



# Prim算法的实现

---

```
#include "heap.h"
template <class T, class E>
void Prim (Graph<T, E>& G, const T u0,
           MinSpanTree<T, E>& MST)
{
    MSTEdgeNode<T, E> ed; //边结点辅助单元
    int i, u, v, count;
    int n = G.NumberOfVertices(); //顶点数
    int m = G.NumberOfEdges();    //边数
    int u = G.getVertexPos(u0);   //起始顶点号
    MinHeap <MSTEdgeNode<T, E>> H(m); //最小堆
    bool Vmst = new bool[n]; //最小生成树顶点集合
```



---

MinHeap <MSTEdgeNode<T, E>> H(m); //最小堆

bool Vmst = new bool[n]; //最小生成树顶点集合

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

    Vmst[i] = false;

Vmst[u] = true; //u 加入生成树



```
count = 1;  
do { //迭代  
    v = G.getFirstNeighbor(u);
```

```
while (v != -1)
```

```
    //检测u所有邻接顶点
```

```
    if (!Vmst[v])
```

```
    { //v不在mst中
```

```
        ed.tail = u; ed.head = v;
```

```
        ed.cost = G.getWeight(u, v);
```

```
        H.Insert(ed); //(u,v)加入堆
```

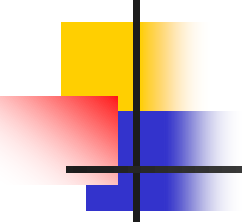
```
    } //堆中存所有u在mst中, v不在mst中的边
```

```
    v = G.getNextNeighbor(u, v);
```

```
}
```

执行e次

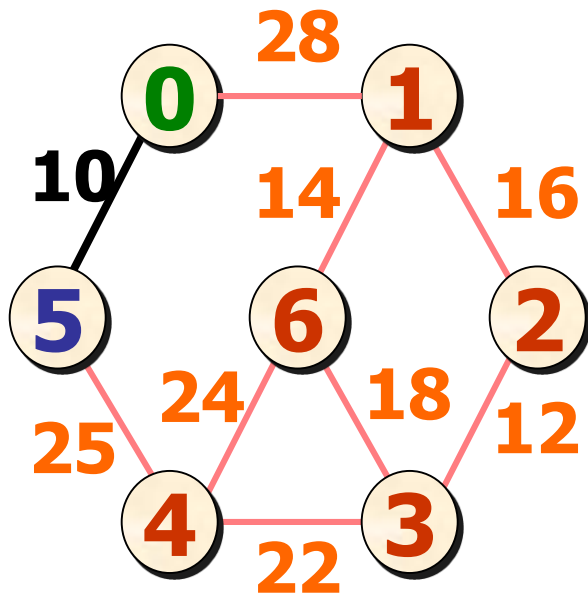
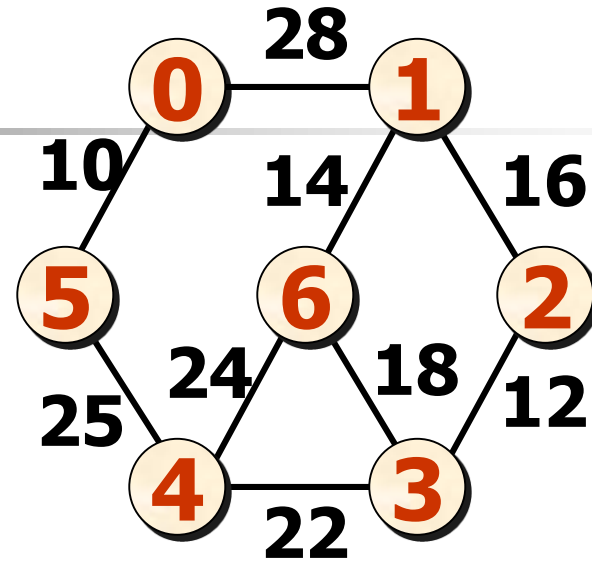
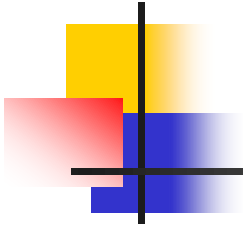
**$O(\log_2 e)$**



```
while (!H.IsEmpty() && count < n)
{
    H.Remove(ed); //选堆中具最小权的边
    if (!Vmst[ed.head])
    {
        MST.Insert(ed); //加入最小生成树
        u = ed.head; Vmst[u] = true;
        //u加入生成树顶点集合
        count++;
        break;
    }
}
```

```
} while (count < n);
} // end of prim
```

