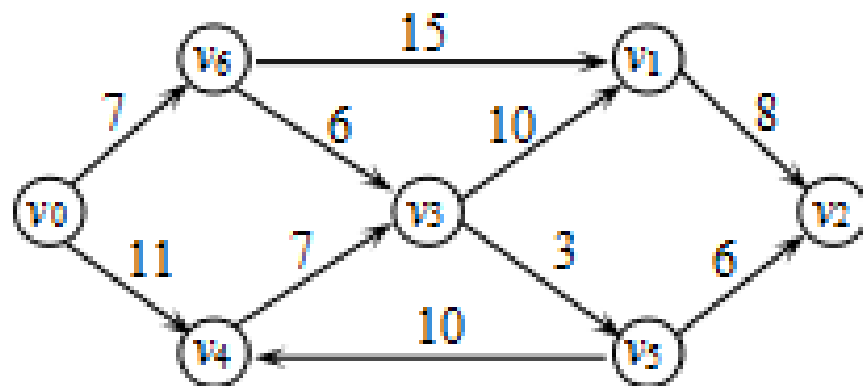


图 6-10 有向网图

- ☐ A (v0v4)11
- ☒ B (v0v6v3)13
- ☐ C (v0v4v3)18
- ☐ D (v0v6v3v5)16

提交

4. Dijkstra算法如何保存迭代过程中当前的最短路径长度？

正常使用主观题需2.0以上版本雨课堂

作答

每一对顶点的最短路径问题

【问题】 给定带权有向图 $G = (V, E)$, 对任意顶点 v_i 和 v_j ($i \neq j$) , 求从顶点 v_i 到顶点 v_j 的最短路径

【想法 1】 每次以一个顶点为源点调用Dijkstra算法。显然, 时间复杂度为 $O(n^3)$

【想法 2】

【算法】 Floyd算法

每一对顶点的最短路径问题

每对顶点最短路径问题-Floyd算法



Floyd算法——基本思想



Floyd算法——存储结构



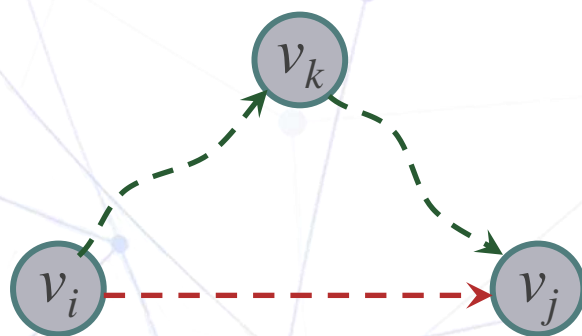
Floyd算法——算法描述



Floyd算法——性能分析

Floyd算法——基本思想

假设任意一对顶点 i 和 j ，对于 i 号顶点到 j 号顶点的最短路径，可能是不通过其他顶点直接走的路径，也可能是经过其它顶点中转到达的路径。



因此Floyd算法采用**试探策略**，对图中的每一个顶点 k ，算法依次检查若将 k 加入到路径中，是否可以得到更短的路径。

Floyd算法——基本思想

算法：Floyd算法

输入：带权有向图 $G=(V, E)$

输出：每一对顶点的最短路径

1. 初始化：假设从 v_i 到 v_j 的弧是最短路径，即 $\text{dist}_{-1}(v_i, v_j) = w\langle v_i, v_j \rangle$;

2. 循环变量 k 从 $0 \sim n-1$ 进行 n 次迭代：

$$\text{dist}_k(v_i, v_j) = \min\{\text{dist}_{k-1}(v_i, v_j), \text{dist}_{k-1}(v_i, v_k) + \text{dist}_{k-1}(v_k, v_j)\}$$

$\text{dist}_{(-1)}[i][j]$ 表示从 i 出发，不经过其他顶点，直接到达顶点 j 的路径长度

$\text{dist}_{(k)}[i][j]$ 表示从 i 到 j ，中间只可能经过 0 号至 k 号顶点而不可能经过 $k+1$ 号至 $n-1$ 号顶点的最短路径长度。



Floyd算法是动态规划算法， $\text{dist}_{-1}, \text{dist}_0, \text{dist}_1, \dots$

Floyd算法——存储结构

算法：Floyd算法

输入：带权有向图 $G=(V, E)$

输出：每一对顶点的最短路径

1. 初始化：假设从 v_i 到 v_j 的弧是最短路径，即 $\text{dist}_{-1}(v_i, v_j) = w\langle v_i, v_j \rangle$;
2. 循环变量 k 从 $0 \sim n-1$ 进行 n 次迭代：

$$\text{dist}_k(v_i, v_j) = \min\{\text{dist}_{k-1}(v_i, v_j), \text{dist}_{k-1}(v_i, v_k) + \text{dist}_{k-1}(v_k, v_j)\}$$



如何存储dist? 如何存储带权有向图?

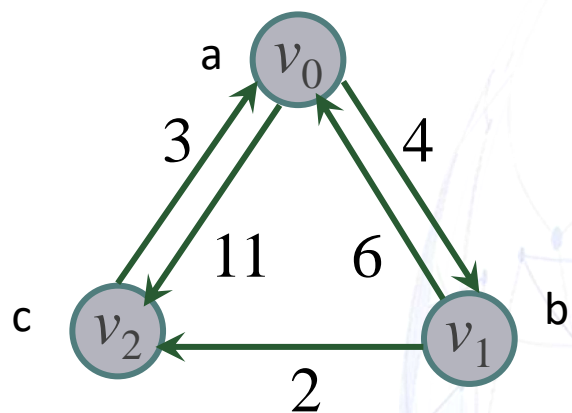


邻接矩阵



由于需要求出任意一对顶点之间的最短路径，除了需要用矩阵dist存储最短路径长度，一般还需要一个矩阵path存储最短路径上的顶点信息。

Floyd算法——运行实例



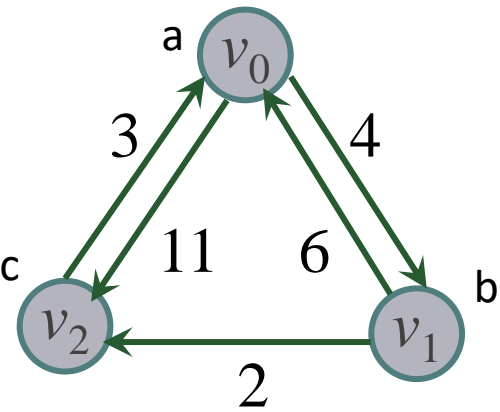
```
void Floyd( )  
{  
    int i, j, k, dist[MaxSize][MaxSize];  
    for (i = 0; i < vertexNum; i++)  
        for (j = 0; j < vertexNum; j++)  
            dist[i][j] = edge[i][j];  
}
```

初始化

$$\text{dist}_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$$\text{path}_{-1} = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & cb & \end{pmatrix}$$

Floyd算法——运行实例



$dist_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$

$path_{-1} = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & cb & \end{pmatrix}$

经过v0

$dist_0 = \begin{matrix} & v0 & v1 & v2 \\ v0 & \begin{pmatrix} 0 & 4 & 11 \end{pmatrix} \\ v1 & \begin{pmatrix} 6 & 0 & 2 \end{pmatrix} \\ v2 & \begin{pmatrix} 3 & 7 & 0 \end{pmatrix} \end{matrix}$

经过v1

$dist_1 = \begin{matrix} & v0 & v1 & v2 \\ v0 & \begin{pmatrix} 0 & 4 & 6 \end{pmatrix} \\ v1 & \begin{pmatrix} 6 & 0 & 2 \end{pmatrix} \\ v2 & \begin{pmatrix} 3 & 7 & 0 \end{pmatrix} \end{matrix}$

经过v2

$dist_2 = \begin{matrix} & v0 & v1 & v2 \\ v0 & \begin{pmatrix} 0 & 4 & 6 \end{pmatrix} \\ v1 & \begin{pmatrix} 5 & 0 & 2 \end{pmatrix} \\ v2 & \begin{pmatrix} 3 & 7 & 0 \end{pmatrix} \end{matrix}$

$path_0 = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & cab & \end{pmatrix}$

$path_2 = \begin{pmatrix} & ab & abc \\ ba & & bc \\ ca & cab & \end{pmatrix}$

$path_3 = \begin{pmatrix} & ab & abc \\ bca & & bc \\ ca & cab & \end{pmatrix}$

Floyd算法——C++实现

```
void Floyd( )  
{  
    int i, j, k, dist[MaxSize][MaxSize];  
    for (i = 0; i < vertexNum; i++)  
        for (j = 0; j < vertexNum; j++)  
            dist[i][j] = edge[i][j];  
    for (k = 0; k < vertexNum; k++)  
        for (i = 0; i < vertexNum; i++)  
            for (j = 0; j < vertexNum; j++)  
                if (dist[i][k] + dist[k][j] < dist[i][j])  
                    dist[i][j] = dist[i][k] + dist[k][j];  
}
```

$O(n^2)$

$O(n^3)$



时间复杂度?

