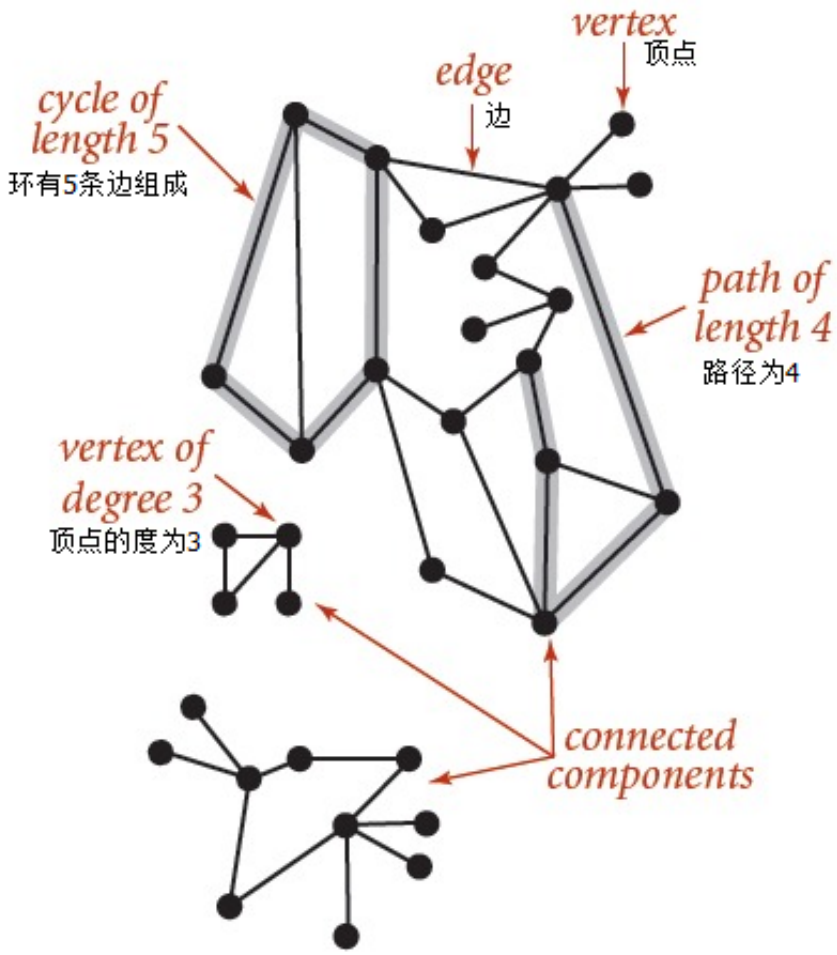
从这篇文章开始介绍图相关的算法，这也是Algorithms在线课程第二部分的第一次课程笔记。 图的应用很广泛，也有很多非常有用的算法，当然也有很多待解决的问题，根据性质，图可以分为无向图和有向图。本文先介绍无向图，后文再介绍有向图。 之所以要研究图，是因为图在生活中应用比较广泛：

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

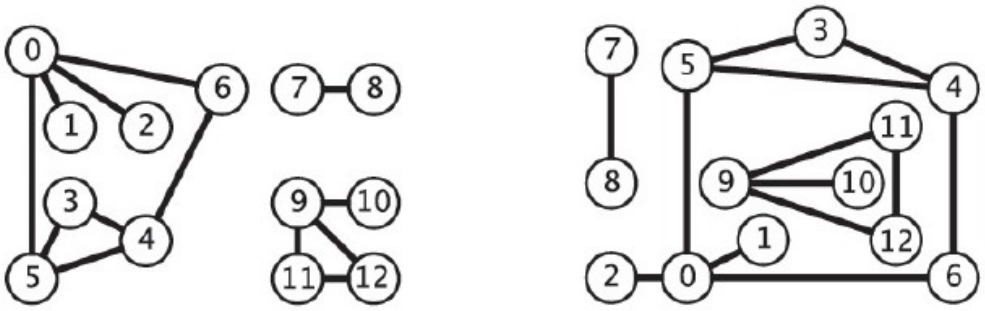
无向图

图是若干个顶点(Vertices)和边(Edges)相互连接组成的。边仅由两个顶点连接，并且没有方向的图称为无向图。 在研究图之前，有一些定义需要明确，下图中表示了图的一些基本属性的含义，这里就不多说明。
(cycle of length 5 : 回路[环]有5条边; path of length 4 : 两个顶点间有4条边)



