数据结构

图的遍历算法

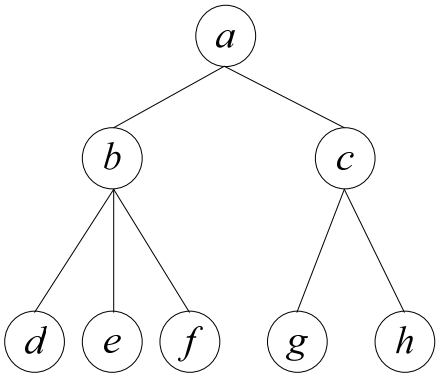
Outline

- 图遍历的概念与困难
- 深度优先搜索算法DFS
- 广度优先搜索算法BFS
- 两种图搜索算法的特性比较
- 图遍历算法的应用

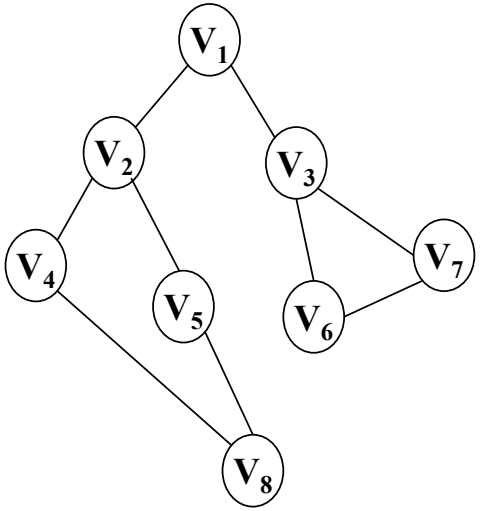
图遍历的概念与困难

- 图的遍历

  从图中某一顶点出发，沿着某条搜索路径访问图中**所有的顶点**，且使每个顶点**仅被访问一次**，就叫做图的遍历。

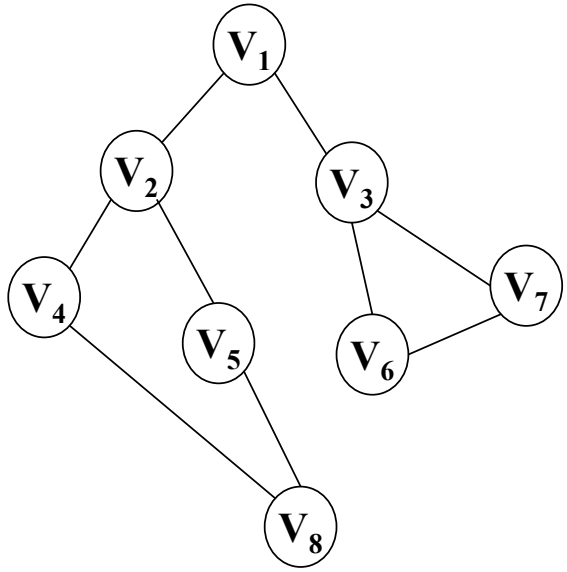


- 相对于树或二叉树结构，图遍历的复杂性是什么？
  - 可能存在结点被重复访问的问题
    - ✓ 图的任一顶点都可能与其它顶点相通，即图中可能存在**回路**，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。



