3.5 深度优先 和 广度优先查找

背景知识

- **图**是一种令人感兴趣的**数据结构**，具有各种**广泛应用**。
- **图的遍历算法**是从一个起点出发，试探性访问其余顶点。

- 它必须处理若干**棘手情况**：
  - 某些图存在**回路**，我们必须确定算法不会因回路而**陷入死循环**。
    - » 为避免发生这些情况，遍历算法通常为图的每个顶点设置一个访问标志(mark)。
  - 从**起点**出发可能**到达不了其他顶点**，非连通图就会发生此种情况。
    - » 所以一次遍历算法结束时要**检查标志数组**，查看算法是否处理了所有顶点。

3.5 深度优先 和 广度优先查找

图的遍历算法主要有两种：

- 广度优先查找
  - BFS (breadth-first search).
- 深度优先查找
  - DFS (depth-first search)

3.5.1 广度优先查找

广度优先思想：

可以**从任何顶点开始**访问图的顶点，每次迭代时，

先处理所有与当前顶点相邻的未访问顶点。

再访问与当前顶点距离为2的所有**未访问顶点。**

再访问与当前顶点距离为3的所有**未访问顶点。**

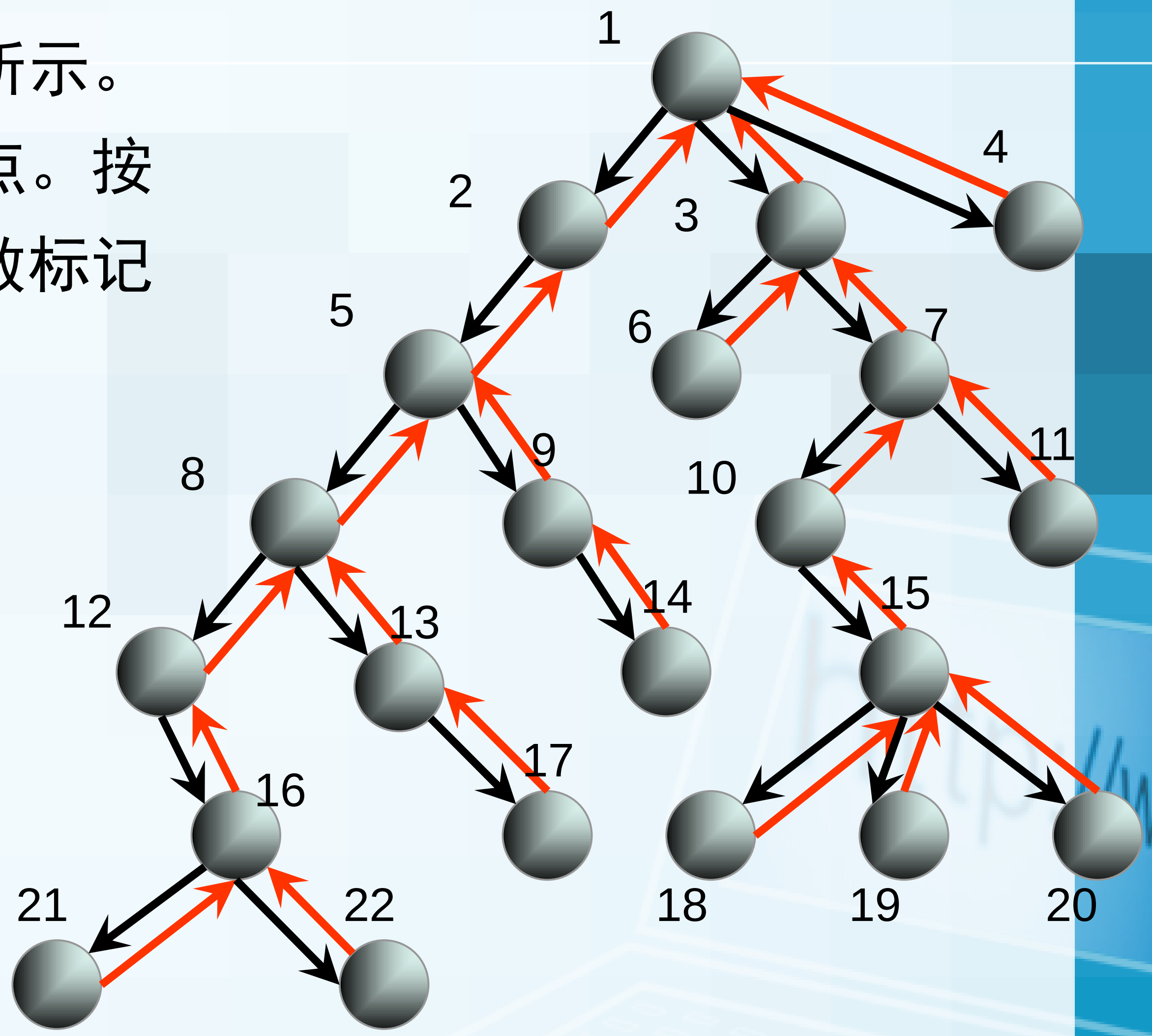
以此类推，……

实现：用队列来跟踪**广度优先**比较方便。

BFS 算法过程图示（以树为例）

广度优先搜索如图所示。

输出：图的各个顶点。按访问顺序用连续整数标记各顶点。



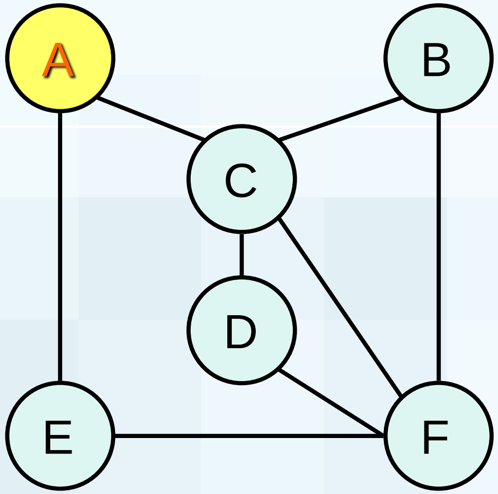
BFS的伪代码

```
BFS(G)    // Breadth-first-search of graph  G=(V,E)
    count ← 0
    for each vertex v∈V do
        Mark[v] ← 0
    for each vertex v∈V do
        if   Mark[v] = 0
             BfsVisit(v)
```

