



# 数据结构与算法（ Python ）-07/图及算法

陈斌 gischen@pku.edu.cn 北京大学地球与空间科学学院

# 目录

- › 本章目标
- › 图抽象数据类型及实现
- › Word Ladder词梯问题
- › 骑士周游问题
- › 拓扑排序和强连通分支
- › 最短路径问题
- › Prim最小生成树算法



# 本章目标

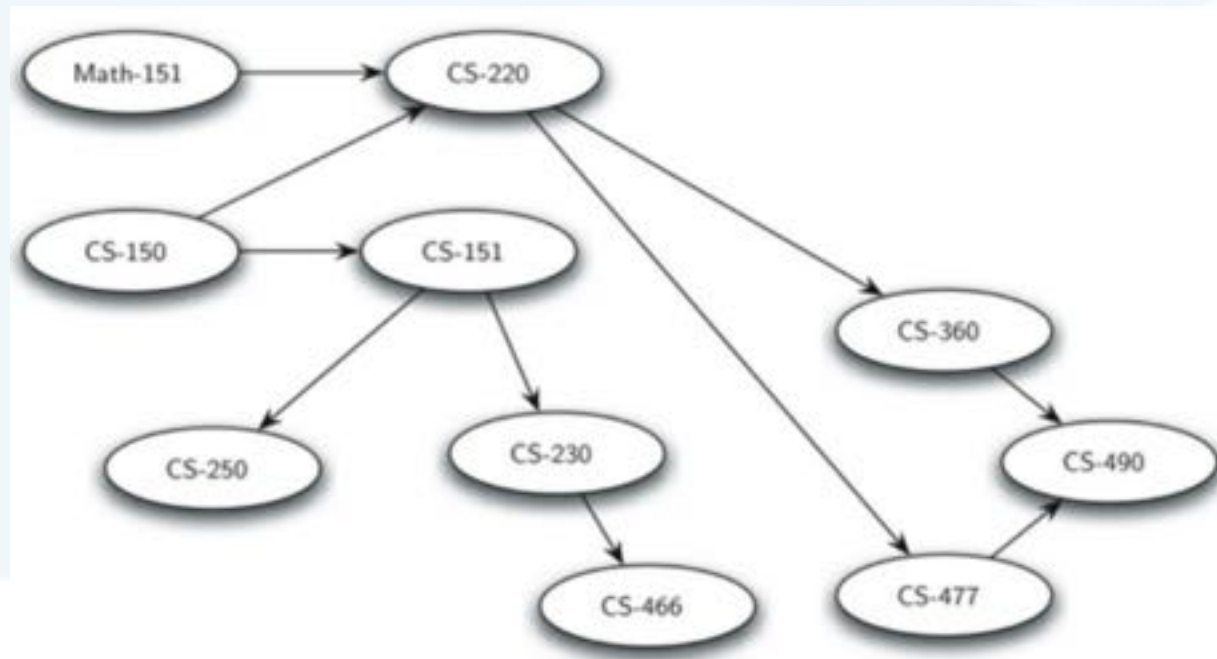
- › 理解图的概念及使用
- › 通过多种方法实现图抽象数据类型
- › 图应用于解决不同领域的问题

# 图Graph的概念

- › 就像“羊”在英文中并不是一个单独的词
- › 中文的“图画”在英文中有很多对应的单词，其意义大不相同
  - painting: 用画刷画的油画
  - drawing: 用硬笔画的素描/线条画
  - picture: 真实形象所反映的画，如照片等，如take picture
  - image: 由印象而来的画，遥感影像叫做image，因是经过传感器印象而来
  - graph: 重在由一些基本元素构造而来的图，如点、线段等
  - figure: 轮廓图的意思，某个侧面的轮廓，所以有figure out的说法
  - diagram: 抽象的概念关系图，如电路图、海洋环流图、类层次图
  - chart: 由数字统计得来的柱状图、饼图、折线图
  - map: 地图；plot: 地图上的一小块

# 图Graph的概念

- › **图Graph是比树更为一般的结构，也是由节点和边构成**  
实际上树是一种具有特殊性质的图
- › **图可以用来表示现实世界中很多事物**  
道路交通系统、航班线路、互联网连接、或者是大学中课程的先修次序



# 图Graph的概念

- › **一旦我们对图相关问题进行了准确的描述，就可以采用处理图的标准算法来解决那些看起来很艰深的问题**

对于人来说，人脑的识别模式能够轻而易举地判断地图中不同地点的相互关联，但计算机并没有这样的能力；

但如果用图来表示地图，就可以解决很多对地图很熟悉甚至专业的人才能解决的问题，甚至能超越；

互联网是由成千上万的计算机所连接起来的复杂网络，也可以通过图算法来确定计算机之间达成通讯的最短、最快或者最有效的路径。

大学课程之间的先修依赖关系，也适合用图来表示

# 北京公共交通

- 北京地铁共有18条运营线路，不重复计算换乘车站则为268座车站，总长约527千米。
- 北京公交系统有1020条运营线路，公交站点近2000个。



## 老人建义务指路队 14年坚持为路人免费指路

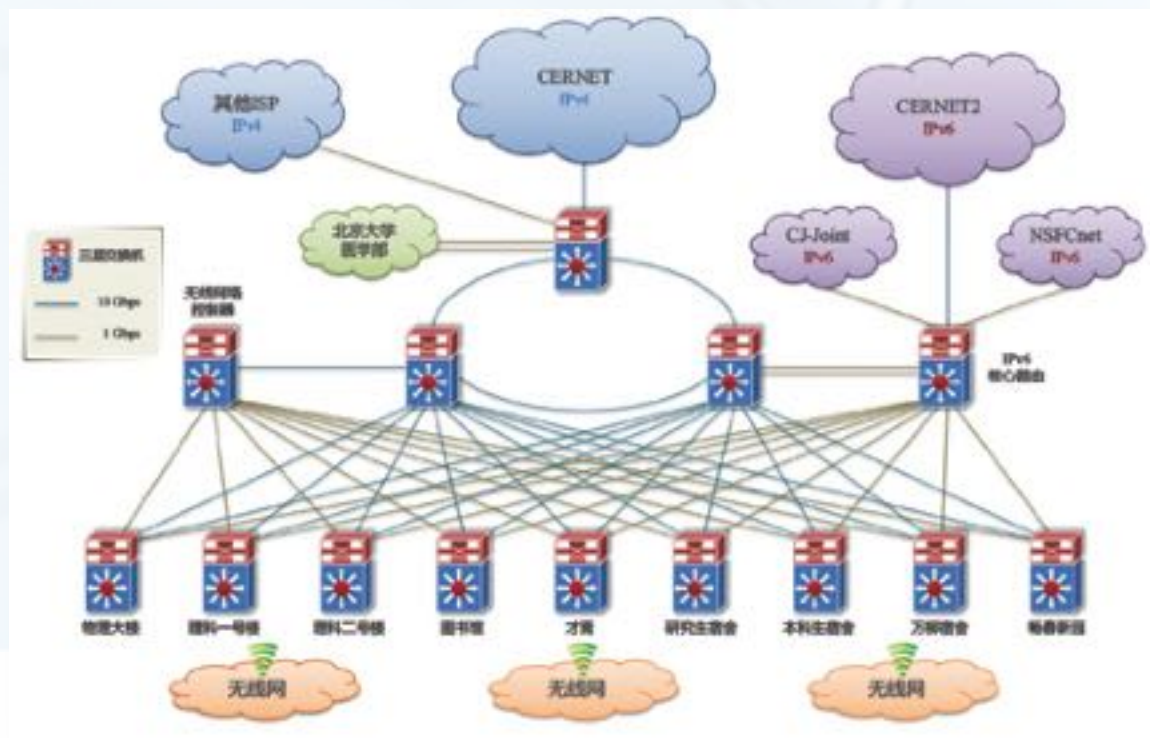
03:59:41 来源：北京青年报 作者：王薇

手机看新闻 | 保存到博客



# 互联网

- › **北京大学校园网目前已经具有近10万信息点**  
通过层层交换机、路由器连接在一起，路由器之间又相互连接
- › **互联网中ipv4的近40亿地址已接近枯竭**  
一张几十亿个信息点的巨型网络
- › **提供内容的Web站点已突破10亿个**  
由超链接相互连接的网页更是不计其数  
Google每天处理的数据量约10PB
- › **在天文数字规模的网络面前**
- › **人脑已经无法处理**



# 社交网络：六度分隔理论

› 世界上任何两个人之间通过最多6个人即可建立联系

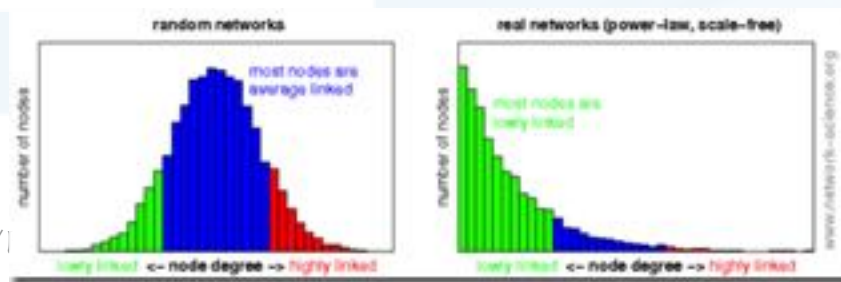
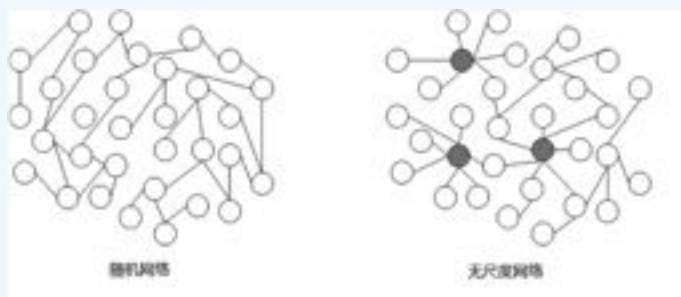
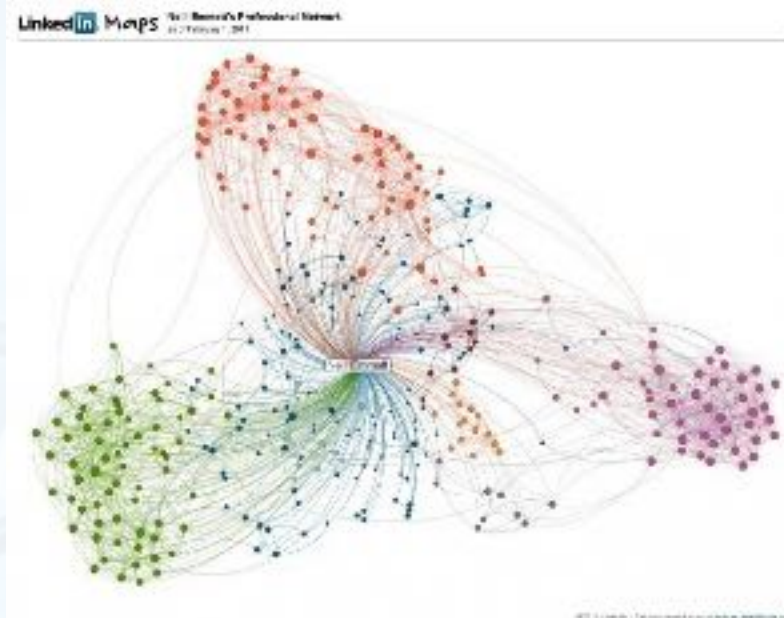
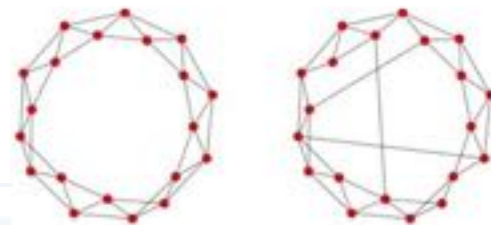
互联网社交网络的兴起将每个人联系到一起

› 在社会中有20%擅长交往的人，建立了80%的连接

区别于随机网络，保证了六度分隔的成立

引出了无尺度网络的研究

现实中的复杂网络多属于无尺度网络



# 术语表

## › 顶点Vertex

也称“节点Node”，是图的基本组成部分，顶点具有名称标识Key，也可以携带数据项payload。

## › 边Edge

也称“弧Arc”，是图的另一个基本组成部分，作为2个顶点之间关系的表示，边连接两个顶点；边可以是单向one-way或者双向two-way的，如果一个图中的所有边都是单向的，就称这个图为“有向图directed graph/digraph”。

## › 权重Weight

为了表达从一个顶点到另一个顶点的“代价”，可以给边赋权；例如公交网络中两个站点之间的“距离”、“通行时间”和“票价”都可以作为权重。

# 图的定义

## 一个图G可以定义为 $G=(V, E)$

其中V是顶点的集合

E是边的集合，E中的每条边 $e=(v, w)$ ，v和w都是V中的顶点；

如果是赋权图，则可以在e中添加第三个权重分量

子图subgraph: V和E的子集

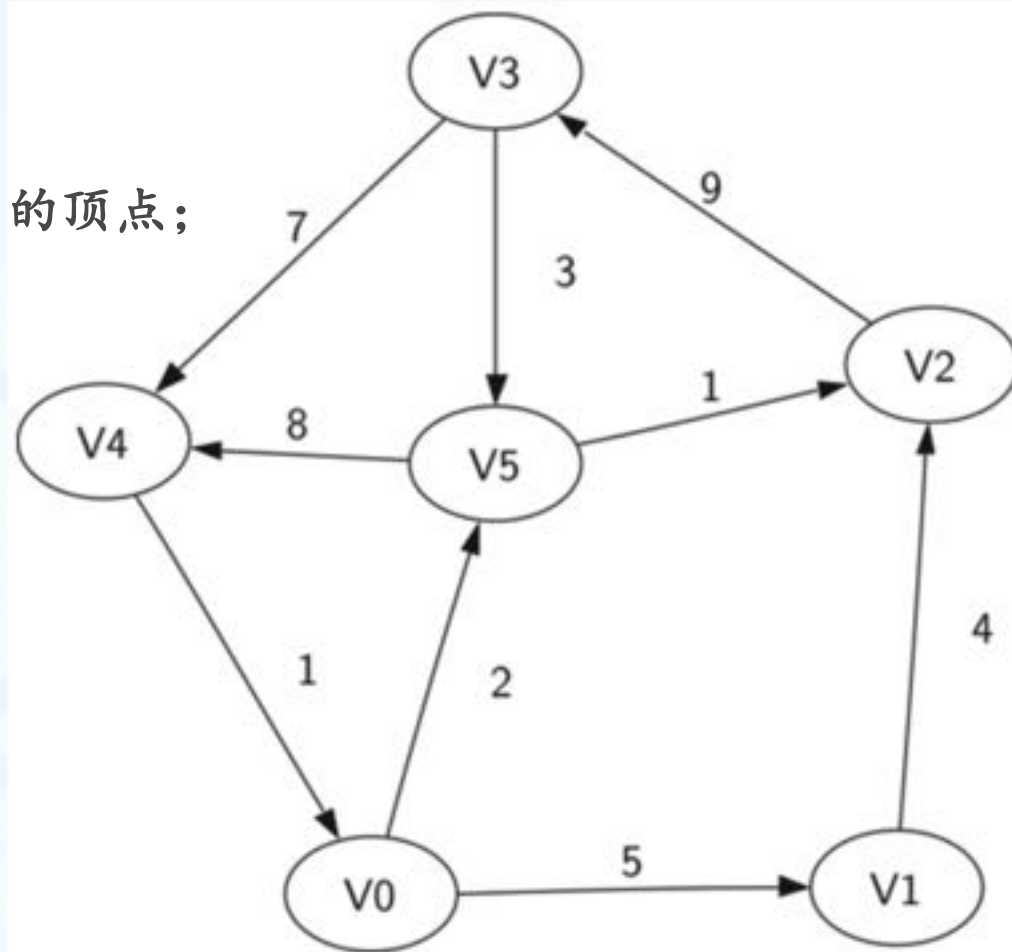
## 赋权图的例子：6个顶点及9条边

有向图

权重为整数

$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$E = \left\{ \begin{array}{l} (v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), \\ (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1) \end{array} \right\}$$



# 术语表

## › 路径Path

图中的路径，是由边依次连接起来的顶点序列；

$P=(w_1, w_2, \dots, w_n)$ ，其中对于所有 $1 \leq i \leq n-1$ ， $(w_i, w_{i+1})$ 属于 $E$ ；

无权路径的长度为边的数量，等于 $n-1$ ；带权路径的长度为所有边权重的和；

如，前页中的一条路径 $(v_3, v_4, v_0, v_1)$ ，其边为 $\{(v_3, v_4, 7), (v_4, v_0, 1), (v_0, v_1, 5)\}$

## › 圈Cycle

圈是首尾顶点相同的路径，如前页中 $(v_5, v_2, v_3, v_5)$ 是一个圈

如果一个图中不存在任何圈，则称作“无圈图acyclic graph”

无圈的有向图称作“有向无圈图directed acyclic graph: DAG”