图的API 表示

在研究图之前，我们需要选用适当的数据结构来表示图，有时候，我们常被我们的直觉欺骗,如下图，这两个其实是一样的，这其实也是一个研究问题，就是如何判断图的形态。

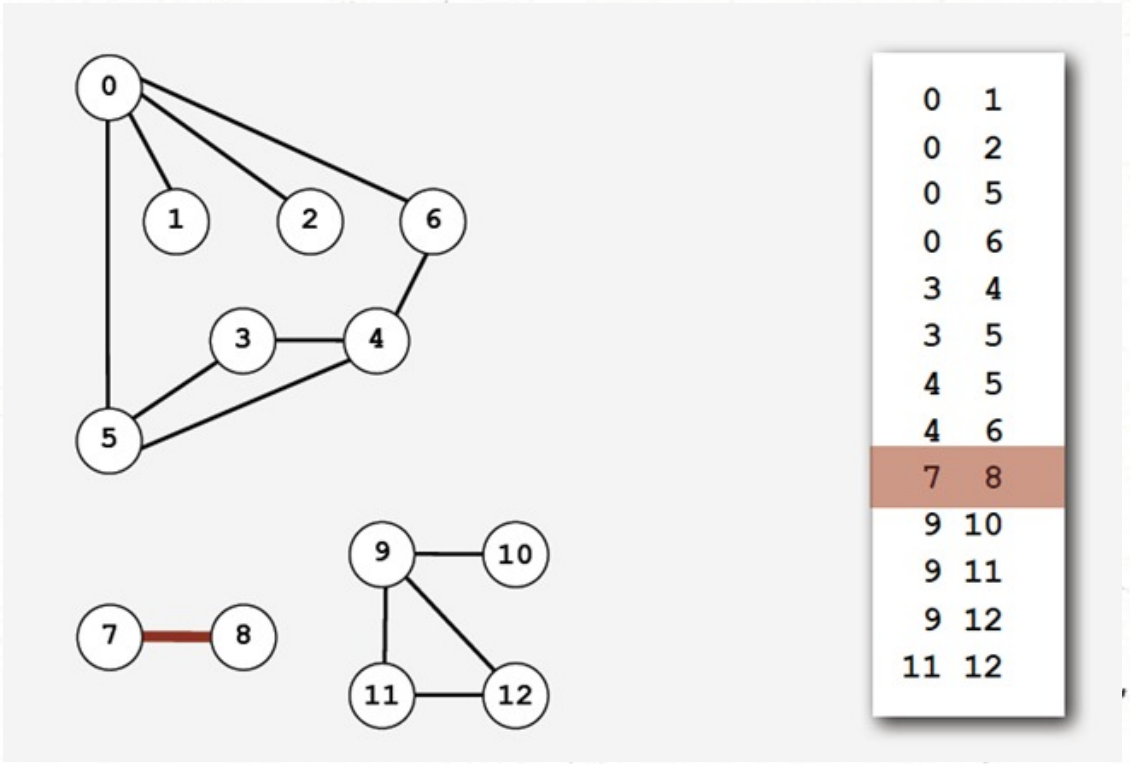


two drawings of the same graph

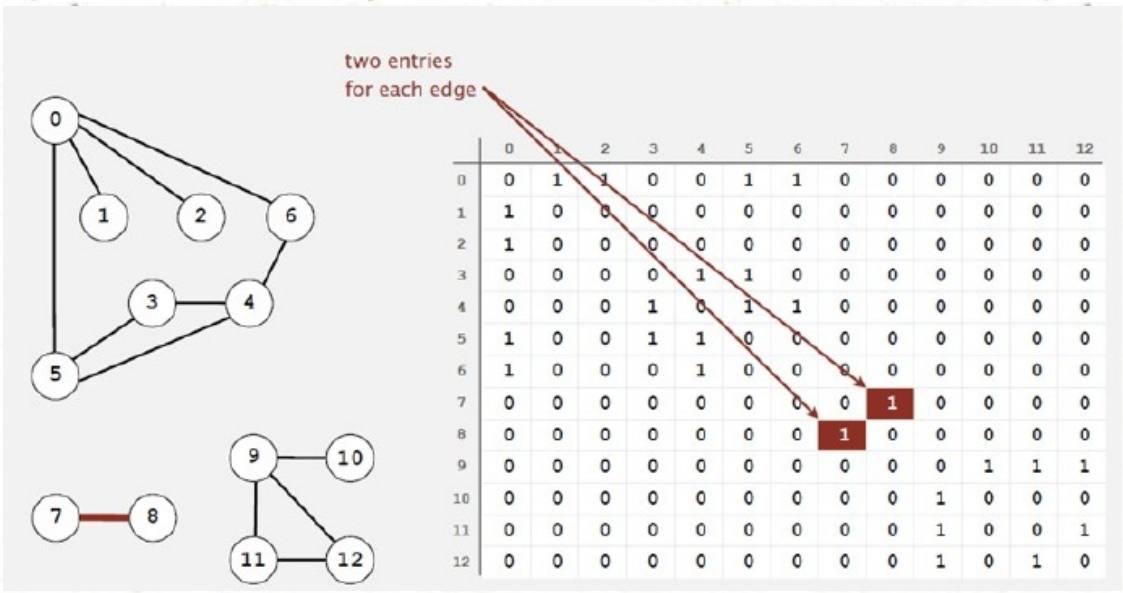
要用计算机处理图，我们可以抽象出以下的表示图的API：

public class Graph	
Graph(int V)	*create an empty graph with V vertices*
Graph(In in)	*create a graph from input stream*
void addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer> adj(int v)	*vertices adjacent to v*
int V()	*number of vertices*
int E()	*number of edges*
String toString()	*string representation*