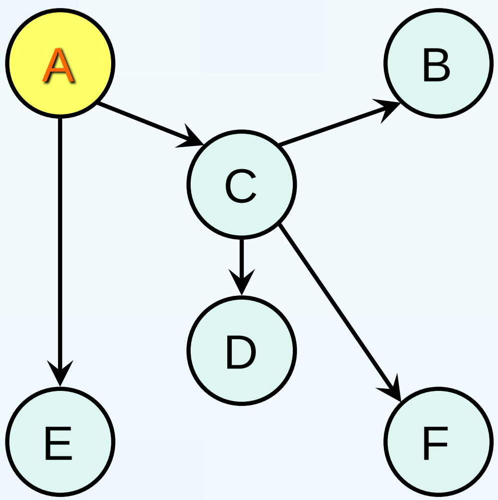
BFS 非递归算法

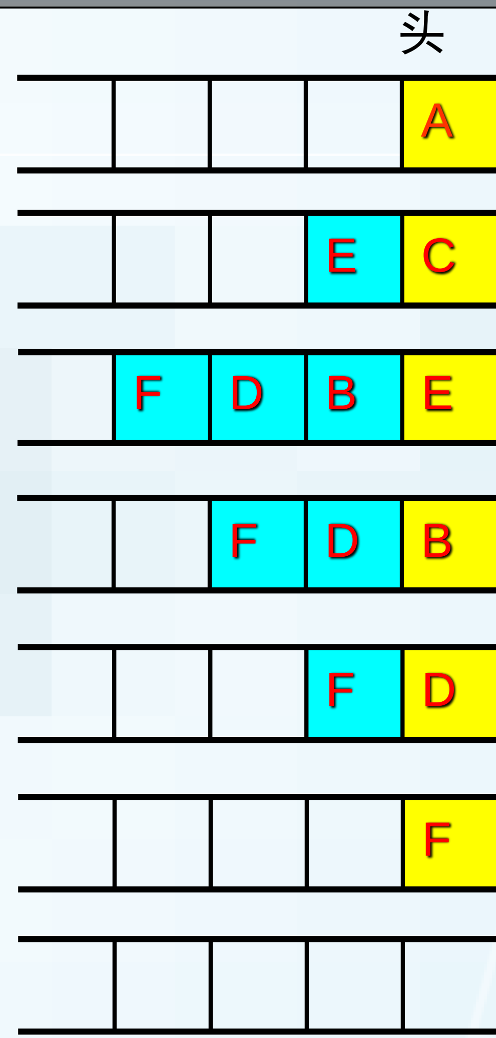
```
BfsVisit(Vertex v)
  visit(v)
  Initialize(Q)  //Q队列初始化为空
  count ← count + 1
  Mark[v] ← count
  Enqueue(Q, v)   //起始顶点入队
  while (isEmpty(Q) = FALSE)  do
    x ← Dequeue(Q)   //队头顶点出队
    for (each vertex w adjacent to x)  do
      if  (Mark[w] = 0)
        visit(w)
        count ← count + 1
        Mark[w] ← count
        Enqueue(Q, w)
```