后面我们可以看到如果一个问题能表示成DAG，就可以用图算法很好地解决

## › 思考：树是一种什么性质的图？是DAG么？

# 抽象数据类型：ADT Graph

## › 抽象数据类型ADT Graph定义如下：

`Graph()`：创建一个空的图；

`addVertex(vert)`：将一个顶点Vertex对象加入图中

`addEdge(fromVert, toVert)`：添加一条有向边

`addEdge(fromVert, toVert, weight)`：添加一条带权的有向边

`getVertex(vertKey)`：查找名称为vertKey的顶点

`getVertices()`：返回图中所有顶点列表

`in`：按照 `vert in graph` 的语句形式，返回顶点是否存在图中 True/False

## › ADT Graph的实现方法有两种主要形式：

邻接矩阵 `adjacency matrix` 和 邻接表 `adjacency list`

两种方法各有优劣，需要在不同的应用中加以选择

# 邻接矩阵Adjacency Matrix

- › 对图的最直观实现方法是采用二维矩阵
- › 矩阵的每行和每列都代表图中的顶点，如果两个顶点之间有边相连，则在相应行列值的矩阵分量中加以体现
  - 无权边则将矩阵分量标注为1，或者0
  - 带权边则将权重保存为矩阵分量值
- › 邻接矩阵实现法的优点是简单
  - 可以很容易得到顶点是如何相连
- › 但如果图中的边数很少则效率低下
  - 成为“稀疏sparse”矩阵
  - 大多数问题所对应的图都是稀疏的
  - 边远远少于 $|V|^2$ 这个量级

|    | V0 | V1 | V2 | V3 | V4 | V5 |
|----|----|----|----|----|----|----|
| V0 |    | 5  |    |    |    | 2  |
| V1 |    |    | 4  |    |    |    |
| V2 |    |    |    | 9  |    |    |
| V3 |    |    |    |    | 7  | 3  |
| V4 | 1  |    |    |    |    |    |
| V5 |    |    | 1  |    | 8  |    |

# 邻接列表Adjacency List

## › 邻接列表adjacency list可以成为稀疏图的更高效实现方案

维护一个包含所有顶点的主列表 (master list)

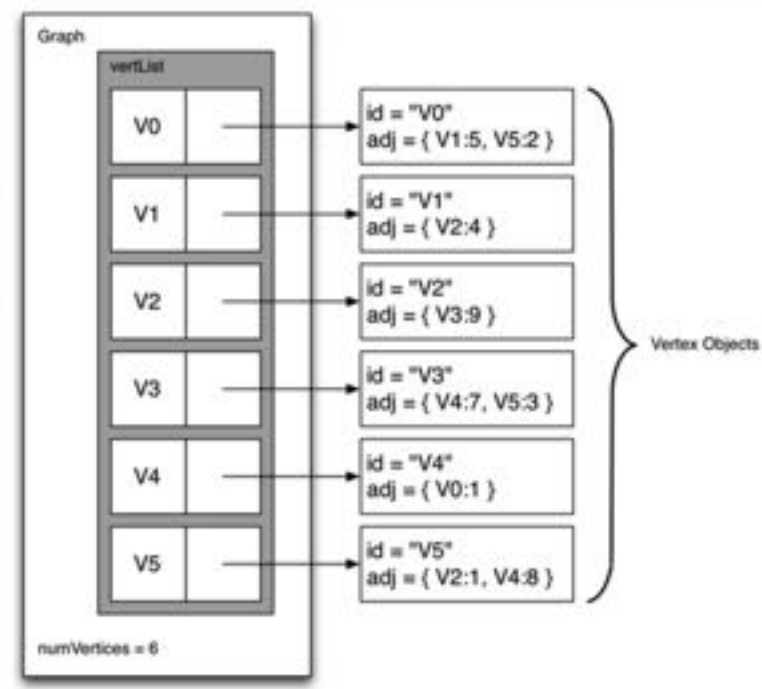
主列表中的每个顶点，再关联一个与自身有边连接的所有顶点的列表

- 在Python实现的Vertex类中，可以采用字典来保存顶点列表
- 字典中的key对应顶点标识，而value则可以保存顶点连接边的权重

## › 邻接列表法的存储空间紧凑高效

## › 很容易获得顶点所连接的所有顶点

## › 以及连接边的信息



# ADT Graph的实现

## › 包括两个类Vertex和Graph

Graph保存了包含所有顶点的主表master list

Vertex则包含了顶点信息，以及顶点连接边的信息

nbr是顶点对象的key

```
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' \
            + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]
```

# ADT Graph的实现

新加顶点

通过key查找顶点

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList
```

# ADT Graph的实现

不存在的顶点先添加

调用起始顶点的方法  
添加邻接边

```
def __contains__(self,n):  
    return n in self.vertList  
  
def addEdge(self,f,t,cost=0):  
    if f not in self.vertList:  
        nv = self.addVertex(f)  
    if t not in self.vertList:  
        nv = self.addVertex(t)  
    self.vertList[f].addNeighbor(self.vertList[t], cost)  
  
def getVertices(self):  
    return self.vertList.keys()  
  
def __iter__(self):  
    return iter(self.vertList.values())
```

# ADT Graph的实现：实例

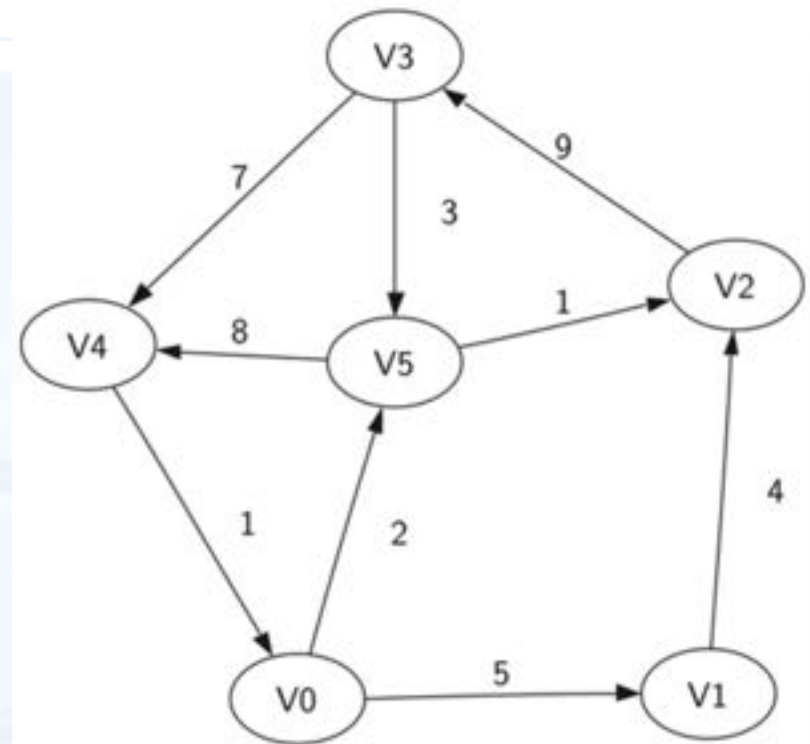
```
>>> g= Graph()
>>> for i in range(6):
    g.addVertex(i)

0 connectedTo: []
1 connectedTo: []
2 connectedTo: []
3 connectedTo: []
4 connectedTo: []
5 connectedTo: []
>>> print g.vertList
{0: 0 connectedTo: [], 1: 1 connectedTo: [], 2: 2 connectedTo: [], 3: 3 connecte
dTo: [], 4: 4 connectedTo: [], 5: 5 connectedTo: []}
```

# ADT Graph的实现：实例

```
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> for v in g:
    for w in v.getConnections():
        print "(%s, %s)" % (v.getId(), w.getId())
```

```
(0, 5)
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(3, 5)
(4, 0)
(5, 4)
(5, 2)
>>> |
```



# 词梯Word Ladder问题

- › 由“爱丽丝漫游奇境”的作者Lewis Carroll在1878年所发明的单词游戏
- › 从一个单词演变到另一个单词，其中的过程可以经过多个中间单词  
要求是相邻两个单词之间差异只能是1个字母，如FOOL变SAGE：  
FOOL >> POOL >> POLL >> POLE >> PALE >> SALE >> SAGE
- › 我们的目标是找到最短的单词变换序列
- › 采用图来解决这个问题的步骤如下：  
将可能的单词之间的演变关系表达为图  
采用“广度优先搜索 BFS”来找到从开始单词到结束单词之间的有效路径

# 词梯问题：构建单词关系图

## › 首先是如何将大量的单词集放到图中

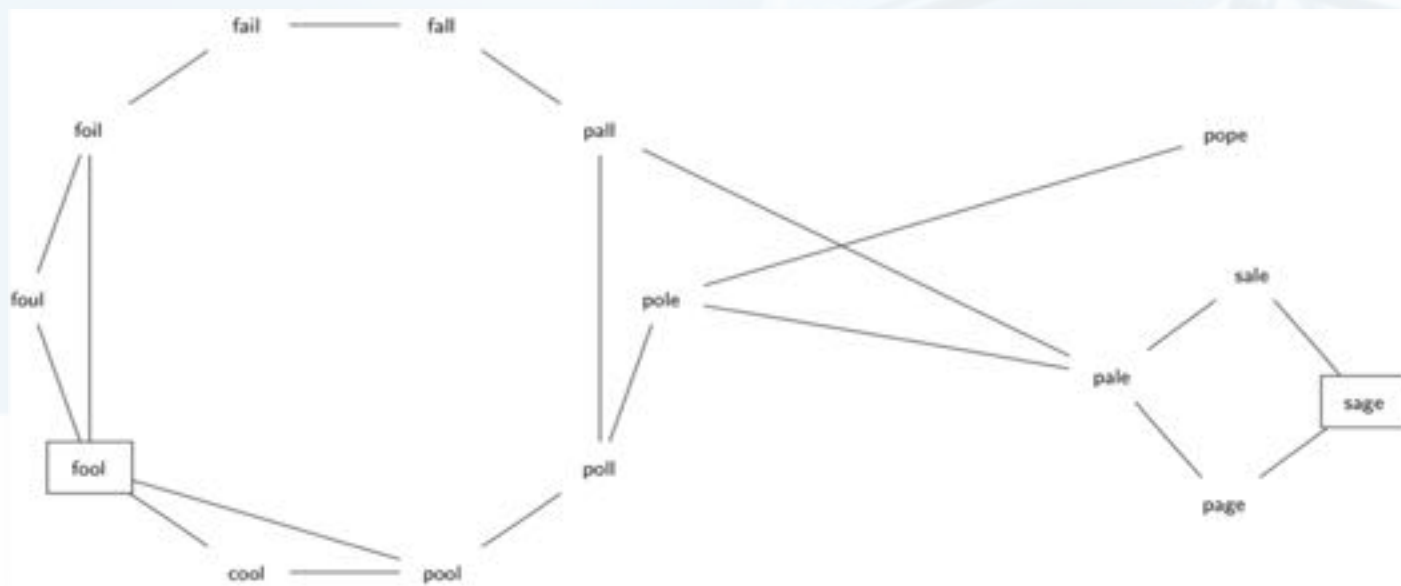
将单词作为顶点的标识Key

如果两个单词之间仅相差1个字母，就在它们之间设一条边

这样，在两个单词之间的任意一条路径，就是词梯问题的一个解

## › 下图是从FOOL到SAGE的词梯解，所用的图是无向图，边没有权重

FOOL到SAGE的每条路径都是一个解



# 词梯问题：构建单词关系图

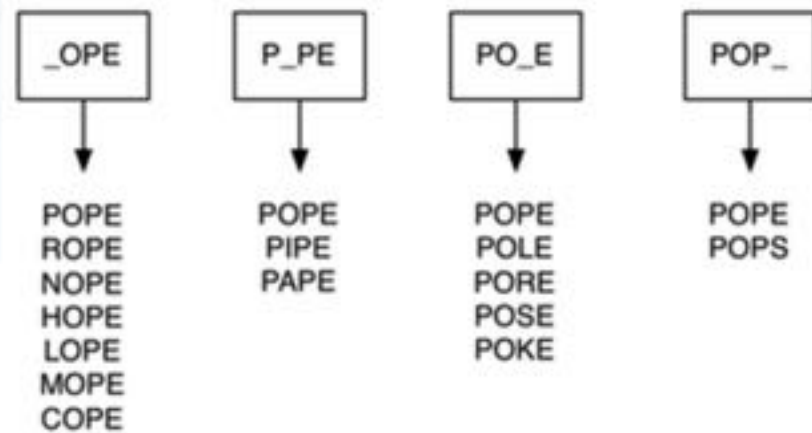
- 单词关系图可以通过不同的算法来构建（以4个字母的单词表为例）

首先是将所有单词作为顶点加入图中，再设法建立顶点之间的边

- 建立边的最直接算法，是对每个顶点（单词），与其它所有单词进行比较，如果相差仅1个字母，则建立一条边

时间复杂度是 $O(n^2)$ ，对于所有4个字母的5110个单词，需要超过2600万次比较

- 改进的算法是创建大量的桶，每个桶可以存放若干单词，桶的标记是去掉1个字母，以通配符“\_”占空的单词，所有匹配标记的单词都放到这个桶里，所有单词就位后，再在同一个桶的单词之间建立边即可。



# 词梯问题：构建单词关系图

› 可以采用Python字典来建立桶

4字母单词可以属于4个桶

同一个桶的单词之间建立边