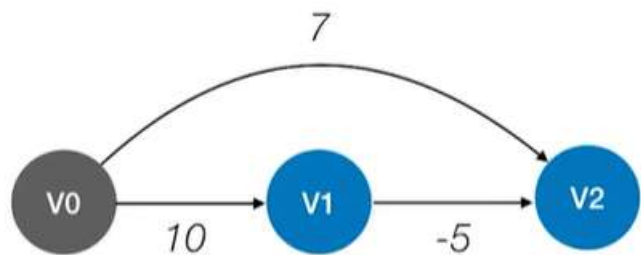


$O(n^3)$

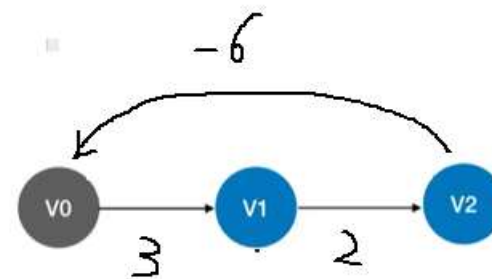
最短路径算法小结

	BFS 算法	Dijkstra 算法	Floyd 算法
无权图	✓	✓	✓
带权图	✗	✓	✓
带负权值的图	✗	✗	✓
带负权回路的图	✗	✗	✗
时间复杂度	$O(V ^2)$ 或 $O(V + E)$	$O(V ^2)$	$O(V ^3)$
通常用于	求无权图的单源最短路径	求带权图的单源最短路径	求带权图中各顶点间的最短路径

带负权值的图



负权的回路



Dijkstra算法不能得到正确答案

$dist[3]$

0	10	7
---	----	---



Floyd算法可以得到正确答案

$$dist_{-1} = \begin{bmatrix} 0 & 10 & 7 \\ \infty & 0 & -5 \\ \infty & \infty & 0 \end{bmatrix}$$

.....

$$dist_2 = \begin{bmatrix} 0 & 10 & 5 \\ \infty & 0 & -5 \\ \infty & \infty & 0 \end{bmatrix}$$



1. Floyd算法采用（ ）作为存储结构。

- ☒ A 邻接矩阵
- ☐ B 邻接表
- ☐ C 边集数组
- ☐ D 多重链表

提交

2. Floyd算法的时间复杂度是（ ）。

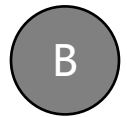
- ☐ A $O(n)$
- ☐ B $O(n^2)$
- ☒ C $O(n^3)$
- ☐ D $O(n^4)$

提交

3. Floyd算法可以求任意两个顶点之间的最短路径。



正确



错误

提交

4. 设有向图的邻接矩阵存储如图6-11(a)所示，第2次迭代结果如图6-11(b)所示，请填写括号。

$$\begin{pmatrix} 0 & 3 & 8 & 9 \\ \infty & 0 & 2 & \infty \\ \infty & \infty & 0 & 6 \\ \infty & 2 & 6 & 0 \end{pmatrix}$$

(a) 有向图的邻接矩阵

$$\begin{pmatrix} 0 & 3 & (\textcircled{1}) & (\textcircled{2}) \\ \infty & 0 & 2 & \infty \\ \infty & \infty & 0 & 6 \\ \infty & 2 & (\textcircled{3}) & 0 \end{pmatrix}$$

(b) Floyd算法第2次迭代结果

图 6-11 Floyd算法的执行过程

图的应用3

拓扑排序

关键路径

拓扑排序



AOV网的定义



拓扑序列的定义



拓扑排序算法——基本思想



拓扑排序算法——存储结构



拓扑排序算法——算法实现



拓扑排序算法——性能分析

AOV 网的定义



什么是工程？工程有什么共性？

几乎所有的工程都可以分为若干个称作**活动**的子工程
某些活动之间通常存在一定的**约束**条件



AOV网（顶点表示活动的网）：在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的**优先**关系

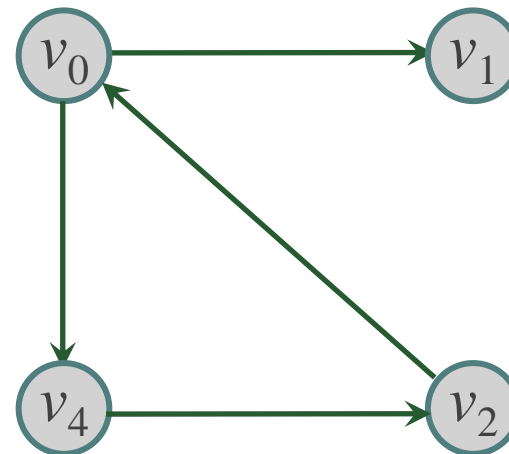
AOV网（activity on vertex network）

AOV网，是指用顶点表示活动，有向边表示活动发生的先后关系的有向图。可用于表示：工程的施工图、产品生产的流程图、程序的数据流图等，教学课程的依赖图。



AOV网中出现回路意味着什么？

活动之间的优先关系是矛盾的



有向无环图 (Directed Acyclic Graph)

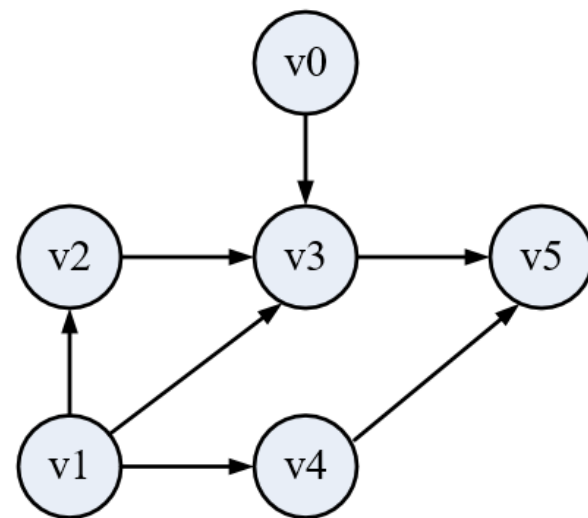
– 简称DAG图，是指不存在回路的有向图。



- 显然，对应真实工程的**AOV**网一定是**DAG**图，否则会出现两个活动互为先修关系这样的情况，导致工程无法正常进行。

课程编号	课程名称	先修课程
v0	计算机导论	无
v1	面向对象程序设计	无
v2	离散数学	v1
v3	数据结构	v0,v1,v2
v4	计算机组成	v1
v5	操作系统	v3,v4

课程及先修关系



拓扑序列的定义

✚ 拓扑序列：设有向图 $G=(V, E)$ 具有 n 个顶点，则顶点序列 v_0, v_1, \dots, v_{n-1} 称为一个拓扑序列，当且仅当满足下列条件：若从顶点 v_i 到 v_j 有一条路径，则在顶点序列中顶点 v_i 必在顶点 v_j 之前