BFS搜索的队列过程图示



从顶点A开始BFS





A入队，开始BFS。

A出队，A的邻接顶点C, E入队。

C出队，C的邻接顶点B, D, F入队。

E出队，E无未访问的邻接顶点入队。

B出队，B无未访问的邻接顶点入队。

D出队，D无未访问邻接顶点入队。

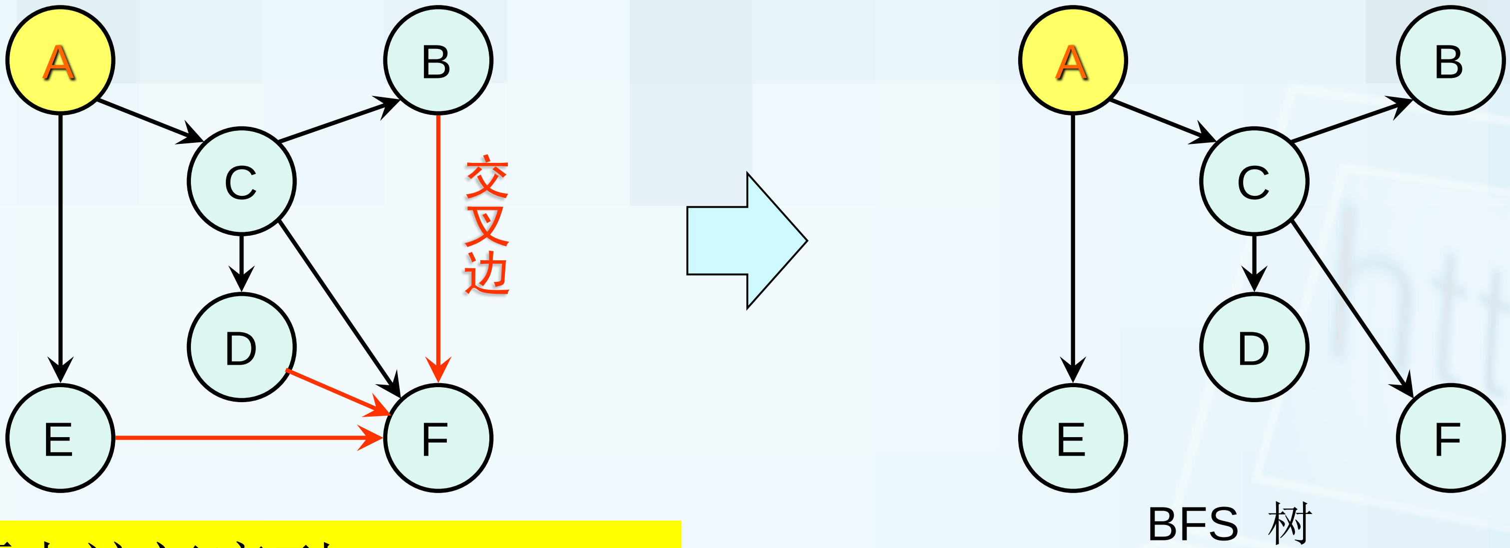
F出队，F无未访问的邻接顶点入队。
此时队列空，BFS过程结束。

顶点的出入队线性序列

ACEBDF

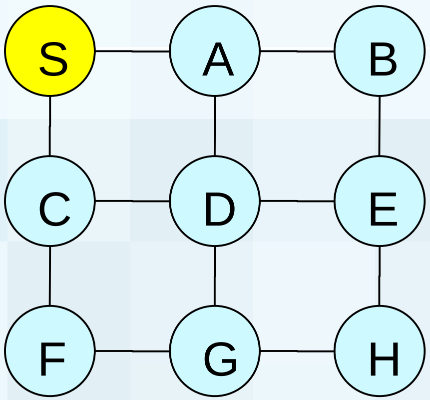
广度优先搜索树

- BFS 可构造出一个**广度优先搜索树**
- 构建广度优先搜索树



顶点访问序列：ACBFDE

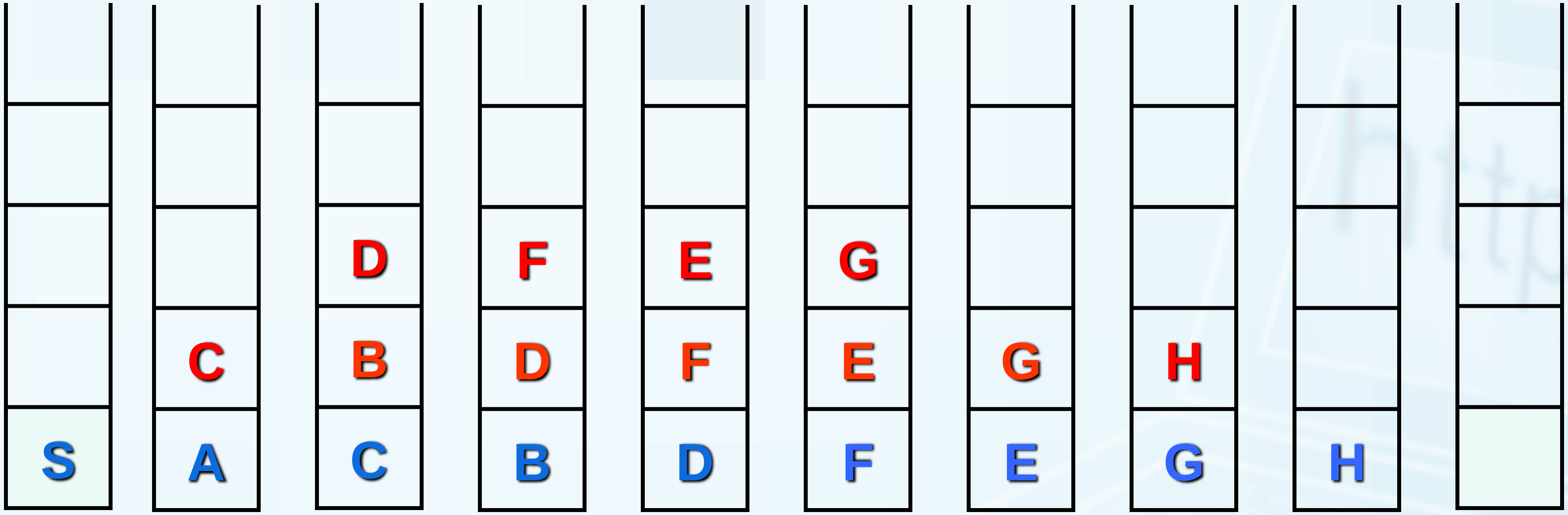
BFS 算法的队列过程图示



顶点**进队**的线性序列：SACBDFEGH

顶点**出队**的线性序列：SACBDFEGH

顶点**访问**的线性序列：SACBDFEGH



BFS应用1

BFS 检查图的连通性：

从任意顶点开始BFS遍历，当遍历算法停止以后，检查是否全部顶点都已访问过。若都访问过，此图是连通的。否则，此图不连通。

BFS 检查图的无环性：

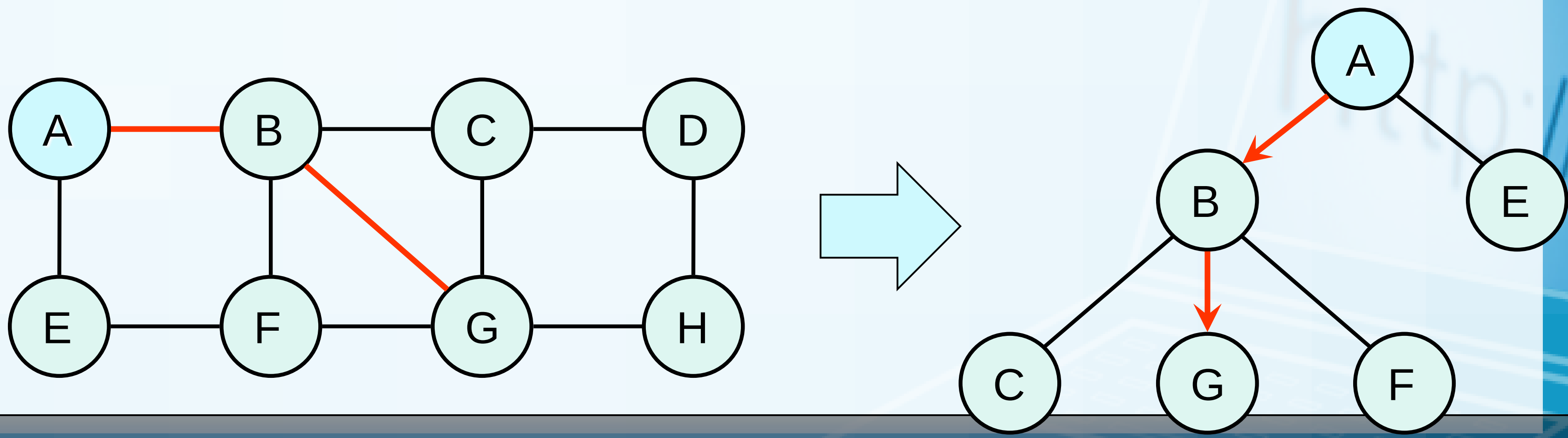
BFS遍历时，**如果发现某个顶点和一个已经访问过的非父顶点相邻，则该图存在一个回路**。否则就是无环的。