```
def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile, 'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1, word2)
    return g
```

# 词梯问题：构建单词关系图

- › 样例数据文件包含了5,110个4字母的单词  
可从课程网站下载
- › 如果采用邻接矩阵表示这个单词关系图，则需要2,600万个矩阵单元  
 $5,110 \times 5,110 = 26,112,100$   
而单词关系图总计有53,286条边，仅仅达到矩阵单元数量的0.2%
- › 单词关系图是一个非常稀疏的图
- › 猜想与验证：这个单词关系图会是一个随机网络，还是一个无尺度网络？单词之间的关系会不会遵循六度分隔理论呢？

# 实现广度优先搜索(Breadth First Search)

- › 在单词关系图建立完成以后，需要继续在图中寻找词梯问题的最短序列。
- › 需要用到“广度优先搜索Breadth First Search:BFS”算法对单词关系图进行搜索
- › BFS是搜索图的最简单算法之一，也是其它一些重要的图算法的基础
- › 给定图G，以及开始搜索的起始顶点s

BFS搜索所有从s可到达顶点的边

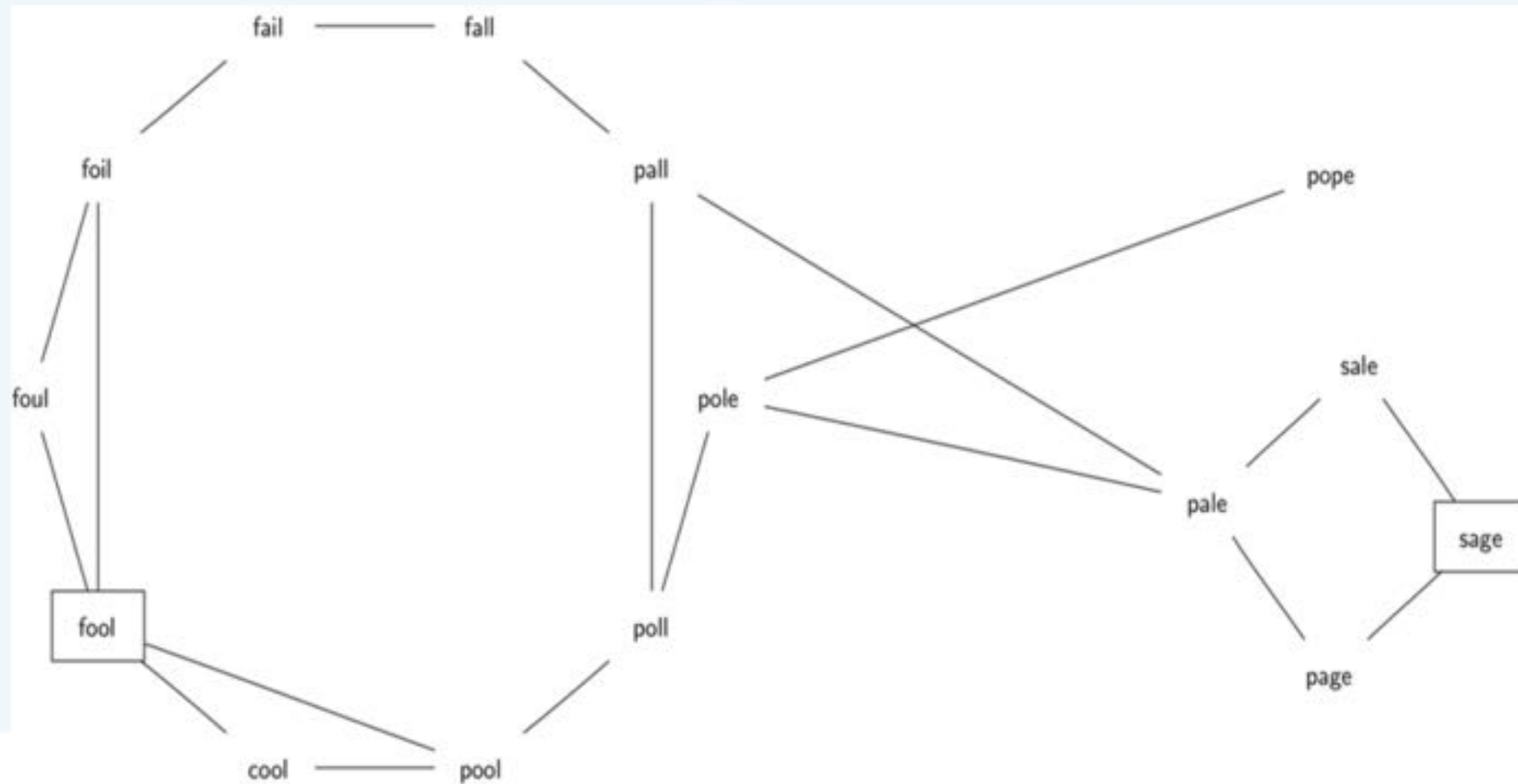
而且在达到更远的距离 $k+1$ 的顶点之前，BFS会找到全部距离为 $k$ 的顶点

可以想象为以s为根，构建一棵树的过程，从顶部向下逐步增加层次

广度优先搜索能保证在增加层次之前，添加了所有兄弟节点到树中

# BFS算法过程

## 我们从FOOL开始搜索



# BFS算法过程

- › **为了跟踪顶点的加入过程，并避免重复顶点，要为顶点增加3个属性**
  - 距离distance：从起始顶点到此顶点的路径长度；
  - 前驱顶点predecessor：可追溯到起始顶点的反向路径；
  - 颜色color：标识了此顶点是尚未发现（白色）、已经发现（灰色）、还是已经完成探索（黑色）
- › **还需要用一个队列Queue来对已发现的顶点进行排列，决定下一个要探索的顶点（队首顶点）**

# BFS算法过程

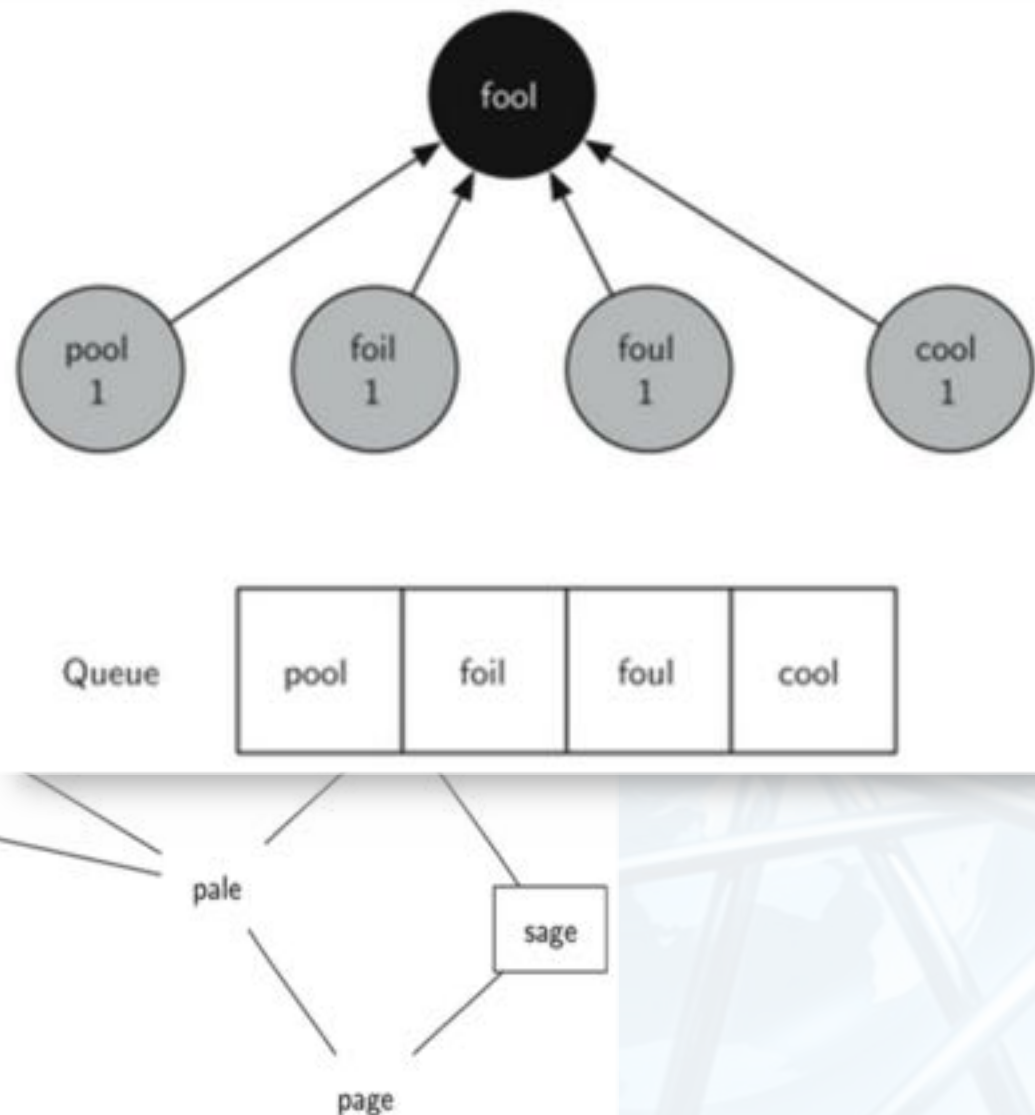
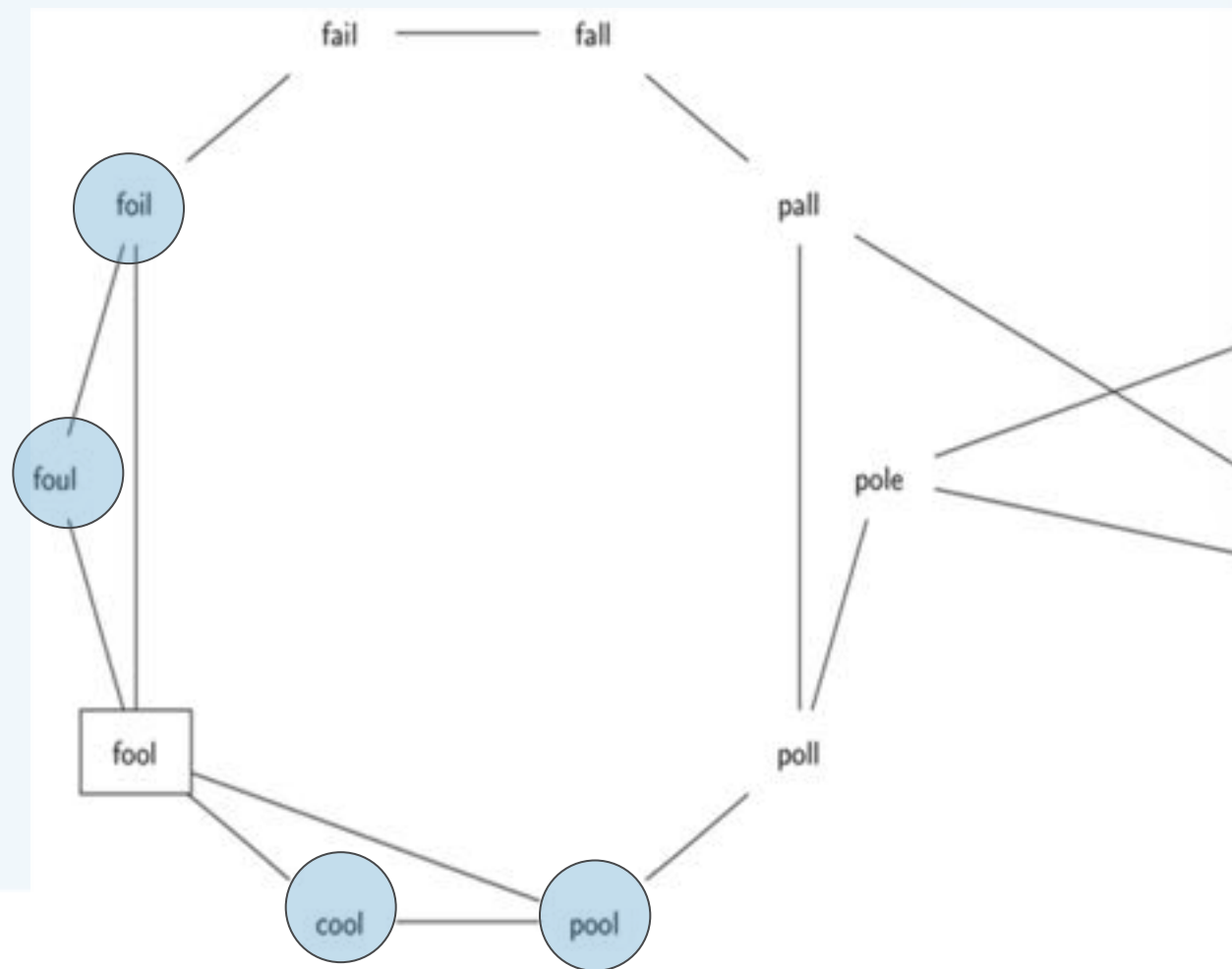
- › 从起始顶点s开始，作为刚发现的顶点，标注为灰色，距离为0，前驱为None，加入队列，接下来是个循环迭代过程：

从队首取出一个顶点作为当前顶点；

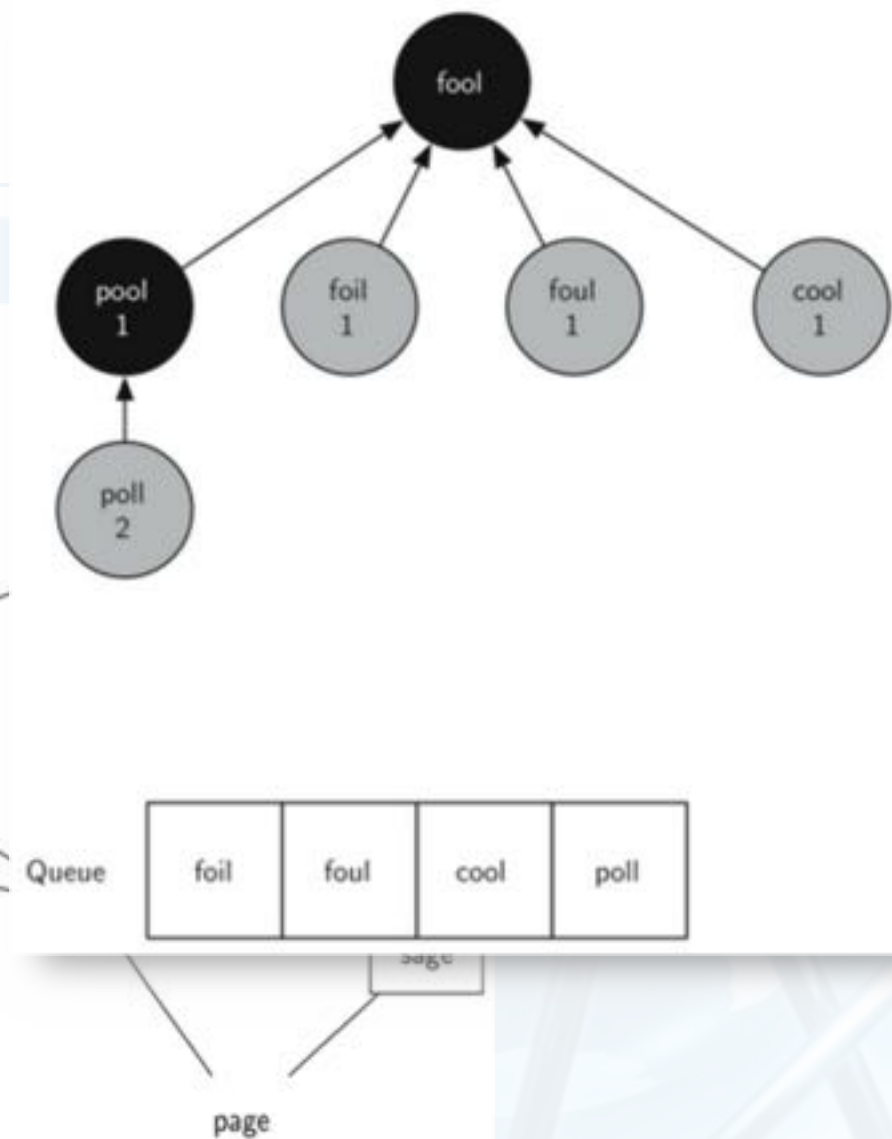
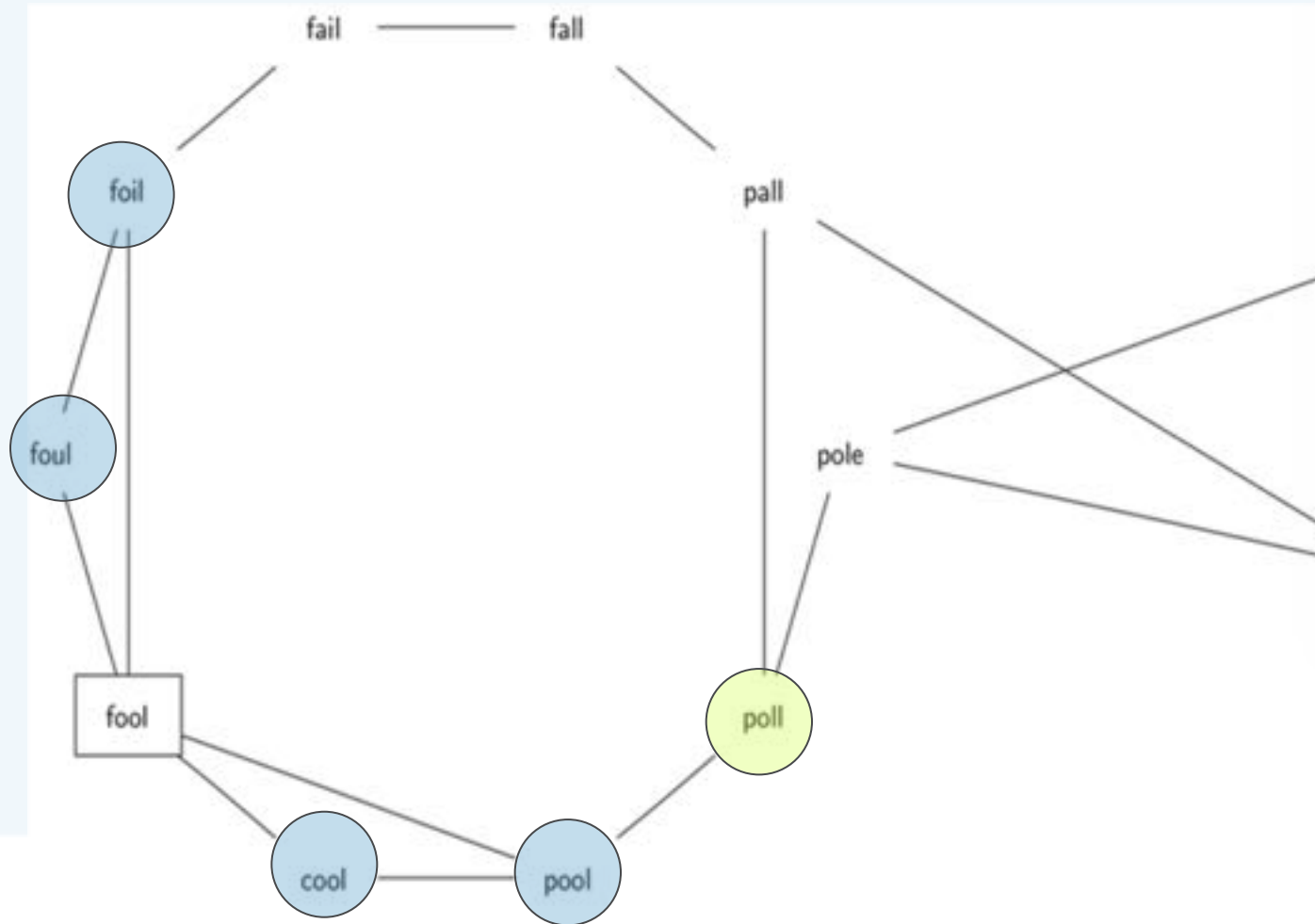
遍历当前顶点的邻接顶点，如果是尚未发现的白色顶点，则将其颜色改为灰色（已发现），距离增加1，前驱顶点为当前顶点，加入到队列中

遍历完成后，将当前顶点设置为黑色（已探索过），循环回到步骤1

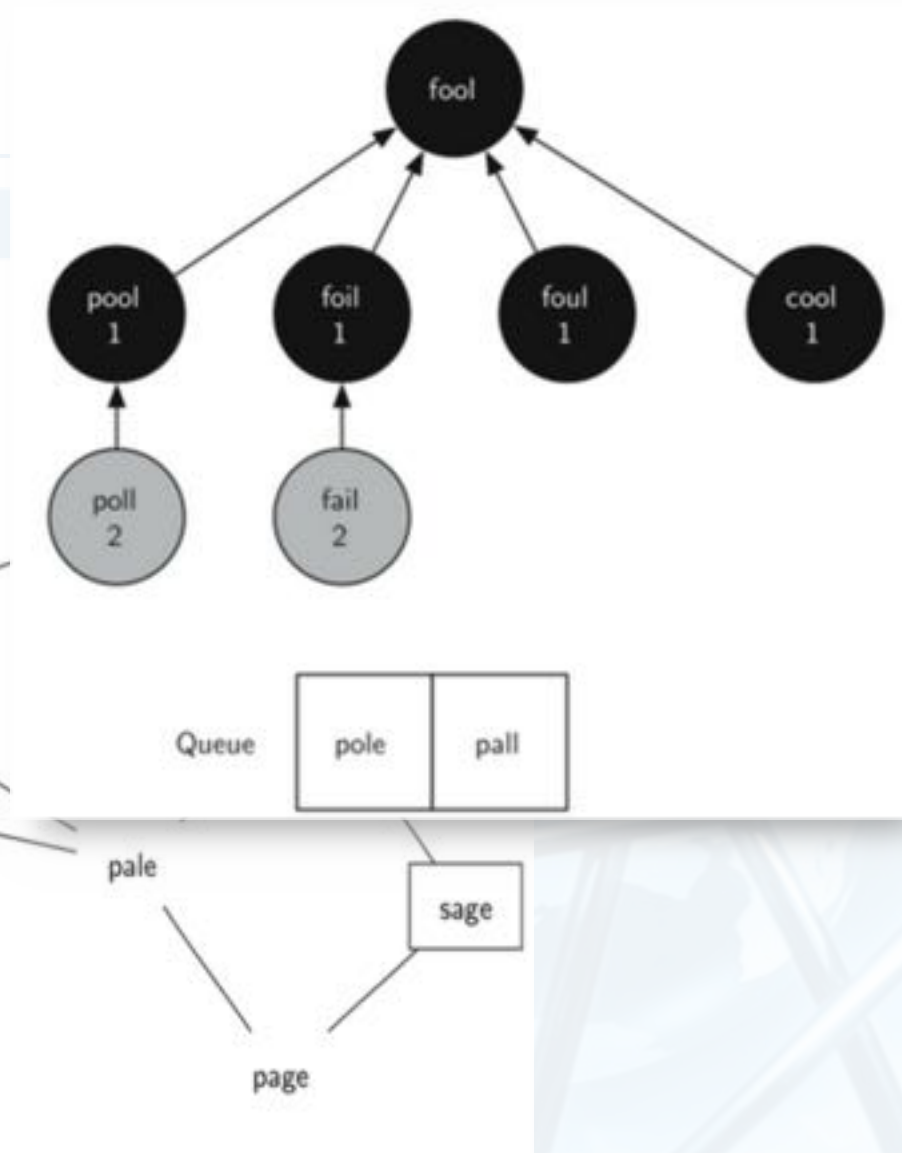
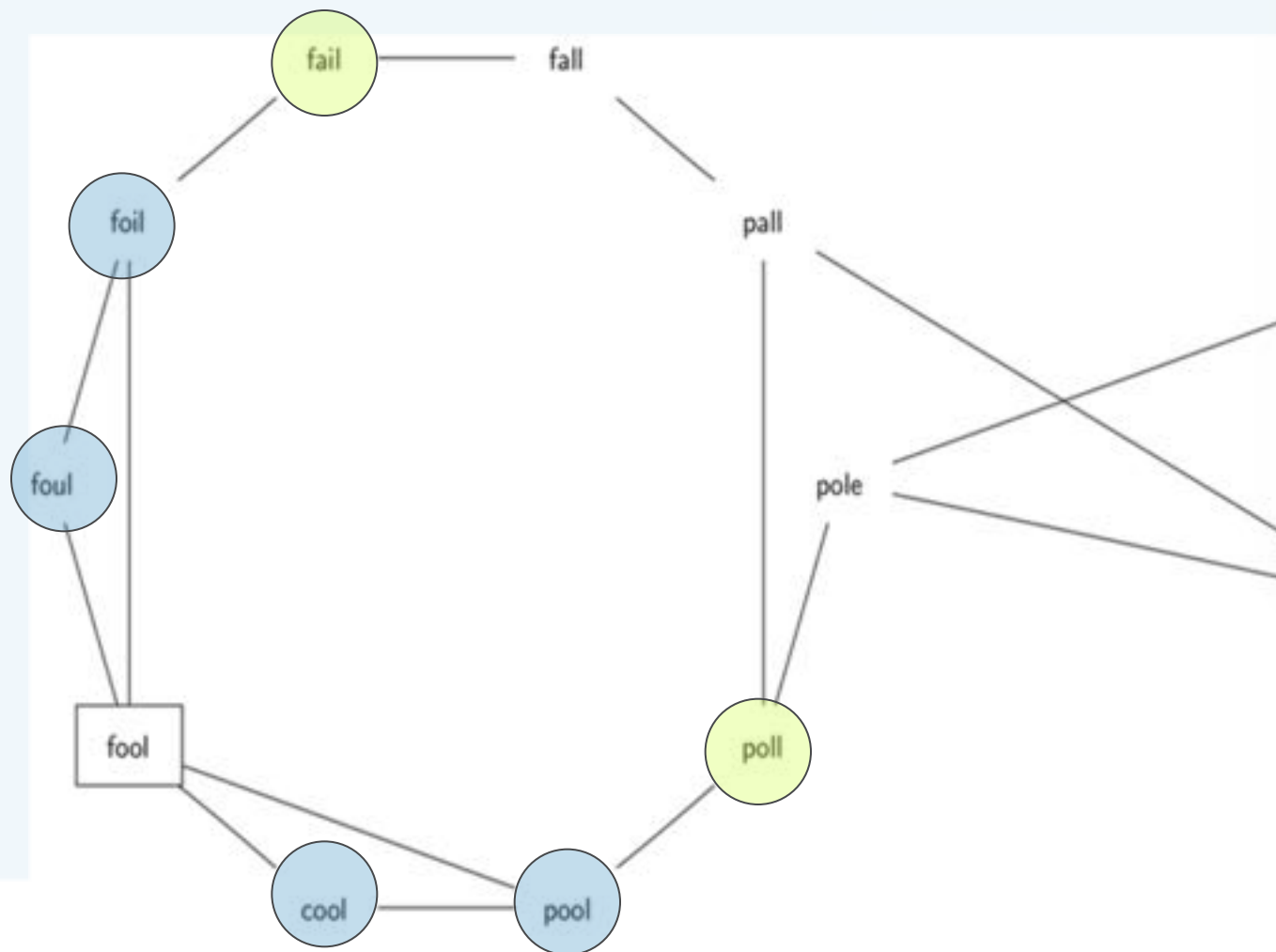
# BFS算法实例运行



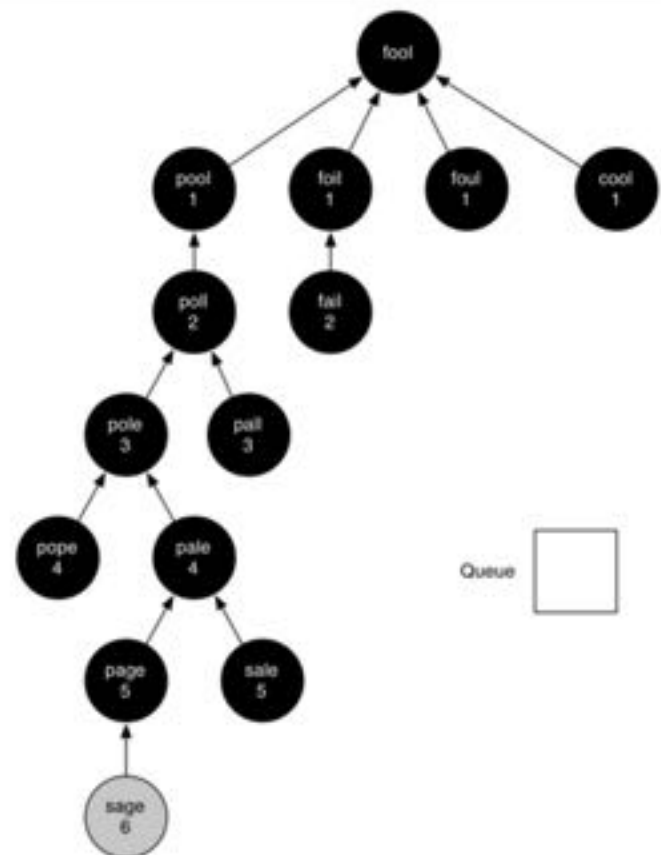
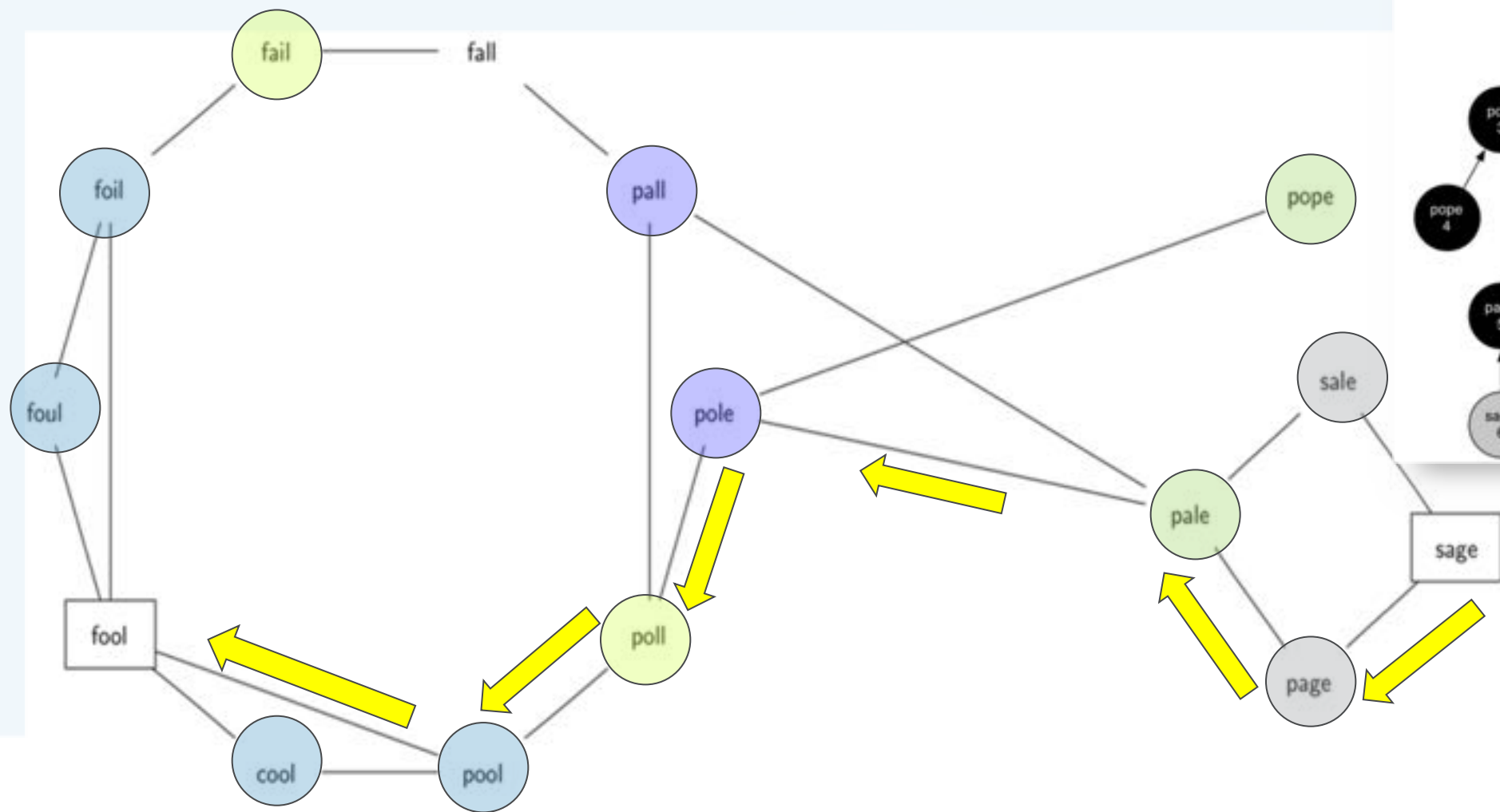
# BFS算法实例运行



# BFS算法实例运行



# BFS算法实例运行



# BFS算法代码

取队首作为当前顶点

遍历当前顶点邻接顶点

当前顶点设黑色