**//prim**算法是对连通图而言



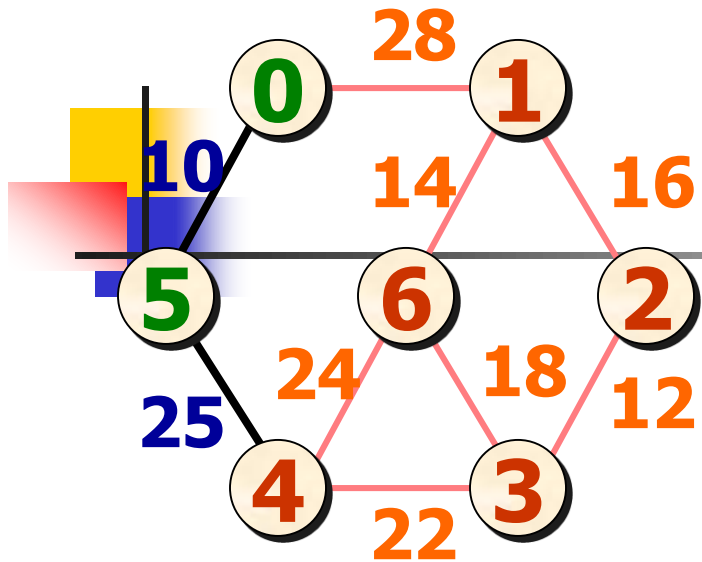
$H = \{(0,5,10), (0,1,28)\}$

$ed = (0, 5, 10)$

$V_{mst} = \{t, f, f, f, f, f, f\}$



$V_{mst} = \{t, f, f, f, f, t, f\}$



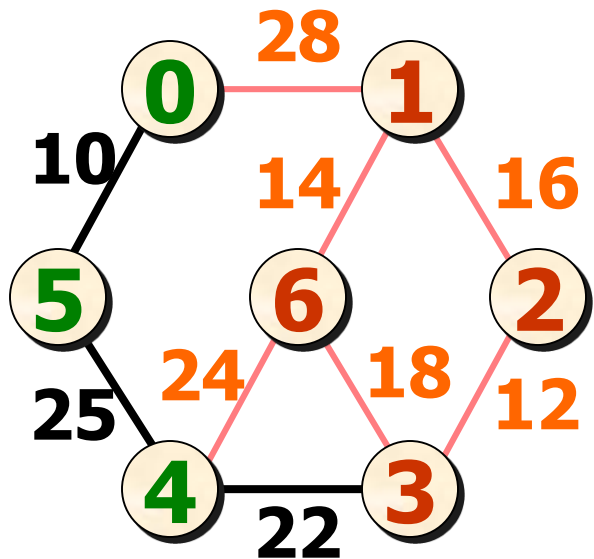
$H = \{(5,4,25), (0,1,28)\}$

$ed = (5, 4, 25)$

$V_{mst} = \{t, f, f, f, f, t, f\}$



$V_{mst} = \{t, f, f, f, t, t, f\}$



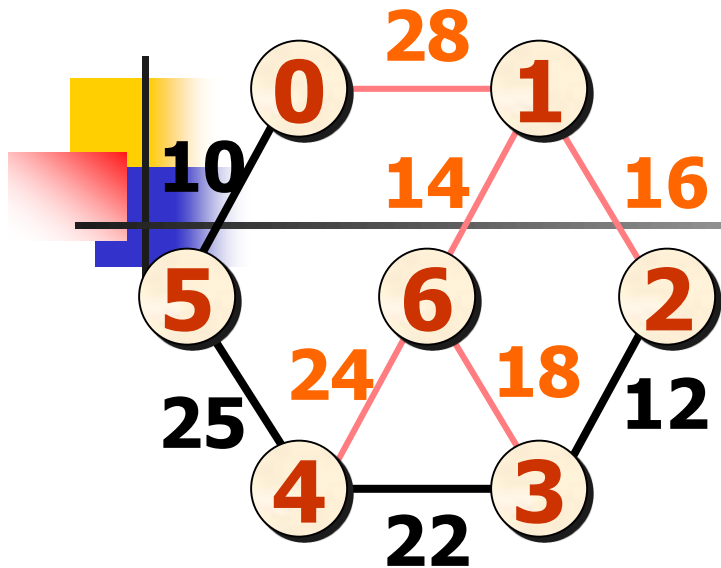
$H = \{(4,3,22), (4,6,24), (0,1,28)\}$

$ed = (4, 3, 22)$

$V_{mst} = \{t, f, f, f, t, t, f\}$



$V_{mst} = \{t, f, f, t, t, t, f\}$



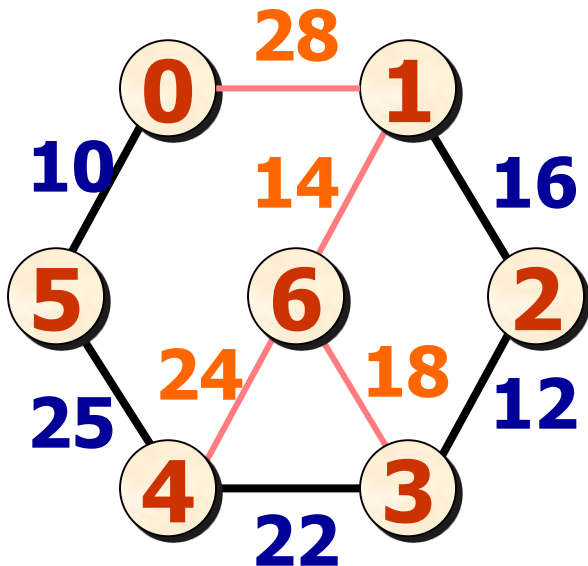
$H = \{(3,2,12), (3,6,18), (4,6,24), (0,1,28)\}$

$ed = (3, 2, 12)$

$V_{mst} = \{t, f, f, t, t, t, f\}$



$V_{mst} = \{t, f, t, t, t, t, f\}$



$H = \{(2,1,16), (3,6,18), (4,6,24), (0,1,28)\}$

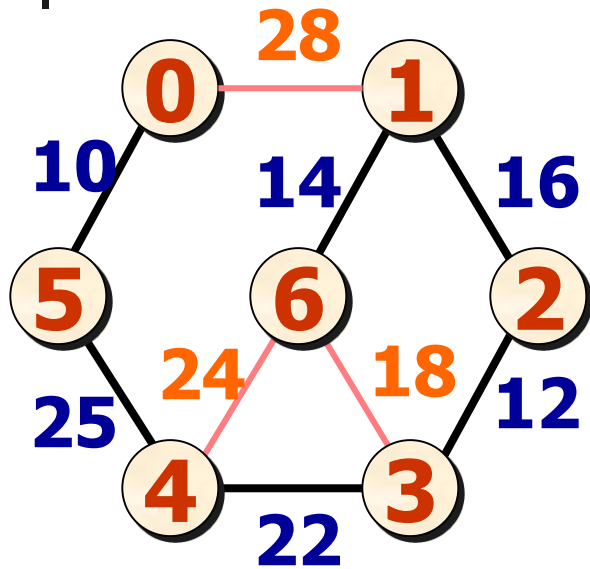
$ed = (2, 1, 16)$

$V_{mst} = \{t, f, t, t, t, t, f\}$



$V_{mst} = \{t, t, t, t, t, t, f\}$





$H = \{ (1,6,14), (3,6,18), (4,6,24), (0,1,28) \}$

$ed = (1, 6, 14)$

$V_{mst} = \{ t, t, t, t, t, t, f \}$

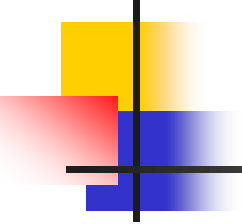


$V_{mst} = \{ t, t, t, t, t, t, t \}$

- 最小生成树中边集合里存入的各条边为:

$(0, 5, 10), (5, 4, 25), (4, 3, 22),$

$(3, 2, 12), (2, 1, 16), (1, 6, 14)$

- 
- 
- prim算法适用于边稠密的网络。
  - Kruskal算法不仅适合于边稠密的情形，也适合于边稀疏的情形。
  - 注意：

当各边有相同权值时，由于选择的随意性，产生的生成树可能不唯一。



# Graph

---

- Basic Concepts
  - Representation
  - Adjacency Matrices
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

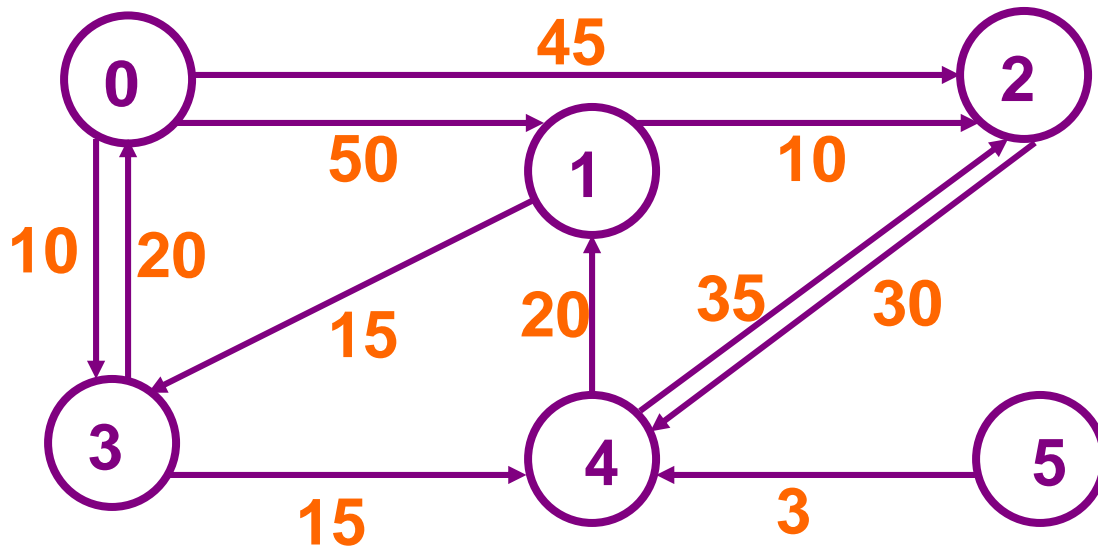


# Shortest Paths

---

- Single Source/All Destinations: Nonnegative Edge Cost
- Problem: given a digraph  $G=(V, E)$ , a length function  $\text{length}(i, j) \geq 0$  for  $\langle i, j \rangle \in E(G)$ , and a source vertex  $v$ , to determine the shortest path from  $v$  to all remaining vertices of  $G$ .

