补充一下关于邻接矩阵和邻接表的`FirstNeighbor(G,x)`和`NextNeighbor(G,x,y)`的函数实现(方便遍历边).此处是以下标为 1开始.

- **邻接矩阵**

```
int FirstNeighbor(MGraph G,int x)
{
    if(x>=1&&x<=G.vexnum)
    {
        for(int i=1;i<=G.vexnum;i++)
            if(G.Edge[x][i]!=inf&&G.Edge[x][i]!=0)
                return i;
    }
    return -1;
}

int NextNeighbor(MGraph G,int x,int y)
{
    if(x>=1&&x<=G.vexnum&&y>=1&&y<=G.vexnum)
    {
        for(int i=y+1;i<=G.vexnum;i++)
            if(G.Edge[x][i]!=inf&&G.Edge[x][i]!=0)
                return i;
    }
    return -1;
}

```

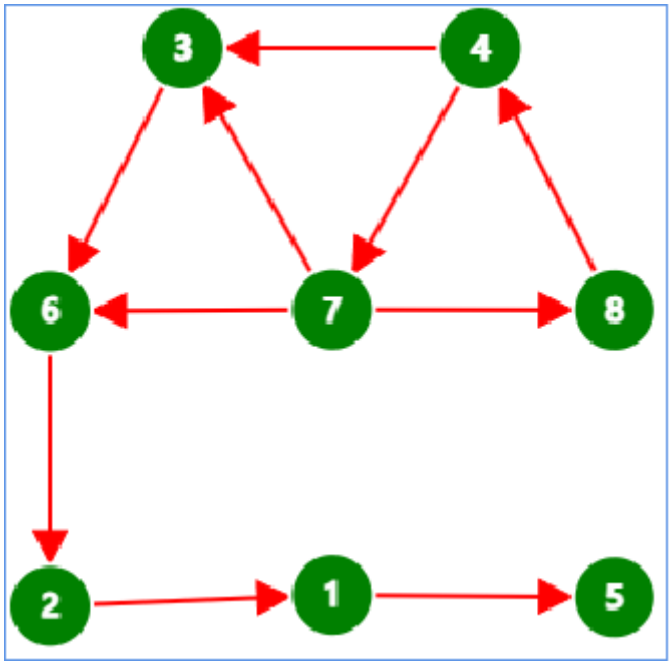
- **邻接表**

```
int FirstNeighbor(ALGraph G,int x)
{
    if(x>=1&&x<=G.vexnum)
    {
        if(G.vertices[x].first!=NULL)
            return G.vertices[x].first->adjvex;
    }
    return -1;
}

int NextNeighbor(ALGraph G,int x,int y)
{
    ArcNode *temp=G.vertices[x].first;

    while(temp!=NULL)
    {
        if(temp->adjvex==y)
            break;
        temp=temp->next;
    }

```

```
    if(temp==NULL||temp->next==NULL)
        return -1;
    else return temp->next->adjvex;
}
```

广度优先搜索

基本思想：首先访问起始顶点 v,接着从 v出发,依次访问 v的各个未访问过的邻接顶点 $w_1, w_2, \cdots, w_i$,然后依次访问 $w_1, w_2, \cdots, w_i$的所有未被访问过的邻接顶点;再从这些访问过的顶点出发,访问它们所有未被访问过的邻接顶点,直至图中所有顶点都被访问过为止.若此时图中尚有顶点未被访问,则另选图中一个未曾被访问的顶点作为始点,重复上述过程,直至图中所有顶点都被访问到为止.常通过**队列**和**标记数组**来进行实现.



假设以 7出发(假设按递增顺序访问邻接顶点),故广度优先搜索序列为 $7, 3, 6, 8, 2, 4, 1, 5$;(调用使用 1次 BFS函数)

假设以 1出发(访问邻接顶点同上),故广度优先搜索序列为 $1, 5, 2, 3, 6, 4, 7, 8$(调用使用 4次 BFS函数,分别在 $1, 2, 3, 4$各自调用)

代码如下(以**邻接矩阵**为例):

```
bool visited[MaxVertexNum];

void visit(MGraph G,int pos)
{
    cout << G.Vex[pos] << " ";
}

void BFS(MGraph G,int start)
{
    queue<int> q;//此处为了方便,使用C++的stl中的队列

    //将起点放入队列中,并标记为true
    q.push(start);
    visited[start]=true;
```

```
15
16      while(q.size()>0)
17      {
18          int temp=q.front();
19          visit(G,temp);//访问该结点
20          q.pop();//弹出队列
21
22          for(int j=FirstNeighbor(G, temp);j!=-1;j=NextNeighbor(G, temp, j))
23          {
24              if(visited[j]==false)//当前邻接顶点未访问,放入队列并标记
25              {
26                  q.push(j);
27                  visited[j]=true;
28              }
29          }
30      }
31      cout << endl << "--------" << endl;//观察使用 BFS次数
32  }
33
34  void BFSTraverse(MGraph G)
35  {
36      for(int i=1;i<=G.vexnum;i++)
37          visited[i]=false;
38
39      for(int i=1;i<=G.vexnum;i++)
40          if(visited[i]==false)
41              BFS(G,i);
42  }
```