```
def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')
```

# BFS算法代码

- › 在BFS以FOOL为起始顶点，遍历了所有顶点，并为每个顶点着色、赋距离和前驱之后
- › 即可以通过一个回途追溯函数来确定FOOL到任何单词顶点的最短词梯！
- › 思考：BFS能得到
- › 所有的最短词梯解么？
- › 为什么？

```
def traverse(y):  
    x = y  
    while (x.getPred()):  
        print(x.getId())  
        x = x.getPred()  
    print(x.getId())  
  
wordgraph = buildGraph("fourletterwords.txt")  
bfs(wordgraph, wordgraph.getVertex('FOOL'))  
traverse(wordgraph.getVertex('SAGE'))  
#traverse(wordgraph.getVertex('COOL'))
```

# 广度优先搜索算法分析

## › BFS算法主体是两个循环的嵌套：while-for

while循环对图中每个顶点访问一次，所以是 $O(|V|)$

而嵌套在while中的for，由于每条边只有在其起始顶点u出队的时候才会被检查一次，而每个顶点最多出队1次，所以边最多被检查1次，一共是 $O(|E|)$

综合起来BFS的时间复杂度为 $O(|V|+|E|)$

## › 词梯问题还包括两个部分算法

建立BFS树之后，回溯顶点到起始顶点的过程，最多为 $O(|V|)$

创建单词关系图也需要时间，思考：其复杂度为多少？

# 广度优先搜索算法分析

## › 课后练习1

整理词梯问题完整算法 (ADT Graph实现、buildGraph、BFS、traverse)

## › 课后练习2

用嵌套列表作为矩阵来实现ADT Graph

# 骑士周游问题Knight's Tour Problem

- 在一个国际象棋棋盘上，一个棋子“马”（骑士），按照“马走日”的规则，从一个格子出发，要走遍所有棋盘格恰好一次。  
把一个这样的走棋序列称为一次“周游”
- 在 $8 \times 8$ 的国际象棋棋盘上，合格的“周游”数量有 $1.305 \times 10^{35}$ 这么多，走棋过程中失败的周游就更多了
- 采用图搜索算法，是解决骑士周游问题最容易理解和编程的方案之一，解决方案还是分为两步：  
首先将合法走棋次序表示为一个图  
采用图搜索算法搜寻一个长度为（行 $\times$ 列-1）的路径  
路径上包含每个顶点恰一次

|    |    |    |    |    |    |                                                                                       |    |
|----|----|----|----|----|----|---------------------------------------------------------------------------------------|----|
| 35 | 40 | 47 | 44 | 61 | 08 | 15                                                                                    | 12 |
| 46 | 43 | 36 | 41 | 14 | 11 | 62                                                                                    | 09 |
| 39 | 34 | 45 | 48 | 07 | 60 | 13                                                                                    | 16 |
| 50 | 55 | 42 | 37 | 22 | 17 | 10                                                                                    | 63 |
| 33 | 38 | 49 | 54 | 59 | 06 | 23                                                                                    | 18 |
| 56 | 51 | 28 | 31 | 26 | 21 |  | 03 |
| 29 | 32 | 53 | 58 | 05 | 02 | 19                                                                                    | 24 |
| 52 | 57 | 30 | 27 | 20 | 25 | 04                                                                                    | 01 |

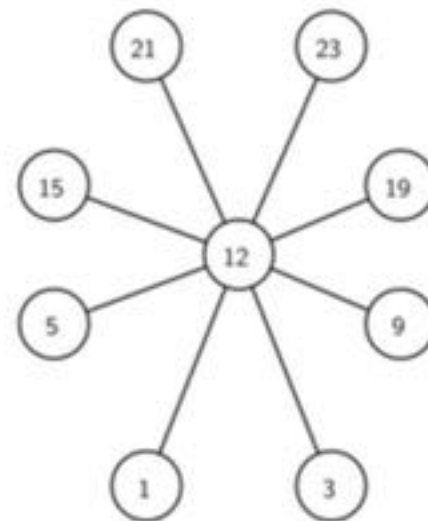
# 构建骑士周游图

## 将棋盘和走棋步骤构建为图的思路，顶点和边

将棋盘格作为顶点

按照“马走日”规则的合法走棋步骤作为顶点之间的连接边

建立每一个棋盘格的所有合法走棋步骤能够到达的棋盘格关系图



遍历每个格

单步合法走棋

添加边及顶点

马走日8个格子

确认不会走出棋盘

```
def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
        for col in range(bdSize):
            nodeId = posToNodeId(row,col,bdSize)
            newPositions = genLegalMoves(row,col,bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0],e[1],bdSize)
                ktGraph.addEdge(nodeId,nid)
    return ktGraph