# Example



Shortest  
paths from 0

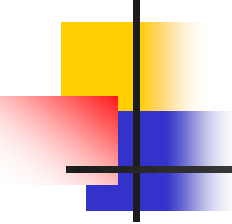
	path	Length
1)	<b>0, 3</b>	10
2)	<b>0, 3, 4</b>	25
3)	<b>0, 3, 4, 1</b>	45
4)	<b>0, 2</b>	45



# 最短路径

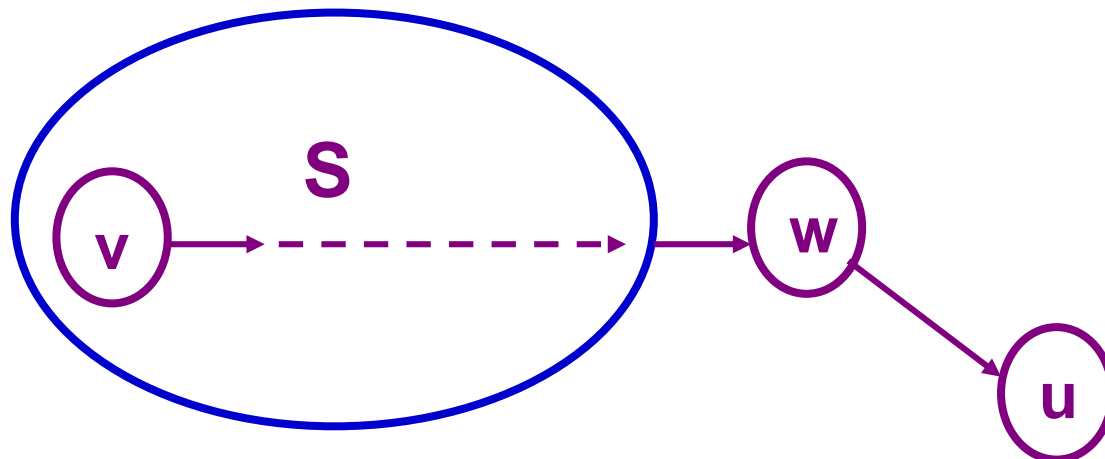
---

- 问题解法
  - ◆ 边上权值非负情形的单源最短路径问题  
— Dijkstra算法
  - ◆ 边上权值为任意值的单源最短路径问题  
— Bellman&Ford算法
  - ◆ 所有顶点之间的最短路径  
— Floyd算法

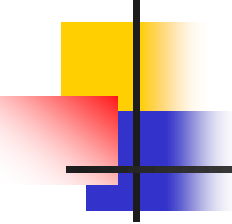
- 
- 
- $S$ : the set of vertices (including  $v$ ) to which the shortest paths have been found.
  - For  $w \notin S$ ,  $\text{dist}[w]$ : the length of the shortest path starting from  $v$ , going through only the vertices in  $S$ , and ending at  $w$ .

- Assume paths are generated in non-decreasing order of length, observe:

(1) If the next shortest path is to  $u$ , it begins at  $v$ , ends at  $u$ , and goes through only vertices in  $S$ .



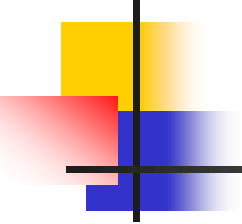




---

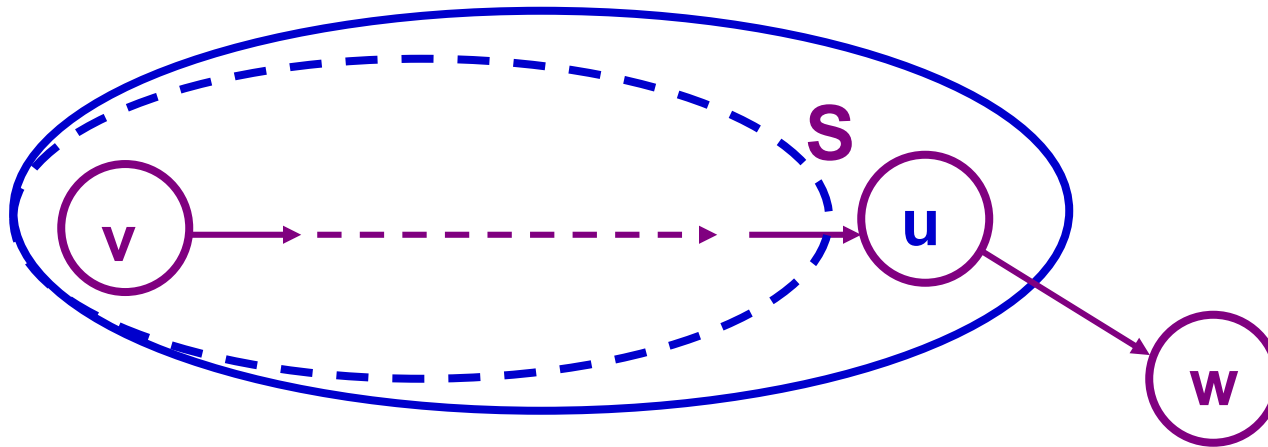
(2) The destination of the next path generated must be  $u$ , such that

$$\text{dist}[u] = \min_{v \notin S} \{ \text{dist}[v] \}$$

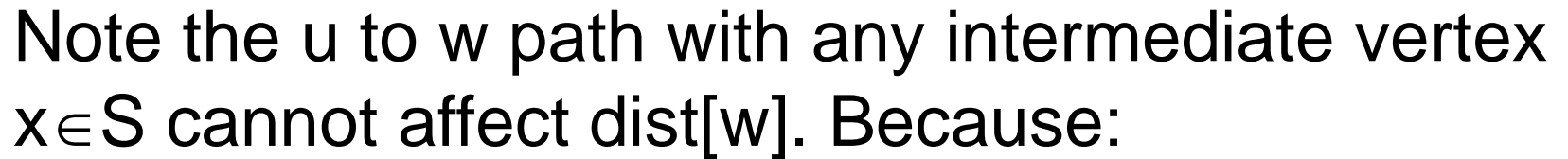


---

(3) Having selected a vertex  $u$  as in (2),  $u$  becomes a member of  $S$ . Now  $\text{dist}[w]$ ,  $w \notin S$ , may change. If it does, it must be due to a shorter path from  $v$  to  $u$  and then to  $w$ .



- The v to u path must be the shortest v to u path, and the u to w path is just the edge  $\langle u, w \rangle$ . If  $\text{dist}[w]$  is to decrease, it is because the current  $\text{dist}[w] > \text{dist}[u] + \text{length}(u, w)$ .



- ①  $\text{length}(v-u-x) > \text{dist}[x]$
- ②  $\text{length}(v-u-x-w) > \text{length}(v-x-w) \geq \text{dist}[w]$

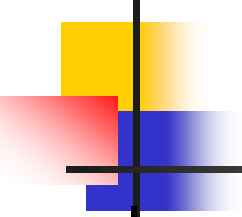




---

Assume:

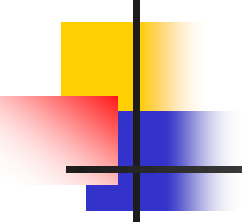
- $n$  vertices numbered  $0, 1, \dots, n-1$ .
- $s[i]=\text{false}$  if  $i \notin S$ ,  $s[i]=\text{true}$  if  $i \in S$ .
- $\text{length}[i][j]$ . If  $\langle i, j \rangle \notin E(G)$  and  $i \neq j$ , set  $\text{length}[i][j]$  to some large number.



---

```
class MatrixWDigraph {  
private:  
    double length[NMAX][NMAX]; // NMAX is a constant  
    double *dist;  
    bool *s;  
public:  
    void ShortestPath(const int, const int);  
    int choose(const int);  
};
```

And we assume the constructor will apply space for array dist and s.



```
1 void MatrixDigraph::ShortestPath(const int n, const int v)
2 { //dist[j], 0 ≤ j < n, is set to the length of the shortest path(v-j)
3 //in a digraph G with n vertices and edge lengths in length[i][j]
4   for (int i=0; i<n; i++) { s[i]=false; dist[i]=length[v][i];}
5   s[v]=true;
6   dist[v]=0;
7   for (i=0; i<n-2; i++) { //determine n-1 paths from v
8     int u=choose(n); //choose returns a value u such that
9       //dist[u]=minimum dist[w], where s[w]=false
10     •   s[u]= true;
11     for ( int w=0; w<n; w++)
12       if (!s[w] && dist[u]+length[u][w]<dist[w])
13         dist[w]=dist[u]+length[u][w];
14   }
15 }
```



# Analysis of ShortestPath

---

- The time for loop of line 4 is  $O(n)$ . The for loop of line 7 is executed  $n-2$  times, each requires  $O(n)$  at line 8 and at line 11 to 13. So the total time is  $O(n^2)$ .
- Even if adjacency lists are used, the overall time for line 11 to 13 can be brought down to  $O(e)$ , but for line 8 remains  $O(n^2)$ .