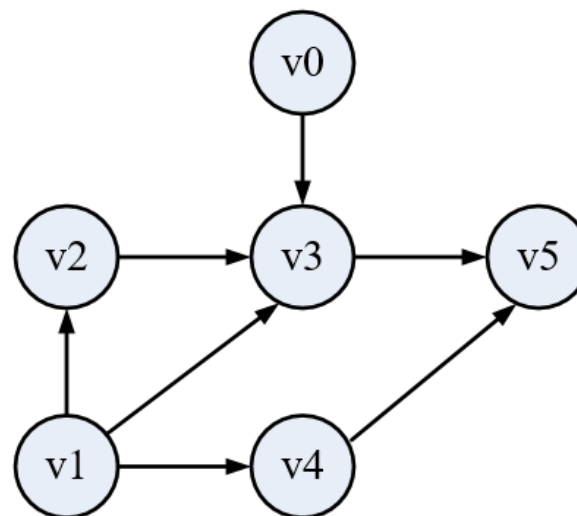
使得AOV网中所有应该存在的前驱和后继关系都能得到满足

✚ 拓扑排序：对一个有向图构造拓扑序列的过程



拓扑排序

- **v0,v1,v2,v3,v4,v5** ✓
- **v0,v1,v2,v4,v3,v5** ✓
- **v0,v1,v2,v3,v5,v4** ✗



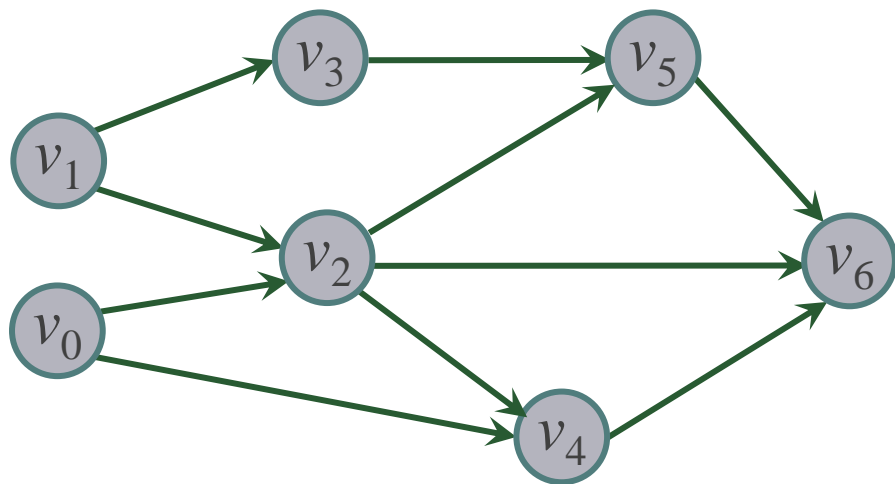
拓扑排序算法——基本思想

算法：拓扑排序TopSort

输入：AOV网 $G=(V, E)$

输出：拓扑序列

1. 重复下述操作，直到输出全部顶点，或AOV网中不存在没有前驱的顶点
 - 1.1 从AOV网中选择一个没有前驱的顶点并且输出；
 - 1.2 从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；



拓扑序列： $v_0 \quad v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6$

拓扑排序算法——存储结构



图采用什么存储结构呢?  邻接表

算法：拓扑排序TopSort

输入：AOV网 $G=(V, E)$

输出：拓扑序列

1. 重复下述操作，直到输出全部顶点，或AOV网中不存在没有前驱的顶点
 - 1.1 从AOV网中选择一个没有前驱的顶点并且输出；
 - 1.2 从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；

拓扑排序算法——存储结构

🕒 图采用什么存储结构呢？ \Rightarrow 邻接表

🕒 在邻接表中，如何求顶点的入度？ \Rightarrow 顶点表中增加入度域

算法：拓扑排序TopSort

输入：AOV网 $G=(V, E)$

输出：拓扑序列

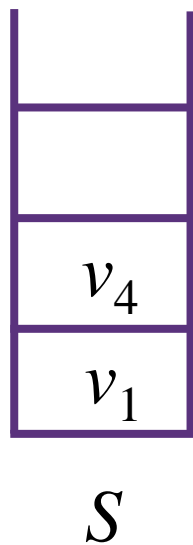
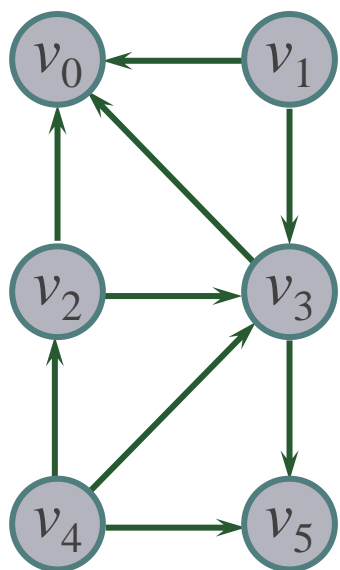
1. 重复下述操作，直到输出全部顶点，或AOV网中不存在没有前驱的顶点
 - 1.1 从AOV网中选择一个没有前驱的顶点并且输出；
 - 1.2 从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；

拓扑排序算法——存储结构

🕒 图采用什么存储结构呢？ ➡ 邻接表

🕒 在邻接表中，如何求顶点的入度？ ➡ 顶点表中增加入度域

🕒 如何查找没有前驱的顶点？ ➡ 设置栈或队列



in vertex firstEdge			
0	3	v_0	\wedge
1	0	v_1	
2	1	v_2	
3	3	v_3	
4	0	v_4	
5	2	v_5	\wedge


```
graph LR; 4[4 | ] --> 3[3 | ^]; 3 --> 5[5 | ^];
```

拓扑排序算法——伪代码



图：带入度的邻接表



栈：入度为 0 的顶点（编号）

算法：TopSort

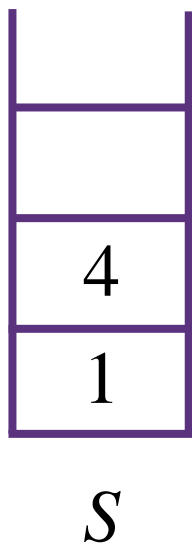
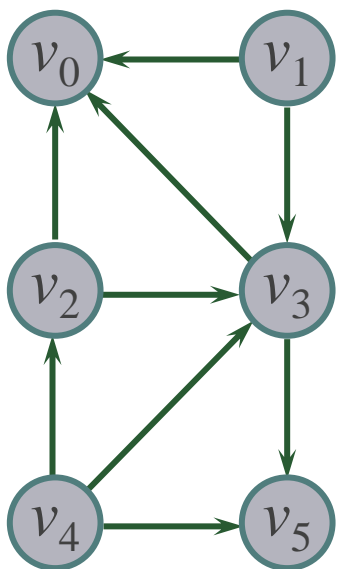
输入：有向图 $G=(V, E)$

输出：拓扑序列

1. 栈 S 初始化；累加器 count 初始化；
2. 扫描顶点表，将入度为 0 的顶点压栈；
3. 当栈 S 非空时循环
 - 3.1 $j =$ 栈顶元素出栈；输出顶点 j ；count++；
 - 3.2 对顶点 j 的每一个邻接点 k 执行下述操作：
 - 3.2.1 将顶点 k 的入度减 1；
 - 3.2.2 如果顶点 k 的入度为 0，则将顶点 k 入栈；
4. if (count < vertexNum) 输出有回路信息；

拓扑排序算法——C++实现

```
void TopSort()  
{  
    int i, j, k, count = 0, S[MaxSize], top = -1;  
    for (i = 0; i < vertexNum; i++)  
        if (adjlist[i].in == 0)  
            S[++top] = i;  
}
```



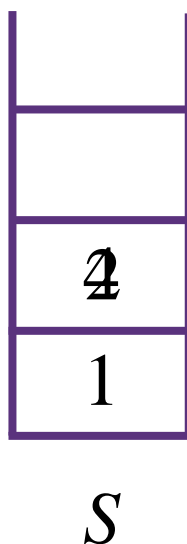
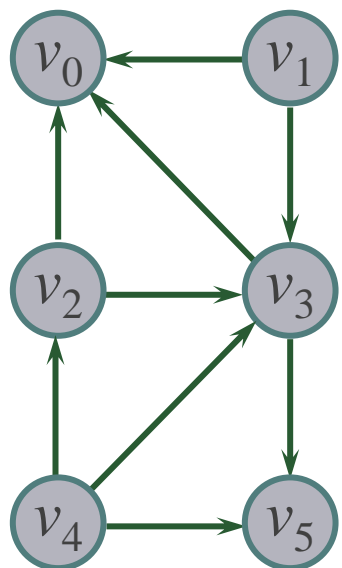
in vertex firstEdge

0	3	v_0	Λ			
1	0	v_1		→	0	→ 3 Λ
2	1	v_2		→	0	→ 3 Λ
3	3	v_3		→	0	→ 5 Λ
4	0	v_4		→	2	→ 3 → 5 Λ
5	2	v_5	Λ			