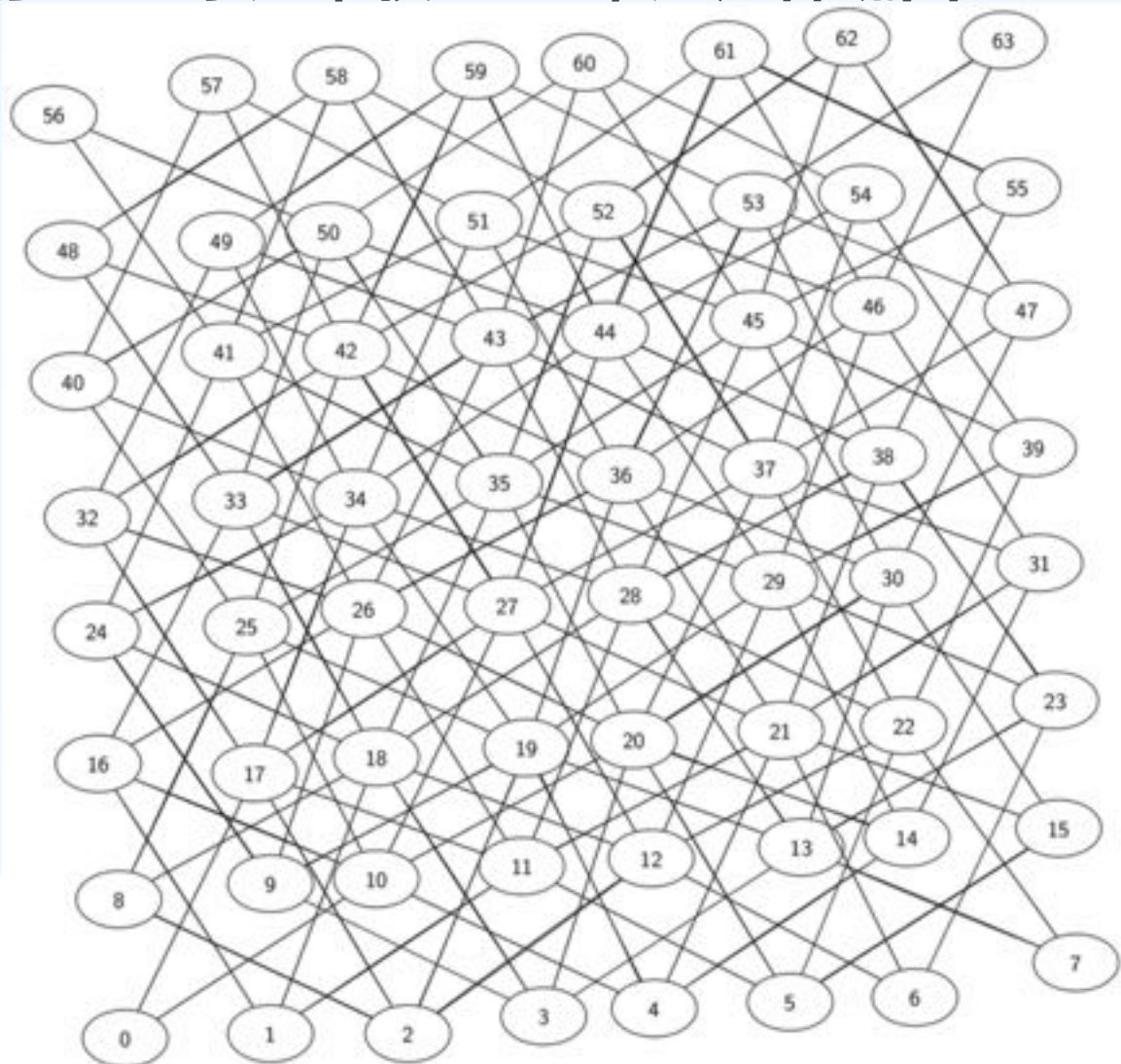
def posToNodeId(row,col,bdSize):
    return row*bdSize+col

def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                   ( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and legalCoord(newY,bdSize):
            newMoves.append((newX,newY))
    return newMoves

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```

# 骑士周游图：8×8棋盘生成的图

› 具有336条边，相比起全连接的4096条边，仅8.2%，还是稀疏图



# 骑士周游算法实现

- › 用于解决骑士周游问题的图搜索算法是深度优先搜索 ( Depth First Search:DFS )
- › 相比前述的广度优先搜索，其逐层建立搜索树的特点，深度优先搜索是沿着树的单支尽量深入向下搜索，如果到无法继续的程度还未找到问题解，就回溯上一层再搜索下一支
- › 下面介绍DFS的两个实现算法
  - 一个DFS算法用于解决骑士周游问题，其特点是每个顶点仅访问一次
  - 另一个DFS算法更为通用，允许顶点被重复访问，可作为其它图算法的基础

# 骑士周游算法实现

- › **深度优先搜索解决骑士周游的关键思路在于：**  
如果沿着单支深入搜索到无法继续（所有合法移动都已经被走过了）时  
路径长度还没有达到预定值（ $8 \times 8$ 棋盘为63）  
那么就清除颜色标记，返回到上一层，换一个分支继续深入搜索。

# 骑士周游算法代码

当前顶点加入路径

选择白色未经过的  
顶点深入

都无法完成总深度，回  
溯，试本层下一个顶点

n:层次；path:路径；u:当前顶点；  
limit:搜索总深度

对所有合法移动逐一深入

层次加  
1，递  
归深入

```
def knightTour(n, path, u, limit):  
    u.setColor('gray')  
    path.append(u)  
    if n < limit:  
        nbrList = list(u.getConnections())  
        i = 0  
        done = False  
        while i < len(nbrList) and not done:  
            if nbrList[i].getColor() == 'white':  
                done = knightTour(n+1, path, nbrList[i], limit)  
            i = i + 1  
        if not done: # prepare to backtrack  
            path.pop()  
            u.setColor('white')  
        else:  
            done = True  
    return done
```

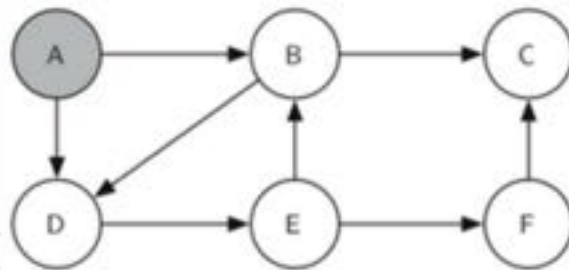


Figure 3: Start with node A

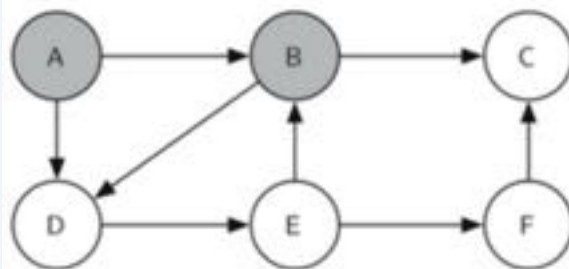


Figure 4: Explore B

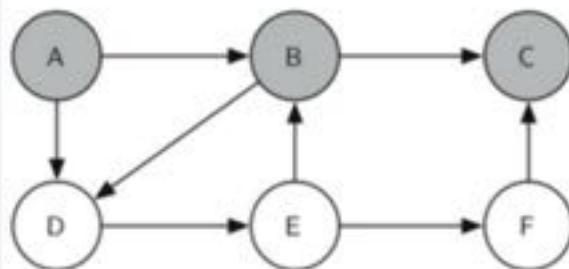


Figure 5: Node C is a dead end

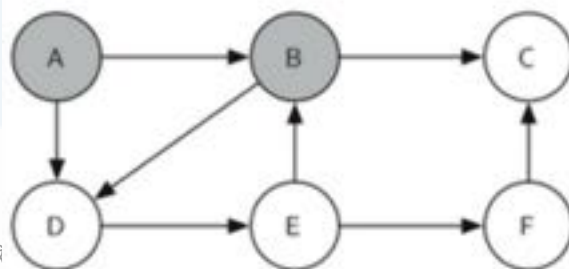


Figure 6: Backtrack to B

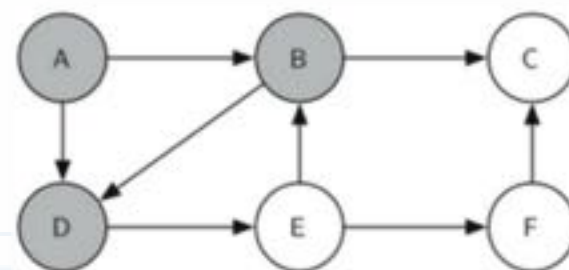


Figure 7: Explore D

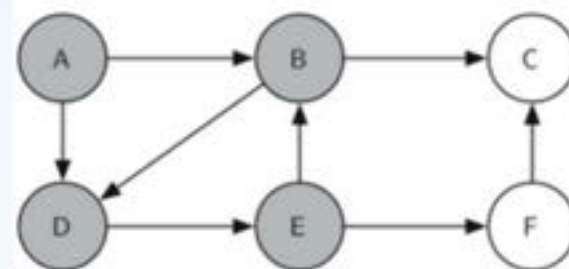


Figure 8: Explore E

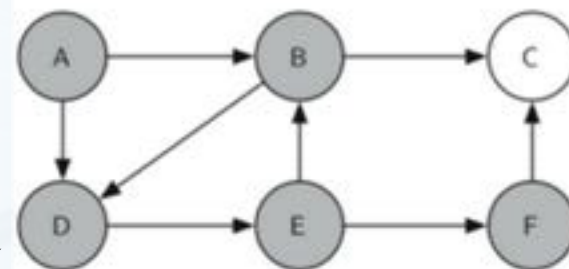


Figure 9: Explore F

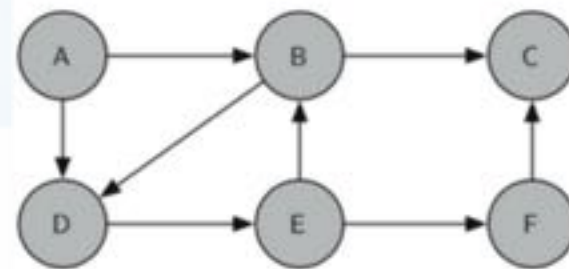
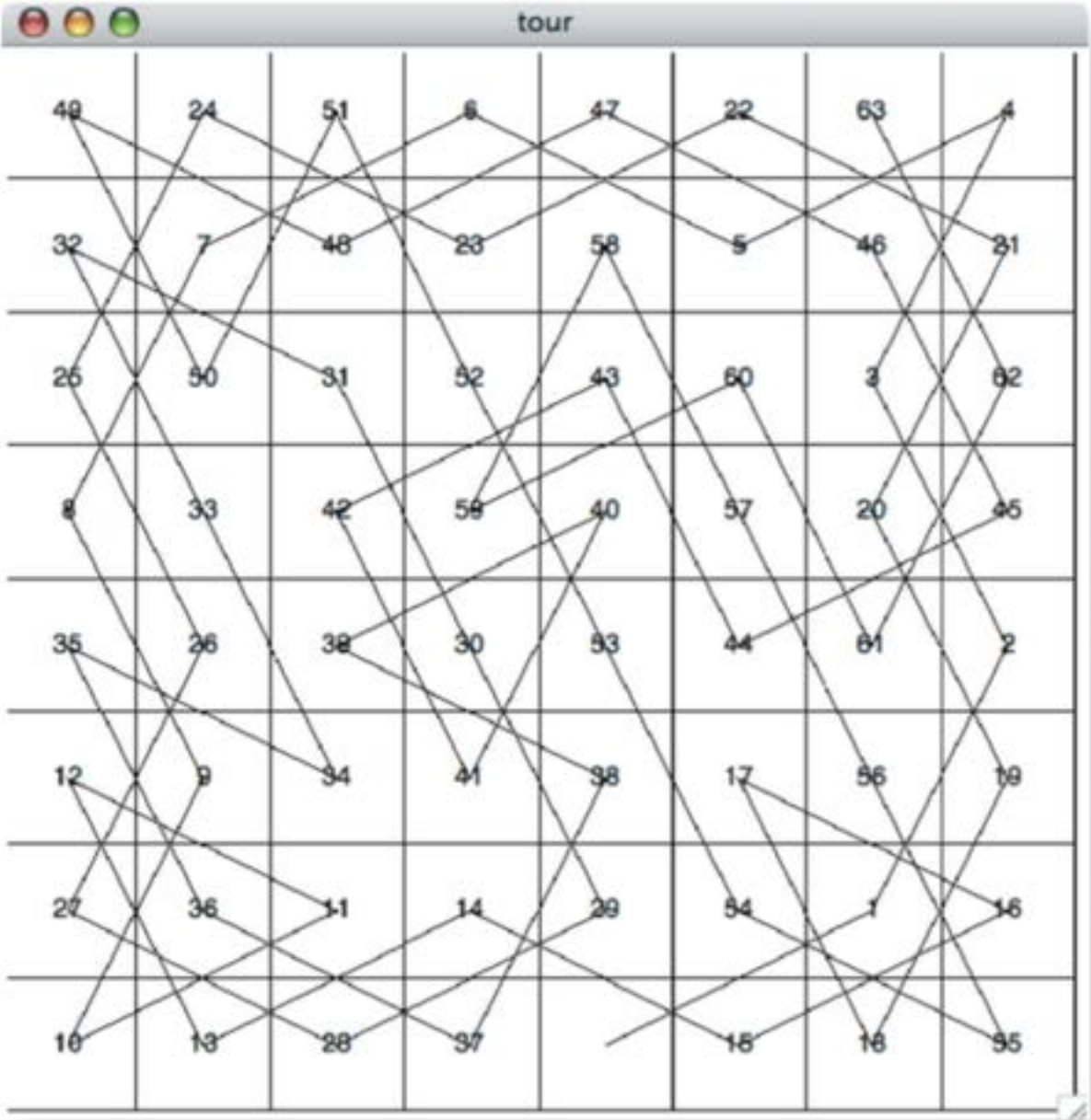


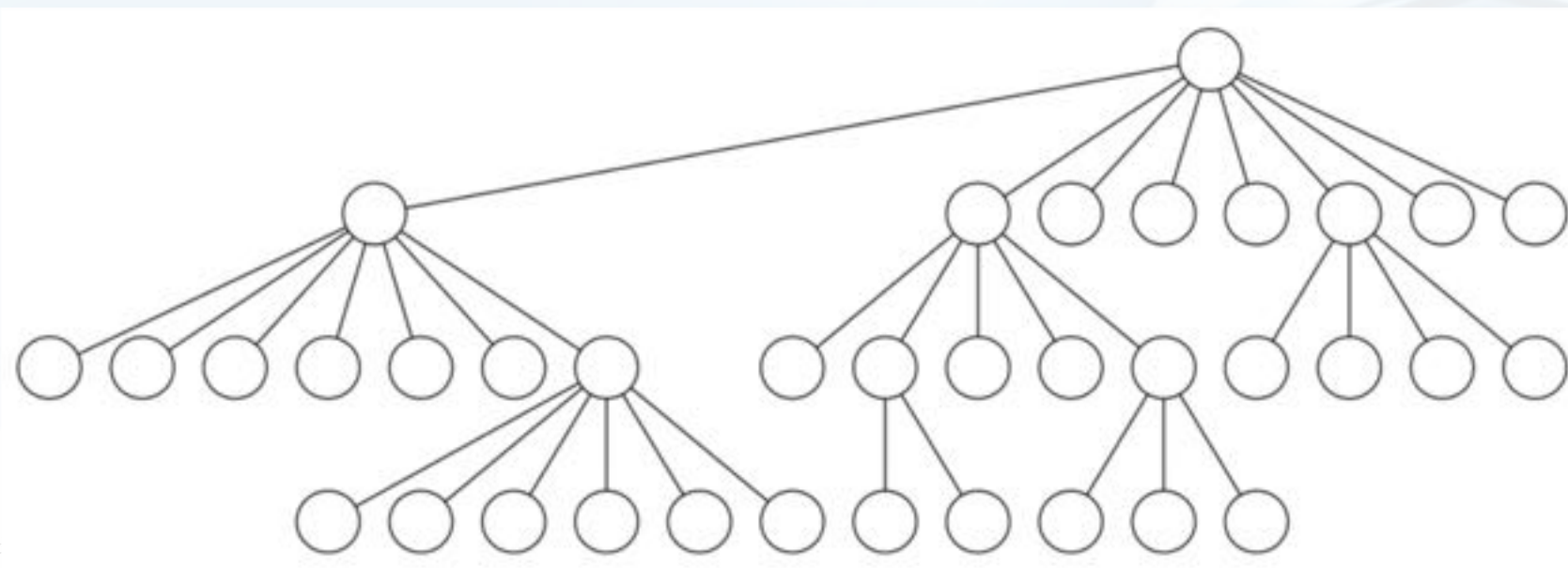
Figure 10: Finish

# 骑士周游问题：一个解



# 骑士周游算法分析

- › 上述算法knightTour的性能高度依赖于深入顶点nbrList的搜索次序：  
就 $5 \times 5$ 的棋盘而言，大约1.5秒可以得到一个周游路径  
但 $8 \times 8$ 的棋盘上，则需要半个小时以上的时间才能得到一个解！
- › 目前实现的算法，其复杂度为 $O(k^n)$ ，其中 $n$ 是棋盘格数目，这是一个指数时间复杂度的算法！其搜索过程表现为一个层次为 $n$ 的树



# 骑士周游算法改进

- › 幸运的是，即便是指数时间复杂度算法也可以在实际性能上加以大幅度改进  
对nbrList的灵巧构造，以特定方式排列顶点访问次序，可以使得 $8 \times 8$ 棋盘的周游路径搜索时间降低到秒级！
- › 初始算法中nbrList= list(u.getConnections())，直接以原始顺序来确定深度优先搜索的分支次序，新的算法，仅修改了nbrList= orderByAvail(u)  
将u的合法移动目标棋盘格排序为：具有最少合法移动目标的格子优先搜索