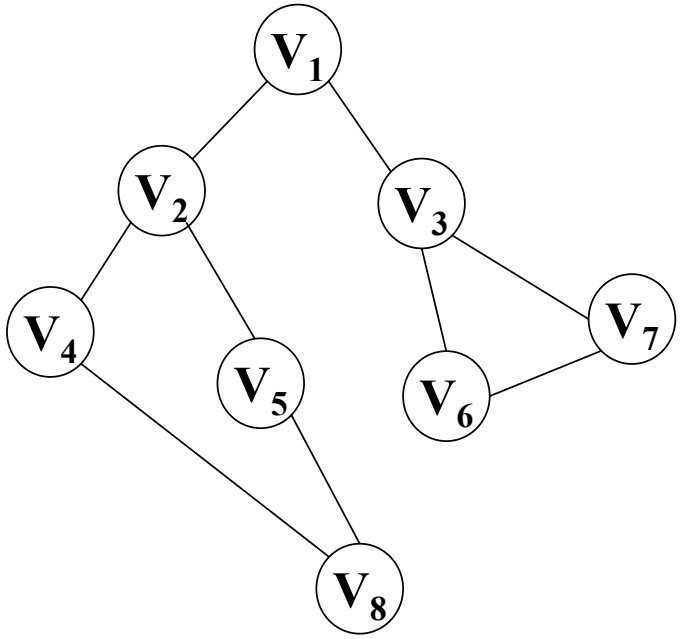
图遍历的概念与困难

- 相对于树或二叉树结构，图遍历的复杂性是什么？
  - 可能存在顶点被重复访问的问题
  - 如何解决?
    - 为了避免顶点重复访问，需设置一个标志顶点是否被访问过的辅助数组 *visited*[ ]，它的初始状态为 0，
    - 在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即修改 *visited*[*i*] 为 1，防止它被多次访问。
  - 搜索的规律如何如何组织?



图搜索的基本方法

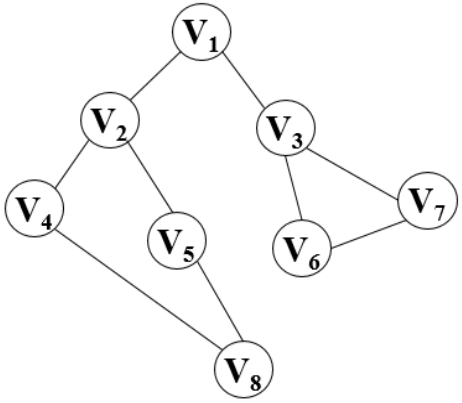
- 递归的遍历算法-- 深度优先遍历DFS
  - ✓ 类似于二叉树的先根遍历，是先根遍历的推广。
- 非递归的基本算法 – 广度优先遍历BFS
  - ✓ 类似于二叉树的层次遍历

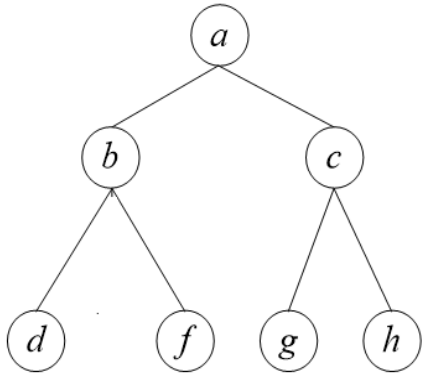


图的深度优先搜索算法DFS

- **DFS的基本过程**
  - ✓ 假设从图中某个顶点v出发，首先访问顶点v；
  - ✓ 然后依次从v的各个未曾访问过的邻接点出发执行深度优先遍历，直至图中所有已被访问的顶点的邻接点都被访问到。
  - ✓ 若此时图中还有未被访问的顶点，则任选其中之一作为新的出发点，重新开始上述过程，直至图中所有顶点都被访问到。