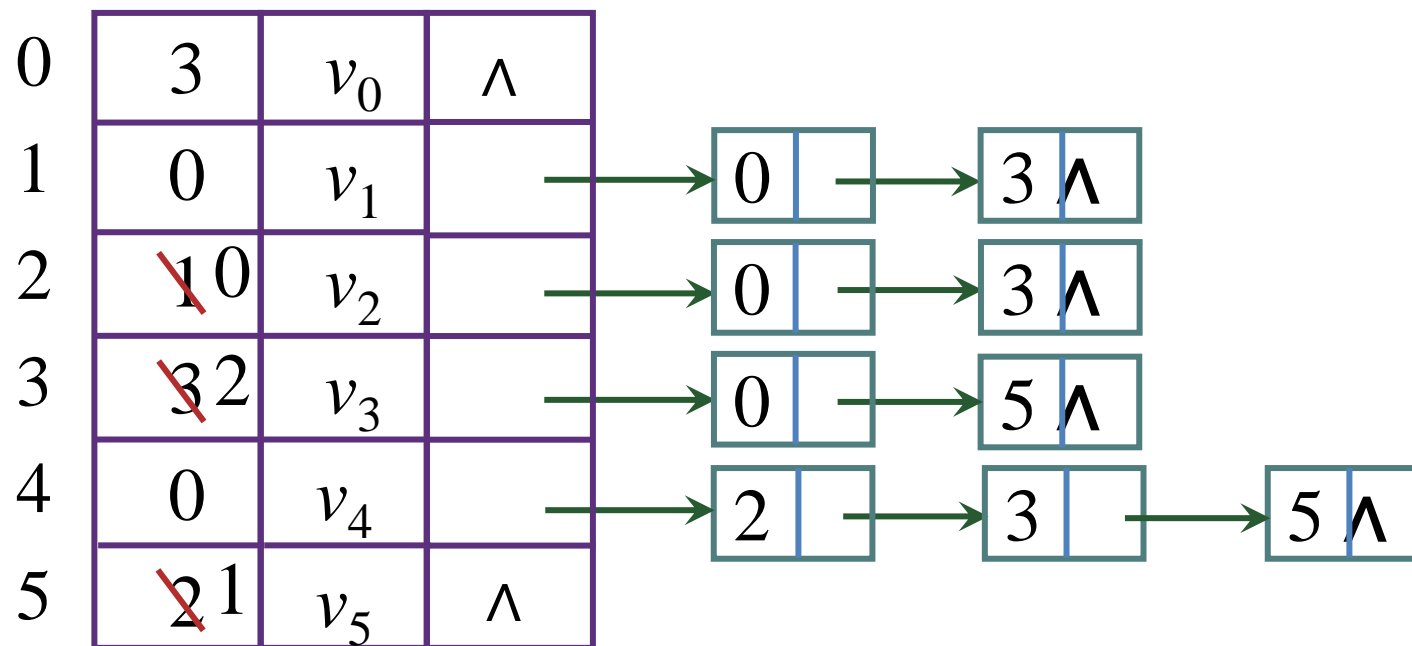
拓扑排序算法——C++实现

```
while (top != -1 )  
{  
    j = S[top--];  
    cout << adjlist[j].vertex;  
    count++;  
}
```

```
p = adjlist[j].first;  
while (p != nullptr)  
{  
    k = p->adjvex; adjlist[k].in--;  
    if (adjlist[k].in == 0) S[++top] = k;  
    p = p->next;  
}
```



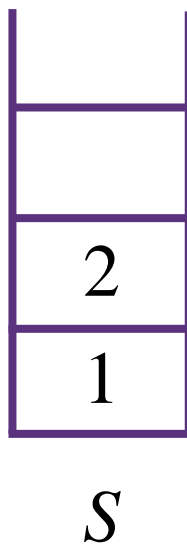
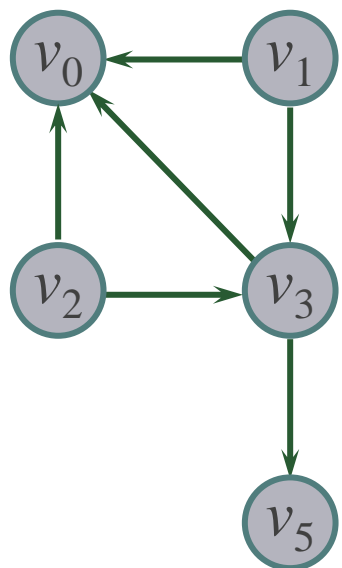
in vertex firstEdge



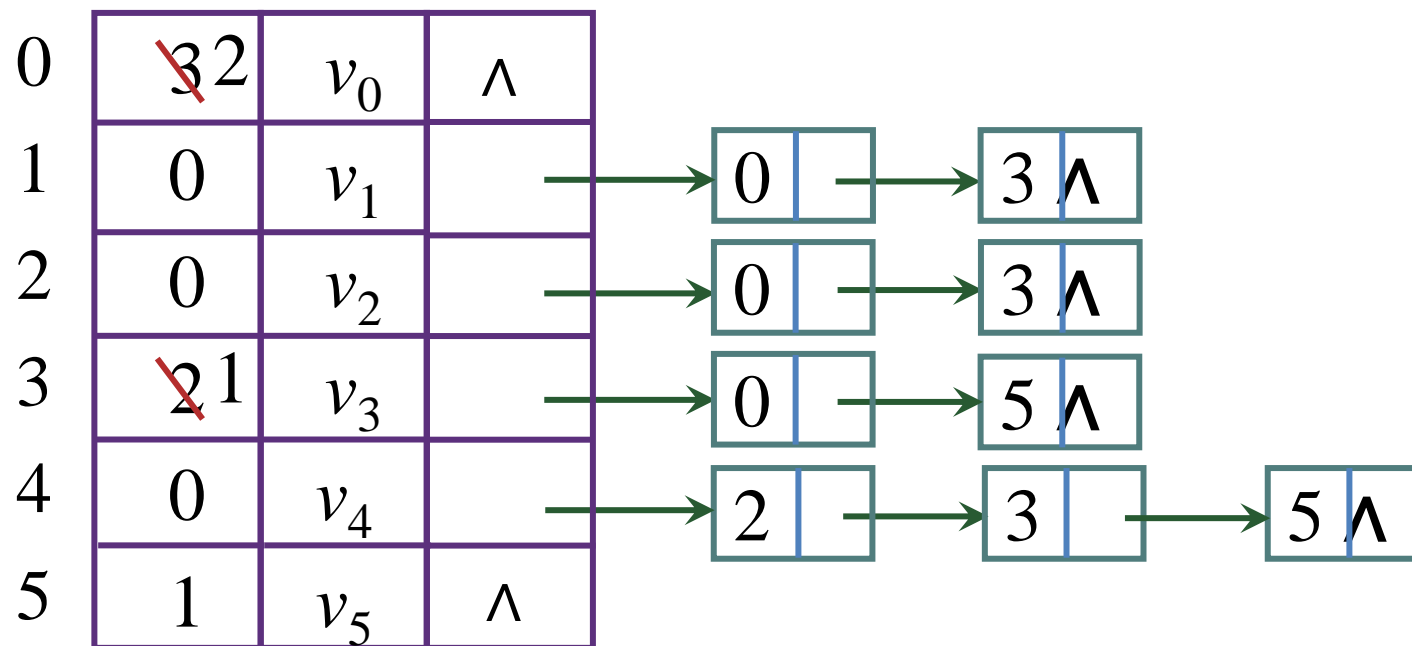
拓扑排序算法——运行实例

```
while (top != -1 )  
{  
    j = S[top--];  
    cout << adjlist[j].vertex;  
    count++;  
}
```

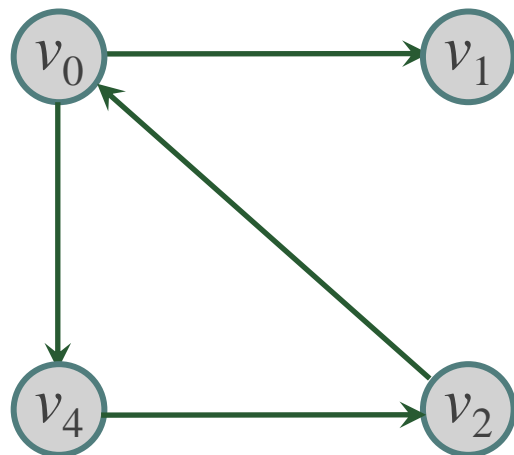
```
p = adjlist[j].first;  
while (p != nullptr)  
{  
    k = p->adjvex; adjlist[k].in--;  
    if (adjlist[k].in == 0) S[++top] = k;  
    p = p->next;  
}
```