# Dijkstra算法

- 用于求带权有向图中某个源点到其余各顶点的最短路径

- **S[]**数组：记录已求得的最短路径的顶点

**dist[]**：记录从源点 $V_0$ 到其他各顶点当前的最短路径长度，**dist[i]**的初值为**arcs[v<sub>0</sub>][i]**

**Path[]**:**path[i]**表示从源点到顶点*i*之间的最短路径的前驱节点

**Arcs[i][j]**:邻接矩阵**arcs**中有向边<*i,j*>的权值

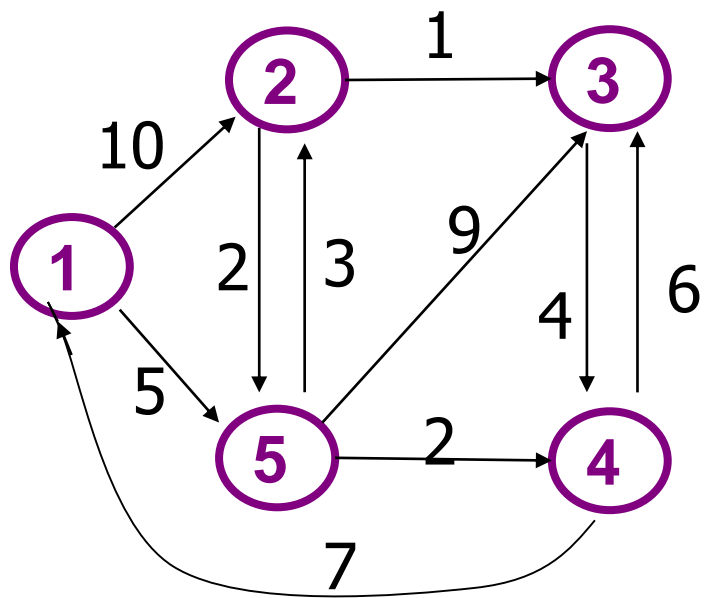


# Dijkstra算法

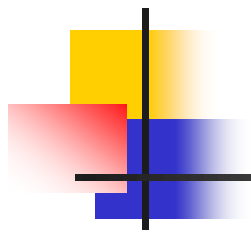
---

- 初始化：集合 $S$ 初始为 $\{0\}$ ， $\text{dist}[]$ 的初始值  
 $\text{dist}[i] = \text{arcs}[0][i]$
- 从顶点集合 $V-S$ 中选出 $v_j$ ，满足  
 $\text{dist}[j] = \min\{\text{dist}[i] \mid v_i \text{ 属于 } V-S\}$
- 修改从 $v_0$ 出发到集合 $V-S$ 上任一顶点 $v_k$ 可达的最短路径长度：若 $\text{dist}[j] + \text{arcs}[j][k] < \text{dist}[k]$ ，则  
 $\text{dist}[k] = \text{dist}[j] + \text{arcs}[j][k]$
- 重复上面两步操作 $n-1$ 次，直到所有的顶点都包含在 $S$ 中。

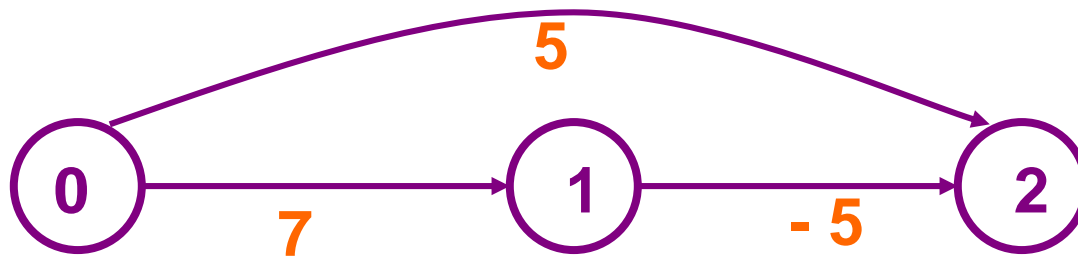
# Dijkstra算法



顶点	第1轮	第2轮	第3轮	第4轮
2	10 $v_1 \rightarrow v_2$	8 $v_1 \rightarrow v_5 \rightarrow v_2$	8 $v_1 \rightarrow v_5 \rightarrow v_2$	
3	无穷大	14 $v_1 \rightarrow v_5 \rightarrow v_3$	13 $v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_3$	9 $v_1 \rightarrow v_5 \rightarrow v_2 \rightarrow v_3$
4	无穷大	7 $v_1 \rightarrow v_5 \rightarrow v_4$		
5	5 $v_1 \rightarrow v_5$			
集合S	{1, 5}	{1, 5, 4}	{1, 5, 4, 2}	{1, 5, 4, 2, 3}

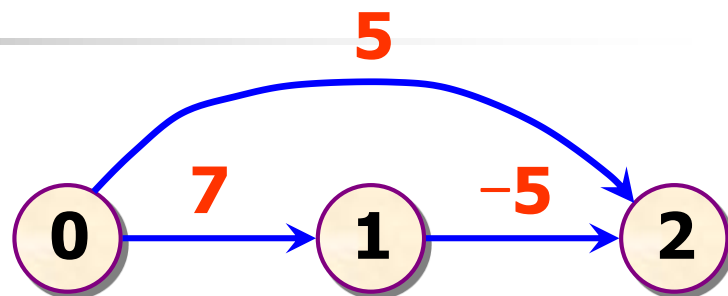


Note the algorithm does not work when some edges may have negative length, as shown below:



# 边上权值为任意值的 单源最短路径问题

- 若设源点  $v = 0$ 
  - 使用 **Dijkstra** 算法所得结果

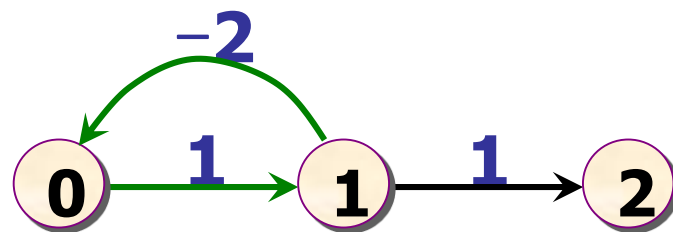
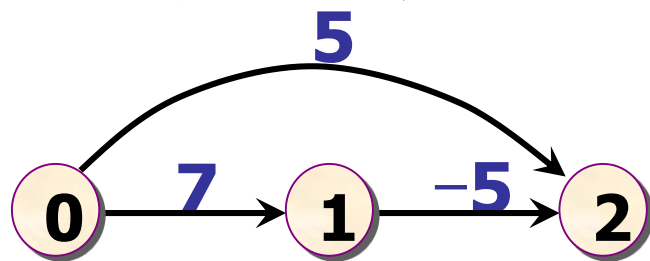


- 源点 **0** -> 终点 **2**
  - 最短路径应是 **<0, 1, 2>**，长度为 **2**，而计算出来的 **dist[2]** 的值比 **2** 大
- 显然，利用 **Dijkstra** 算法，**不一定能得到正确的结果**