Graph的API的实现可以由多种不同的数据结构来表示，最基本的是维护一系列边的集合，如下：

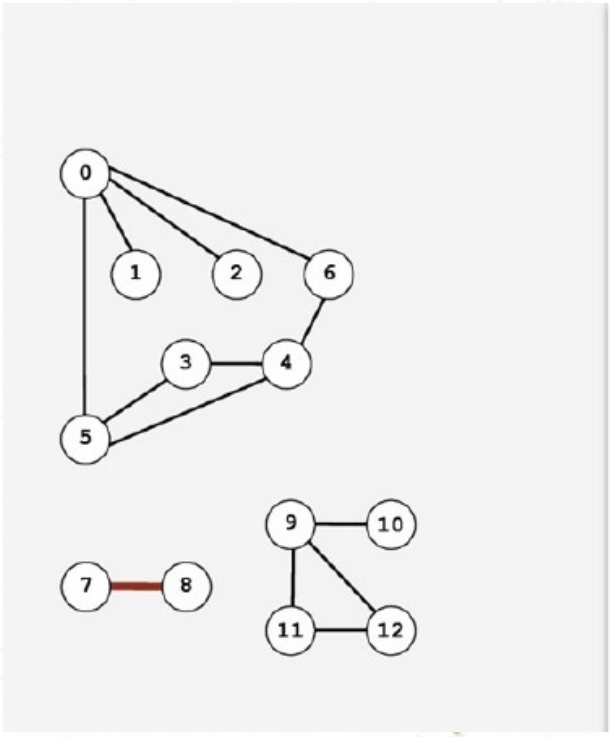


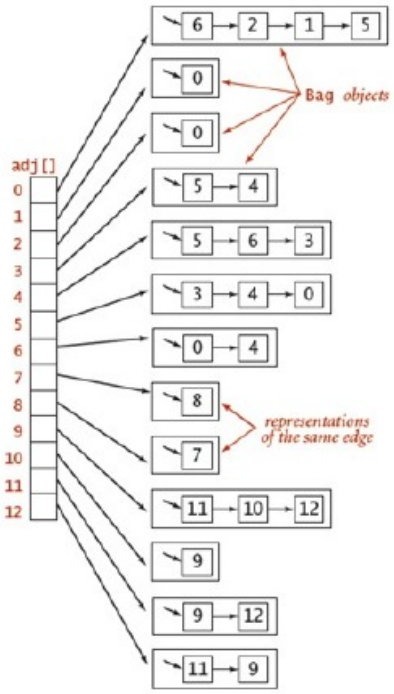
还可以使用邻接矩阵来表示：



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

也可以使用邻接列表来表示：



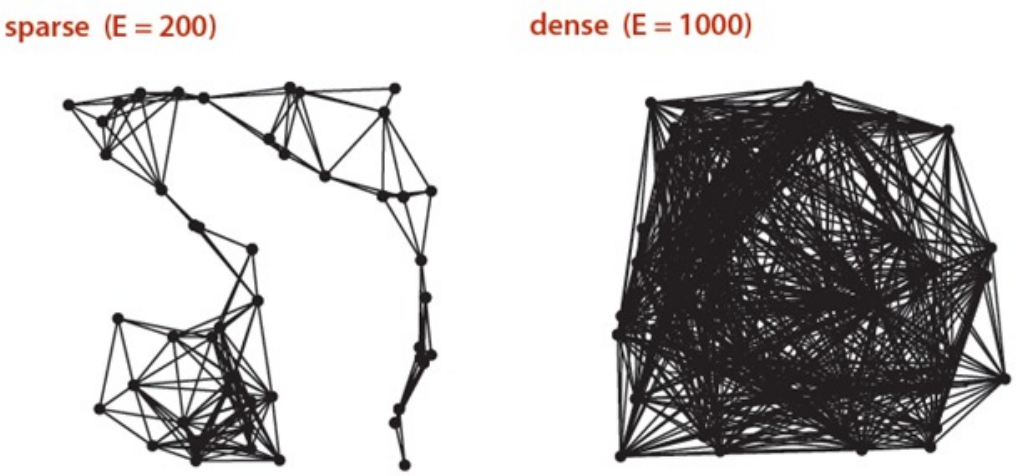


由于采用如上方式具有比较好的灵活性，采用邻接列表来表示的话，可以定义如下数据结构来表示一个Graph对象。

```
public class Graph
{
    private readonly int verticals;//顶点个数
    private int edges;//边的个数
    private List<int>[] adjacency;//顶点联接列表

    public Graph(int vertical)
    {
        this.verticals = vertical;
        this.edges = 0;
        adjacency=new List<int>[vertical];
        for (int v = 0; v < vertical; v++)
        {
            adjacency[v]=new List<int>();
        }
    }

    public int GetVerticals ()
    {
```

```
        return verticals;
    }

    public int GetEdges()
    {
        return edges;
    }

    public void AddEdge(int verticalStart, int verticalEnd)
    {
        adjacency[verticalStart].Add(verticalEnd);
        adjacency[verticalEnd].Add(verticalStart);
        edges++;
    }

    public List<int> GetAdjacency(int vetical)
    {
        return adjacency[vetical];
    }
}
```

图也分为稀疏图和稠密图两种，如下图： 在这两个图中，顶点个数均为50，但是稀疏图中只有200个边，稠密图中有1000个边。在现实生活中，大部分都是稀疏图，即顶点很多，但是顶点的平均度比较小。



Two graphs (V = 50)