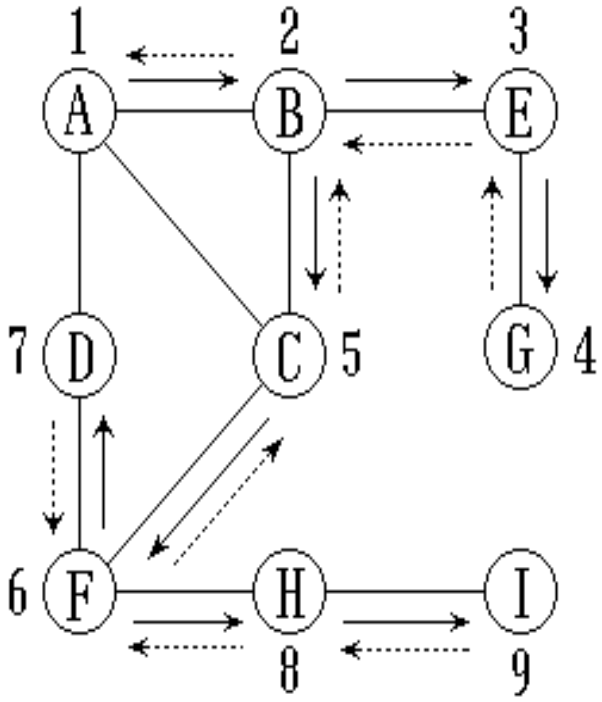
- **二叉树先根遍历过程**
  - ✓ 首先访问根结点；
  - ✓ 对根的左子树执行先根遍历；
  - ✓ 对根的右子树执行先根遍历。





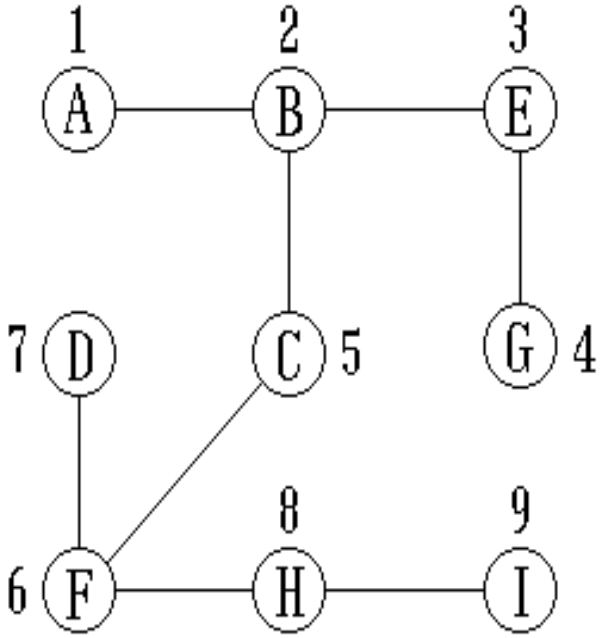
z1

图的深度优先搜索过程的示例



搜索

回退



DFS输出的顶点序列：A B E G C F D H I

深度优先生成树

z1

DFS的递归算法描述（**邻接矩阵表示法**）：

```
template<class T>
void MGraph<T>::DFSTraverse( )
{
    bool* visited=new bool[vexnum];
    for(v=0; v<vexnum; v++)
        visited[v]=false;
    for(v=0;v<vexnum;v++)
        if(!visited[v])
            DFS(v, visited);
    delete []visited;
}
```

```
template<class T>
void MGraph<T>::DFS( int v, bool* visited )
{
    cout<<vexs[v];
    visited[v]=true;
    for(i=0; i<vexnum; i++)
        if( edges[v][i] == 1 && !visited[i] )
            DFS( i, visited );
}
```

问题：

1. 如何判断一个图是否是连通的？
2. 如何求出一个非连通图中的连通分量个数？

z1

DFS的递归算法描述（**邻接矩阵表示法**）：

```
template<class T>
void MGraph<T>::DFSTraverse( )
{
    bool* visited=new bool[vexnum];
    for(v=0; v<vexnum; v++)
        visited[v]=false;
    for(v=0;v<vexnum;v++)
        if(!visited[v])
            DFS(v, visited);
    delete []visited;
}
```

```
template<class T>
void MGraph<T>::DFS( int v, bool* visited )
{
    cout<<vexs[v];
    visited[v]=true;
    for(i=0; i<vexnum; i++)
        if( edges[v][i] == 1 && !visited[i] )
            DFS( i, visited );
}
```