BFS 应用2

- **应用2：** **可求给定两个顶点间的最短路径（边数最少）**。

从两个给定的顶点之一开始BFS，当访问到另一个顶点就结束BFS。从起始顶点开始到另一个顶点之间的简单路径就是所求最短路径。

从BFS操作过程看，正确性不言而喻，但数学上的证明并不简单。说明：这样的最短路径可能不止一条。



3.5.1 广度优先查找

时间效率：

输入规模：一个图的顶点数 n

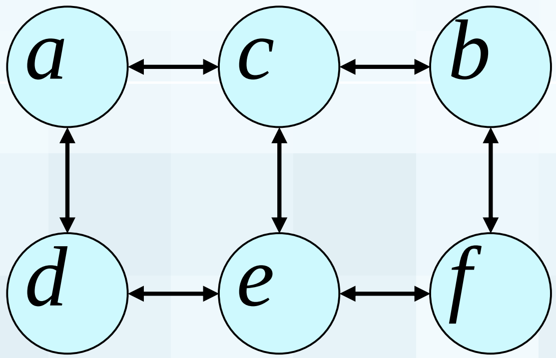
基本操作：判断是否邻接顶点

效率类别：图一定时，BFS算法没有最佳、最差、平均效率之分，但是与图的表示方法（邻接矩阵、邻接链表）有关。

邻接矩阵：给定一个n个顶点的有向图，对每个顶点判断是否邻接顶点时，都需检查所有其他顶点来判断它们**是否相邻**。

总的访问顶点数： $T(n) = n(n-1) \in \Theta(n^2) = \Theta(|V|^2)$

邻接链表：下一个邻接顶点在链表中是确定的。



表头顶点

$$
\left\{
\begin{array}{ll}
a \rightarrow c \rightarrow d & (2) \\
b \rightarrow c \rightarrow f & (2) \\
c \rightarrow a \rightarrow b \rightarrow e & (3) \\
d \rightarrow a \rightarrow e & (2) \\
e \rightarrow c \rightarrow d \rightarrow f & (3) \\
f \rightarrow b \rightarrow e & (2)
\end{array}
\right\}
$$

边数

图的邻接链表表示

现讨论访问顶点数与规模n的关系：

1. 每个**表头顶点**需要访问，以找到该顶点开始的邻接顶点链；
2. 每个链表的**剩余顶点**数正好等于该顶点的**边数（有向边）**。

以每个顶点开始的全部链表都需要访问，完成图的遍历。遍历结束，访问顶点总数等于1、2 顶点之和：

$T(n) \in \Theta(|V|+|E|)$

3.5.1 广度优先查找

效率类别：

邻接矩阵表示，该遍历算法的时间效率属于$\Theta(|V|^2)$

邻接链表表示，该遍历算法的时间效率属于$\Theta(|V|+|E|)$

$|E|_{max} = n \times n = n^2$, $|E| \in [\ 0, |V|^2\ ]$

$|E|=n^2$：完全图。

$|E|=0$: 孤立顶点

结论：**对于稠密图，邻接矩阵表示效率较高（无链表的额外开销）；**

对于稀疏图，邻接链表表示更好。

3.5 深度优先 和 广度优先查找

3.5.2 深度优先查找

算法策略：

- 可以从任何顶点开始访问图的顶点，**每次迭代时，先处理与当前顶点相邻的第一个未访问顶点**。

- 这与**广度优先先处理所有相邻顶点**不同。

3.5.2 深度优先查找

搜索过程：

- 当搜索到一条路径的末端时（即该顶点的所有相邻顶点都已访问过），**沿原路后退一条边**，并从那里继续访问未访问过的顶点（回溯）。
- 访问过程中，若某顶点有**多个邻接顶点**，那么可以按顶点编号或其他策略**顺序进行访问**。

- 当**后退到开始顶点**，并且**开始顶点是一个末端时**，一次搜索**停止**。

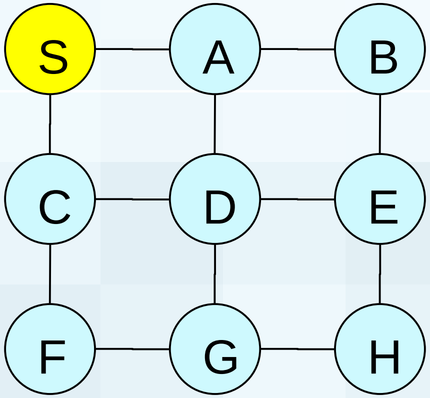
- 实现：用**递归**、或者**栈**

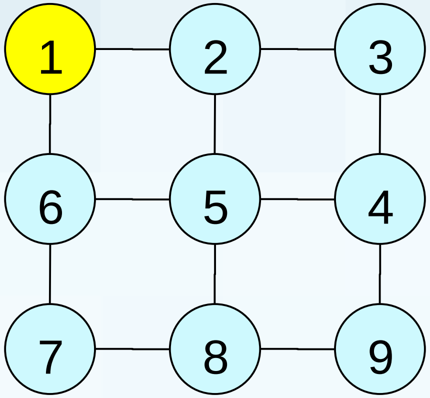
深度优先的递归算法

```
DFS(G)    // Depth-first-search of graph G=(V,E)
    count ← 0
    for each vertex v∈V do
        Mark[v] ← 0
    for each vertex v∈V do
        if  Mark[v] = 0
            DfsVisit(v)
```

```
DfsVisit(Vertex v)
   //count 为全局变量，初始化为0
   count ← count + 1;  // > 0: 已访问
   visit(v);
   Mark[v] ← count;  //按访问序标记顶点
   for (each vertex w adjacent to v) do
       if  Mark[w] = 0
           DfsVisit(w);
```

深度优先递归算法的实例





***DfsVisit*(*Vertex v*)**
  //*count* 为全局变量，初始化为0
  count $\leftarrow$ ***count*** + **1;** // > 0: 已访问
  ***visit*(*v*);**
  ***Mark*[*v*]** $\leftarrow$ ***count*;** //按访问序标记顶点
  for* (*each vertex w adjacent to v*) *do
    ***if* *Mark*[*w*] = 0**
      ***DfsVisit*(*w*);**