in vertex firstEdge





G有回路，应用上述拓扑排序算法后必然有未处理结点，即会出现 $\text{count} < \text{vertexNum}$



拓扑排序算法也应用于判断图中是否存在回路

拓扑排序算法——C++实现

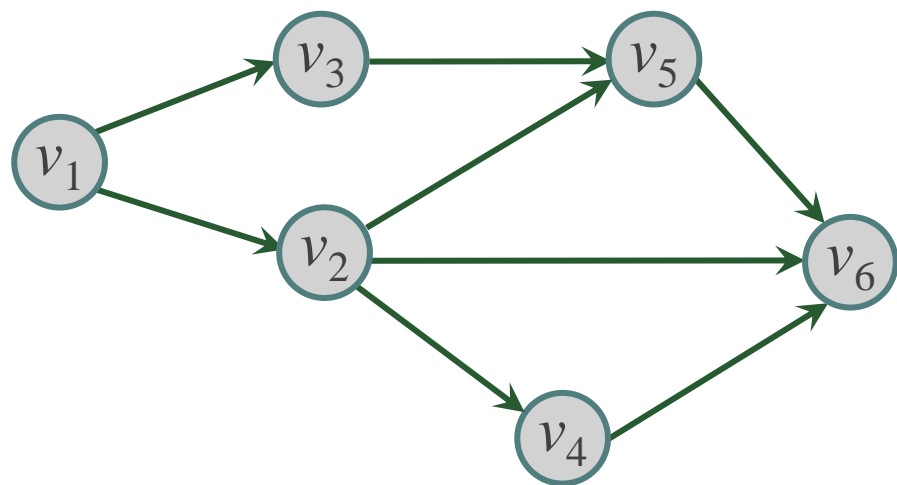
```
void TopSort( )
{
    int i, j, k, count = 0, S[MaxSize], top = -1;
    EdgeNode *p = nullptr;
    for (i = 0; i < vertexNum; i++)           /*扫描顶点表*/
        if (adjlist[i].in == 0) S[++top] = i;   }  $O(n)$ 
    while (top != -1 )                         /*当栈中还有入度为0的顶点时*/
    {
        j = S[top--]; cout << adjlist[j].vertex; count++;
        p = adjlist[j].first;
        while (p != nullptr)                  /*描顶点表，找出顶点j的所有出边*/
        {
            k = p->adjvex; adjlist[k].in--;
            if (adjlist[k].in == 0) S[++top] = k; /*将入度为0的顶点入栈*/
            p = p->next;
        }
    }
    if (count < vertexNum) cout << "有回路";
}
```

 时间复杂度?  $O(n+e)$

基于广度优先的拓扑排序

基于深度优先的拓扑排序

深度优先遍历次序（编号小者优先）



v1 v2 v4 v6 v5 v3

上述深度遍历序列符合拓扑序列定义吗？

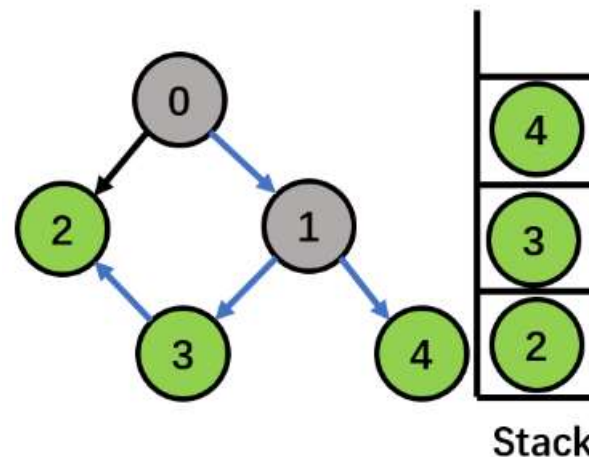
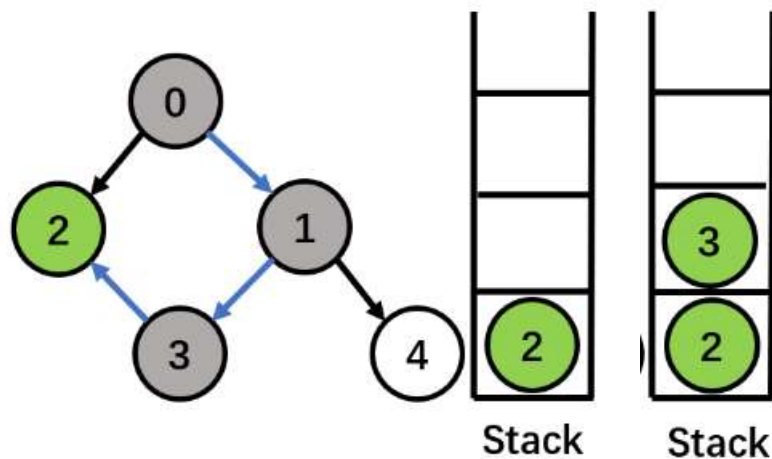
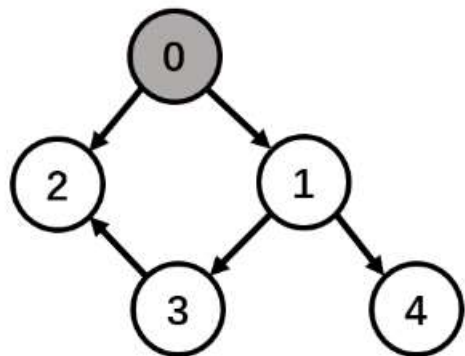
基于深度优先的拓扑排序

- 深度优先实现拓扑排序的基本思想是：

对于一个特定节点，如果该节点的**所有相邻节点都已经搜索完成**，则**该节点也会变成已经搜索完成的节点**，在拓扑排序中，该节点位于其所有**相邻节点**的前面。

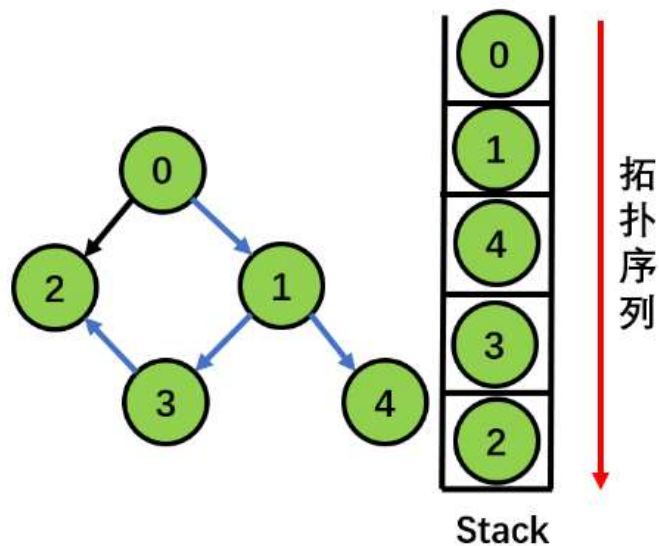
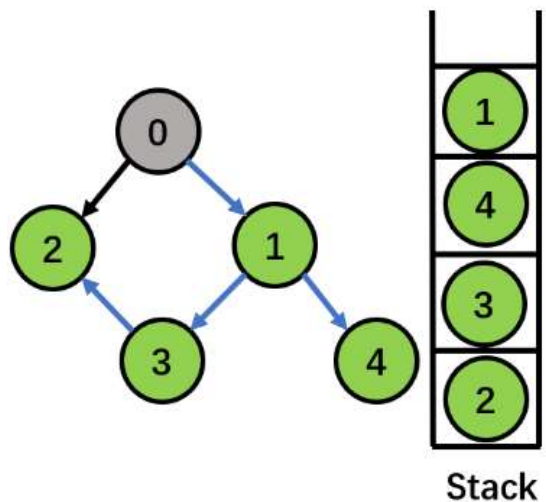
一个节点的**相邻节点**指的是从该节点出发通过一条有向边可以到达的节点。

基于深度优先的拓扑排序



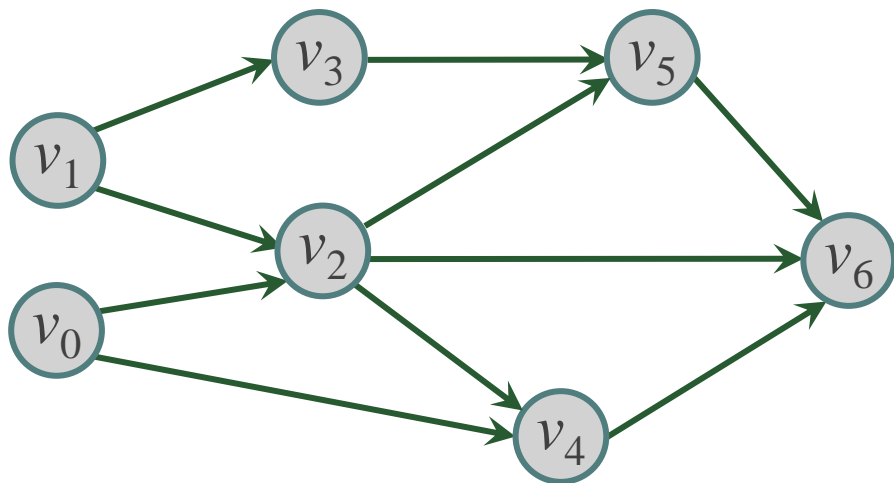
注意：这里与深度遍历不同，不是边遍历边打印，而是要走到无后继结点，才开始进栈

注意：要把1的所有邻接点都处理完才可以访问1





深度优先搜索拓扑排序



v1 v3 v0 v2 v5 v4 v6



深度优先搜索拓扑排序递归算法

```
template <typename DataType>
void ALGraph<DataType> :: DFTopo(int v, vector<int> &topoOrder)
{
    int j; EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1;
    p = adjlist[v].firstEdge;
    while (p != nullptr)
    {
        j = p->adjvex;
        if (visited[j] == 0) DFTopo(j,topoOrder);
        p = p->next;
    }
    topoOrder.insert(topoOrder.begin(),v);
}
```