- **算法的时间复杂度是多少?**
  - 函数DFS被递归调用的次数为n次
  - 函数DFS每次执行的代价是$O(n)$
  - 因而总的时间复杂度是： $O(n^2)$

z1

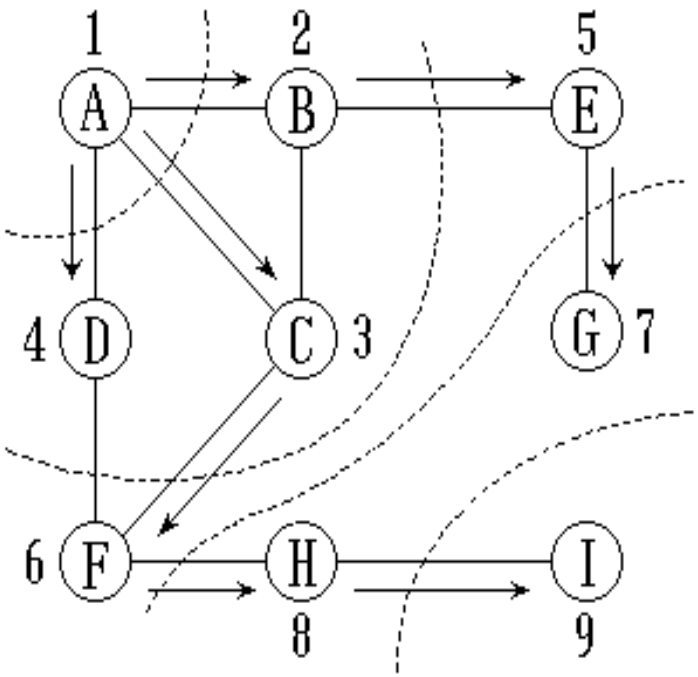
图的广度优先搜索算法BFS

- **BFS的基本过程**
  - ✓ 假设从图中某个顶点v出发，首先访问顶点v；
  - ✓ 然后依次访问v的各个未曾访问过的邻接点，并保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问。直至图中所有已被访问的顶点的邻接点都被访问到。
  - ✓ 若此时图中还有未被访问的顶点，则任选其中之一作为新的出发点，重新开始上述过程，直至图中所有顶点都被访问到。

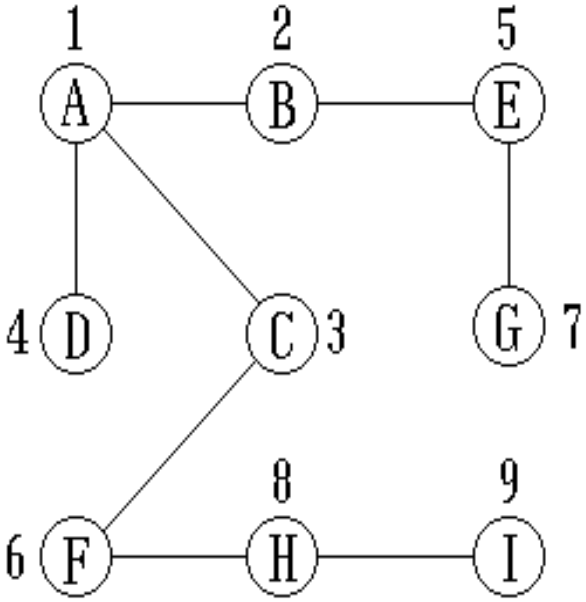
逐层访问的思想

z1

图的广度优先搜索过程的示例



搜索



广度优先搜索过程　　**广度优先生成树**

BFS输出的顶点序列：A B C D E F G H I

z1

广度优先搜索算法BFS的实现

- ✓ **其关键之处在于怎么保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问?**
- ✓ **为了实现逐层访问，算法需要使用一个*队列*，以便按序逐层访问。**
- ✓ **为避免重复访问，需要一个辅助数组 visited[ ]，给被访问过的顶点加标记。**