BFS算法的性能分析

> 无论是邻接表还是邻接矩阵的存储方式, BFS算法都需要借助一个**辅助队列** Q, n个顶点均需入队一次, 在最坏情况下,空间复杂度为 $O(|V|)$.
>
> 采用**邻接表**存储方式时,每个顶点只需搜索一次(或入队一次),故时间复杂度为 $O(|V|)$,在搜索任意一个顶点的邻接点时,每条边至少访问一次,故时间复杂度为 $O(|E|)$,算法总的时间复杂度为 $O(|V|+|E|)$; 采用邻接矩阵存储方式时,查找每个顶点的邻接点所需的时间为 $O(|V|)$,故算法总的时间复杂度为 $O(|V|^2)$.

广度优先搜索树

在广度遍历的过程中,可得到一棵遍历树,称为广度优先生成树.同一个图的**邻接矩阵**存储表示是唯一的,故其广度优先生成树也是`唯一`的,但由于**邻接表**存储表示不唯一,故其广度优先生成树也是`不唯一`的.

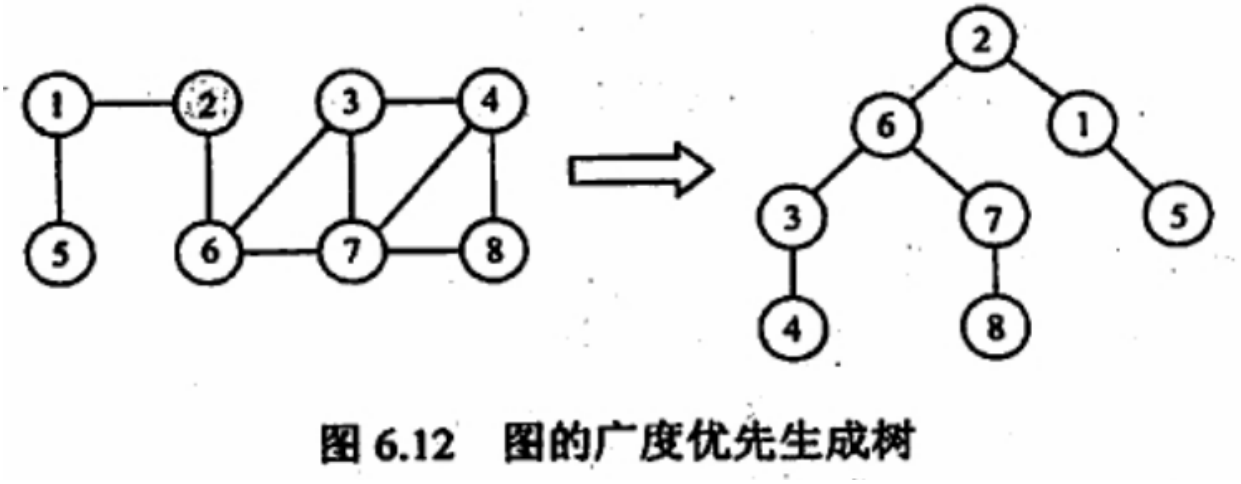
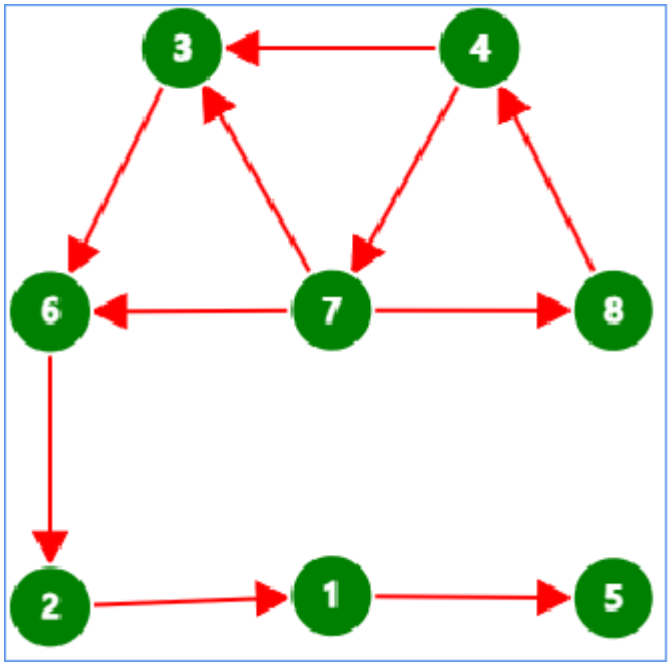


图 6.12　图的广度优先生成树

对**非连通图**的广度优先遍历,可得到**广度优先森林**

深度优先搜索

基本思想：首先访问图中某一起始顶点 v,然后由 v出发,访问与其相邻且未被访问的任意一个顶点 w_1,再访问与 w_1邻接且未被访问的任意一个顶点 w_2, $\cdots\cdots$重复上述过程.当不能再继续向下访问时,依次退回到最近被访问的顶点,若它还有邻接顶点未被访问过,则从该点开始继续上述搜索过程,直至图中所有顶点均为访问过为止.