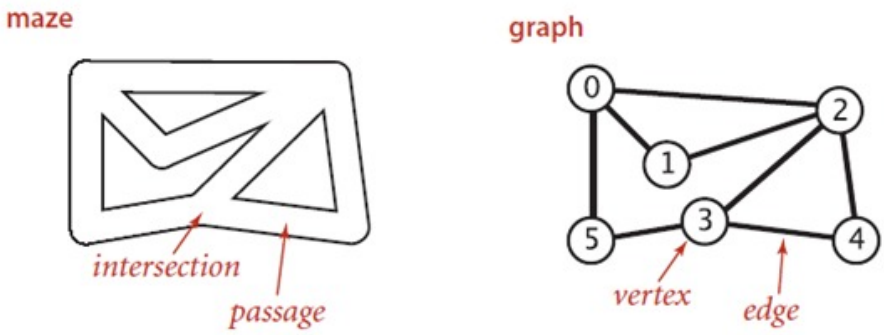
采用以上三种表示方式的效率如下：

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

在讨论完图的表示之后，我们来看下在图中比较重要的一种算法，即深度优先算法：

深度优先算法

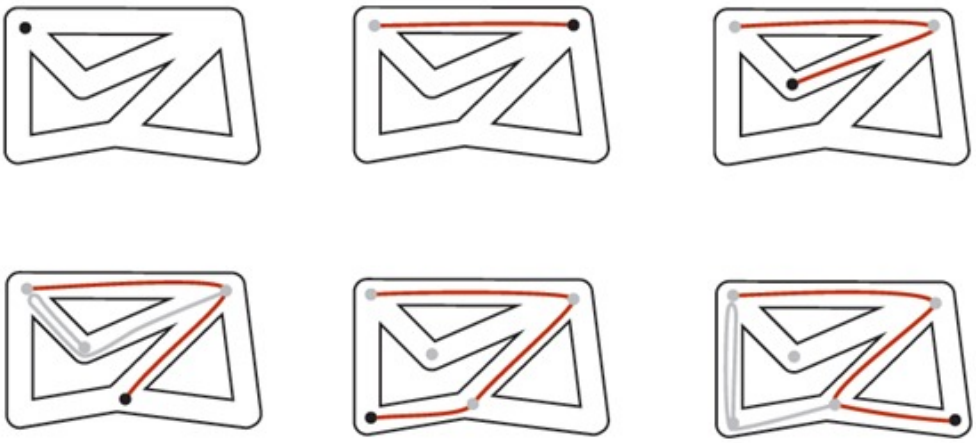
在谈论深度优先算法之前，我们可以先看看迷宫探索问题。下面是一个迷宫和图之间的对应关系： 迷宫中的每一个交会点代表图中的一个顶点，每一条通道对应一个边。



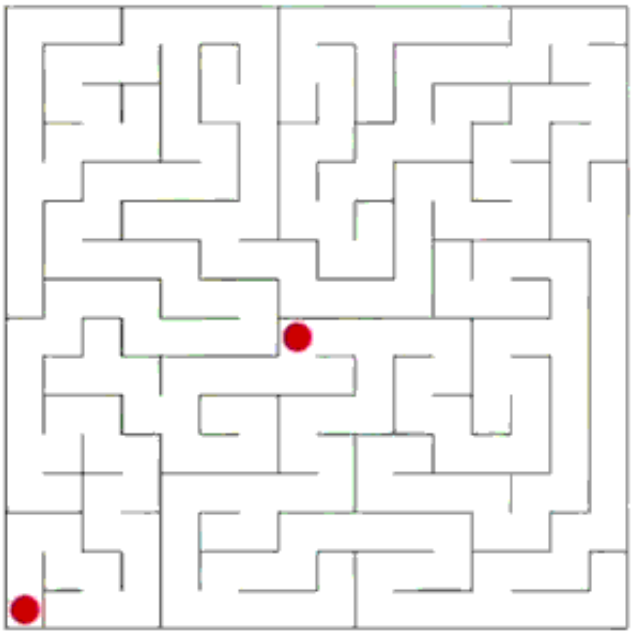
迷宫探索可以采用Trémaux绳索探索法。即：

- 在身后放一个绳子
- 访问到的每一个地方放一个绳索标记访问到的交会点和通道
- 当遇到已经访问过的地方，沿着绳索回退到之前没有访问过的地方：

图示如下：



下面是迷宫探索的一个小动画：



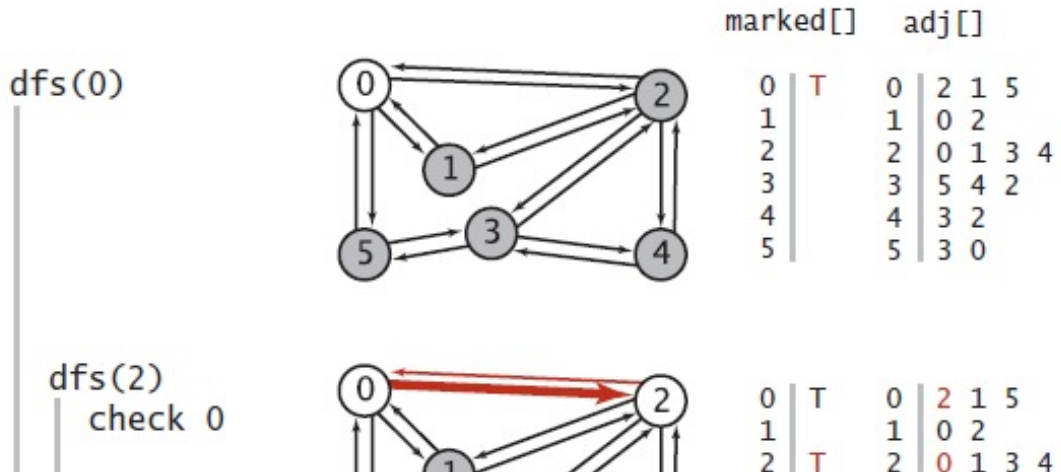
深度优先搜索算法模拟迷宫探索。在实际的图处理算法中，我们通常将图的表示和图的处理逻辑分开来。所以算法的整体设计模式如下：