z1

BFS算法描述（**邻接矩阵表示法**）：

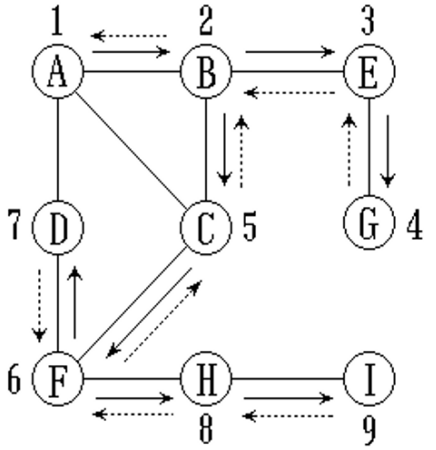
```
template<class T>
void MGraph<T>::BFSTraverse()  //v为遍历出发点序号 {
    SeqQueue Q;
    bool* visited=new bool[vexnum];
    for(i=0; i<vexnum; i++) visited[i]=0; // 置初值
    for(v=0;v<vexnum;v++)
        if(!visited[v]){
            cout<<vexs[v];  visited[v]=1;  //设置出发点访问标志为已访问
            Q.EnQueue(v);
            while( ! Q. Empty() ) {
                u=Q.DeQueue();
                for(j=0; j<vexnum; j++ )
                    if(edges [u][j]==1 && !visited[j]){
                        cout<<vexs[j];  visited[j]=1;
                        Q.EnQueue( j);
                    }
            }
    delete [] visited;
}
```

算法的时间复杂度为$O(n^2)$

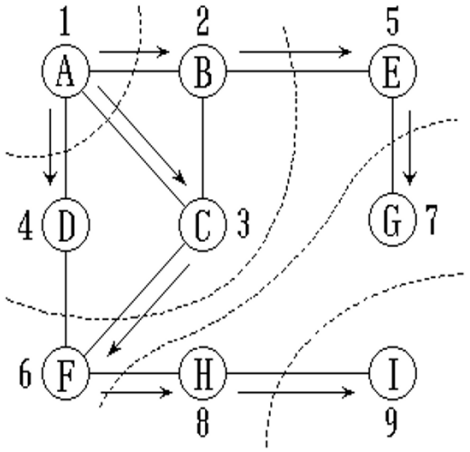
z1

DFS与BFS算法特性的比较

- 对每个顶点的处理机会?
  - DFS: 2
    - ✓ At discovering
    - ✓ At finishing
  - BFS: only 1, when dequeued



DFS



BFS

z1

DFS与BFS算法的应用场景

- BFS
  - 适合用来求从一个源点到另一个顶点的最短距离（指路径上的边数最少）的路径。

- DFS
  - 相对于BFS，DFS因为有2次处理时机，因而有非常广泛的应用
  - 例如：搜索从一个源点到另一个顶点的所有路径、判别有向图的有向环、有向图强连通分量的划分、无向图双连通分量的划分等。

图遍历算法的应用

- 图遍历算法是很多图处理算法的核心与基础，因为很多基于图的问题求解往往是在遍历过程中完成的。
- 图的遍历算法包含深度优先遍历和广度优先遍历两种算法，在不同的应用场合可选择不同的遍历算法。

- **例8-1：设计一个算法，找出图G中从顶点u到顶点v的所有简单路径。假设图G采用邻接表存储结构。**

  问题1：采用哪种搜索方法？

  - 深度优先搜索

  问题2：如何找出图中从顶点u到顶点v的所有简单路径？

  - 允许顶点能重复到达

```
void Find_All_Path(ALGraph<T> &G, int u, int v, int k)
// k表示当前路径长度
{
    path[k]=u;  //加入当前路径中
    visited[u]=true;
    if( u==v ) {   //找到了一条简单路径
          cout<<"Found one path!\n";
          for(int i=0; i<=k; i++)
             cout<<path[i]<<endl;  //输出路径
    }
    else
          for(p=G.firstEdge(u); p; p=p->nextedge){
             w=p->adjvex;
             if(!visited[w])
                  Find_All_Path(G, w, v, k+1);  //继续寻找
           }
    visited[u]=false;
    path[k]=0;      //回溯
}
```