深度优先搜索拓扑排序主方法

```
template <typename DataType>
void ALGraph<DataType> :: DFTopo()
    vector<int> topoOrder;
    for (i = 0; i < MaxSize; i++)
        visited[i] = 0;
    for (i = 0; i < vexnum; i++)
        if (!visited[i])
            ALG.DFTopo(i,topoOrder);
            //从顶点i出发进行深度优先拓扑排序
    for (v:topoOrder)
        cout<<v;    //输出拓扑序列
```



关键路径

AOE 网的定义

🕒 什么是工程？工程有什么共性？

几乎所有的工程都可以分为若干个称作活动的子工程

活动之间存在某些制约关系

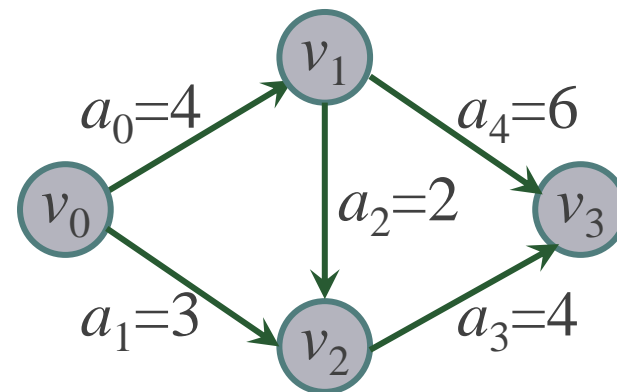
每个活动通常需要一个持续的时间

📌 源点：整个工程的开始点，其入度为0

📌 终点：整个工程的结束点，其出度为0

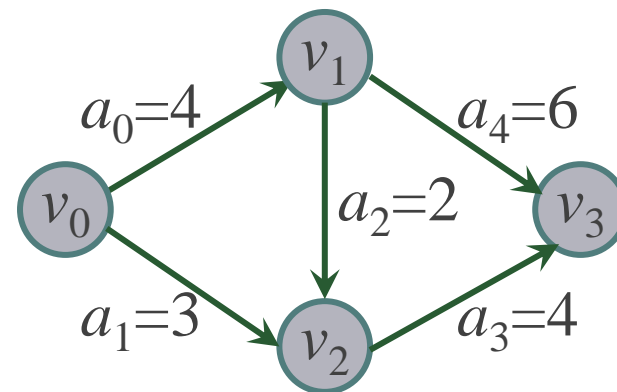
📌 AOE网（边表示活动的网）：在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动的持续时间

AOE网 (activity on edge network)



AOE 网的定义

事件	事件含义
v_0	源点, 整个工程开始
v_1	活动 a_0 完成, 活动 a_2 和 a_4 可以开始
v_2	活动 a_1 和 a_2 完成, 活动 a_3 可以开始
v_3	活动 a_3 和 a_4 完成, 整个工程结束



AOE网的性质:

- (1) 只有在进入某顶点的各活动都已经结束, 该顶点所代表的事件才能发生
- (2) 只有在某顶点所代表的事件发生后, 从该顶点出发的各活动才能开始

AOE 网的定义

🕒 AOE网能够解决什么问题？

- (1) 完成整个工程至少需要多少时间？
- (2) 为缩短完成工程所需的时间，应当加快哪些活动？

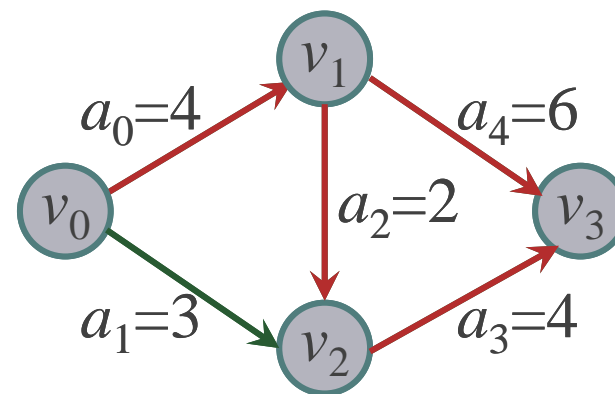
🕒 这个AOE网的最短工期是多少？

📌 关键路径：AOE网中从源点到终点的**最长**路径

📌 关键活动：关键路径上的活动

不按期完成关键活动就会影响整个工程的进度

换言之，要缩短整个工期，必须加快关键活动的进度



关键路径算法——基本思想

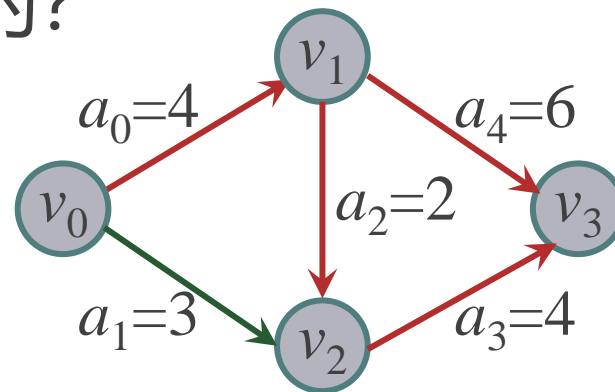
🕒 如何求关键路径呢？ \Rightarrow 求关键活动

🕒 如何求关键活动呢？ 关键活动为什么是关键的呢？

关键活动的开始时间不能推迟



关键活动的最早开始时间和最晚开始时间相等



算法：关键路径算法

输入：带权有向图 $G=(V, E)$

输出：关键活动

1. 计算各个活动的最早开始时间和最晚开始时间
2. 计算各个活动的时间余量，时间余量为 0 即为关键活动

关键路径算法——存储结构

算法：关键路径算法

输入：带权有向图 $G=(V, E)$

输出：关键活动

1. 计算各个活动的最早开始时间和最晚开始时间
2. 计算各个活动的时间余量，时间余量为 0 即为关键活动

设带权有向图 $G=(V, E)$ 含有 n 个顶点 e 条边，设置 **4 个一维数组**：

- (1) 事件的**最早**发生时间 $ve[n]$
- (2) 事件的**最迟**发生时间 $vl[n]$ ：
- (3) 活动的**最早**开始时间 $ae[e]$
- (4) 活动的**最晚**开始时间 $al[e]$

关键路径算法——存储结构

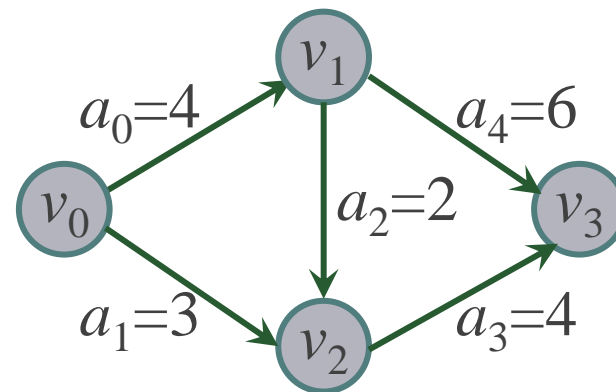
(1) 事件的**最早**发生时间 $ve[n]$

$$\begin{cases} ve[0] = 0 \\ ve[k] = \max\{ve[j] + \text{len}\langle v_j, v_k \rangle\} \end{cases}$$

$(\langle v_j, v_k \rangle \in p[k])$ $p[k]$: 所有到达 v_k 的有向边

 事件 v_2 的最早发生时间是多少?