若按字母顺序，

顶点**访问**的线性序列为：SABEDCFGH

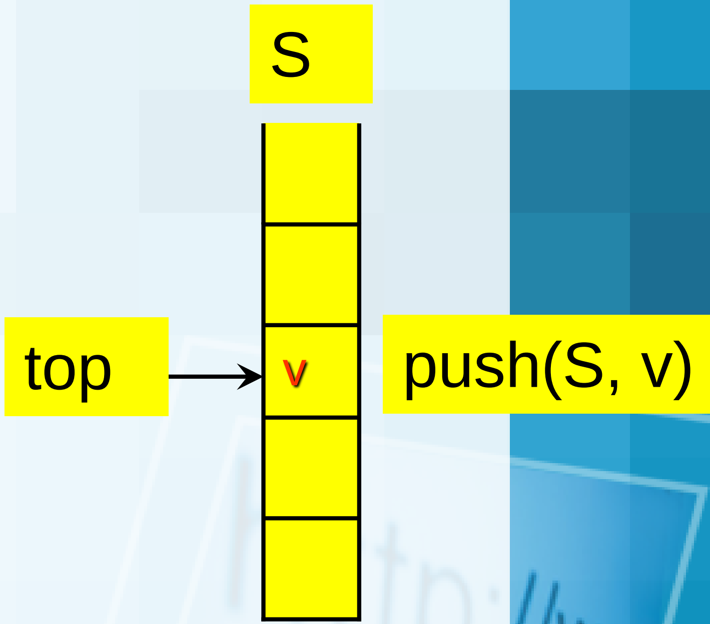
DFS的非递归算法

- 通常，递归算法转为非递归可获得以下好处：
  - 节省内存空间
  - 提高执行效率
  - 有些语言不支持递归
- DFS转为非递归，需要用栈来存储中间结果。

DFS的伪代码2（非递归算法）

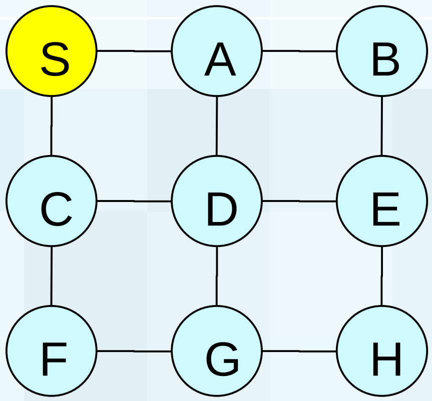
```
DfsVisit( Vertex v)
    visit(v)
    count ← count + 1;  Mark[v] ← count
    Initialize(S) //栈S初始化为空
    Push(S, v)   //起始顶点v入栈
    while (isEmpty(S) = FALSE)  do
        x ← Pop(S) //栈顶top的顶点出栈
        for (each vertex w adjacent to x)  do
            if  (Mark[w] = 0)
                visit(w)
                count ← count + 1
                Mark[w] ← count
                Push(S, w)
```