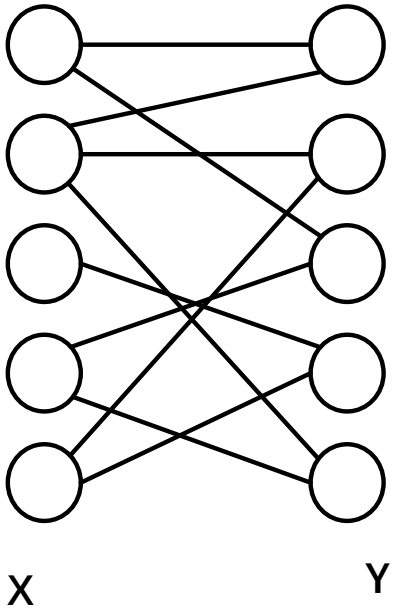
例8-2：设计一个算法，找出图G中从顶点u到顶点v长度为n的所有简单路径。假设图G采用邻接表存储结构。

- 在例8-1算法的基础上增加一个长度参数n。

```
int path[MAXSIZE];
bool visited[MAXSIZE];
void Find_All_Path(ALGraph<T> &G, int u, int v, int k, int n){
    path[k]=u; //加入当前路径中
    visited[u]=true;
    if(u==v && k==n){ //找到了一条简单路径
        cout<<"Found one path!\n";
        for(int i=0; i<=n; i++)
            cout<<path[i]<<endl; //输出路径
    }
    else
        for(p=G.firstEdge(u); p; p=p->nextedge){
            w=p->adjvex;
            if(!visited[w])
                Find_All_Path(G, w, v, n, k+1);
        }
    visited[u]=false;
    path[k]=0;      //回溯
}
```

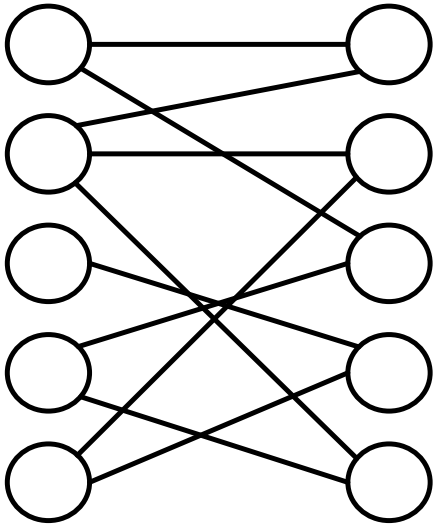
- **例8-3：设计一个算法，判断一个给定的连通图G是否为二部图（Bipartite）。假设图G采用邻接表存储结构。**

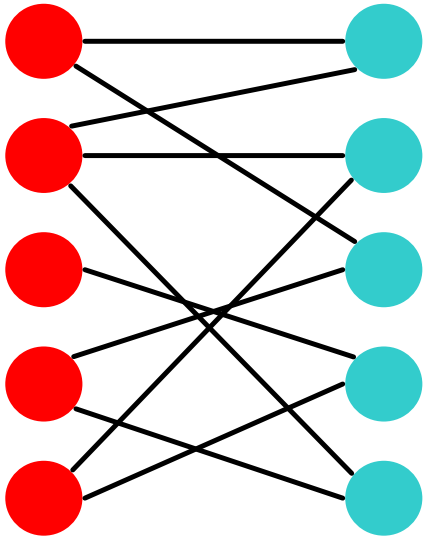
首先给出二部图的定义：一个二部图G=(V, E)是一个无向图，它的顶点集合可以被划分成X和Y，并具有下述性质：每条边有一个端点在X中且另一个端点在Y中。下图给出了一个二部图的示例。