仍以该图做例子,假设以 7出发(假设按递增顺序访问邻接顶点),故深度优先搜索序列为 $7, 3, 6, 2, 1, 5, 8, 4$.

代码如下(以**邻接矩阵**为例):

```
bool visited[MaxVertexNum];

void visit(MGraph G,int pos)
{
    cout << G.Vex[pos] << " ";
}

void DFS(MGraph G,int start)
```

```
{
    visit(G,start);//访问该点,并标记
    visited[start]=true;
    for(int j=FirstNeighbor(G, start);j!=-1;j=NextNeighbor(G, start, j))//访问其未被访问的邻接顶点
        if(visited[j]==false)
            DFS(G,j);
}

void DFSTraverse(MGraph G)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;

    for(int i=1;i<=G.vexnum;i++)
        if(visited[i]==false)
        {
            DFS(G,i);
            cout << endl << "--------" << endl;//观察使用 DFS次数
        }
}
```

DFS算法的性能分析

> DFS算法是一个递归算法,需要借助一个递归工作栈,故空间复杂度为 $O(|V|)$.
>
> 遍历图的过程实质是对每个顶点查找其邻接点的过程,其耗费的时间取决于所用的存储结构.以**邻接矩阵**表示时,查找每个顶点的邻接点的时间为 $O(|V|)$,故总的时间复杂度为 $O(|V|)^2$;以邻接表表示时,查找所有顶点的邻接点所需的时间为 $O(|E|)$,访问顶点所需的时间为 $O(|V|)$,总的时间复杂度为 $O(|V|+|E|)$.

深度优先搜索树

与广度优先搜索类似,深度优先搜索也会产生一棵深度优先生成树.对连通图调用 DFS才能产生深度优先生成树,否则产生的是深度优先生产森林.与 BFS类似,基于邻接表存储的深度优先生产树不唯一.

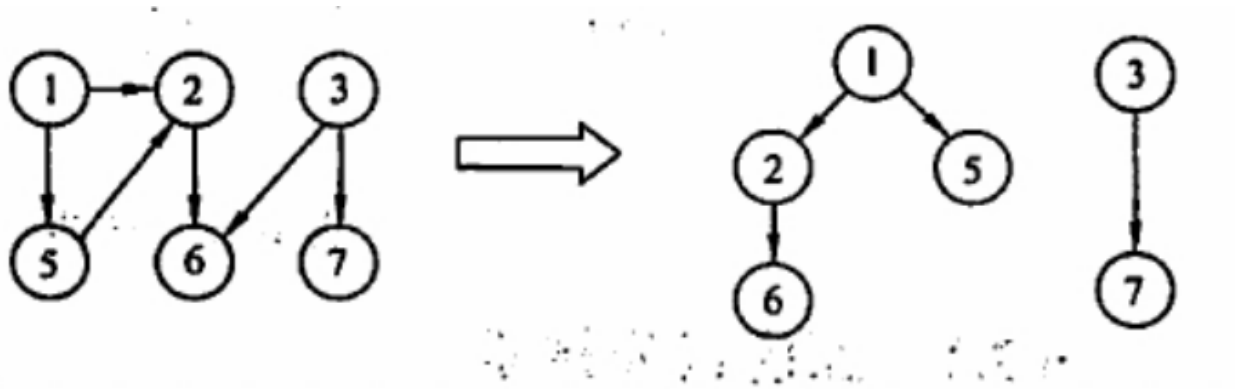


图 6.13　图的深度优先生成森林

图的遍历和连通性

对于**无向图**,两个函数调用 $BFS(G,i)$和 $DFS(G,i)$的次数等于该图的**连通分量数**;