思考：为什么能提高性能？

尝试：相反的次序会如何？

```
def orderByAvail(n):  
    resList = []  
    for v in n.getConnections():  
        if v.getColor() == 'white':  
            c = 0  
            for w in v.getConnections():  
                if w.getColor() == 'white':  
                    c = c + 1  
            resList.append((c,v))  
    resList.sort(key=lambda x: x[0])  
    return [y[1] for y in resList]
```

# 骑士周游算法改进

- › 这个改进算法被特别以发明者名字命名：Warnsdorff算法
- › 采用先验的知识来改进算法性能的做法，称作为“启发式规则heuristic”  
启发式规则经常用于人工智能领域；  
可以有效地减小搜索范围、更快达到目标等等；  
如棋类程序算法，会预先存入棋谱、布阵口诀、高手习惯等“启发式规则”，能够在最短时间内从海量的棋局落子点搜索树中定位最佳落子。
  - 例如：黑白棋中的“金角银边”口诀，指导程序优先占边角位置等等

# 通用的深度优先搜索

- › 骑士周游问题是一种特殊的对图进行深度优先搜索，其目的是建立一个没有分支的最深的深度优先树
- › 一般的深度优先搜索实际上更为容易，其目标是在图上进行尽量深的搜索，连接尽量多的顶点，必要时可以进行分支（创建了树）

有时候深度优先搜索会创建多棵树，称为“深度优先森林”

深度优先搜索同样要用到顶点的“前驱”属性，来构建树或森林

另外还要设置“发现时间”和“结束时间”两个属性

- 前者是算法在第几步访问到这个顶点（设置灰色）
- 后者则是算法在第几步完成了此顶点的探索（设置黑色）

这两个新属性对后面的图算法很重要

# 通用的深度优先搜索

- › 我们把带有DFS算法的图实现为Graph的子类
  - 顶点Vertex增加了成员Discovery及Finish
  - 图Graph增加了成员time用于记录算法执行的步骤数目