# Bellman-Ford算法

从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法

- 一个限制条件: 要求图中不能包含有由带负权值的边组成的回路。



- 当图中没有由带负权值的边组成的回路时
  - 有  $n$  个顶点的图中任意两个顶点之间，如果存在最短路径，此路径最多有  $n-1$  条边。
- 以此为依据
  - 计算从源点  $v$  到其他顶点  $u$  的最短路径长度  $\text{dist}[u]$ 。



# Bellman-Ford算法

---

构造一个最短路径长度数组序列  $\text{dist}^1[u], \text{dist}^2[u], \dots, \text{dist}^{n-1}[u]$

其中：

- ◆  $\text{dist}^1[u]$ 是从源点 $v$ 到终点 $u$ 的只经过一条边的最短路径的长度  
 $\text{dist}^1[u] = \text{Edge}[v][u]$
- ◆  $\text{dist}^2[u]$ 是从源点 $v$ 出发最多经过两条边到达终点 $u$ 的最短路径长度；
- ◆  $\text{dist}^3[u]$ 是从源点 $v$ 出发最多经过不构成带负长度边回路的三条边到达终点 $u$ 的最短路径长度
- ◆ ...
- ◆  $\text{dist}^{n-1}[u]$ 是从源点 $v$ 出发最多经过不构成带负长度边回路的 $n-1$ 条边到达终点 $u$ 的最短路径长度

■ 算法的最终目的是计算出  $\text{dist}^{n-1}[u]$ 。



采用递推方式计算  $\text{dist}^k[u]$

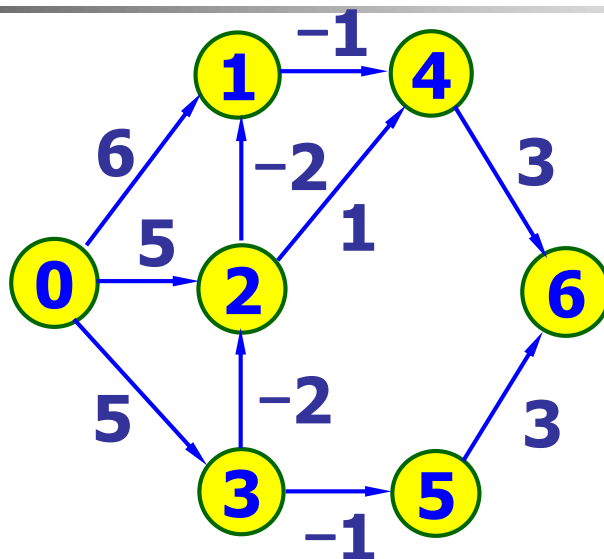
$$\text{dist}^1[u] = \text{Edge}[v][u];$$

$$\text{dist}^k[u] = \min \{ \text{dist}^{k-1}[u], \min \{ \text{dist}^{k-1}[j] + \text{Edge}[j][u] \} \}$$

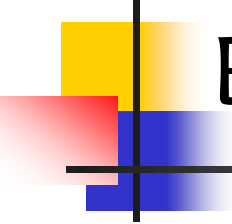
- ◆ ①设：已经求出  $\text{dist}^{k-1}[j], j = 0, 1, \dots, n-1$ 
  - ◆ 即从源点 $v$ 出发最多经过不构成带负长度边回路的  $k-1$  条边到达终点 $j$ 的最短路径长度。
- ◆ ②计算  $\min \{ \text{dist}^{k-1}[j] + \text{Edge}[j][u] \}$ 
  - ◆ 可得从源点 $v$ 绕过各顶点 $j$ ，最多经过不构成带负长度边回路的 $k$ 条边到达终点 $u$ 的最短路径长度。
- ◆ ③将②的计算值与 $\text{dist}^{k-1}[u]$ 比较，取小者作为 $\text{dist}^k[u]$ 的值。



# 图的最短路径长度



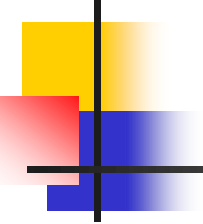
$k$	$d^k[0]$	$d^k[1]$	$d^k[2]$	$d^k[3]$	$d^k[4]$	$d^k[5]$	$d^k[6]$
1	0	6	5	5	$\infty$	$\infty$	$\infty$
2	0	3	3	5	5	4	$\infty$
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3



# Bellman-Ford 算法

---

```
template <class T>
void Bellman-Ford (Graph<T>& G, int v, E dist[], int path[])
{
    //在有向带权图中有的边具有负的权值。从顶点v
    //找到所有其他顶点的最短路径。
    E w;
    int i, k, u, n = G.NumberOfVertices();
    for (i = 0; i < n; i++)
    {
        //计算dist1[i]
        dist[i] = G.getWeight(v, i);
        if (i != v && dist[i] < maxValue)    path[i] = v;
        else path[i] = -1;
    }
}
```



```
for (k = 2; k < n; k++)    // 计算 $\text{dist}^2[i]$ 到 $\text{dist}^{n-1}[i]$ 
    for (u = 0; u < n; u++)
        if (u != v)
            for (i = 0; i < n; i++)
            {
                w = G.getWeight(i, u);
                if (w > 0 && w < maxValue && dist[u] > dist[i]+w)
                {
                    dist[u] = dist[i]+w;
                    path[u] = i;
                }
            }
    }
```



# Floyd算法

---

- 求所有顶点之间的最短路径问题
- 已知一个各边权值均大于0的带权有向图，对每对顶点 $v_i \neq v_j$ ，求出 $v_i$ 与 $v_j$ 之间的最短路径和最短路径长度



# Floyd算法

---

## ■ 基本思想

递推产生一个 $n$ 阶方阵序列 $A^{(-1)}, A^{(0)}, \dots, A^{(k)}, \dots, A^{(n-1)}$ ，其中 $A^{(k)}[i][j]$ 表示从顶点 $v_i$ 到顶点 $v_j$ 的路径长度， $k$ 表示绕行第 $k$ 个顶点的运算步骤。

逐步尝试在原路径中加入顶点 $k(k=0, 1, \dots, n-1)$ 作为中间顶点。若增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径。



# Graph

---

- Basic Concepts
  - Representation
  - Adjacency Matrices
  - Adjacency Lists
- Traversal Algorithms
  - Depth-first Search
  - Breadth-first Search
- Applications
  - Spanning Tree
  - Shortest Paths
  - Activity Networks