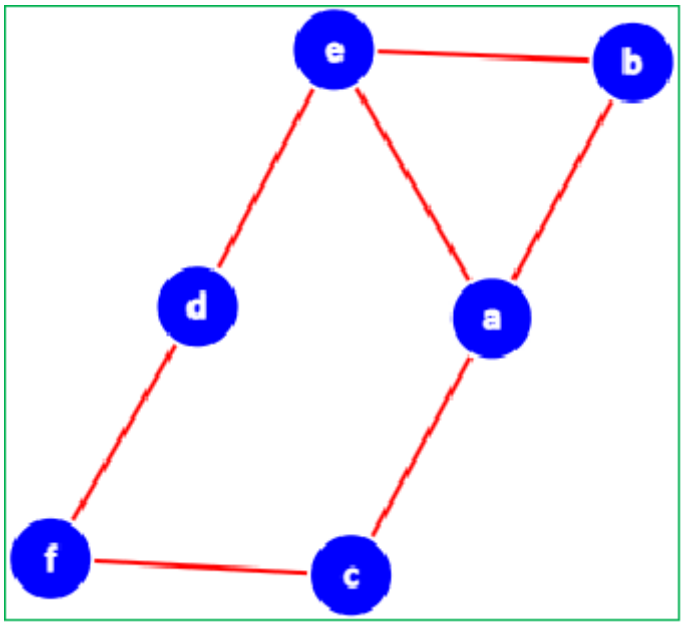
对于**有向图**,由于一个连通的有向图分为强连通的和非强连通的,它的连通子图也分为强连通分量和非强连通分量,非强连通分量一次调用 $BFS(G,i)$或 $DFS(G,i)$无法访问到该连通分量的所有顶点.

图 6.14　有向图的非强连通分量

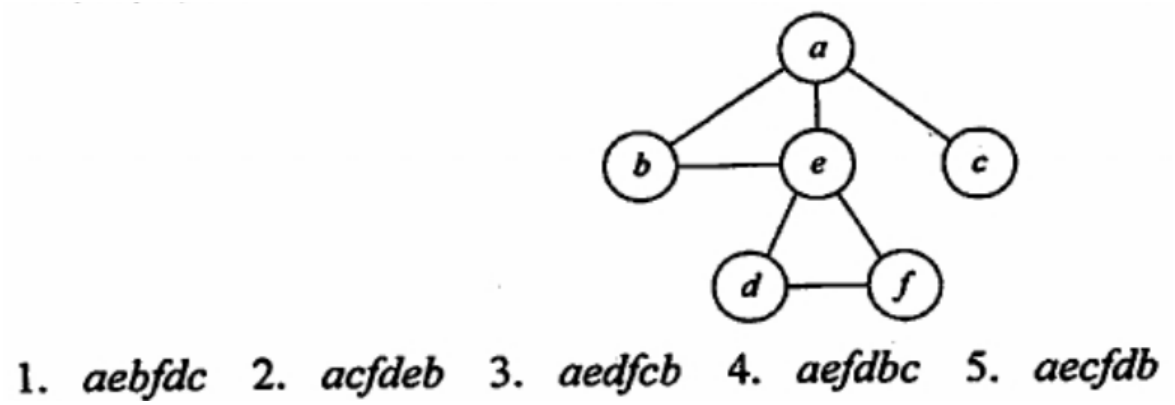
> 对一个有 n个顶点、 e条边的图采用**邻接表**表示时,进行 DFS遍历的时间复杂度为(C),空间复杂度为(A);进行 BFS遍历的时间复杂度为(C),空间复杂度为(A).
>
> A. $O(n)$
>
> B. $O(e)$
>
> C. $O(n+e)$
>
> D. $O(1)$
>
> 对有 n个顶点、 e条边的图采用**邻接矩阵**表示时,进行 DFS遍历的时间复杂度为(A),进行 BFS遍历的时间复杂度为(A).
>
> A. $O(n^2)$
>
> B. $O(e)$
>
> C. $O(n+e)$
>
> D. $O(e^2)$

> 无向图 $G=(V,E)$,其中 `V={a,b,c,d,e,f}`,`E={(a,b),(a,e),(a,c),(b,e),(c,f),(f,d),(e,d)}`,对该图从 a开始进行深度优先遍历,得到的顶点序列正确的是().
>
> A. a,b,e,c,d,f
>
> B. a,c,f,e,b,d
>
> C. a,e,b,c,f,d
>
> D. a,e,d,f,c,b

逐个顶点去检查是否存在`(v,w)`的边,若不存在证明该深度优先遍历不合法.该无向图如下图所示:



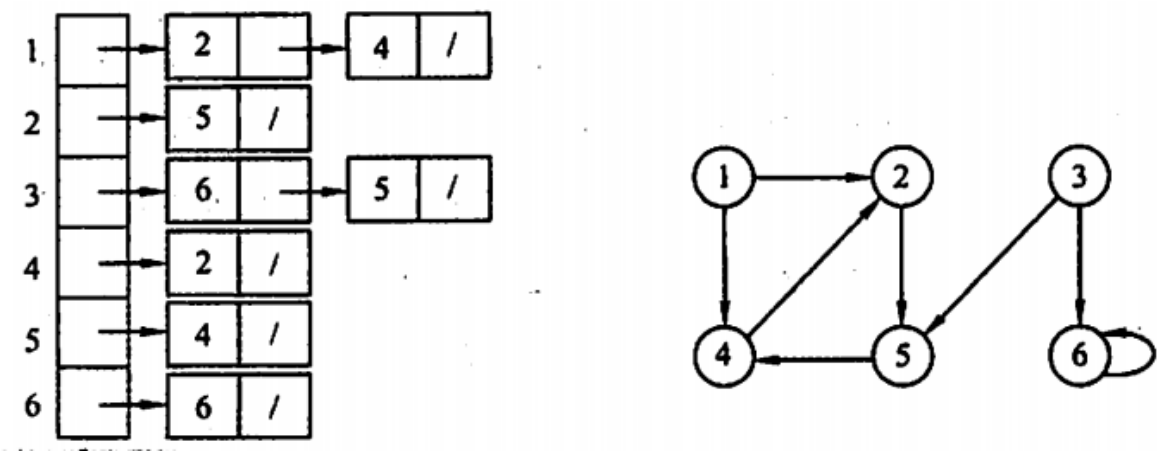
- 选项 A,由于 e到 c无直接相连的边,故 A错误.
- 选项 B,由于 f到 e无直接相连的边,故 B错误;
- 选项 C,由于 e(此处是从 b回溯回来)到 c无直接相连的边,故 C错误.

> 如下图所示,在下面的 5个序列中,符合深度优先遍历的序列个数是().
>
> 
>
>
> A. 5
>
> B. 4
>
> C. 3
>
> D. 2

符合深度优先遍历的个数是 2,分别是第 1个和第 4个.

- 2中 a(从 c回溯的)到 f无直接相连的边;
- 3中访问完 a, e, d, f此时回溯到 e有 b,不满足条件;
- 5中 e到 c无直接相连的边.

> 一个有向图 G的邻接表存储如下图所示,从顶点 1出发,对图 G调用深度优先遍历所得顶点序列是(A);按广度优先遍历所得顶点序列是(B).



A. 125436

B. 124536

C. 124563

D. 362514

深度优先遍历：从 1出发,邻接表中 1的邻接顶点分别是 $2, 4$,访问 2;从 2出发,邻接表中 2的邻接顶点为 5,访问 5;从 5出发,邻接表中 5的邻接顶点为 4,访问 4;从 4出发,邻接表中 4的邻接顶点为 2, 2被访问过;回溯,发现 $5, 2, 1$均无未被访问的邻接顶点;由于深度优先遍历仍要对未被访问的结点遍历,故此时访问 3;从 3出发,邻接表中 3的邻接顶点分别是 $6, 5$,访问 6;从 6出发,邻接表中 6的邻接顶点为 6,已经访问完;回溯,此时 3的另一个邻接顶点已被访问.故深度优先遍历的顶点序列为 $1, 2, 5, 4, 3, 6$.

广度优先遍历：初始队列为空,将起点 1放入队列中,队列中元素有 1.将 1从队列中弹出,将它未被访问的的邻接顶点中 $2, 4$依次放入队列中,队列中元素有 $2, 4$;将 2从队列中弹出,将它未被访问的邻接顶点 5放入队列中,队列中元素有 $4, 5$;将 4从队列中弹出,发现它的邻接顶点均已被访问过,此时队列中元素有 5;将 5从队列中弹出,发现它的邻接顶点均已被访问过,此时队列为空;此时将图中未被访问的结点 3放入队列中,此时队列元素有 3;将 3从队列中弹出,将它未被访问的邻接顶点 6放入队列中(邻接顶点 5已被访问过),此时队列元素有 6;将 6从队列中弹出,它的邻接顶点均已被访问过,此时队列为空,结束.故广度优先遍历的顶点序列为 $1, 2, 4, 5, 3, 6$.

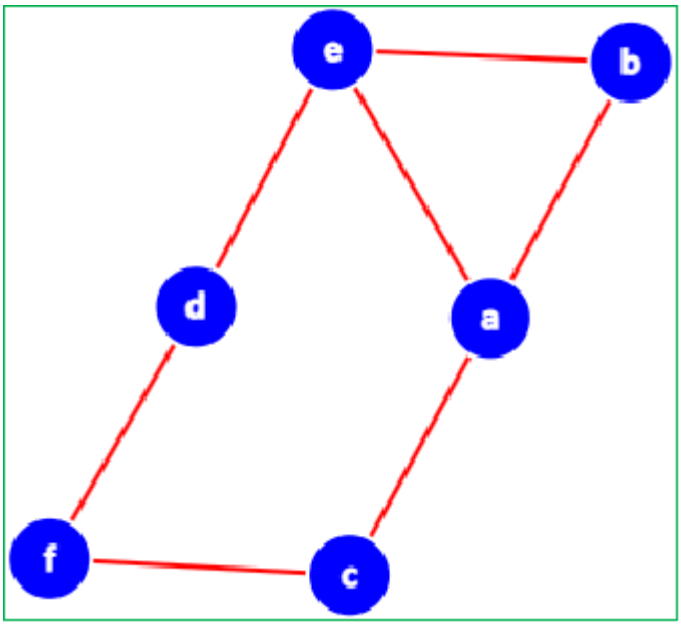
无向图 $G = (V, E)$,其中`V={a,b,c,d,e,f}`,`E={(a,b),(a,e),(a,c),(b,e),(c,f),(f,d),(e,d)}`.对该图进行深度优先遍历,不能得到的序列是().

