Topological Sorting
拓扑排序

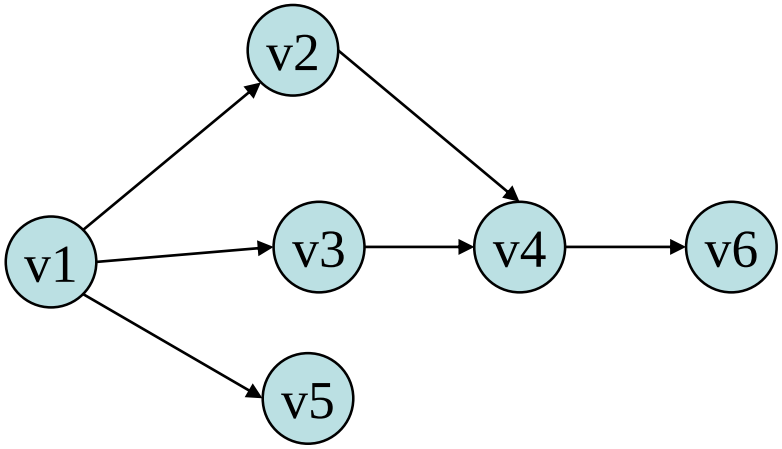
定义

❖Let *G* be a directed graph with no cycles. A *topological order*(*拓扑有序序列*) for *G* is a sequential listing of all the vertices in *G* such that, for all vertices $v,w \in G$, if there is an edge from *v* to *w*, then *v* precedes(*在...之前*) *w* in the sequential listing.

❖实际意义：

- ✓ 有向无环图中，用顶点表示活动，边表示活动之间的先后次序。
- ✓ 比如：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。
- ✓ 通过对该有向无环图进行拓扑排序，可以对活动执行的先后次序进行排定。

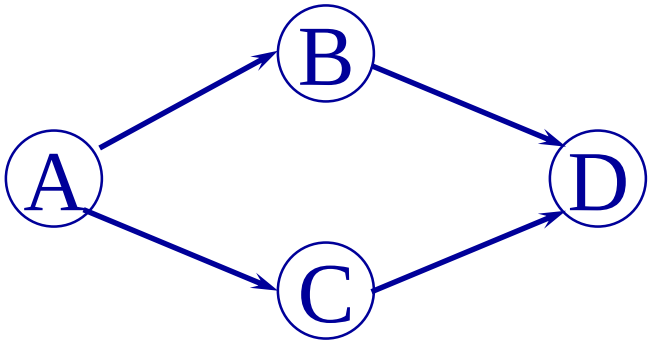
例：课程依赖关系	先决条件
v1(程序设计基础)	无
v2(面向对象程序设计)	v1
v3(离散数学)	v1
v4(面向对象的数据结构)	v3,v2
v5(汇编语言)	v1
V6(编译原理)	v4