$$ve[2] = \max\{ve[0] + a_1, ve[1] + a_2\} = \{0 + 3, 4 + 2\} = 6$$



AOE网的性质：只有进入 v_k 的所有活动 $\langle v_j, v_k \rangle$ 都结束， v_k 代表的事件才能发生

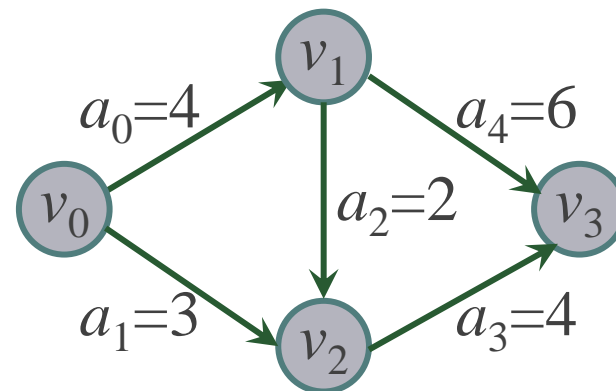
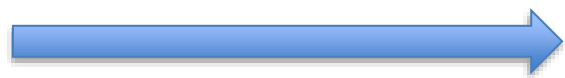
关键路径算法——运行实例

(1) 事件的最早发生时间 $ve[n]$

$$\begin{cases} ve[0] = 0 \\ ve[k] = \max\{ve[j] + \text{len}\langle v_j, v_k \rangle\} \end{cases}$$

$p[k]$: 所有到达 v_k 的有向边

	v_0	v_1	v_2	v_3
$ve[k]$	0	4	6	10



注意：这里要按拓扑序列进行处理

(v_0, v_1, v_2, v_3)

关键路径算法——存储结构

(2) 事件的最迟发生时间 $vl[n]$

$$\begin{cases} vl[n-1] = ve[n-1] \\ vl[k] = \min\{vl[j] - \text{len}\langle v_k, v_j \rangle\} \quad (\langle v_k, v_j \rangle \in s[k]) \end{cases} \quad s[k]: \text{所有从 } v_k \text{ 发出的有向边}$$

🕒 事件 v_3 的最迟发生时间是多少？

$$vl[3] = ve[3] = 10$$

🕒 事件 v_2 的最迟发生时间是多少？

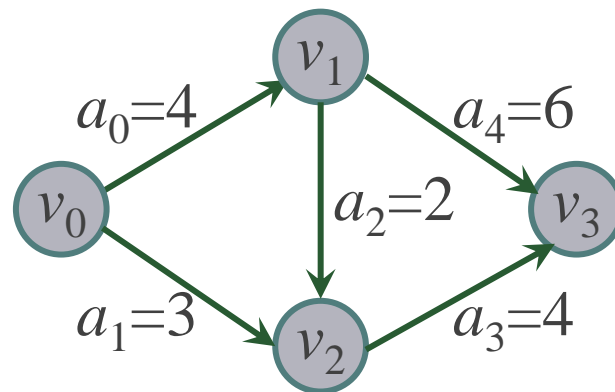
$$vl[2] = vl[3] - a_3 = 6$$

🕒 事件 v_1 的最迟发生时间是多少？

$$vl[1] = \min\{vl[3] - a_4, vl[2] - a_2\} = \{10 - 4, 6 - 2\} = 4$$

🕒 事件 v_0 的最迟发生时间是多少？

$$vl[0] = \min\{vl[1] - a_0, vl[2] - a_1\} = \{4 - 4, 6 - 3\} = 0$$

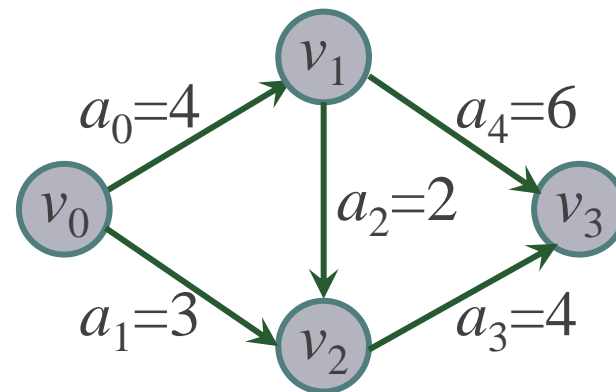


关键路径算法——运行实例

(2) 事件的最迟发生时间 $vl[n]$

$$\begin{cases} vl[n-1] = ve[n-1] \\ vl[k] = \min\{vl[j] - \text{len}\langle v_k, v_j \rangle\} \quad (\langle v_k, v_j \rangle \in s[k]) \end{cases} \quad s[k] : \text{所有从 } v_k \text{ 发出的有向边}$$

	v_0	v_1	v_2	v_3
$ve[k]$	0	4	6	10
$vl[k]$	0	4	6	10



注意：求 vl 要按逆拓扑序列进行处理

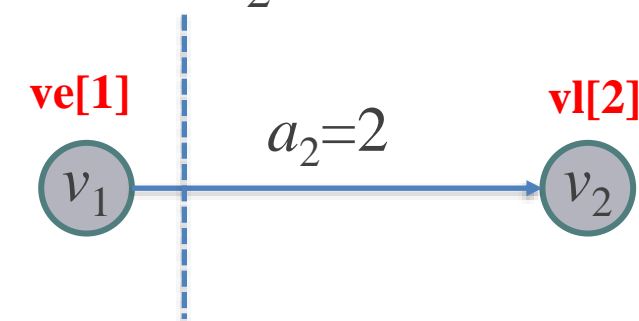
关键路径算法——存储结构

(3) 活动的最早开始时间 $ae[e]$

(4) 活动的最晚开始时间 $al[e]$



活动 a_2 的最早、最晚开始时间是多少？



a_2 活动的最晚开始时间就是
 a_2 最晚结束时间-持续长度

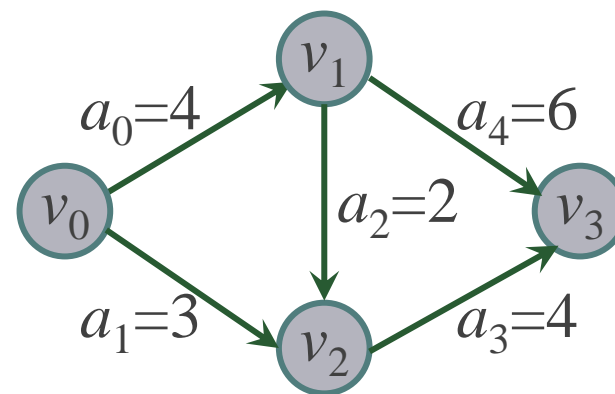
v_2 事件的最晚开始时间就是
 a_2 可以最晚结束的时间

$$ae[2] = ve[1] = 4$$

$$al[2] = vl[2] - 2 = 4$$

若活动 a_i 由有向边 $\langle v_k, v_j \rangle$ 表示, 则

$$\begin{cases} ae[i] = ve[k] \\ al[i] = vl[j] - len\langle v_k, v_j \rangle \end{cases}$$



	v_0	v_1	v_2	v_3
$ve[k]$	0	4	6	10
$vl[k]$	0	4	6	10

关键路径算法——运行实例

(3) 活动的**最早**开始时间 $ae[e]$

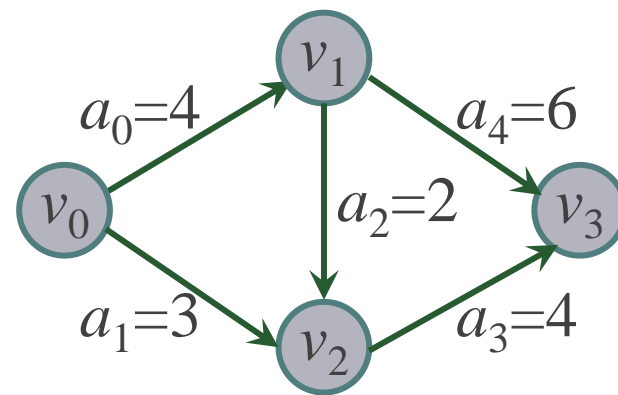
若活动 a_i 由有向边 $\langle v_k, v_j \rangle$ 表示, 则

$$\begin{cases} ae[i] = ve[k] \\ al[i] = vl[j] - len\langle v_k, v_j \rangle \end{cases}$$