A. $acfdeb$

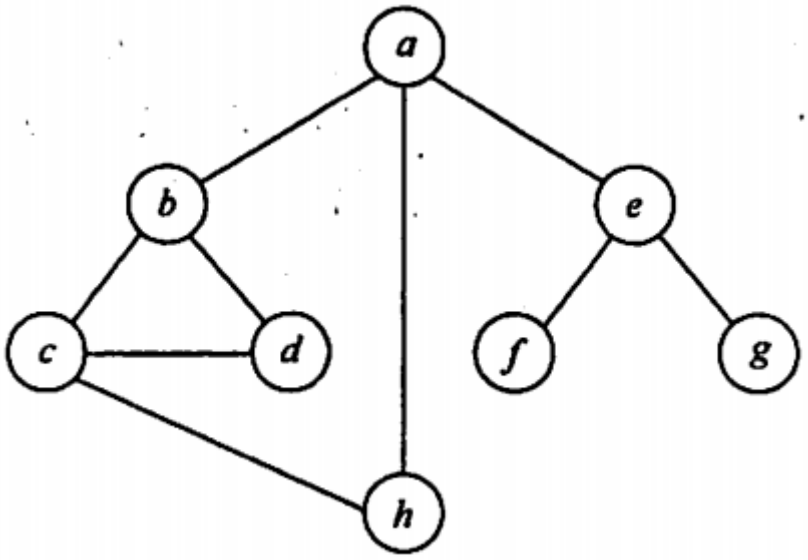
B. $aebdfc$

C. $aedfcb$

D. $abecdf$



选项 D中,由于 e到 c无直接相连的边,故不满足条件.

> **2013统考真题**：若对如下无向图进行遍历,则下列选项中,不是广度优先遍历序列的是().
>
> 
>
>
> A. h, c, a, b, d, e, g, f
>
> B. e, a, f, g, b, h, c, d
>
> C. d, b, c, a, h, e, f, g
>
> D. a, b, c, d, h, e, f, g

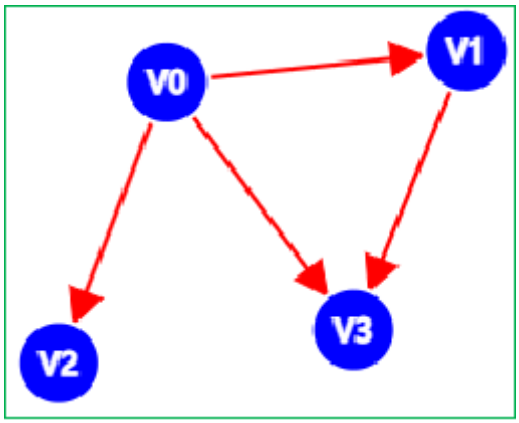
在广度优先搜索中,在当遍历一个结点时会把它的所有邻接顶点放入队列中. 而在选项 D中,在遍历 a时,会把 b, h, e放入队列中,故不满足条件.

> **2015统考真题**：设有向图 $G=(V, E)$,顶点集 `V={` V_0, V_1, V_2, V_3 `}`,边集 `E={` $\langle v_0, v_1\rangle, \langle v_0, v_2\rangle, \langle v_0, v_3\rangle, \langle v_1, v_3\rangle$ `}`,若从顶点 V_0开始对图进行深度优先遍历,则可能得到的不同遍历个数是().
>
> A. 2
>
> B. 3

C. 4

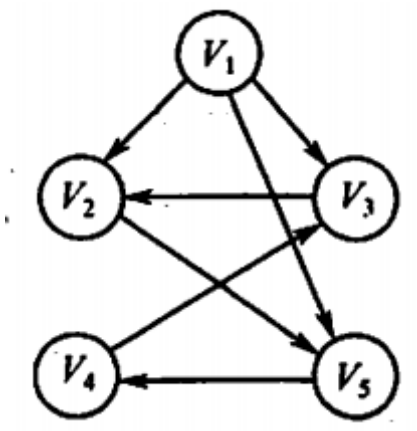
D. 5

已知起点从 V_0出发,故可枚举 6种情况判断是否满足深度优先遍历. 注意是**有向图**G如下图所示:



- V_0, V_1, V_2, V_3,不满足条件;
- V_0, V_1, V_3, V_2,满足条件;
- V_0, V_2, V_1, V_3,满足条件;
- V_0, V_2, V_3, V_1,满足条件;
- V_0, V_3, V_1, V_2,满足条件;
- V_0, V_3, V_2, V_1,满足条件;