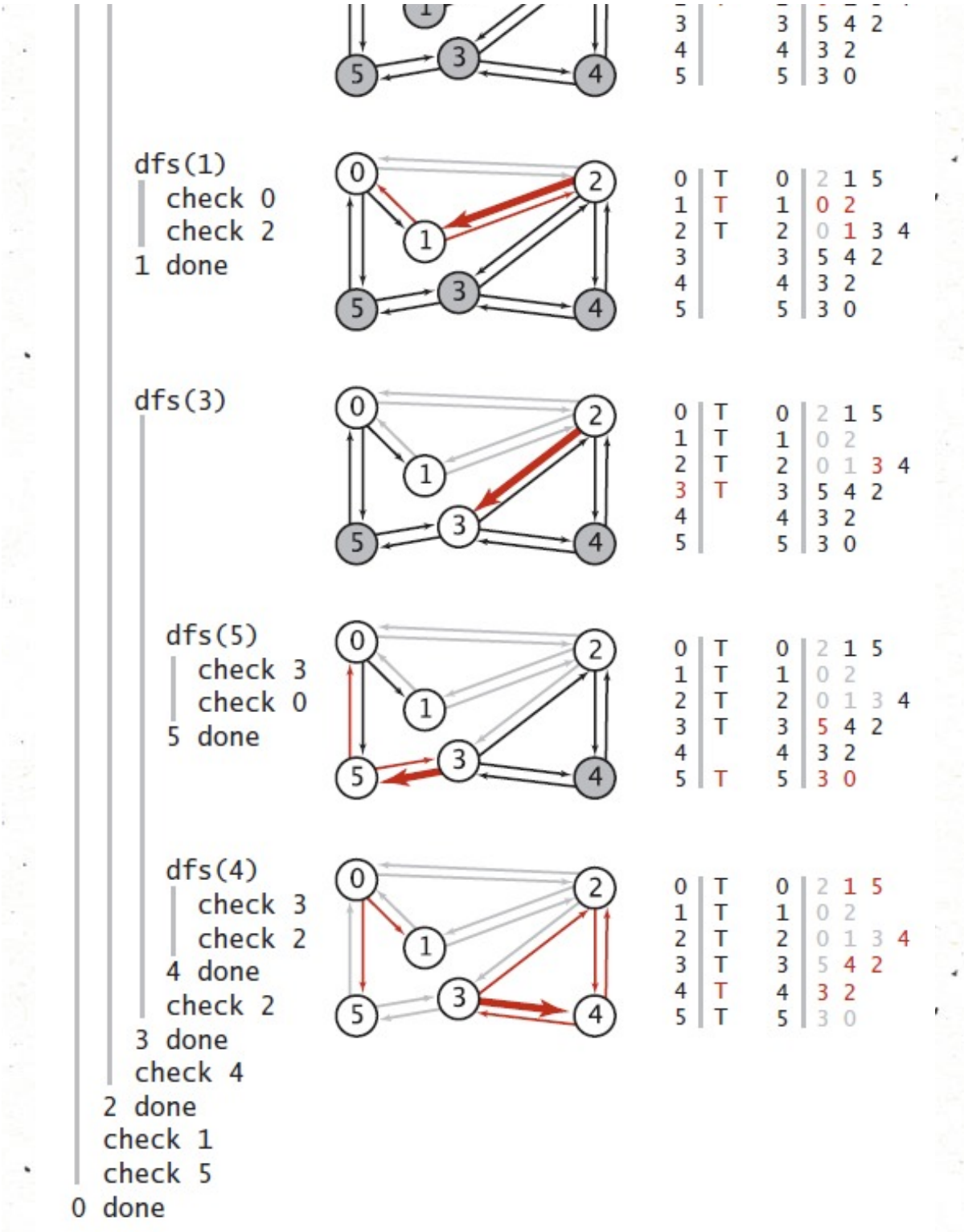
- 创建一个Graph对象
- 将Graph对象传给图算法处理对象，如一个Paths对象
- 然后查询处理后的结果来获取信息

下面是深度优先的基本代码，我们可以看到，递归调用dfs方法，在调用之前判断该节点是否已经被访问过。

```
public class DepthFirstSearch {
    private bool[] marked;//记录顶点是否被标记
    private int count;//记录查找次数

    private DepthFirstSearch(Graph g, int v) {
        marked = new bool[g.GetVerticals()];
        dfs(g, v);
    }

    private void dfs(Graph g, int v) {
        marked[v] = true;
        count++;
        foreach (int vertical in g.GetAdjacency(v)) {
            if (!marked[vertical])
                dfs(g,vertical);
        }
    }

    public bool IsMarked(int vertical) {
        return marked[vertical];
    }

    public int Count() {
        return count;
    }
}
```