用栈来跟踪



简单模仿BFS，入栈时访问，得到的是错误的算法！

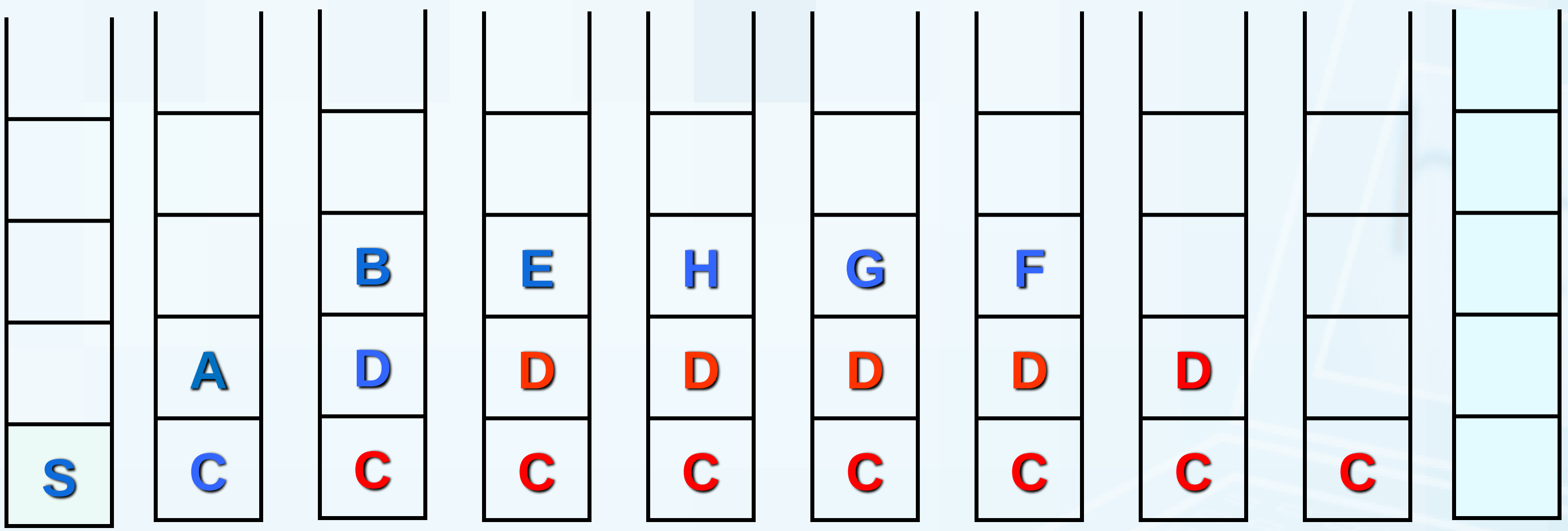
DFS 算法2的栈过程图示 2



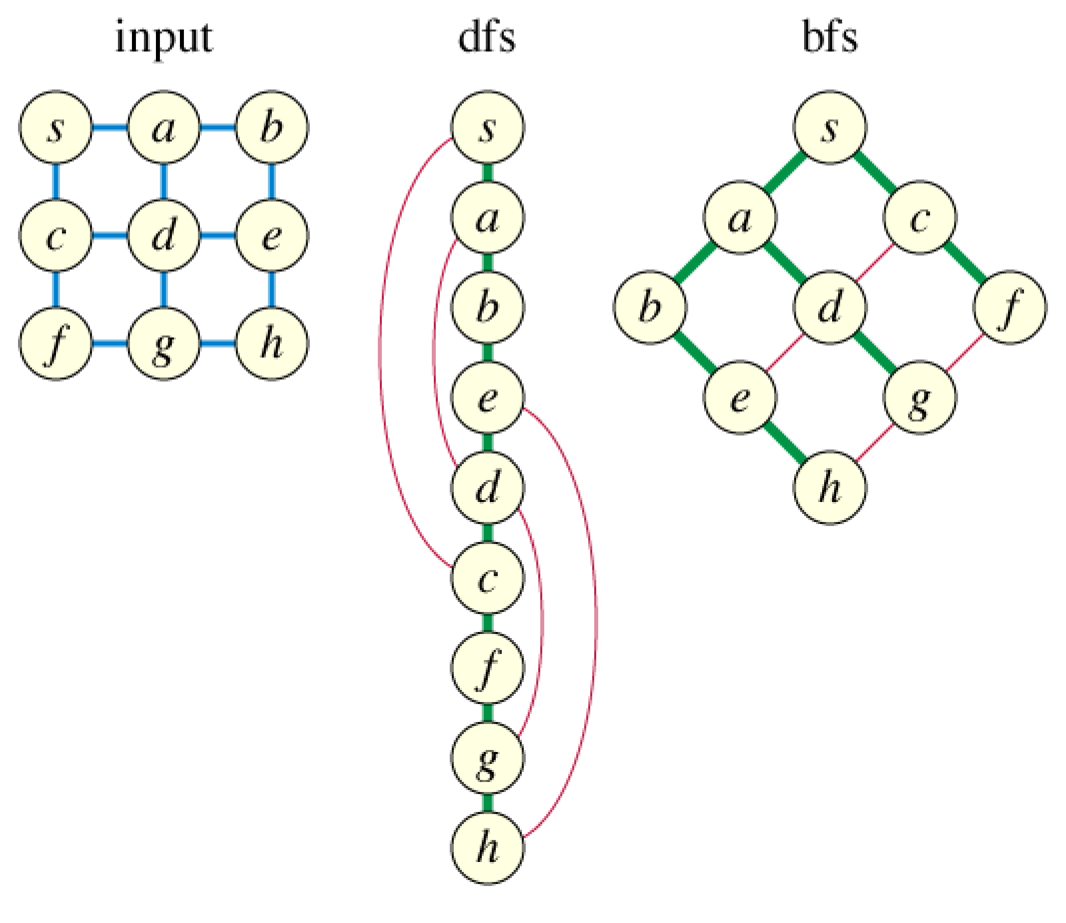
顶点**进栈**的线性序列：SCADBEHGF

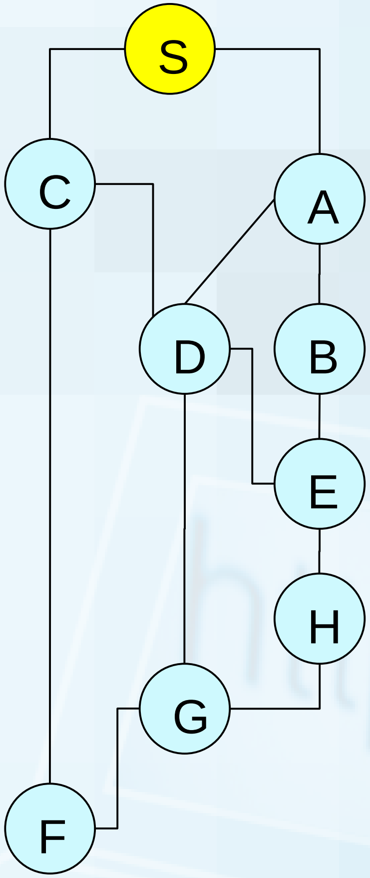
顶点**出栈**的线性序列：SABEHGFDC

顶点**访问**的线性序列：SCADBEHGF



顶点访问的线性序列：SCADBEHGF





错误的访问序列！

教材中对DFS非递归算法的错误表述

- Skiena's *Algorithm Design Manual* p. 169;
- Jeff Edmonds' *How to Think about Algorithms*, pp. 175–178;
- Gilberg and Forouzan, *Data Structures: A Pseudocode Approach Using C*, 2nd ed., p. 497

DFS的伪代码3（非递归算法）

```
DfsVisit(Vertex v)
  Initialize(S)  //栈S初始化为空
  Push(S, v)   //起始顶点v入栈
  while (isEmpty(S) = FALSE) do
     x ← Pop(S) ; Push(S, x)
     if ( Mark[x] = 0)
         visit(x)
         count ← count + 1
         Mark[x] ← count
     if (exist a vertex w adjacent to x AND Mark[w] = 0)
         Push(S, w)
    else
         Pop(S)
```