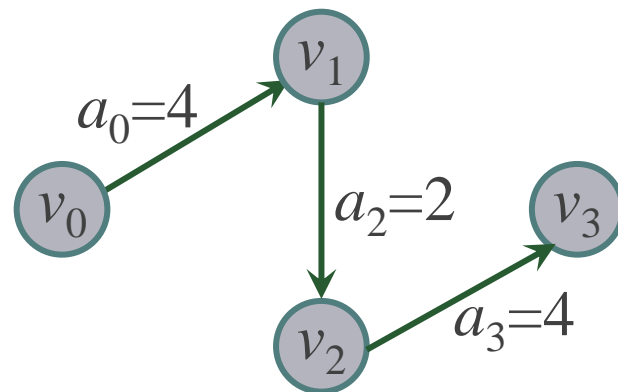
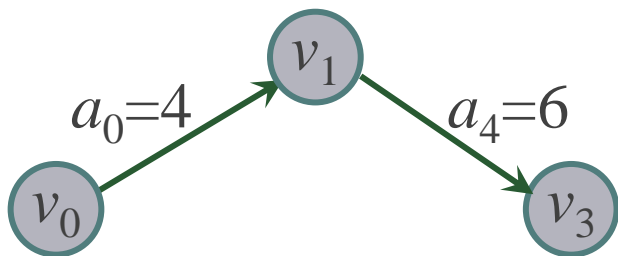
	v_0	v_1	v_2	v_3
$ve[k]$	0	4	6	10
$vl[k]$	0	4	6	10

	a_0	a_1	a_2	a_3	a_4
$ae[i]$	0	0	4	6	4
$al[i]$	0	3	4	6	4

(4) 活动的**最晚**开始时间 $al[e]$



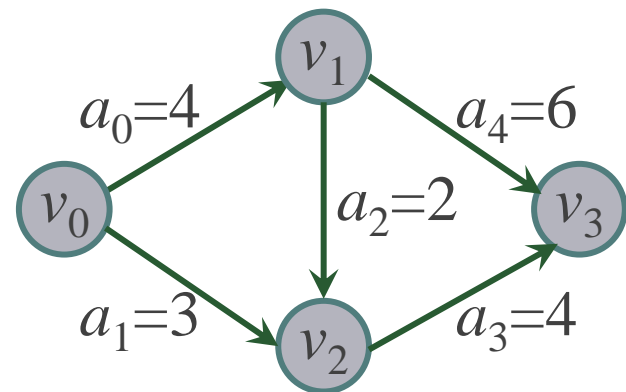
关键路径算法——运行实例



关键路径

	v_0	v_1	v_2	v_3
ve[k]	0	4	6	10
vl[k]	0	4	6	10

	a_0	a_1	a_2	a_3	a_4
ae[i]	0	0	4	6	4
al[i]	0	3	4	6	4





求关键路径算法

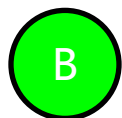
求解关键路径步骤

- ① 求解AOE网络的一个拓扑序列 $T = \{v_1, v_2, \dots, v_n\}$;
- ② $\text{earliest}[v_1] = 0$;
- ③ For $i = 2$ to n do
- ④ $\text{earliest}[v_i] = \max_{(v_j, v_i) \in E} \{\text{earliest}[v_j] + \text{weight}(v_j, v_i)\}$;
- ⑤ $\text{latest}[v_n] = \text{earliest}[v_n]$;
- ⑥ For $i = n$ to 1 do
- ⑦ $\text{latest}[v_i] = \min_{(v_i, v_j) \in E} \{\text{latest}[v_j] - \text{weight}(v_i, v_j)\}$;
- ⑧ For $a_i = (u, v) \in E$ do
- ⑨ $e[i] = \text{earliest}[u]$;
- ⑩ $l[i] = \text{latest}[v] - \text{weight}(u, v)$;
- ⑪ if $e[i] = l[i]$ then
- ⑫ a_i 为关键活动;
- ⑬ 选取所有的关键活动组成关键路径;

1. 在一个有向图的拓扑序列中，若顶点a在顶点b之前，则图中必有一条弧 $\langle a, b \rangle$ 。



正确



错误

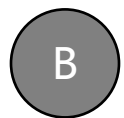
提交

2. 若一个有向图的邻接矩阵中对角线以下元素均为零，则该图的拓扑序列必定存在。



A

正确



B

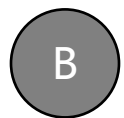
错误

提交

3. 拓扑排序算法可以用栈或者队列保存入度为0的顶点。



正确



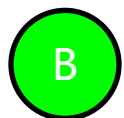
错误

提交

4. 在AOV网中不可能出现回路，因此一定存在拓扑序列。



正确



错误

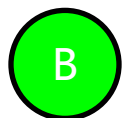
提交

1. 在AOE网中一定只有一条关键路径。



A

正确



B

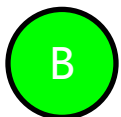
错误

提交

2. 加快关键活动的进度一定会缩短最短工期。



正确



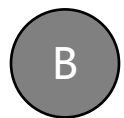
错误

提交

3. 关键活动的最早开始时间和最晚开始时间都不能推迟，否则会影响整个工期。



正确



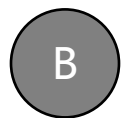
错误

提交

4. 求出所有事件的最早发生时间后，即可确定最短工期。



正确



错误

提交

5. 如图6-14所示AOE网，活动a2的最早开始时间和最晚开始时间是（ ）。

- ☐ A 3, 4
- ☒ B 4, 4
- ☐ C 3, 5
- ☐ D 4, 5

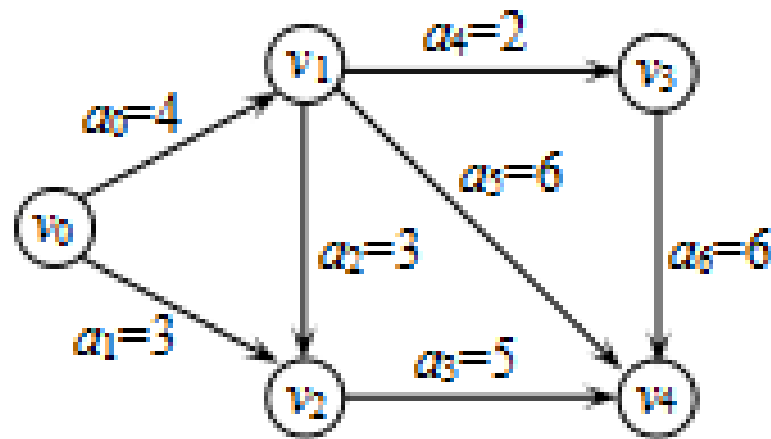


图 6-14 一个 AOE 网

6. 如图6-14所示AOE网，该AOE网的最短工期是（ ）。

- ☐ A 8
- ☐ B 10
- ☒ C 12
- ☐ D 16

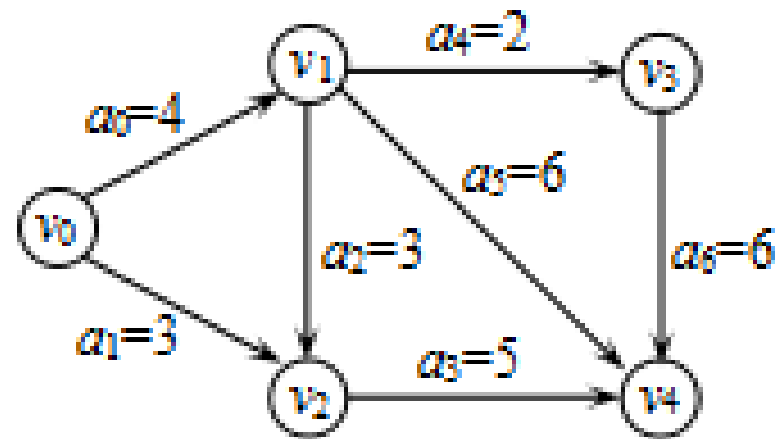


图 6-14 一个 AOE 网

提交

7. 如图6-14所示AOE网，事件 v_2 的含义是什么？

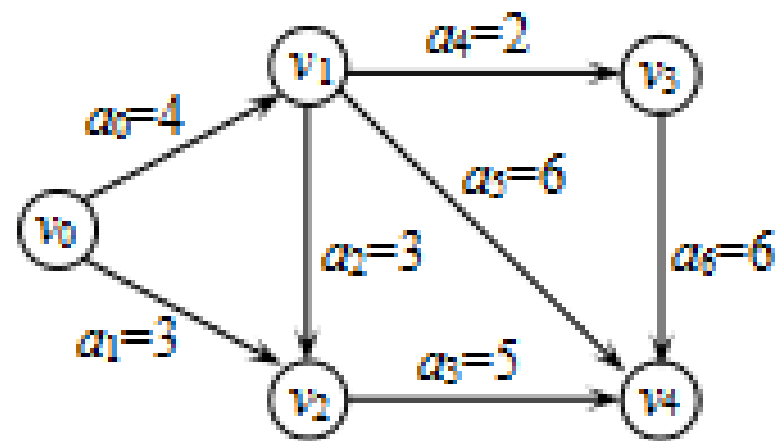


图 6-14 一个 AOE 网