试验一个算法最简单的办法是找一个简单的例子来实现。





Trace of depth-first search to find vertices connected to 0

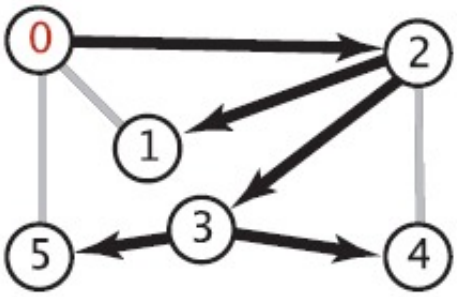
深度优先路径查询

有了这个基础，我们可以实现基于深度优先的路径查询，要实现路径查询，我

们必须定义一个变量来记录所探索到的路径。 所以在上面的基础上定义一个edgesTo变量来后向记录所有到s的顶点的记录，和仅记录从当前节点到起始节点不同，我们记录图中的**每一个节点到开始节点的路径**。为了完成这一日任务，通过设置edgesTo[w]=v，我们记录从v到w的边，换句话说，v-w是做后一条从s到达w的边。 edgesTo[]其实是一个指向其父节点的树。

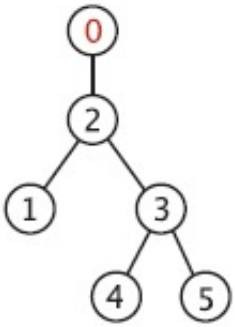
```
public class DepthFirstPaths {
    private bool[] marked; //记录是否被dfs访问过
    private int[] edgesTo; //记录最后一个到当前节点的顶点
    private int s; //搜索的起始点

    public DepthFirstPaths(Graph g, int s) {
        marked = new bool[g.GetVerticals()];
        edgesTo = new int[g.GetVerticals()];
        this.s = s;
        dfs(g, s);
    }

    private void dfs(Graph g, int v) {
        marked[v] = true;
        foreach (int w in g.GetAdjacency(v)) {
            if (!marked[w]) {
                edgesTo[w] = v;
                dfs(g,w);
            }
        }
    }

    public bool HasPathTo(int v) {
        return marked[v];
    }

    public Stack<int> PathTo(int v) {

        if (!HasPathTo(v)) return null;
        Stack<int> path = new Stack<int>();

        for (int x = v; x!=s; x=edgesTo[x]) {
```

```
            path.Push(x);
        }
        path.Push(s);
        return path;
    }
}
```



edgeTo[]

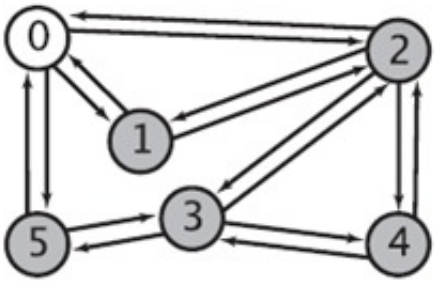
0	
1	2
2	0
3	2
4	3
5	3



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

上图中是黑色线条表示 深度优先搜索中，所有定点到原点0的路径， 他是通过edgeTo[]这个变量记录的，可以从右边可以看出，他其实是一颗树，树根即是原点，每个子节点到树根的路径即是从原点到该子节点的路径。 下图是深度优先搜索算法的一个简单例子的追踪。

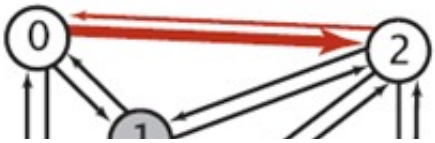
dfs(0)



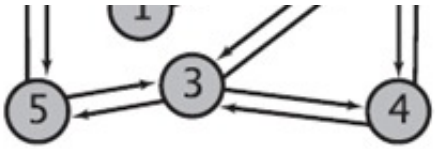
edgeTo[]

0
1
2
3
4
5

dfs(2)
check 0

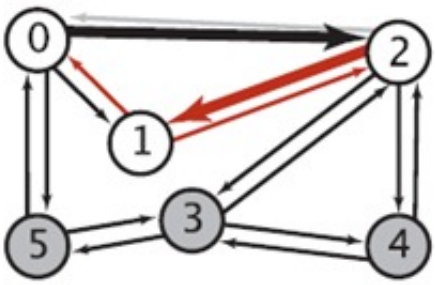


0	
1	
2	0



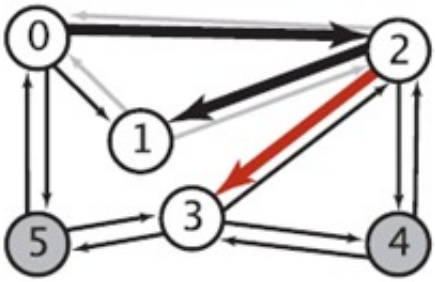
3	
4	
5	

```
dfs(1)
  check 0
  check 2
1 done
```