# 通用的深度优先搜索算法代码

BFS采用队列存储待访问顶点  
DFS则是通过递归调用，隐式使用了栈

颜色初始化

如果还有未包括的顶点，则建森林

算法的步数

深度优先递归访问

```
from pythonds.graphs import Graph
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

# 通用的深度优先搜索算法：示例

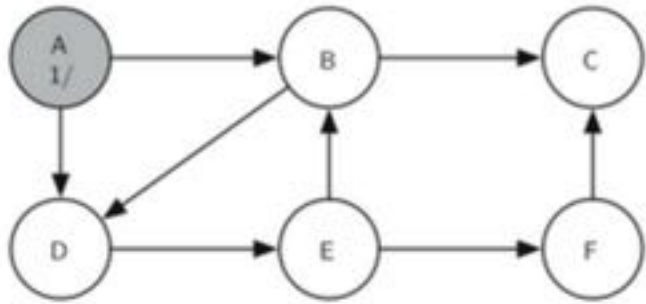


Figure 14: Constructing the Depth First Search Tree-10

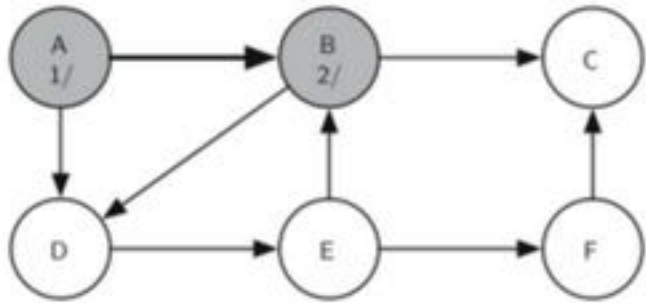


Figure 15: Constructing the Depth First Search Tree-11

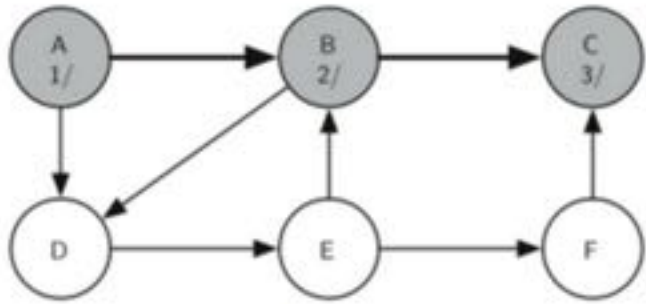


Figure 16: Constructing the Depth First Search Tree-12

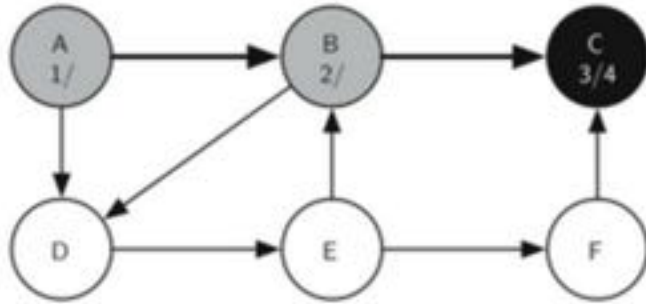


Figure 17: Constructing the Depth First Search Tree-13

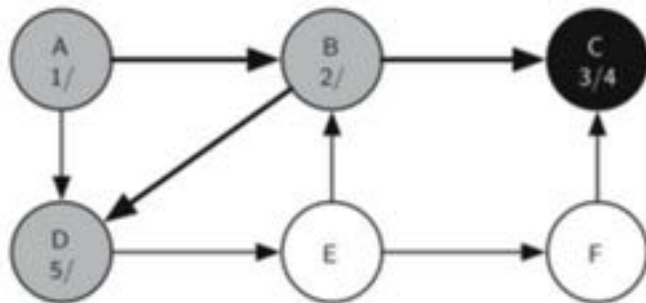


Figure 18: Constructing the Depth First Search Tree-14

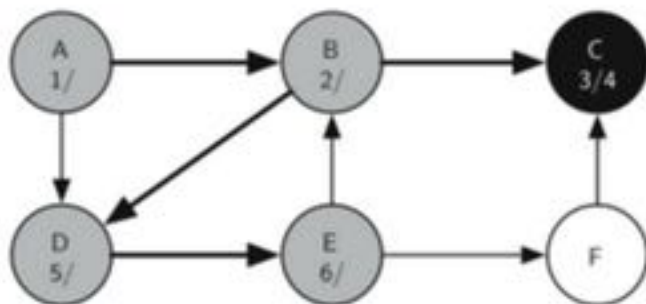


Figure 19: Constructing the Depth First Search Tree-15

# 通用的深度优先搜索算法：示例

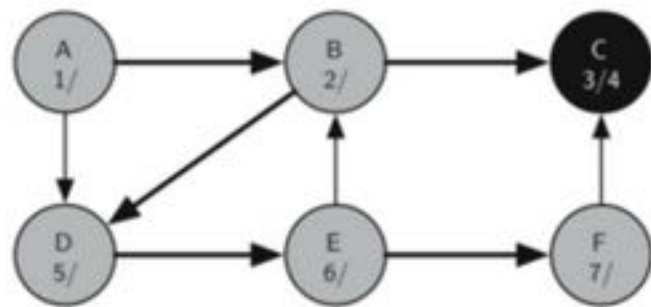


Figure 20: Constructing the Depth First Search Tree-16

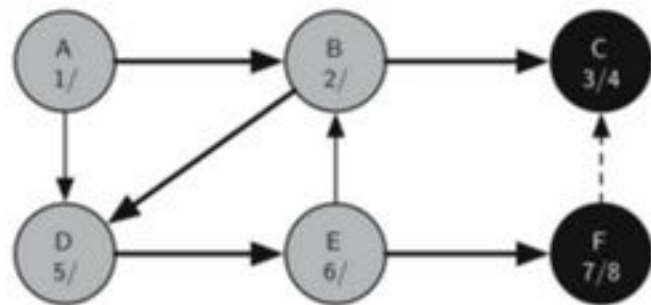


Figure 21: Constructing the Depth First Search Tree-17

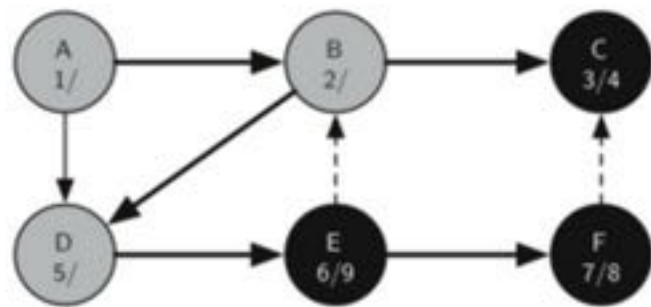


Figure 22: Constructing the Depth First Search Tree-18

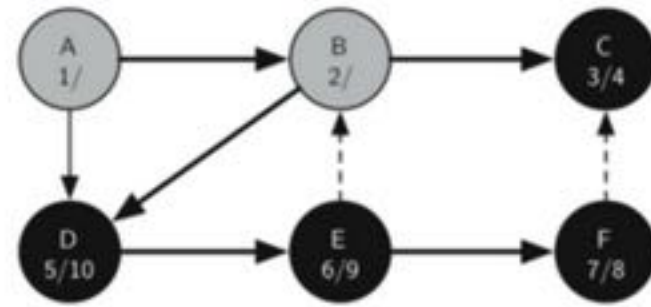


Figure 23: Constructing the Depth First Search Tree-19

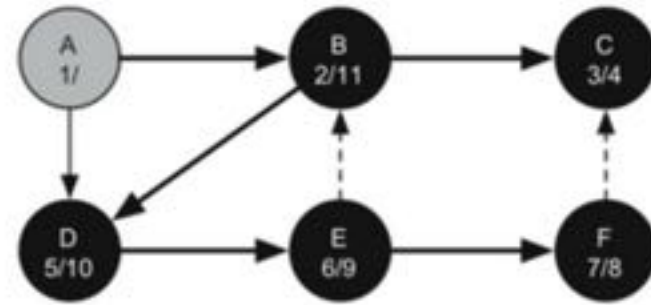


Figure 24: Constructing the Depth First Search Tree-20

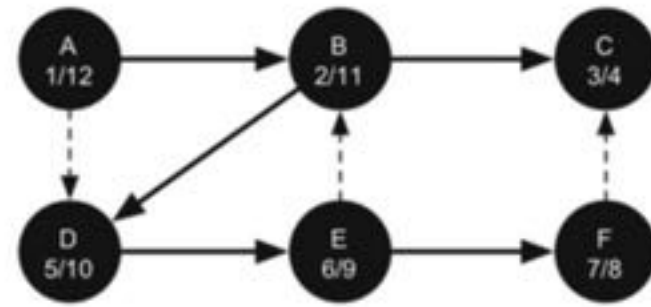


Figure 25: Constructing the Depth First Search Tree-21

# 通用的深度优先搜索算法：分析

- › DFS构建的树，其顶点上的“发现时间”和“结束时间”属性，具有类似括号的性质：即一个顶点的“发现时间”总小于所有子顶点的“发现时间”，而“结束时间”则大于所有子顶点的“结束时间”  
比子顶点更早被发现，更晚被结束探索
- › DFS运行时间同样也包括了两方面：  
dfs函数中有两个循环，每个都是 $|V|$ 次，所以是 $O(|V|)$   
而dfsvisit函数中的循环则是对当前顶点所连接的顶点进行，而且仅有在顶点为白色的情况下才进行递归调用，所以对每条边来说只会运行一步，所以是 $O(|E|)$   
加起来就是和BFS一样的 $O(|V|+|E|)$

# 拓扑排序Topological Sort

› 很多问题都可以转化为图，利用图算法来解决

› 例如早餐吃薄煎饼的过程

以动作为顶点

以先后次序为有向边

› 问题是对整个过程而言

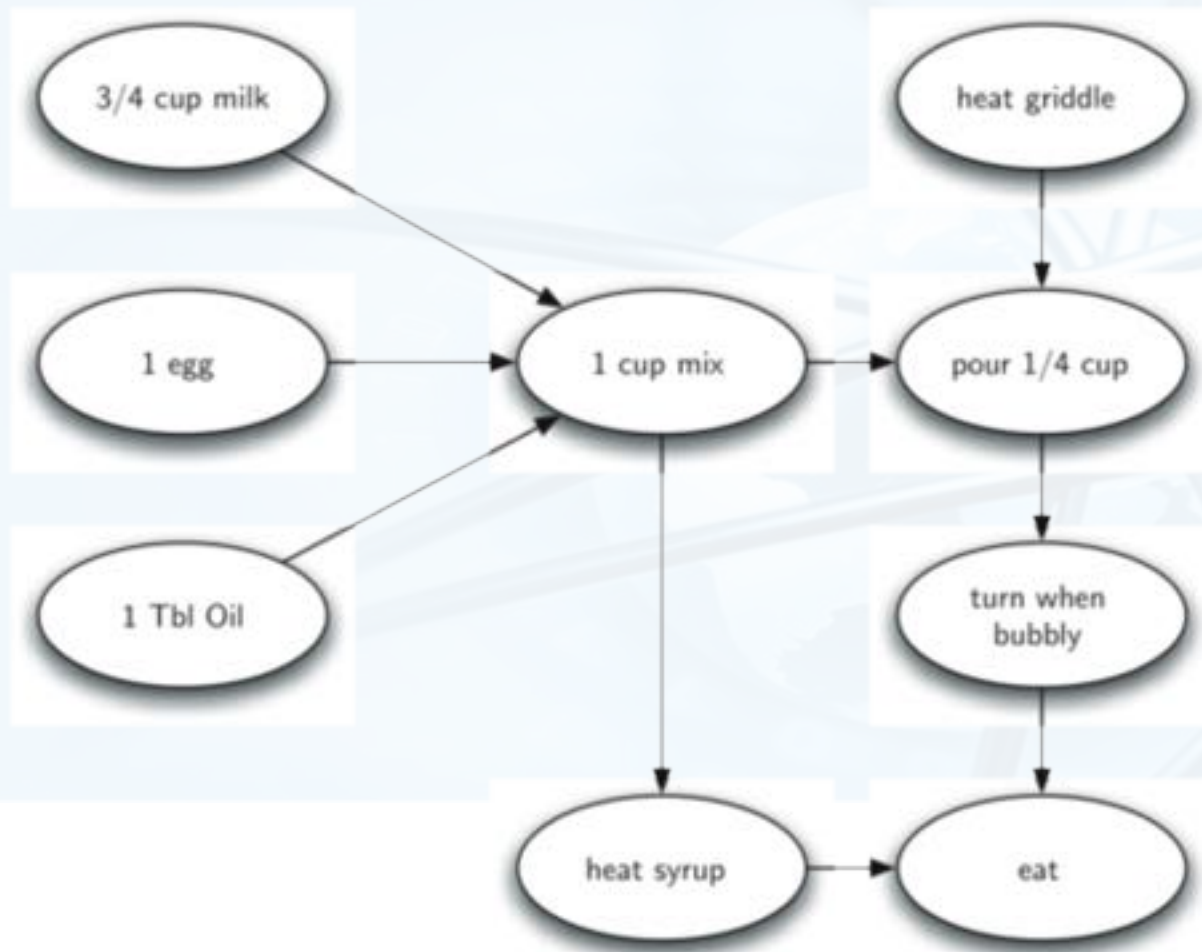
如果一个人独自做，

所有动作的先后次序？

› 从加料开始？

还是从加热烤盘开始？

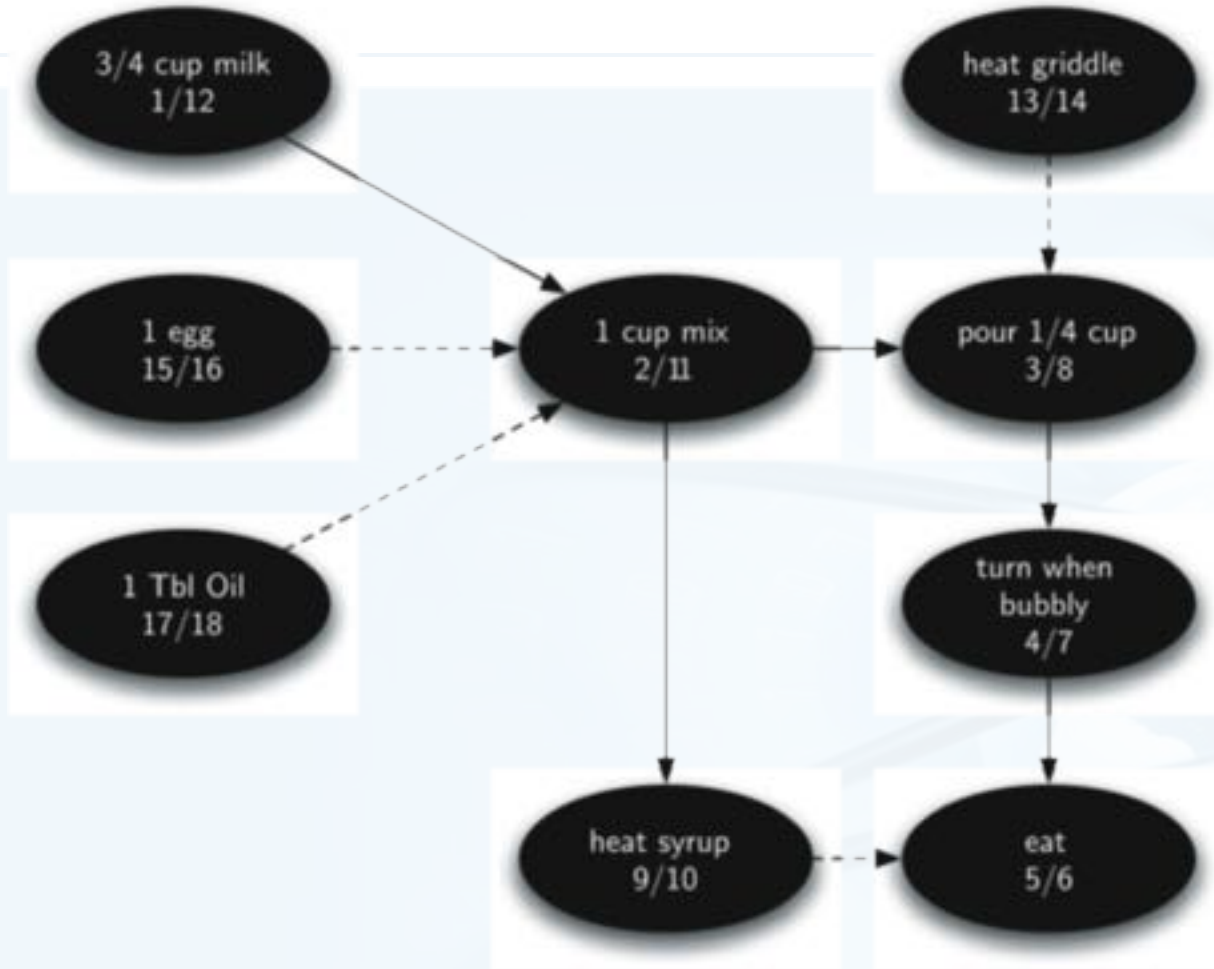
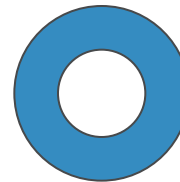
› 思考：整个过程次序



# 拓扑排序Topological Sort

- › 从工作流程图得到工作次序排列的算法，称为“拓扑排序”
- › 拓扑排序处理一个DAG，输出顶点的线性序列，使得两个顶点 $v, w$ ，如果 $G$ 中有 $(v, w)$ 边，在线性序列中 $v$ 就出现在 $w$ 之前。
- › DAG的广泛应用在依赖事件的排期上，除了吃早餐过程之外，还可以用在项目管理、数据库查询优化和矩阵乘法的次序优化上
- › 拓扑排序可以采用DFS很好地实现：
  - 对DAG图调用DFS算法，以得到每个顶点的“结束时间”
  - 按照每个顶点的“结束时间”从大到小排序
  - 输出这个次序下的顶点列表
- › 课后练习：整理出骑士周游算法、通用DFS以及拓扑排序算法的代码

# 拓扑排序：示例



# 数据结构与算法 (Python)

-

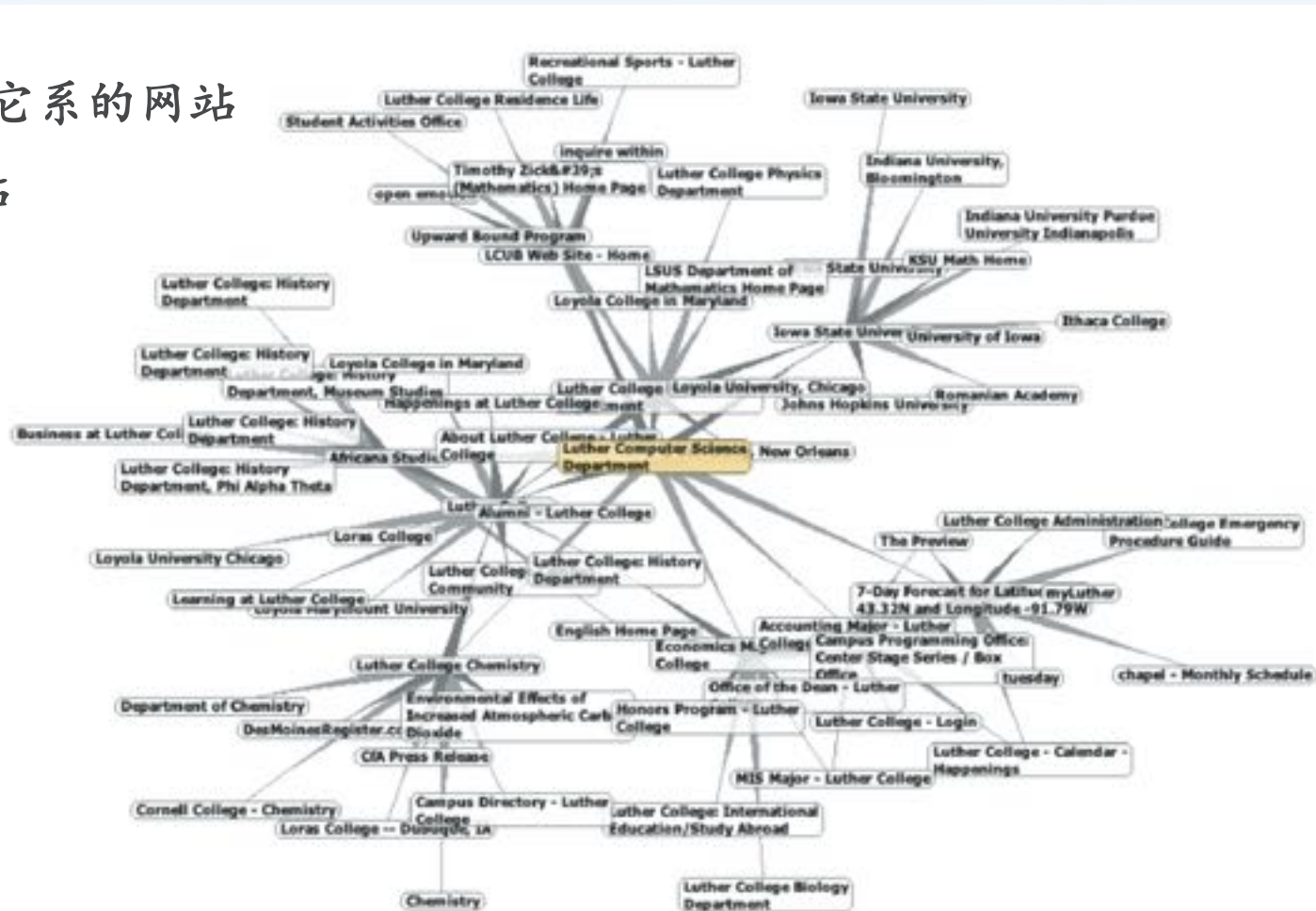
# 强连通分支

## › 上图是Luther College的Computer Science网站链接情况，有三个有趣的现象

图中包含了许多Luther College其它系的网站

包含了一些Iowa其它大学学院的网站

还包含了一些人文学院的网站

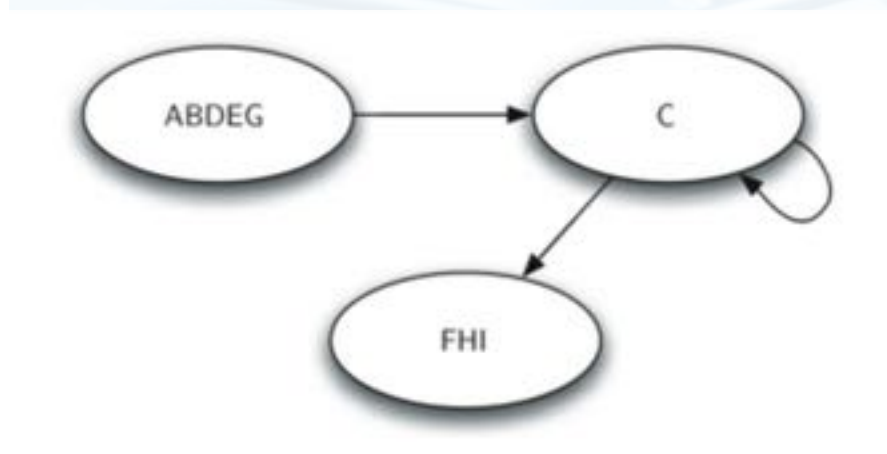
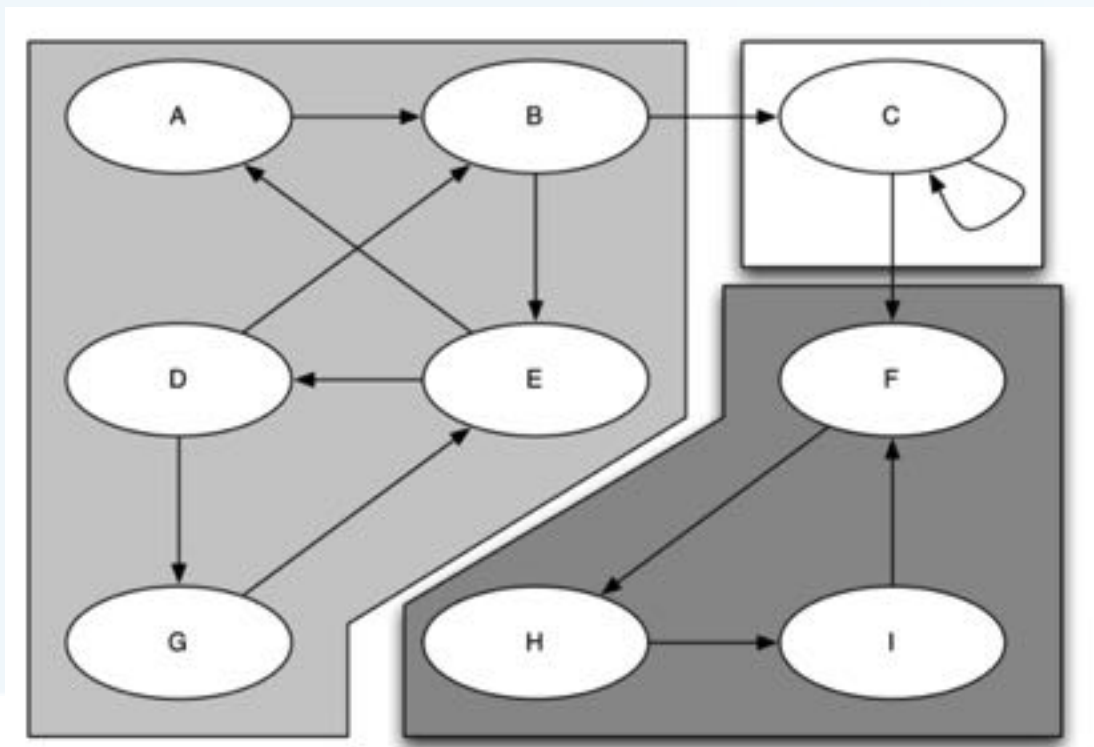


# 强连通分支

- › 我们可以猜想，Web的底层结构可能存在某些同类网站的聚集
- › 在图中发现高度聚集节点群的算法，即寻找“强连通分支 Strongly Connected Components”算法。
- › 强连通分支SCC，定义为图G的一个子集C
- › C中的任意两个顶点 $v, w$ 之间都有路径来回  
即 $(v, w)$   $(w, v)$ 都是C的路径，
- › 而且C是具有这样性质的最大子集

# 强连通分支例子

- 右图是具有3个强连通分支的9顶点有向图
- 一旦找到强连通分支，可以据此对图的顶点进行分类，并对图进行化简。

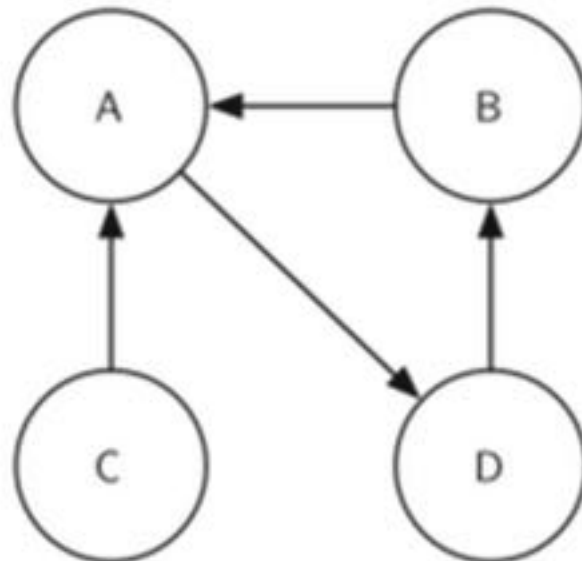
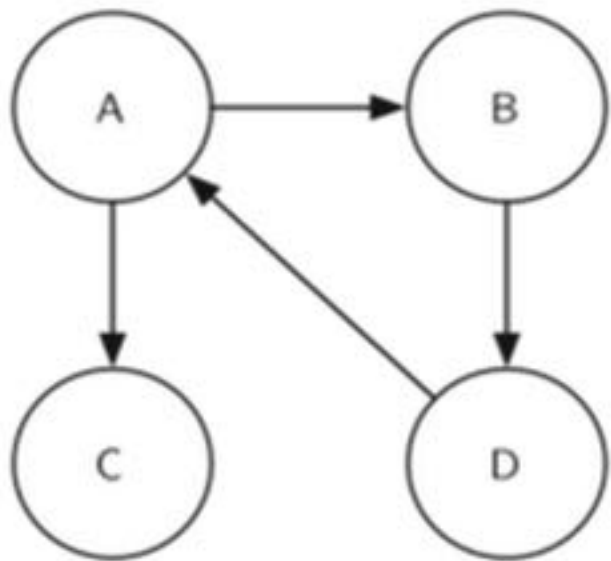


# 强连通分支算法：转置Transposition概念

- › 在使用深度优先搜索DFS算法来发现强连通分支之前，先熟悉一个概念：**Transposition转置**

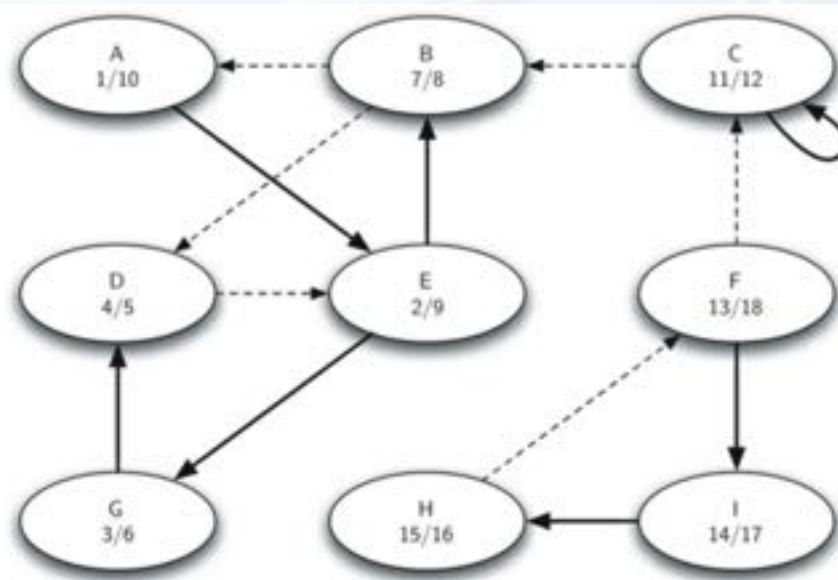
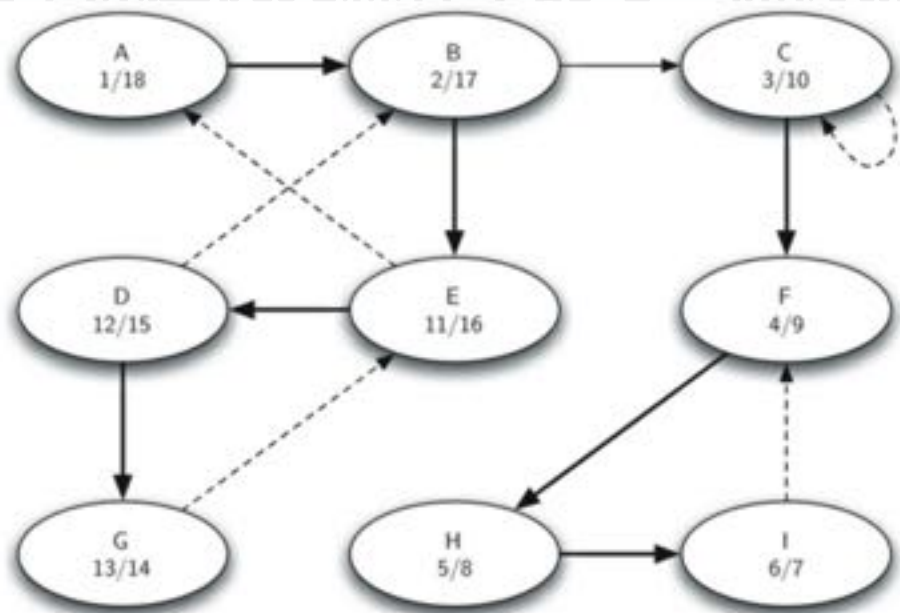
一个有向图 $G$ 的转置 $G^T$ ，定义为将图 $G$ 的所有边的顶点交换次序，如将 $(v, w)$ 转换为 $(w, v)$ ，如图所示的图 $G$ 和转置 $G^T$ ：

可以观察到图和转置图在强连通分支的数量和划分上，是相同的。



# 强连通分支算法：思路

- 首先，对图G调用dfs算法，为每个顶点计算“结束时间”；
- 然后，将图G进行转置，得到 $G^T$ ；
- 再对 $G^T$ 调用dfs算法，但在dfs函数中，对每个顶点的搜索循环里，要以顶点的“结束时间”倒序的顺序来搜索
- 最后，深度优先森林中的每一棵树就是一个强连通分支



# 强连通分支算法：思路

› 输出的强连通分支如图

› 本算法的延伸阅读

<http://edward-mj.com/archives/455>

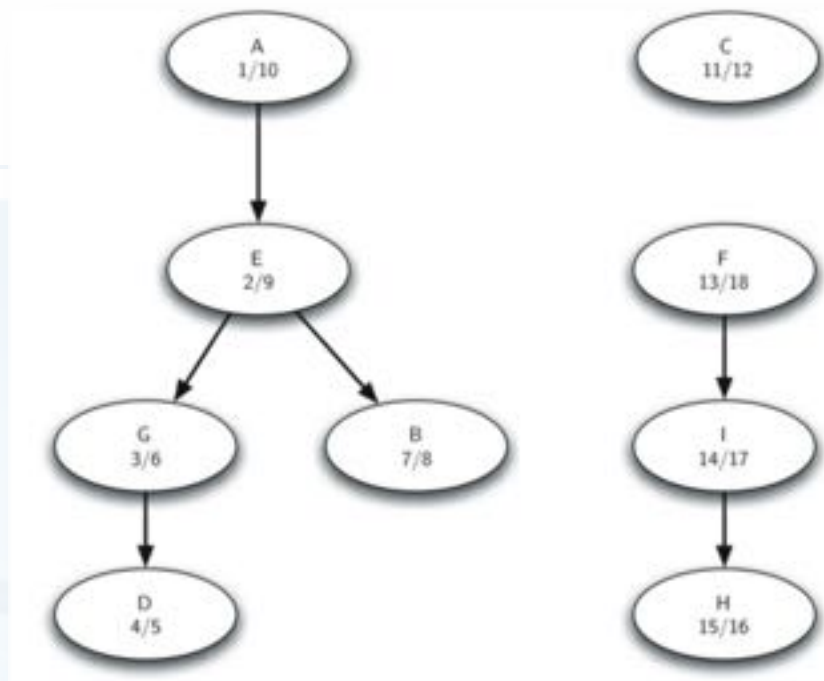
› 另一个常用的强连通分支算法

Tarjan算法

<http://www.cnblogs.com/luweiseu/archive/2012/07/14/2591370.html>

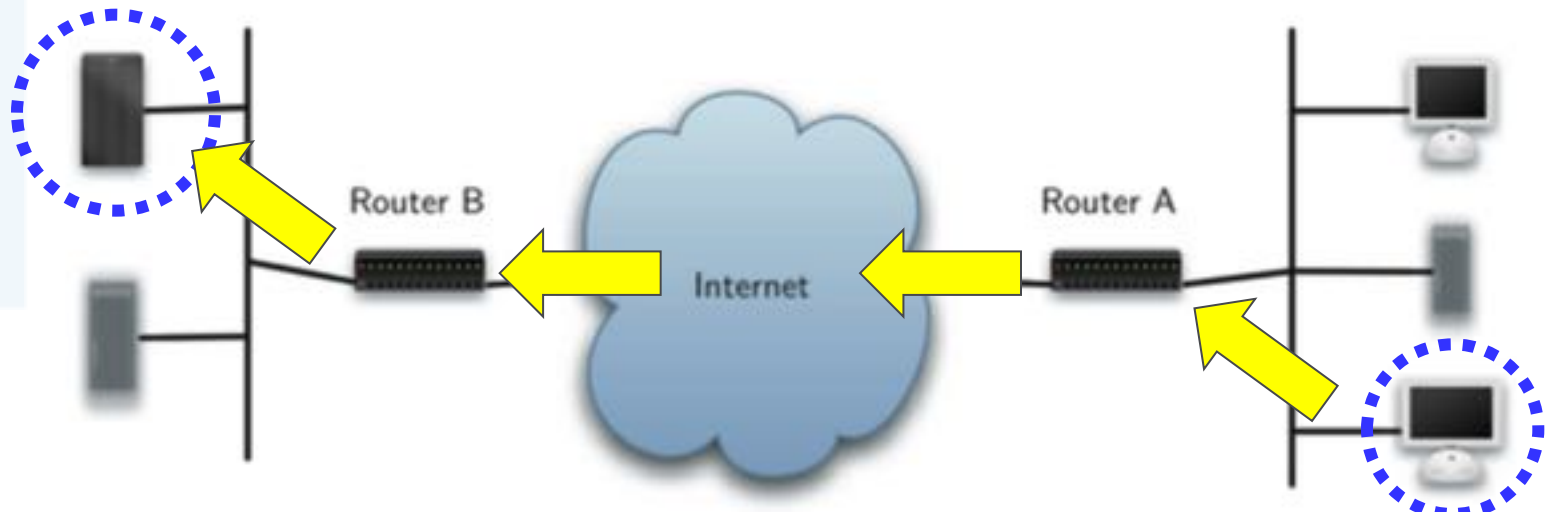
› 课后练习1：写出本算法（Kosaraju算法）的代码

› 课后练习2：根据参考资料写出Tarjan算法的代码



# 最短路径问题：介绍

- 当我们通过网络浏览网页、发送电子邮件、QQ消息传输的时候，数据会在联网设备之间流动，计算机网络专业领域会详尽地研究网络各层面上的技术细节
- 我们对Internet工作方式感兴趣的主要是其中包含的图算法
- 如图，当PC上的浏览器向服务器请求一个网页时，请求信息需要先通过本地局域网，由路由器A发送到Internet，请求信息沿着Internet中的众多路由器传播，最后到达服务器本地局域网所属的路由器B，从而传给服务器。



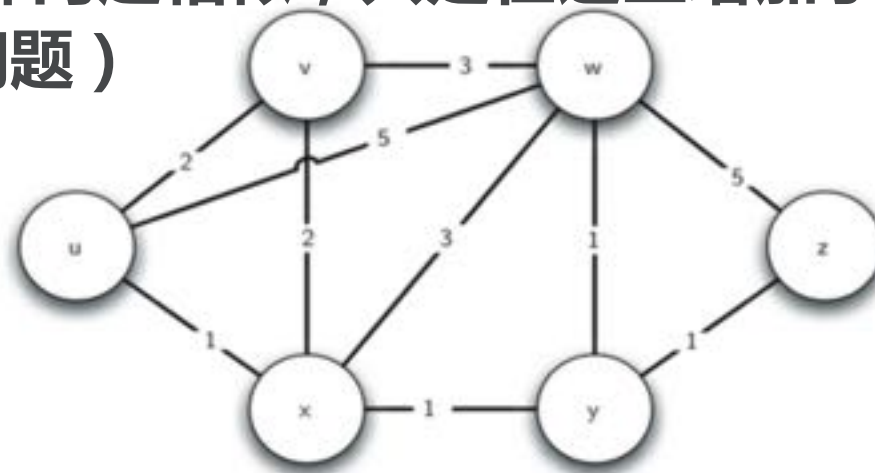
# 最短路径问题：介绍

- 图中标注“Internet”的云状结构，实际上是一个由路由器连接成的网络，这些路由器各自独立而又协同工作，负责将信息从Internet的一端传送到另一端。
- 我们可以通过“traceroute”命令来跟踪信息传送的路径（由于某种原因，北大校园网某些部分关闭了路由追踪，无法正常使用本命令），我们来看看从Luther College的web服务器到University of Minnesota的mail服务器之间的一条路由器路径，包含了13个路由器。
- 由于网络流量的状况会影响
- 路径选择算法，在不同的时
- 间，路径可能不同。