拓扑序列：
v1,v2,v3,v5,v4,v6
v1,v3,v2, v4,v6,v5
……

× v1,v4,v3,v5,v2,v6

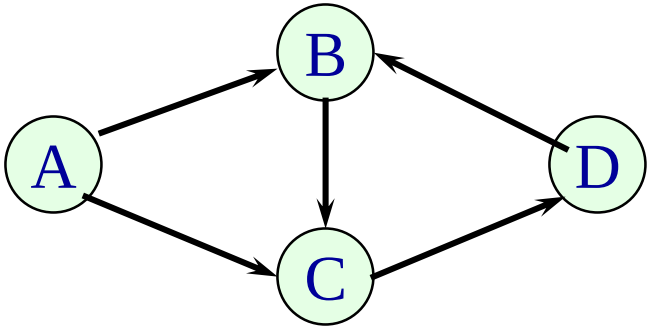
对于下列有向图



可求得拓扑有序序列：

A B C D　或　A C B D

下列有向图



不能求得它的拓扑有序序列。

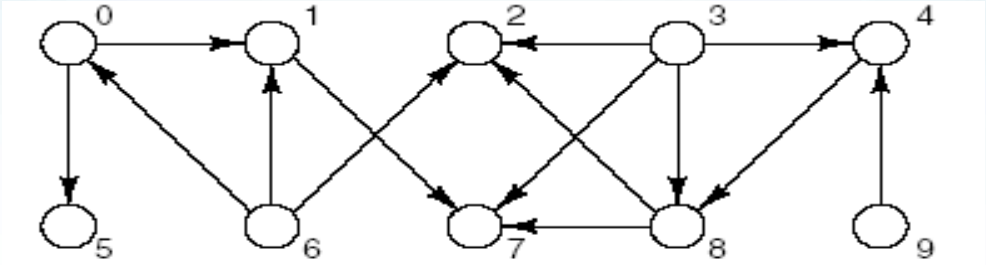
因为图中存在一个回路 {B, C, D}

拓扑序列求解可作为有向图中是否有回路的判断方法。

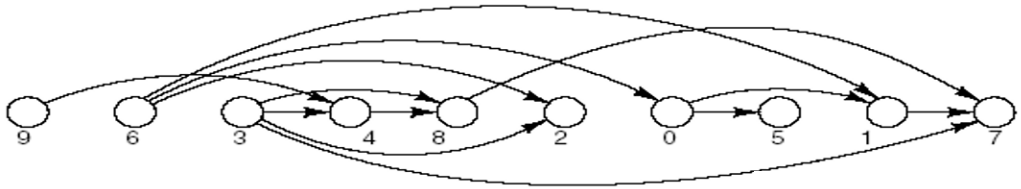
拓扑排序

- 排序策略：
  - 深度优先
    - 深度优先搜索遍历中，若从该顶点开始的遍历已经结束，即它的邻接点已加到序列中了，则将该顶点放在序列的最前面
  - 广度优先
    - 有向图中没有前驱的顶点可先实施

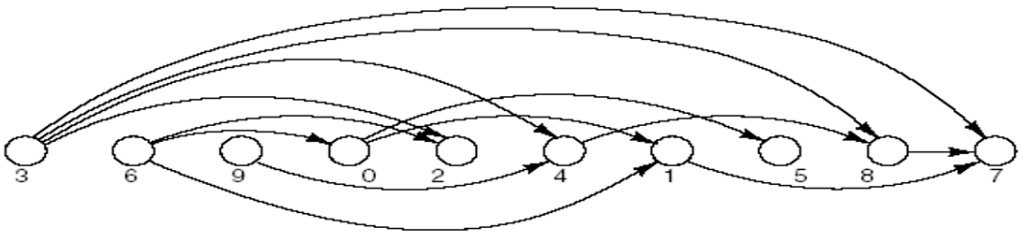
拓扑排序



Directed graph with no directed cycles



Depth-first ordering



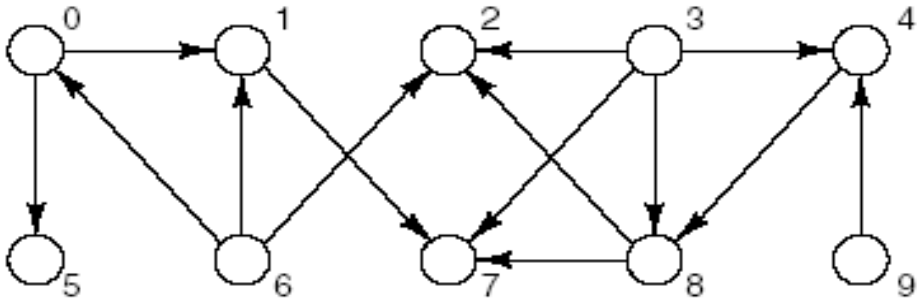
Breadth-first ordering

拓扑排序——specification

```
typedef int Vertex;
template <int graph_size>
class Digraph {
public:
    Digraph( );
    void read( );
    void write( );
    // methods to do a topological sort
    void depth_sort(List<Vertex> &topological_order);
    void breadth_sort(List<Vertex> &topological_order);
private:
    int count;
    List <Vertex> neighbors[graph_size];
    void recursive_depth_sort(Vertex v, bool visited[],
    List<Vertex> &topological_order);
};
```