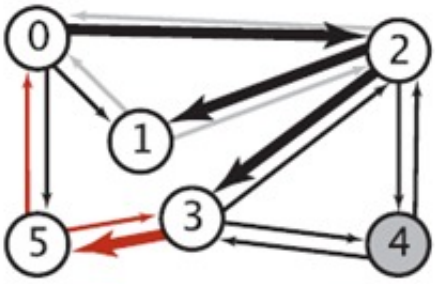
0	
1	2
2	0
3	
4	
5	

```
dfs(3)
```



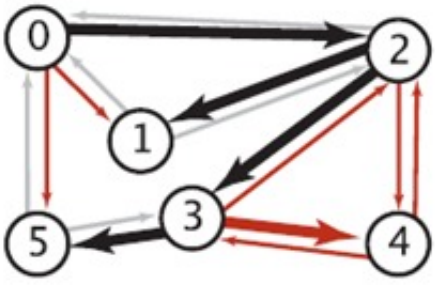
0	
1	2
2	0
3	2
4	
5	

```
  dfs(5)
    check 3
    check 0
  5 done
```



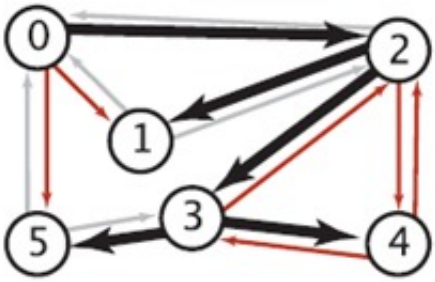
0	
1	2
2	0
3	2
4	
5	3

```
  dfs(4)
    check 3
    check 2
  4 done
  check 2
3 done
check 4
2 done
check 1
```



0	
1	2
2	0
3	2
4	3
5	3

```
check 5
0 done
```



0	
1	2
2	0
3	2
4	3
5	3

Trace of depth-first search to find all paths from 0

广度优先算法

通常我们更关注的是一类单源最短路径的问题，那就是给定一个图和一个源S，是否存在一条从s到给定定点v的路径，如果存在，找出最短的那条(这里最短定义为边的条数最小) 深度优先算法是将未被访问的节点放到一个堆中(stack)，虽然在上面的代码中没有明确在代码中写stack，但是 递归 间接的利用递归堆实现了这一原理。 和深度优先算法不同， 广度优先是将所有未被访问的节点放到了队列中。其主要原理是：

- 将 s放到FIFO中，并且将s标记为已访问
- 重复直到队列为空

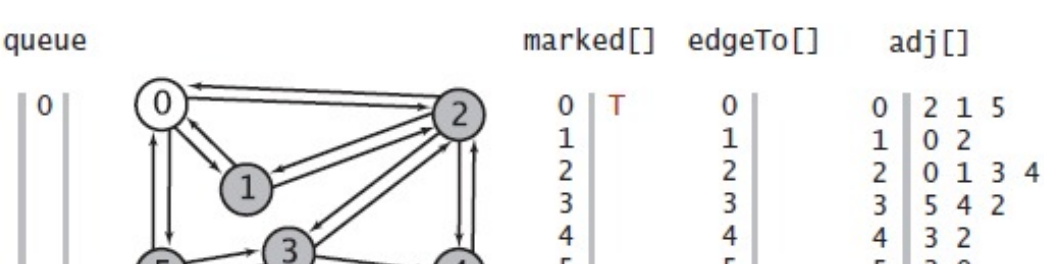
1. 移除最近最近添加的顶点v
2. 将v未被访问的节点添加到队列中
3. 标记他们为已经访问