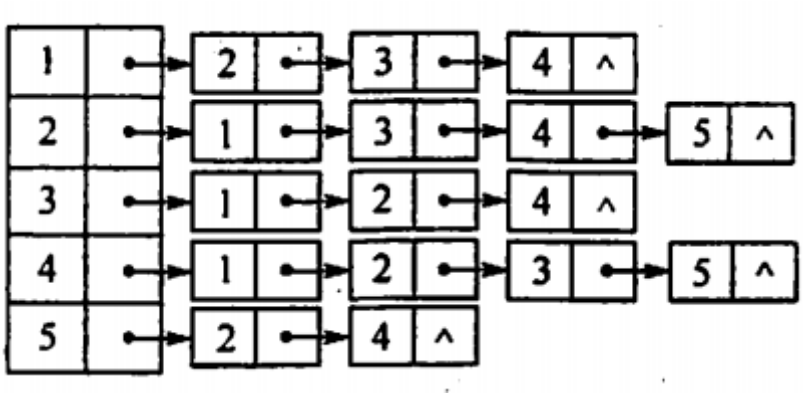
满足深度优先遍历的共有 5种结果.

> **2016统考真题**：下列选项中,不是下图深度优先搜索序列的是().
>
> 
>
>
> A. V_1, V_5, V_4, V_3, V_2
>
> B. V_1, V_3, V_2, V_5, V_4
>
> C. V_1, V_2, V_5, V_4, V_3
>
> D. V_1, V_2, V_3, V_4, V_5

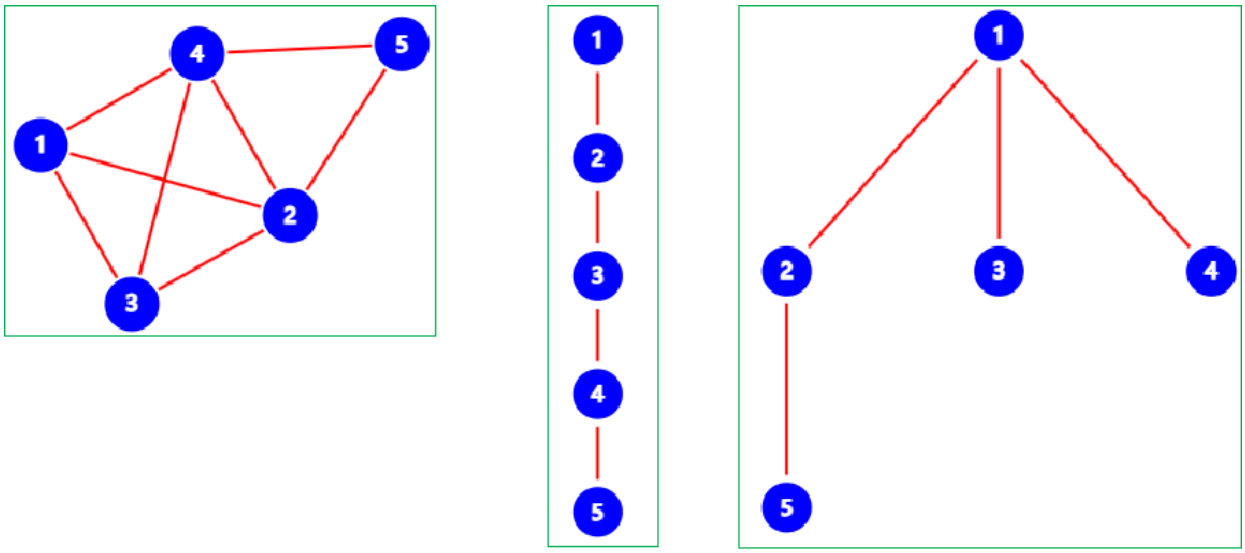
选项 D中,由于 V_2到 V_3无直接相连的边,故不满足条件.

6.3.1

> 图 $G=(V,E)$以邻接表存储,如下图所示,试画出图 G的深度优先生成树和广度优先生成树(假设从结点 1开始遍历)



左图为图 G,中图为深度优先生成树,右图为广度优先生成树.



6.3.2

> 设设计一个算法,判断一个无向图 G是否为一棵树.若是一棵树,则算法返回 $true$,否则返回 $false$.