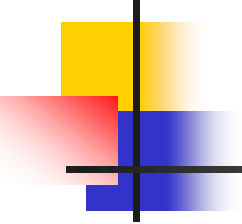


# Activity Network

---

## Activity-on-Vertex (AOV) Networks

- 计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。



课程代号

课程名称

先修课程

C<sub>1</sub>

高等数学

C<sub>2</sub>

程序设计基础

C<sub>3</sub>

离散数学

C<sub>1</sub>, C<sub>2</sub>

C<sub>4</sub>

数据结构

C<sub>3</sub>, C<sub>2</sub>

C<sub>5</sub>

高级语言程序设计

C<sub>2</sub>

C<sub>6</sub>

编译方法

C<sub>5</sub>, C<sub>4</sub>

C<sub>7</sub>

操作系统

C<sub>4</sub>, C<sub>9</sub>

C<sub>8</sub>

普通物理

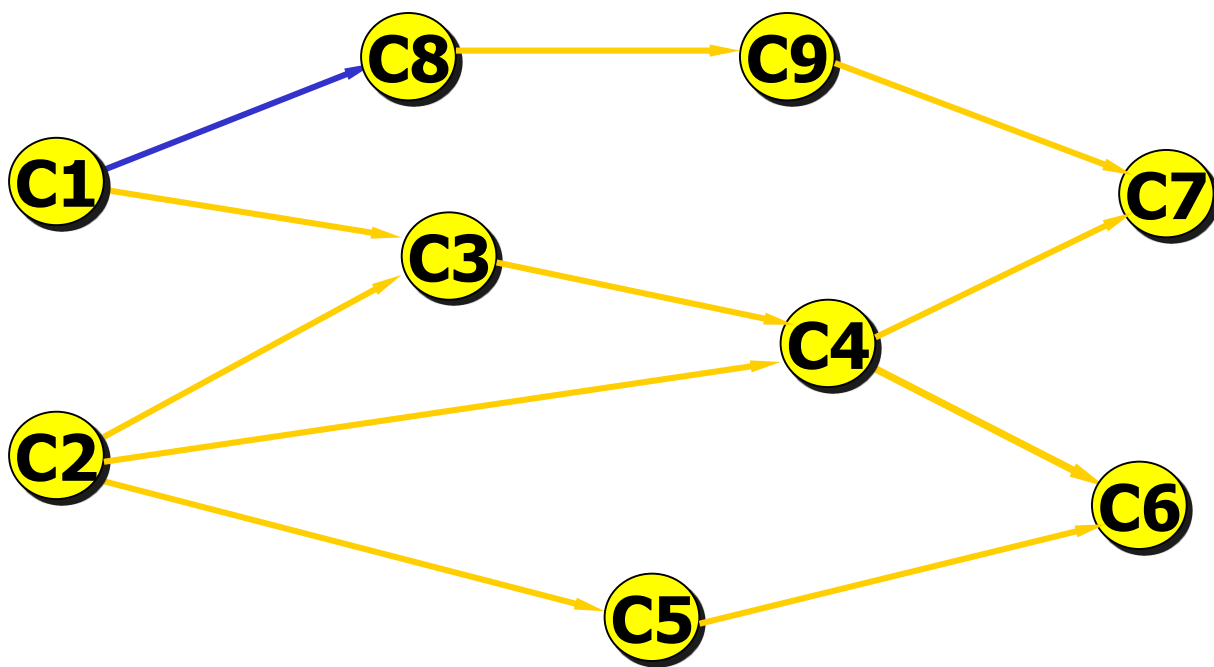
C<sub>1</sub>

C<sub>9</sub>

计算机原理

C<sub>8</sub>





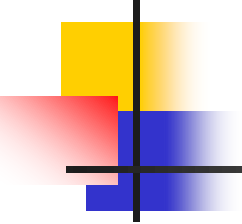
学生课程学习工程图

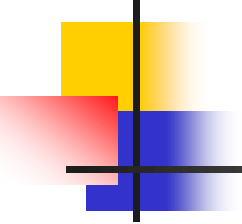


# Activity-on-Vertex (AOV) Networks

---

Definition: A directed graph  $G$  in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an activity-on-vertex network or AOV network.

- 
- 可以用有向图表示一个工程
  - 在这种有向图中，用顶点表示活动，用有向边 $\langle V_i, V_j \rangle$ 表示活动 $V_i$ 必须先于活动 $V_j$ 进行
  - 这种有向图叫做顶点表示活动的AOV网络 (Activity On Vertices)
  - 在AOV网络中不能出现有向回路，即有向环
  - 如果出现了有向环，意味着某项活动应以自己作为先决条件
  - 对给定的AOV网络，必须先判断它是否存在有向环

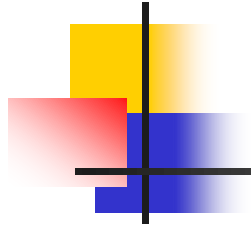


---

**Definition:** Vertex  $i$  in an AOV network  $G$  is a predecessor of  $j$  iff there is a directed path from  $i$  to  $j$ . If  $\langle i, j \rangle$  is an edge in  $G$  then  $i$  is an immediate predecessor of  $j$  and  $j$  immediate successor of  $i$ .

**Definition:** A precedence relation that is both transitive and irreflexive is a partial order.

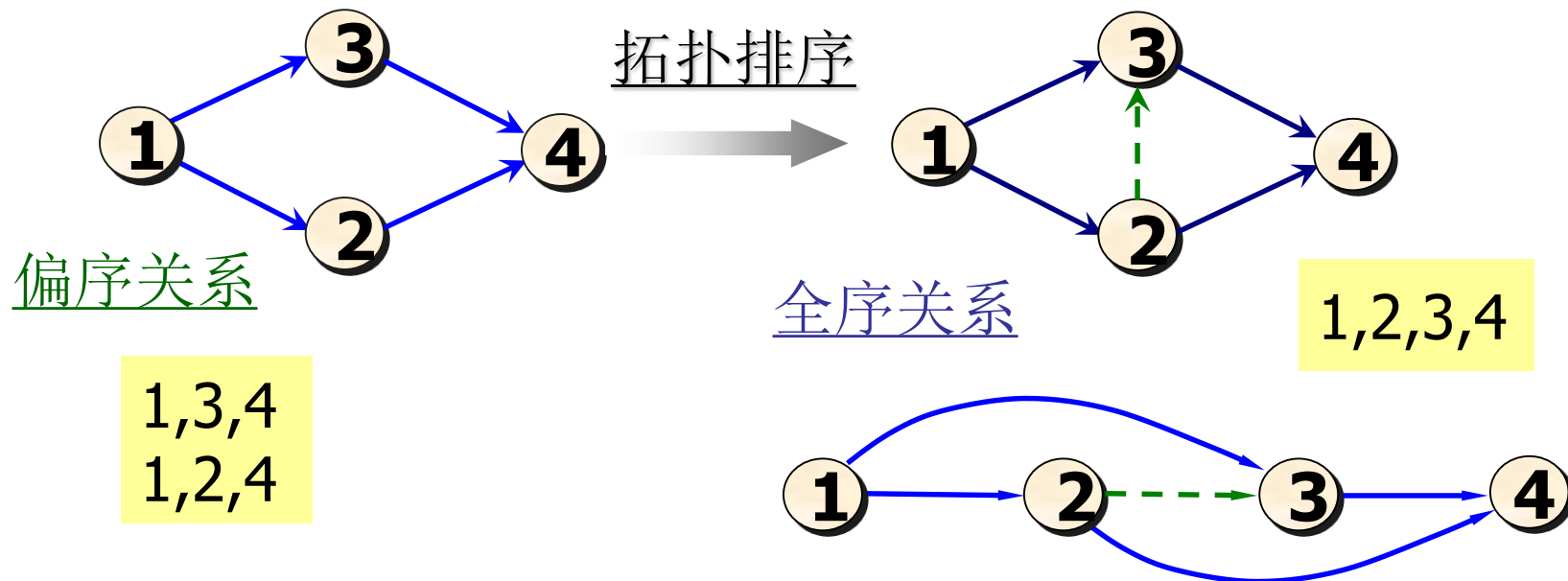
A directed graph with no cycle is an acyclic graph.



- Given a AOV network  $G$ , we focus on determining whether or not it is irreflexive, i.e., acyclic.
- We need to generate the topological order of vertices in  $G$ .

