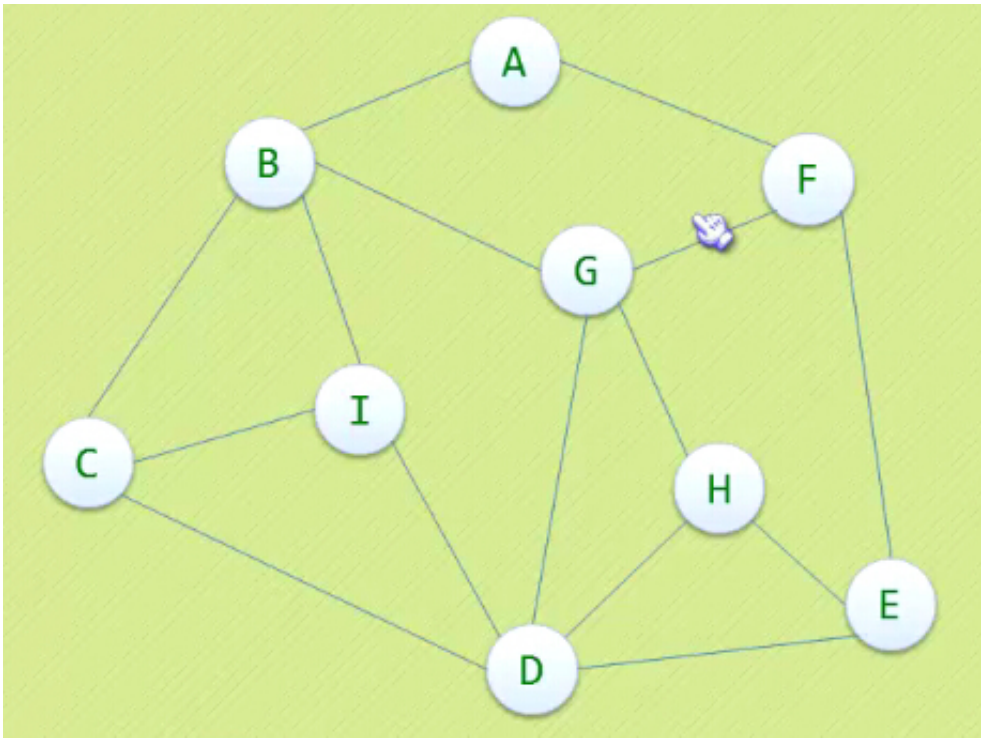
https://my.oschina.net/u/140462/blog/281268

深度优先遍历是连通图的一种遍历策略。其基本思想如下：
设x是当前被访问顶点，在对x做过访问标记后，选择一条从x出发的未检测过的边(x，y)。若发现顶点y已访问过，则重新选择另一条从x出发的未检测过的边，否则沿边(x，y)到达未曾访问过的y，对y访问并将其标记为已访问过；然后从y开始搜索，直到搜索完从y出发的所有路径，即访问完所有从y出发可达的顶点之后，才回溯到顶点x，并且再选择一条从x出发的未检测过的边。上述过程直至从x出发的所有边都已检测过为止。代码示例中遍历如下图所示的图。




```
package test.algorithm.FastSlowPointer;

import java.util.Stack;

/**
 * 图的深度优先遍历
 * @author serenity
 *
 */
```

```java
public class Graph {

    // 存储节点信息
    private char[] vertices;

    // 存储边信息（邻接矩阵）
    private  int[][] arcs;

    // 图的节点数
    private int vexnum;

    // 记录节点是否已被遍历
    private boolean[] visited;

    // 初始化
    public Graph(int n) {
        vexnum = n;
        vertices = new char[n];
        arcs = new int[n][n];
        visited = new boolean[n];
        for (int i = 0; i < vexnum; i++) {
            for (int j = 0; j < vexnum; j++) {
                arcs[i][j] = 0;
            }
        }
    }

    // 添加边(无向图)
    public void addEdge(int i, int j) {
        // 边的头尾不能为同一节点
        if (i == j) return;

        arcs[i][j] = 1;
        arcs[j][i] = 1;
    }

    // 设置节点集
    public void setVertices(char[] vertices) {
        this.vertices = vertices;
```

```
}

// 设置节点访问标记
public void setVisited(boolean[] visited) {
    this.visited = visited;
}

// 打印遍历节点
public void visit(int i){
    System.out.print(vertices[i] + " ");
}

// 从第i个节点开始深度优先遍历
private void traverse(int i) {
    // 标记第i个节点已遍历
    visited[i] = true;
    // 打印当前遍历的节点
    visit(i);

    // 遍历邻接矩阵中第i个节点的直接联通关系
    for(int j=0;j<vexnum;j++) {
        // 目标节点与当前节点直接联通，并且该节点还没有被访问，递归
        if(arcs[i][j]==1 && visited[j]==false) {
            traverse(j);
        }
    }
}

// 图的深度优先遍历（递归）
public void DFSTraverse() {
    // 初始化节点遍历标记
    for(int i=0; i<vexnum; i++) {
        visited[i] = false;
    }

    // 从没有被遍历的节点开始深度遍历
    for(int i=0;i<vexnum;i++) {
        if(visited[i]==false) {
            // 若是连通图，只会执行一次
```

```
                traverse(i);
            }
        }
    }

    // 图的深度优先遍历（非递归）
    public void DFSTraverse2() {
        // 初始化节点遍历标记
        for (int i = 0; i < vexnum; i++) {
            visited[i] = false;
        }

        Stack<Integer> s = new Stack<Integer>();
        for(int i=0;i<vexnum;i++){
            if(!visited[i]){
                //连通子图起始节点
                s.add(i);
                do{
                    // 出栈
                    int curr = s.pop();

                    // 如果该节点还没有被遍历，则遍历该节点并将子节点入
栈
                    if(visited[curr]==false){
                        // 遍历并打印
                        visit(curr);
                        visited[curr] = true;

                        // 没遍历的子节点入栈
                        for(int j=vexnum-1; j>=0 ; j-- ){
                            if(arcs[curr][j]==1 &&
visited[j]==false){
                                s.add(j);
                            }
                        }
                    }
                }while(!s.isEmpty());
            }
        }
```

```
}

public static void main(String[] args) {
    Graph g = new Graph(9);
    char[] vertices = {'A','B','C','D','E','F','G','H','I'};
    g.setVertices(vertices);

    g.addEdge(0, 1);
    g.addEdge(0, 5);
    g.addEdge(1, 0);
    g.addEdge(1, 2);
    g.addEdge(1, 6);
    g.addEdge(1, 8);
    g.addEdge(2, 1);
    g.addEdge(2, 3);
    g.addEdge(2, 8);
    g.addEdge(3, 2);
    g.addEdge(3, 4);
    g.addEdge(3, 6);
    g.addEdge(3, 7);
    g.addEdge(3, 8);
    g.addEdge(4, 3);
    g.addEdge(4, 5);
    g.addEdge(4, 7);
    g.addEdge(5, 0);
    g.addEdge(5, 4);
    g.addEdge(5, 6);
    g.addEdge(6, 1);
    g.addEdge(6, 3);
    g.addEdge(6, 5);
    g.addEdge(6, 7);
    g.addEdge(7, 3);
    g.addEdge(7, 4);
    g.addEdge(7, 6);
    g.addEdge(8, 1);
    g.addEdge(8, 2);
    g.addEdge(8, 3);

    System.out.print("深度优先遍历（递归）：");
```

```
        g.DFSTraverse();

        System.out.println();

        System.out.print("深度优先遍历（非递归）：");
        g.DFSTraverse2();
    }

}
```