拓扑排序

- 深度优先策略：
  - 采用深度优先搜索遍历，若从该顶点开始的遍历已经结束，即它的邻接点已加到序列中了，则将该顶点放在序列的最前面

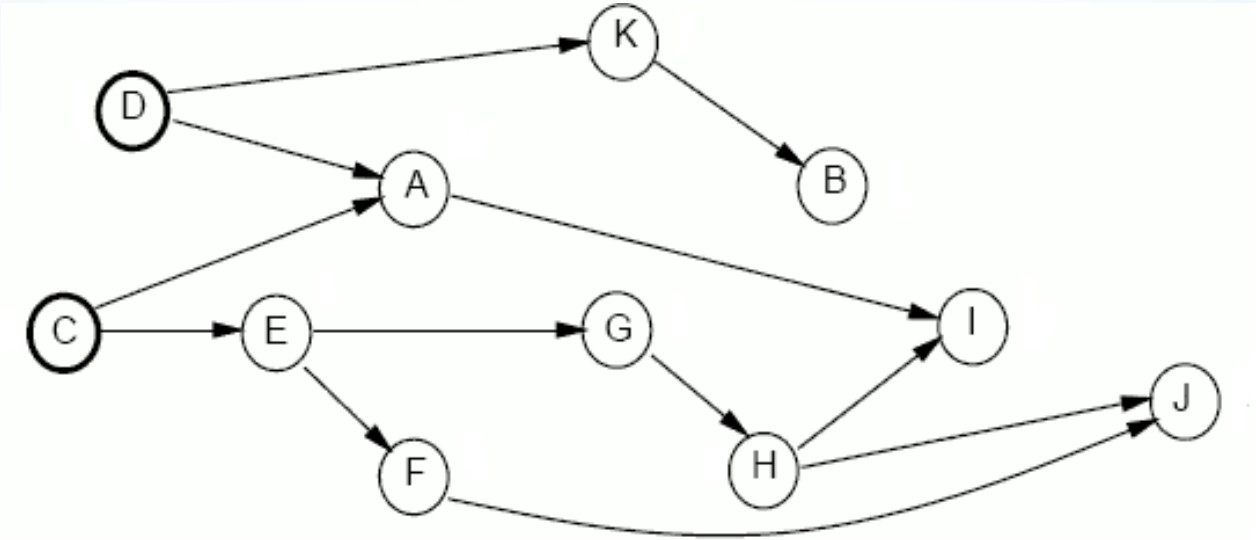


Directed graph with no directed cycles

DF策略拓扑排序结构：9 6 3 4 8 2 0 5 1 7

拓扑排序

- 深度优先策略：



DF结果：D K B C E G H F J A I

拓扑排序

- Depth-First算法

```
template <int graph_size>
void Digraph<graph_size> ::depth_sort(List<Vertex>
        &topological_order)
/* Post: The vertices of the Digraph are placed into List
        topological_order with a depth-first traversal of those
        vertices that do not belong to a cycle.
Uses: Methods of class List , and function
        recursive_depth_sort to perform depth-first traversal.
        */
```

拓扑排序

- Depth-First算法（续）

```
{
    bool visited[graph_size];
    Vertex v;
    for (v = 0; v < count; v++) visited[v] = false;
    Topological_order.clear( );
    for (v = 0; v < count; v++)
        if (!visited[v])
            //Add v and its successors into topological order.
            recursive_depth_sort(v, visited, topological_order);
}
```

拓扑排序

❖ Depth-First算法

```
template <int graph_size> void Digraph<graph_size> ::
recursive_depth_sort(Vertex v, bool *visited, List<Vertex> &topological_order)
/* Pre: Vertex v of the Digraph does not belong to the partially completed List
        topological_order .
Post: All the successors of v and finally v itself are added to topological order
        with a depth-first search.
Uses: Methods of class List and the function recursive_depth_sort . */
```