# 检测有向环的方法

- 对AOV网络构造它的拓扑有序序列
- 将各个顶点 (代表各个活动)排列成一个线性有序的序列，使得AOV网络中所有应存在的前驱和后继关系都能得到满足。

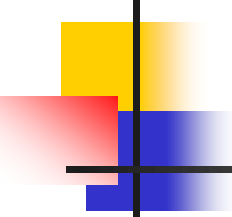




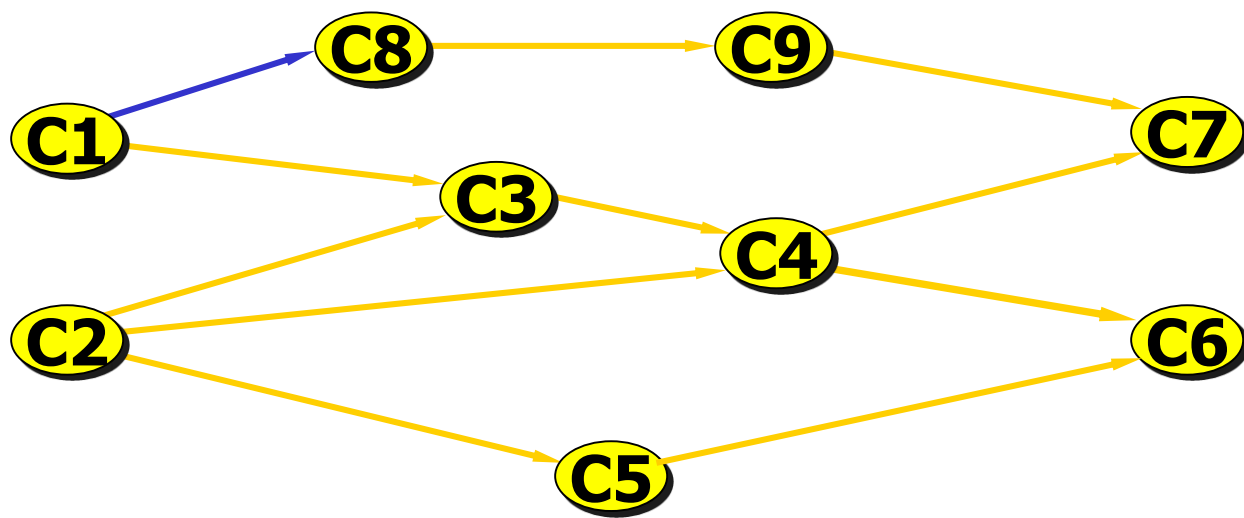
# Topological Order

---

**Definition:** A topological order is a linear ordering of vertices of a graph such that, for any two vertices  $i$  and  $j$ , if  $i$  is a predecessor of  $j$  in the network, then  $i$  precedes  $j$  in the linear ordering.

- 
- 对学生选课工程图进行拓扑排序, 得到的拓扑有序序列为:

$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$   
或  $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$







# Topological sorting algorithm

---

**Idea:** list a vertex with no predecessor, then delete this vertex together with all edges leading out of it from the network. Repeat the above until all vertices have been listed or all remaining vertices have predecessors.



① 输入：AOV网络

令： $n$  为顶点个数

② 在AOV网络中，  
选一个没有直接前驱的顶点，并输出之。

③ 从图中删去该顶点，  
同时，删去所有它发出的有向边。

④ 重复以上 ②、③步，直到

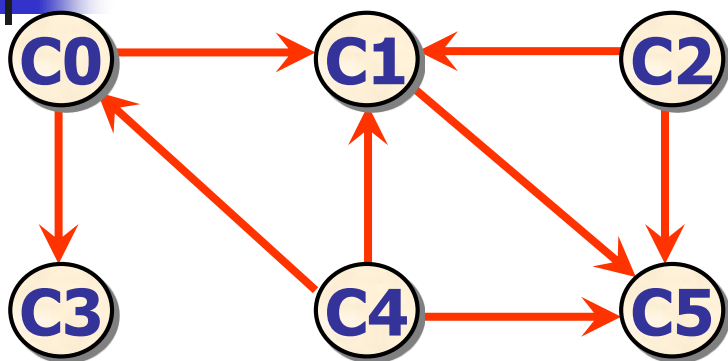
- ◆ 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成  
或

- ◆ 图中还有未输出的顶点，但已跳出处理循环。

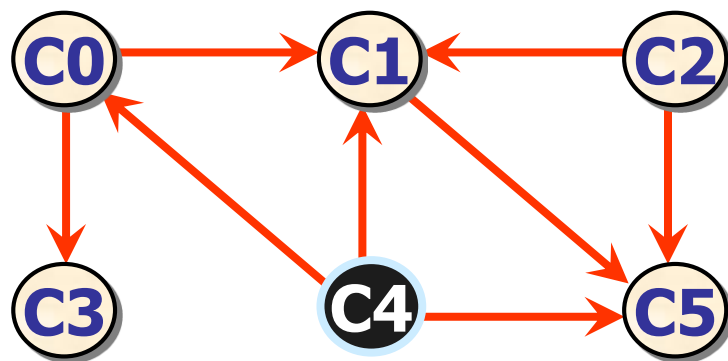
说明：图中还剩下一些顶点，它们都有直接前驱。

结论：这时网络中必存在有向环。

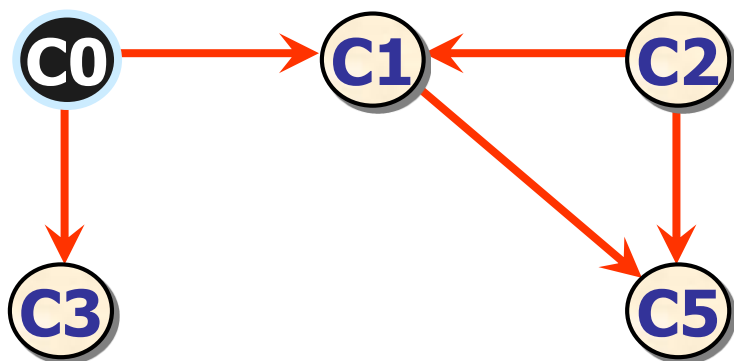
# The Procedure



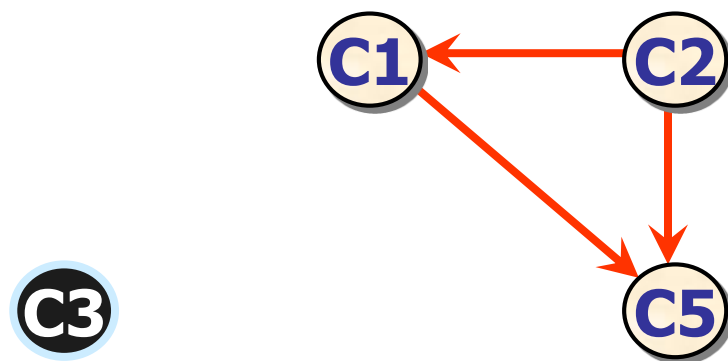
(a) 有向无环图



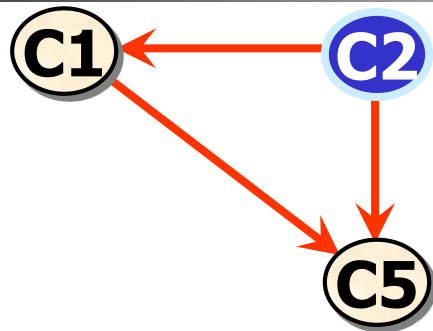
(b) 输出顶点C4



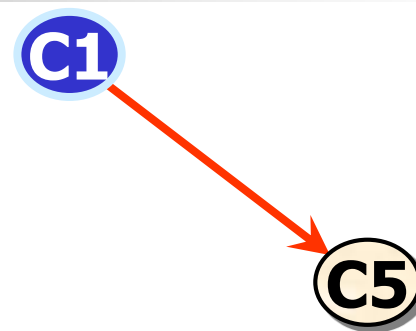
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