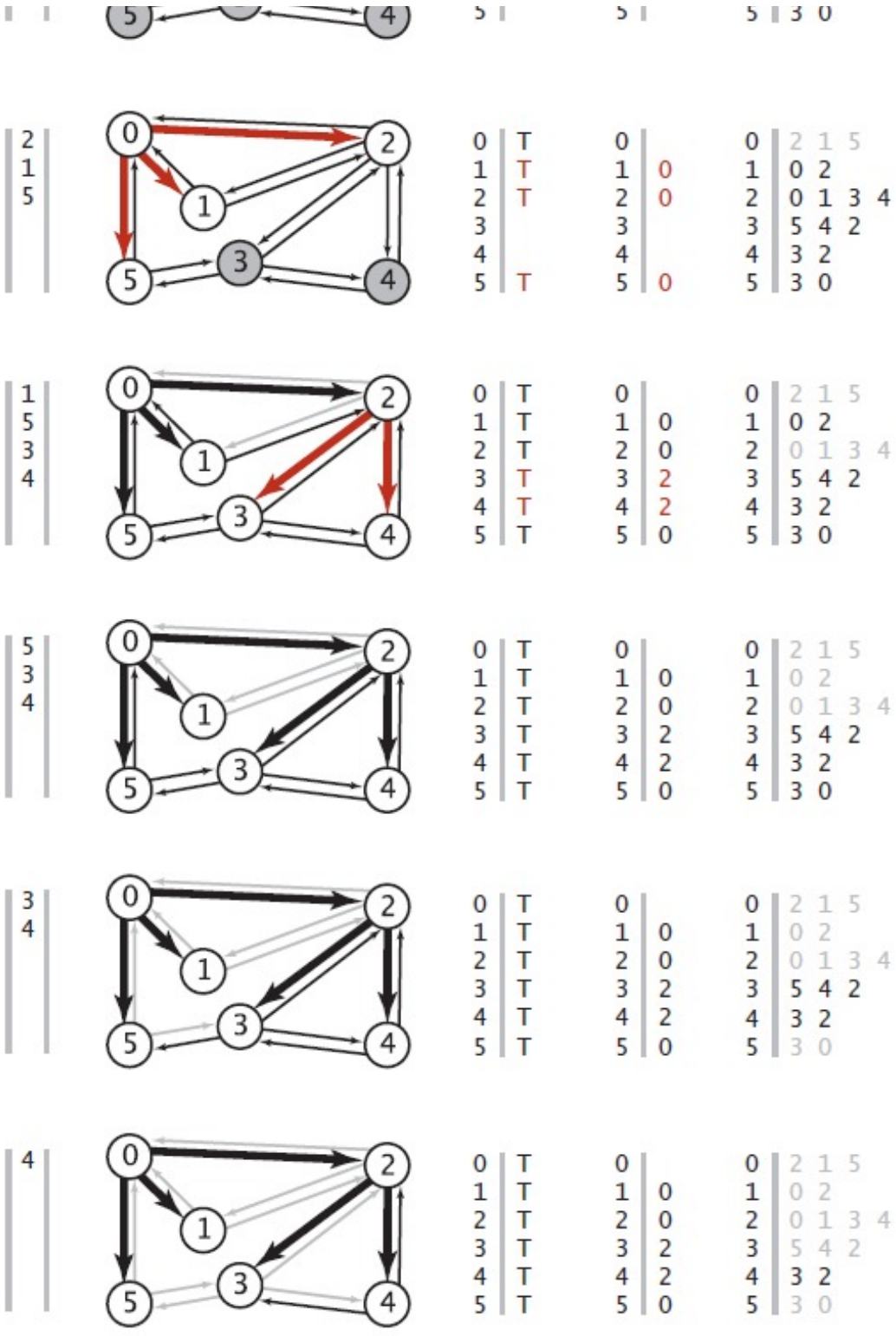
广度优先是以距离递增的方式来搜索路径的。

```
class BreadthFirstSearch {
    private bool[] marked;
    private int[] edgeTo;
    private int sourceVetical;//Source vertical

    public BreadthFirstSearch(Graph g, int s) {
        marked=new bool[g.GetVerticals()];
        edgeTo=new int[g.GetVerticals()];
        this.sourceVetical = s;
        bfs(g, s);
    }
```

```
    private void bfs(Graph g, int s) {
        Queue<int> queue = new Queue<int>();
        marked[s] = true;
        queue.Enqueue(s);
        while (queue.Count()!=0) {
            int v = queue.Dequeue();
            foreach (int w in g.GetAdjacency(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    marked[w] = true;
                    queue.Enqueue(w);
                }
            }
        }
    }

    public bool HasPathTo(int v) {
        return marked[v];
    }

    public Stack<int> PathTo(int v) {
        if (!HasPathTo(v)) return null;

        Stack<int> path = new Stack<int>();
        for (int x = v; x!=sourceVetical; x=edgeTo[x]) {
            path.Push(x);
        }
        path.Push(sourceVetical);
        return path;
    }
}
```

广度优先算法的搜索步骤如下：

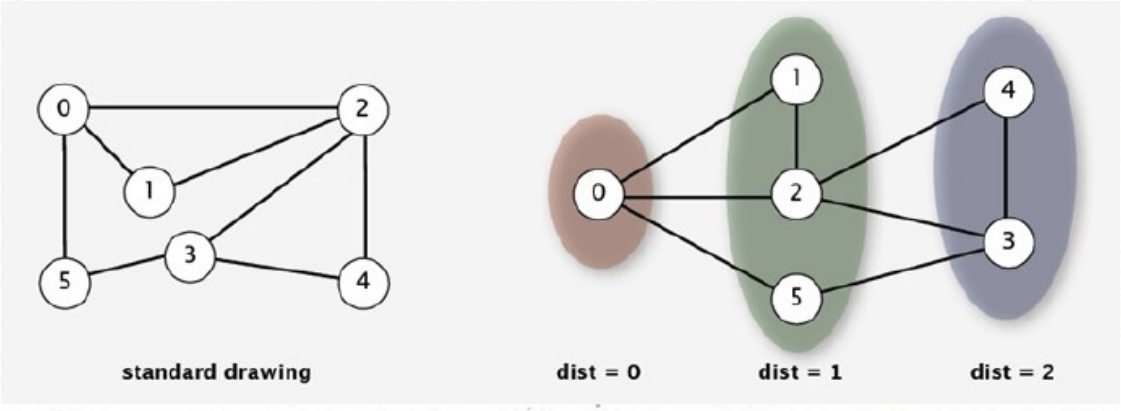




Trace of breadth-first search to find all paths from 0

广度优先搜索首先是在距离起始点为1的范围内的所有邻接点中查找有没有到达目标结点的对象，如果没有，继续前进在距离起始点为2的范围内查找，依次向

前推进。



总结

本文简要介绍了无向图中的深度优先和广度优先算法，这两种算法时图处理算法中的最基础算法，也是后续更复杂算法的基础。其中图的表示，图算法与表示的分离这种思想在后续的算法介绍中会一直沿用，下文将讲解无向图中深度优先和广度优先的应用，以及利用这两种基本算法解决实际问题的应用。