入栈时不访问！每次访问未标识的栈顶元素，算法正确！
但是占用空间较多，时间效率差！

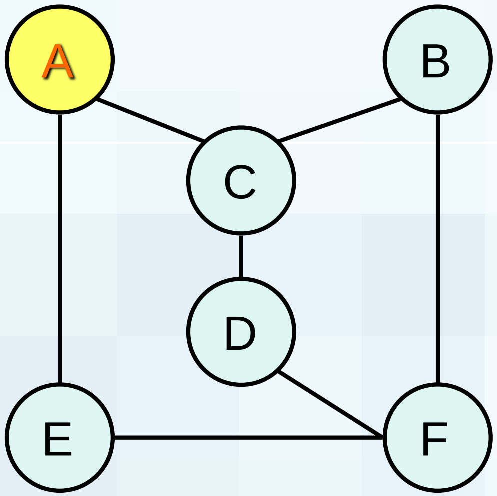
DFS 算法3的栈过程图示

顶点**进栈**的线性序列：SABEDCFGH

顶点**访问**的线性序列：SABEDCFGH

与递归算法一致！

DFS 算法3的栈过程图示



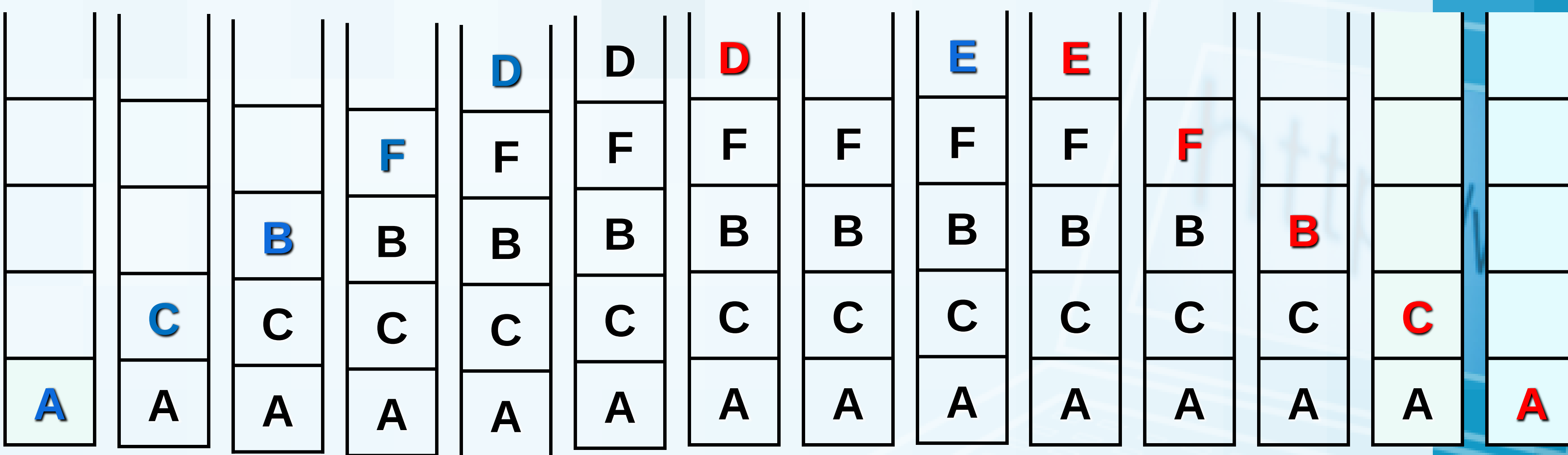
从顶点A开始DFS

顶点**进栈**的线性序列：ACBFDE

顶点**出栈**的线性序列：DEFBCA

顶点**访问**的线性序列：ACBFDE

与递归算法一致！



DFS的伪代码4（非递归算法）

```
DfsVisit( Vertex v )
  Initialize(S)  //栈S初始化为空
  Push(S, v)    //起始顶点v入栈
  while (isEmpty(S) = FALSE) do
    x ← Pop(S)  //栈顶top的顶点出栈
    if ( Mark[x] = 0)
      visit(x)
      count ← count + 1
      Mark[x] ← count
      for (each vertex w adjacent to x) do
        if ( Mark[w] = 0)
          Push(S, w)
```

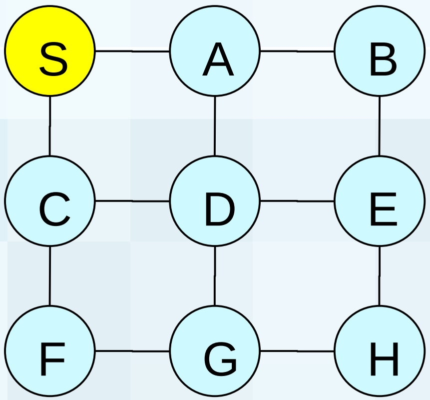
```
DFS ( G=(V,E) )
    count ← 0
    for each vertex v∈V do
        Mark[v] ← 0
    for each vertex v∈V do
        if  Mark[v] = 0
            DfsVisit(v)
```

出栈时访问！多个邻接点时用逆序入栈！

不足：可能多次入栈。

DFS 算法4的栈过程图示

递归顶点**访问**的线性序列：SABEDCFGH

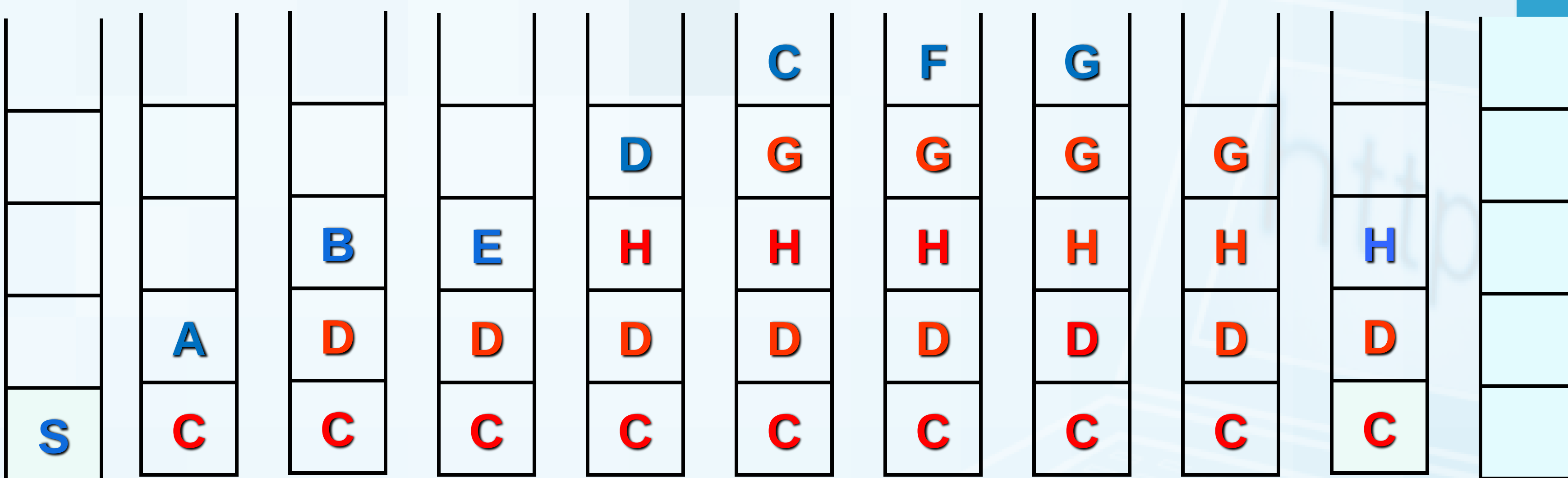


顶点**进栈**的线性序列：SCADBEHDGCFG

顶点**出栈**的线性序列：SABEDCFGH

顶点**访问**的线性序列：SABEDCFGH

与递归算法一致！



DFS的时间效率

- 输入规模：一个图的顶点数 n
- 基本操作：判断是否邻接顶点
- 效率类别：图一定时，DFS算法没有最佳、最差、平均效率之分，但是与图的表示方法（邻接矩阵、邻接链表）有关。