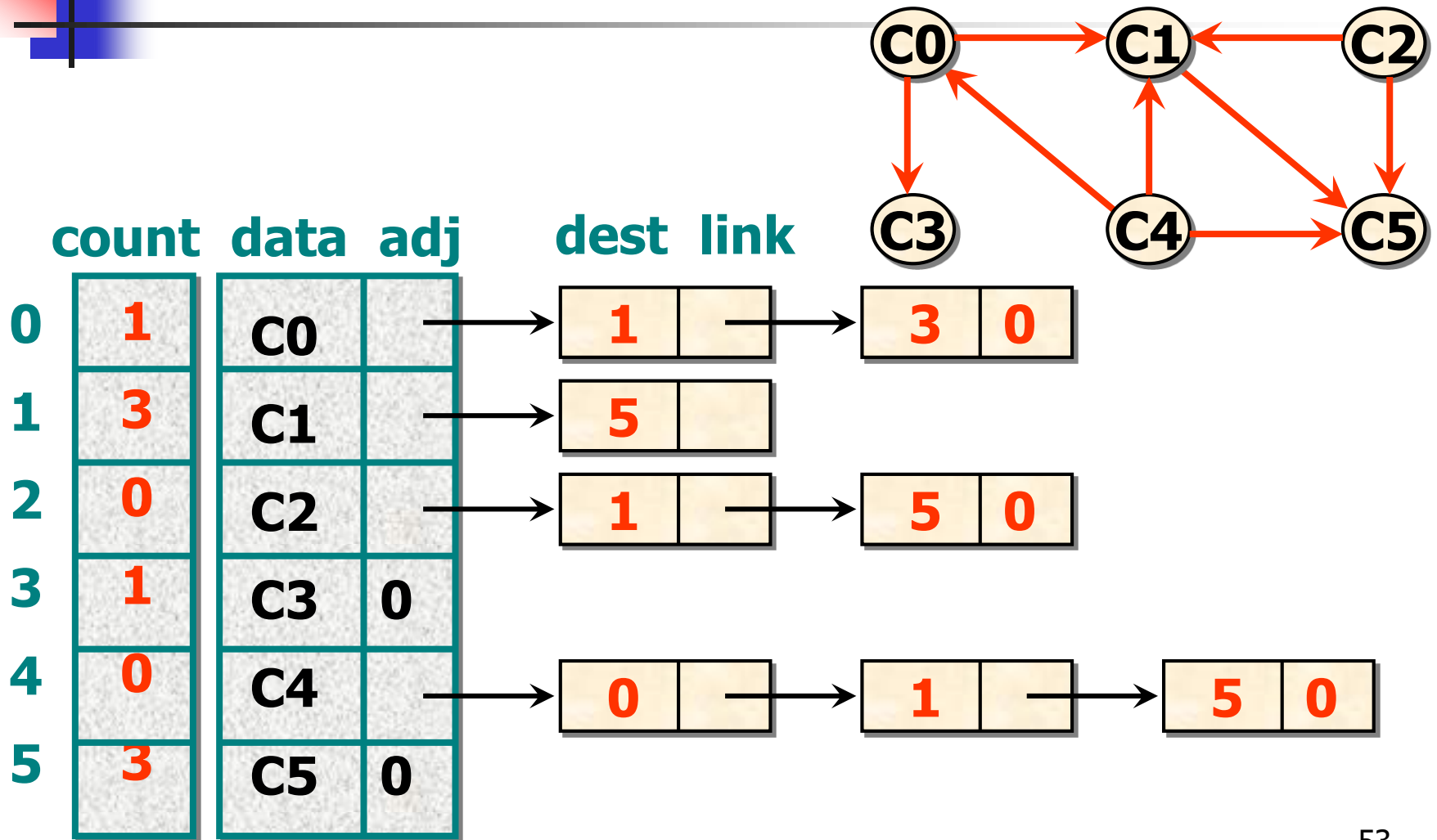


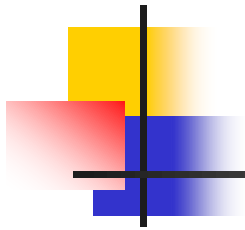
(g) 输出顶点C5

(h) 拓扑排序完成

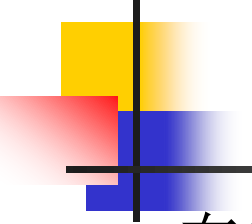
- 得到的拓扑有序序列为  $C_4$  ,  $C_0$  ,  $C_3$  ,  $C_2$  ,  $C_1$  ,  $C_5$
- 它满足图中给出的所有前驱和后继关系
- 对于本来没有这种关系的顶点, 如 $C_4$ 和 $C_2$ , 也排出了先后次序关系

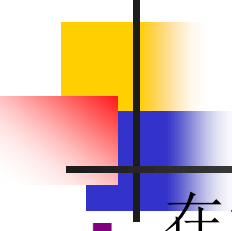
# Representation of AOV





- 在邻接表中增设一个数组count[], 记录各顶点入度
- 入度为零的顶点, 即: 无前驱顶点
- 在输入数据前, 顶点表NodeTable[]和入度数组count[]全部初始化
- 在输入数据时, 每输入一条边<j, k>, 就需要建立一个边结点, 并将它链入相应边链表中, 统计入度信息

- 
- 在算法中,
    - 使用一个存放入度为零的顶点的链式栈,
    - 供选择和输出无前驱的顶点。
  - 拓扑排序算法描述如下:
    - ◆ 建立入度为零的顶点栈;
    - ◆ 当入度为零的顶点栈不空时, 重复执行
      - ✿ 从顶点栈中退出一个顶点, 并输出之;
      - ✿ 从AOV网络中删去这个顶点和它发出的边, 边的终顶点入度减一;
      - ✿ 如果边的终顶点入度减至0, 则该顶点进入度为零的顶点栈;
    - ◆ 如果输出顶点个数少于AOV网络的顶点个数, 则报告网络中存在有向环。

- 
- 在算法实现时,
    - 为了建立入度为零的顶点栈,可以不另外分配存储空间,直接利用入度为零的顶点的count[ ]数组元素。
    - 设立一个栈顶指针top, 指示当前栈顶位置, 即某一个入度为零的顶点。
    - 栈初始化时置 $top = -1$ 。
  - 将顶点i 进栈时执行以下指针的修改:  
     $count[i] = top; top = i;$   
    // top指向新栈顶i, 原栈顶元素在count[i]中
  - 退栈操作可以写成:  
     $j = top; top = count[top];$   
    //位于栈顶的顶点记于j, top退到次栈顶

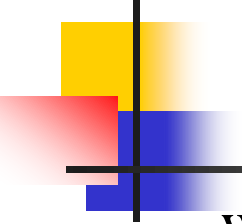




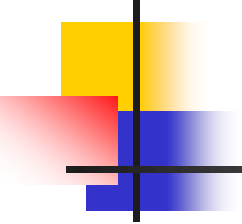
# Algorithm of Topological Sorting

---

```
template <class T, class E>
void TopologicalSort (Graph<T, E>& G)
{
    int i, j, w, v;
    int top = -1;           //入度为零顶点的栈初始化
    int n = G.NumberOfVertices(); //网络中顶点个数
    int *count = new int[n];
                           //入度数组兼入度为零顶点栈
    for (i = 0; i < n; i++) count[i] = 0;
    cin >> i >> j;        //输入一条边(i, j)
```



```
while (i > -1 && i < n && j > -1 && j < n) {
    G.insertEdge (i, j); count[j]++;
    cin >> i >> j;
}
for (i = 0; i < n; i++)           //检查网络所有顶点
    if (count[i] == 0)             //入度为零的顶点进栈
        { count[i] = top; top = i; }
for (i = 0; i < n; i++)           //期望输出n个顶点
    if (top == -1) {               //中途栈空, 转出
        cout << "网络中有回路! " << endl;
        return;
    }
```



```

else {                                     //继续拓扑排序
    v = top; top = count[top];           //退栈v
    cout << G.getValue(v) << " " << endl; //输出
    w = G.GetFirstNeighbor(v);
    while (w != -1) {                     //扫描顶点v的出边表
        count[w]--;                       //邻接顶点入度减一
        if (!count[w])                   //入度减至零, 进栈
            { count[w] = top; top = w; }
        w = G.GetNextNeighbor (v, w);
    } //一个顶点输出后, 调整其邻接顶点入度
} //输出一个顶点, 继续for循环
}

```



# Analysis

---

- 如果：AOV网络有 $n$ 个顶点， $e$ 条边
- 在拓扑排序的过程中，搜索入度为零的顶点，建立链式栈所需要的时间是 $O(n)$ 。
- 在正常的情况下，有向图有 $n$ 个顶点，每个顶点进一次栈，出一次栈，共输出 $n$ 次。
- 顶点入度减一的运算，共执行了 $e$ 次。
- 所以：总的时间复杂度为 $O(n+e)$ 。