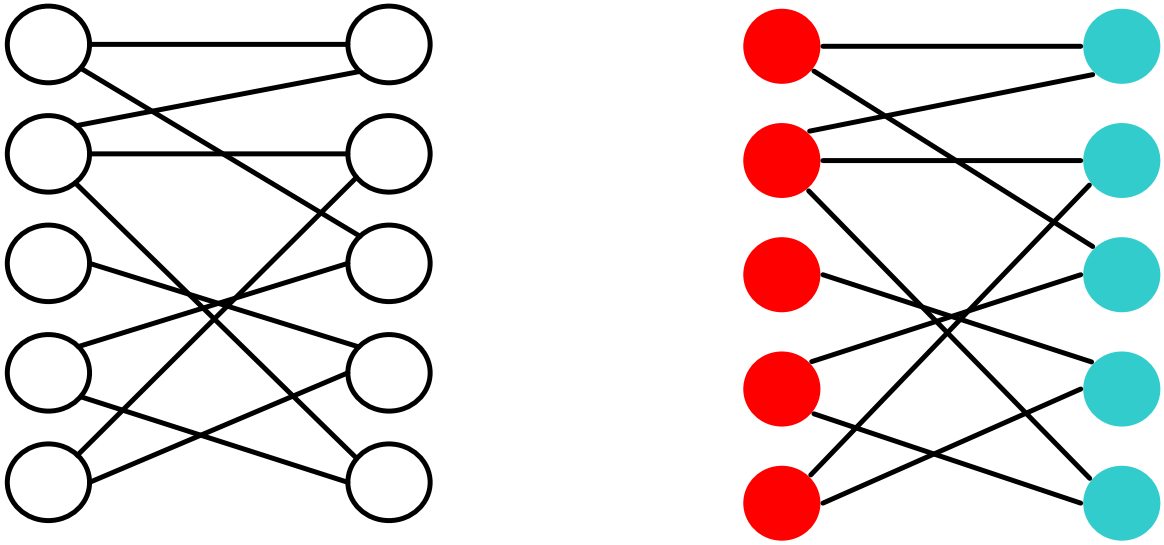
算法设计思路:

- 问题1: 判断的基本方法?
  - 为了便于实现二部图的判定，可以想象将集合X中的顶点着红色，集合Y中的顶点着蓝色。
  - 这样将判断问题转换成着色问题。
  - 则若发现某条边的两个端点颜色相同，该图为非二部图。





- 问题2: 如何实现顶点的着色？
  - 任取一个顶点s并且将它着红色。接着，找出顶点s 的所有邻接点，并将这些顶点都着为蓝色；接下来再分别将这些顶点的所有邻接点都着为红色，…，如此间替对顶点进行着色，直到图中所有顶点都被着色为止。
  - 可以在广度优先遍历算法的实现中再加上一个记录所有顶点颜色的数组Color[]，并在遍历过程中相应地记录每个顶点的颜色。



```
bool check_Bipartite( ALGraph<T> &G ) {
    int* visited=new int[G.vexterNum()];
    int* color=new int[G.vexterNum()];
    SeqQueue Q;
   for(int i=0;i<G.vexterNum();i++)  visited[i]=0;
   visited[0]=1;  //从第0个顶点开始处理
   color[0]=1;    //1表示红色，-1表示蓝色
   Q.EnQueue(0);
   while(! Q.Empty() ){
          u=DeQueue( Q );
          for(p=G.firstEdge(u); p; p=p->nextedge ){
                  k=p->adjvex;
                  if(!visited[k]){
                          visited[k]=1;
                          color[k]= -color[u];
                          Q.EnQueue( k );
                  }
          }
   }
```

```
    for( i=0; i<G.vexterNum(); i++)
        for(p=G.firstEdge(i); p; p=p->nextedge){
            k=p->adjvex;
            if(color[i]==color[k]){
                delete []visited;
                delete []color;
                return false;
             }
        }
    delete []visited;
    delete []color;
    return true;
}
```

z1