一个无向图 G是一棵树的条件是： G必须是无回路的连通图或者是有 $n-1$条边的连通图,这里采用后者作为判断条件.对连通图的判定,可用能否遍历全部顶点来实现;可以采用**深度优先搜索**算法在遍历图的过程中统计可能访问到的**顶点个数和边的条数**，如果一次遍历就能访问到 n个顶点和 $n-1$条边，则可判定此图是一棵树。

```
1  bool visited[MaxVertexNum];
2
3  void DFS(MGraph G,int start,int &node_num,int &edge_num)
4  {
5      visited[start]=true;//标记该点,并增加顶点的个数
6      node_num++;
7
8      for(int j=FirstNeighbor(G, start);j!=-1;j=NextNeighbor(G, start, j))
9      {
10         //边数+1
```

```
        edge_num++;
        if(visited[j]==false)
            DFS(G,j,node_num,edge_num);
    }
}

bool isTree(MGraph G)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;
    int node_num=0,edge_num=0;

    DFS(G,1,node_num,edge_num);

    if(node_num==G.vexnum&&edge_num==2*(G.vexnum-1))
        return true;
    return false;
}
```

6.3.3

> 写出图的深度优先搜索 DFS算法的非递归算法(图采用邻接表形式).

```
bool visited[MaxVertexNum];

void visit(ALGraph G,int pos)
{
    cout << G.vertices[pos].data << " ";
}


void DFS(ALGraph G,int start)
{
    stack<int> sta;
    sta.push(start);//把起点放入栈中
    visited[start]=true;

    while(sta.size()>0)
    {
        int temp=sta.top();//取出栈顶元素
        visit(G,temp);
        sta.pop();
        for(int j=FirstNeighbor(G, temp);j!=-1;j=NextNeighbor(G, temp, j))//访问其未被访问的邻接顶点
            if(visited[j]==false)
            {
                sta.push(j);
                visited[j]=true;//标记,防止再次入栈
            }
    }
}

```

```
void DFSTraverse(ALGraph G)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;

    for(int i=1;i<=G.vexnum;i++)
        if(visited[i]==false)
            DFS(G,i);
}
```

6.3.4

> 分别采用基于深度优先遍历和广度优先遍历算法判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j的路径($i \neq j$).注意,算法中涉及图的基本操作必须在此存储结构上实现.

- **深度优先搜索**

```
bool visited[MaxVertexNum];

void DFS(ALGraph G,int start_,int end_,bool &can_reach)
{
    if(start_==end_)
    {
        can_reach=true;
        return ;
    }
    visited[start_]=true;

    for(int j=FirstNeighbor(G, start_);j!=-1;j=NextNeighbor(G, start_, j))
        if(visited[j]==false)
            DFS(G,j,end_,can_reach);
}

bool DFSTraverse(ALGraph G,int start_,int end_)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;
    bool can_reach=false;

    DFS(G,start_,end_,can_reach);

    return can_reach;
}
```

- **广度优先搜索**

```
bool visited[MaxVertexNum];

bool BFSTraverse(ALGraph G,int start_,int end_)
{
    for(int i=1;i<=G.vexnum;i++)
```

```
        visited[i]=false;

    queue<int> q;
    q.push(start_);
    visited[start_]=true;

    while(q.size()>0)
    {
        int temp=q.front();
        q.pop();

        if(temp==end_)
            return true;
        for(int j=FirstNeighbor(G, temp);j!=-1;j=NextNeighbor(G, temp, j))
            if(visited[j]==false)
            {
                q.push(j);
                visited[j]=true;
            }
    }
    return false;
}
```

6.3.5

> 假设图用邻接表表示,设计一个算法,输出从顶点 V_i到顶点 V_j的所有简单路径.

- *vector*版

```
bool visited[MaxVertexNum];

void DFS(ALGraph G,int start_,int end_,vector<VertexType> &path)
{

    //cout << start_ << " " << end_ << endl;
    if(start_==end_)
    {
        for(int i=0;i<path.size();i++)
            cout << G.vertices[path[i]].data << " ";
        cout << endl;
        return ;
    }

    for(int j=FirstNeighbor(G, start_);j!=-1;j=NextNeighbor(G, start_, j))
    {
        if(visited[j]==false)
        {
            visited[j]=true;
            path.push_back(j);
            DFS(G, j, end_, path);
            path.pop_back();
            visited[j]=false;
```

```
        }
    }
}

void DFSTraverse(ALGraph G,int start_,int end_)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;
    vector<VertexType> path;
    path.push_back(start_);
    visited[start_]=true;
    DFS(G, start_, end_,path);
    path.pop_back();
    visited[start_]=false;
}
```

- 非*vector*版

```
bool visited[MaxVertexNum];
VertexType path[MaxVertexNum];

void DFS(ALGraph G,int start_,int end_,int tot)
{
    visited[start_]=true;
    path[tot]=G.vertices[start_].data;
    if(start_==end_)
    {
        for(int i=0;i<=tot;i++)
            cout << path[i] << " ";
        cout << endl;
        visited[start_]=false;
        return ;
    }

    for(int j=FirstNeighbor(G, start_);j!=-1;j=NextNeighbor(G, start_, j))
        if(visited[j]==false)
            DFS(G,j,end_,tot+1);
    visited[start_]=false;
}

void DFSTraverse(ALGraph G,int start_,int end_)
{
    for(int i=1;i<=G.vexnum;i++)
        visited[i]=false;
    DFS(G, start_, end_, 0);
}
```