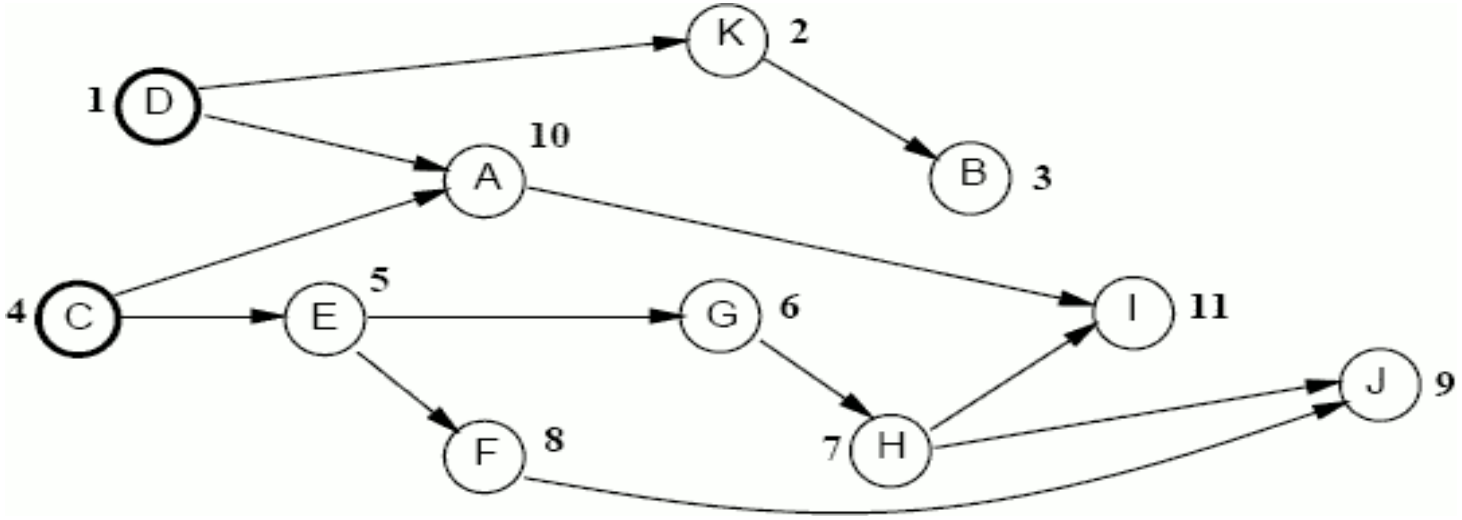
拓扑排序

- Depth-First算法（续）

```
{
    visited[v] = true;
    int degree = neighbors[v].size( );
    for (int i = 0; i < degree; i++) {
      Vertex w; //A (neighboring) successor of v
      neighbors[v].retrieve(i, w);
      if (!visited[w]) // Order the successors of w.
                recursive_depth_sort(w, visited, topological_order);
    }
    topological_order.insert(0, v);
      // Put v into topological_order .
}
```

拓扑排序

- 广度优先策略：
  - 从有向图中选取一个没有前驱的顶点，并输出之；
  - 从有向图中删去此顶点以及所有以它为尾的弧；
  - 重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。



没有前驱的顶点 == 入度为零的顶点

删除顶点及以它为尾的弧 == 弧头顶点的入度减1

拓扑排序

- Breadth-First算法：

```
template <int graph_size> void Digraph<graph_size> ::
breadth_sort(List<Vertex> &topological_order)
/* Post: The vertices of the Digraph are arranged into the List
         topological_order which is found with a breadth-first traversal
         of those vertices that do not belong to a cycle.
Uses: Methods of classes Queue and List . */
```

拓扑排序

- Breadth-First算法：（续）

```
{
    topological_order.clear( );
    Vertex v, w;
    int predecessor_count[graph_size];
    for (v = 0; v < count; v++) predecessor_count[v] = 0;
    for (v = 0; v < count; v++)
       for (int i = 0; i < neighbors[v].size( ); i++) {
                   neighbors[v].retrieve(i, w); // Loop over all edges v-w.
                   predecessor_count[w]++;
       }
    Queue ready_to_process;
    for (v = 0; v < count; v++)
    if (predecessor_count[v] == 0)
       ready_to_process.append(v);
```

拓扑排序

- Breadth-First算法：（续）

```
while (!ready_to_process.empty( )) {
        ready_to_process.retrieve(v);
        topological_order.insert(topological_order.size( ), v);
        for (int j = 0; j < neighbors[v].size( ); j++) {
                neighbors[v].retrieve(j, w); // Traverse successors of v .
                predecessor_count[w]--;
                if (predecessor_count[w] == 0)
                        ready_to_process.append(w);
        }
        ready_to_process.serve( );
        }
}
```