效率类别：

邻接矩阵表示，该遍历算法的时间效率属于$\Theta(|V|^2)$

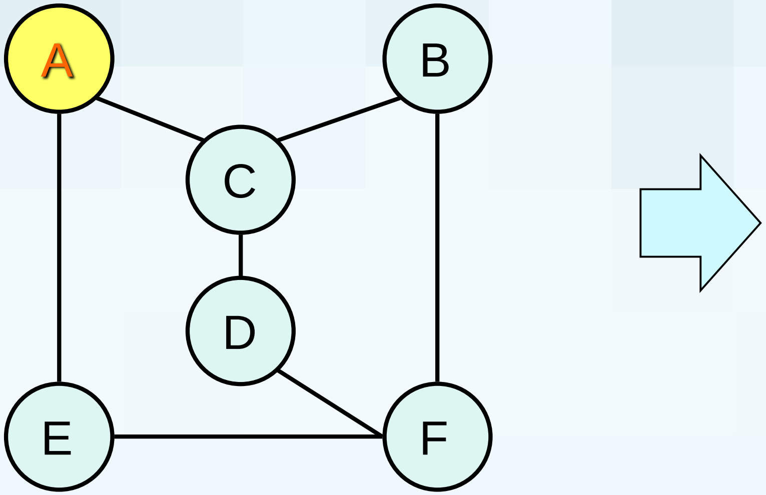
邻接链表表示，该遍历算法的时间效率属于$\Theta(|V|+|E|)$

DFS树与森林

- DFS 可构造出一个**深度优先搜索树或森林**。

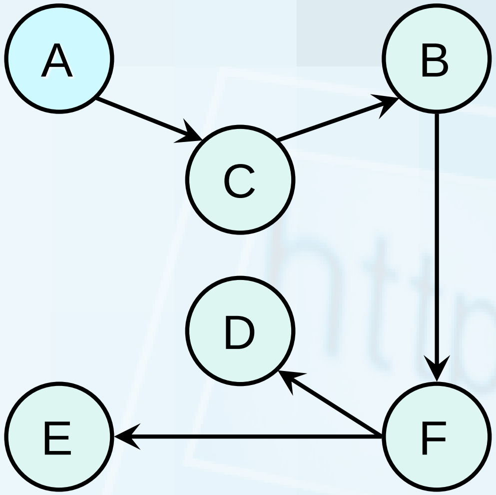
- 构建深度优先搜索树

顶点访问序列：ACBFDE



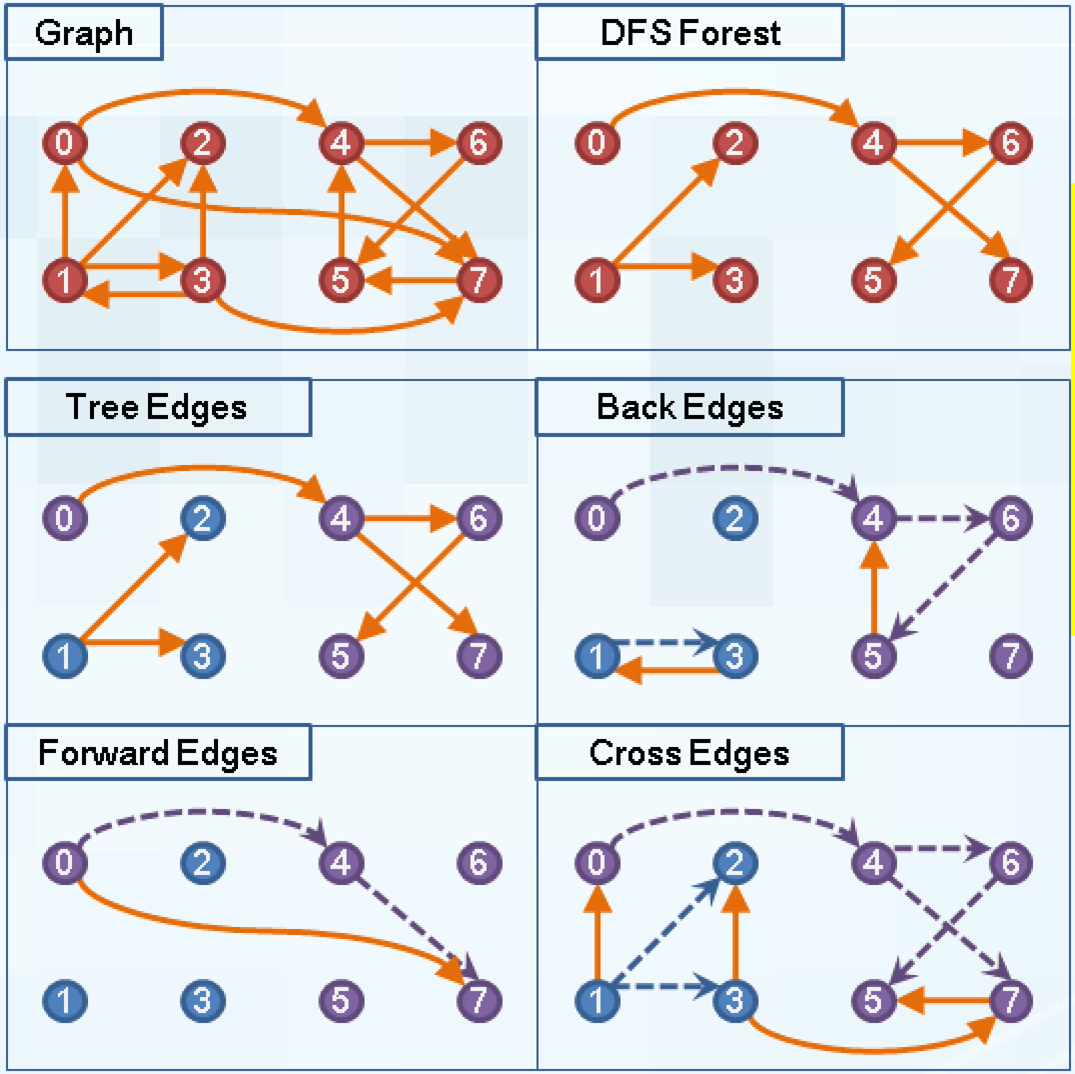
从顶点A开始DFS

DFS树



DFS树与森林

深度优先搜索森林



先从**0**点开始搜索

Tree Edge：树的边。

Back Edge：连向祖先结点。

Forward Edge：连向子孙结点。

Cross Edge：连向姊妹结点或其他树结点

DFS简单应用

DFS 检查图的连通性：

从任意顶点开始DFS遍历，当遍历算法停止以后，检查是否全部顶点都已访问过。若都访问过，此图是连通的。否则，此图不连通。

DFS 检查图的无环性：

如果DFS遍历的过程中，发现从某个顶点到它的祖先顶点（非父顶点）之间有一条回边（Back Edge），则该图存在一个回路。若不存在这样的回边，则图是无环的。

DFS 其他应用：

求图的**关节点**（从图中移走某个顶点及其与它相连的边以后，图被分为若干个不相交的部分，该顶点称为关节点）

讨论

- **二叉树**有三种遍历方法：
  - 前序、中序、后序

- 讨论它们和**深度优先遍历**及**广度优先遍历**的区别与联系。