```
1 192.203.196.1
2 hilda.luther.edu (216.159.75.1)
3 ICN-Luther-Ether.icn.state.ia.us (207.165.237.137)
4 ICN-ISP-1.icn.state.ia.us (209.56.255.1)
5 p3-0.hsa1.chi1.bbnplanet.net (4.24.202.13)
6 ae-1-54.bbr2.Chicago1.Level3.net (4.68.101.97)
7 so-3-0-0.mpls2.Minneapolis1.Level3.net (64.159.4.214)
8 ge-3-0.hsa2.Minneapolis1.Level3.net (4.68.112.18)
9 p1-0.minnesota.bbnplanet.net (4.24.226.74)
10 TelecomB-BR-01-V4002.ggnet.umn.edu (192.42.152.37)
11 TelecomB-BN-01-Vlan-3000.ggnet.umn.edu (128.101.58.1)
12 TelecomB-CN-01-Vlan-710.ggnet.umn.edu (128.101.80.158)
13 baldrick.cs.umn.edu (128.101.80.129)(N!) 88.631 ms (N!)
```

# 最短路径问题：介绍

- › 所以我们可以将互联网路由器体系表示为一个带权边的图  
路由器作为顶点，路由器之间的网络连接作为边  
权重可以包括网络连接的速度、网络负载程度、分时段优先级等影响因素  
作为一个抽象，我们把所有影响因素合成为单一的权重
- › 解决信息在路由器网络中选择传播速度最快路径的问题，就转变为在带权图上最短路径的问题。
- › 这个问题与广度优先搜索BFS算法解决的词梯问题相似，只是在边上增加了权重（如果所有权重相等，还是还原到词梯问题）



# 最短路径问题：Dijkstra算法

/ˈdɛɪkstrə/

- › 解决带权最短路径问题的经典算法是以发明者命名的“Dijkstra算法”，这是一个迭代算法，得出从一个顶点到其余所有顶点的最短路径，很接近于广度优先搜索算法BFS的结果。
- › 具体实现上，在顶点Vertex类中的成员dist用于记录从开始顶点到本顶点的最短带权路径长度（**权重之和**），算法对图中的每个顶点迭代一次。
- › 顶点的访问次序由一个**优先队列**Priority Queue来控制，优先队列中作为优先级的是顶点的dist属性。
- › 最初，只有开始顶点dist设为0，而其他所有顶点dist设为sys.maxsize（最大整数），全部加入优先队列。
- › 随着开始顶点率先出队，并计算它与邻接顶点的权重，会引起其它顶点dist的减小和修改，引起堆重排，并据此依次出队。

# 最短路径问题：Dijkstra算法代码

对所有顶点建堆，  
形成优先队列

优先队列出队

修改出队顶点所邻  
接顶点的dist，并逐  
个重排队列

```
from pythonds.graphs import PriorityQueue, Graph, Vertex
def dijkstra(aGraph, start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance(newDist)
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert, newDist)
```

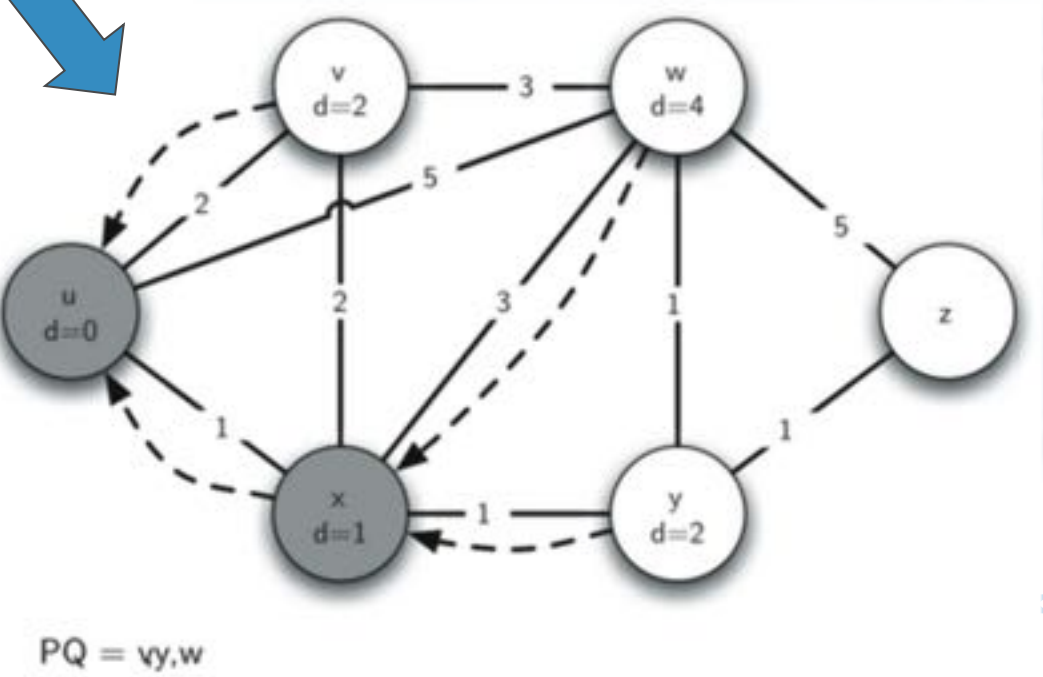
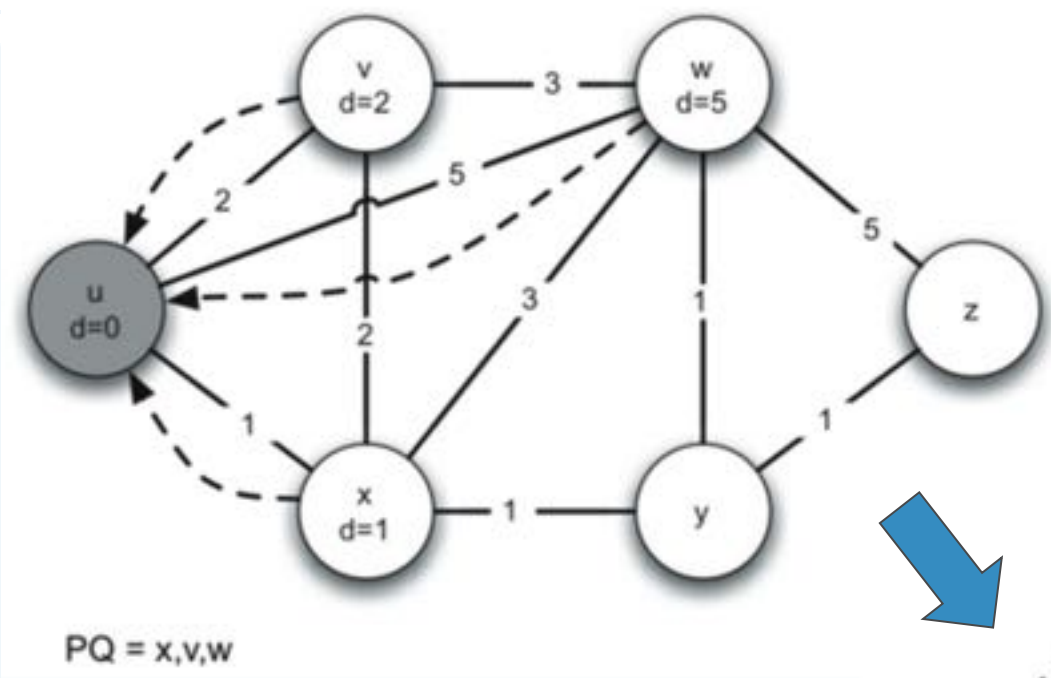
重  
排  
队  
列

```
def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')
```

BFS对比

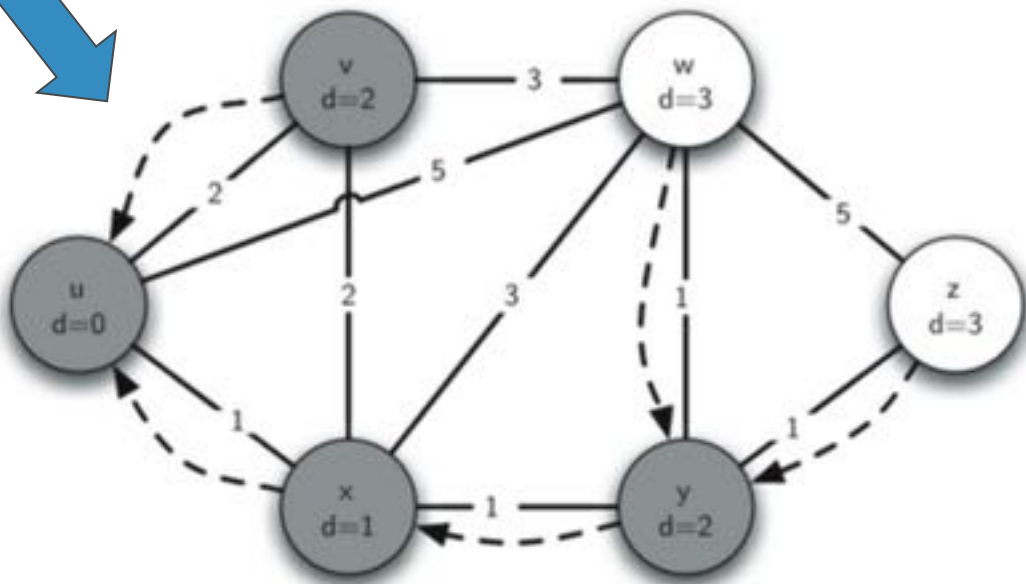
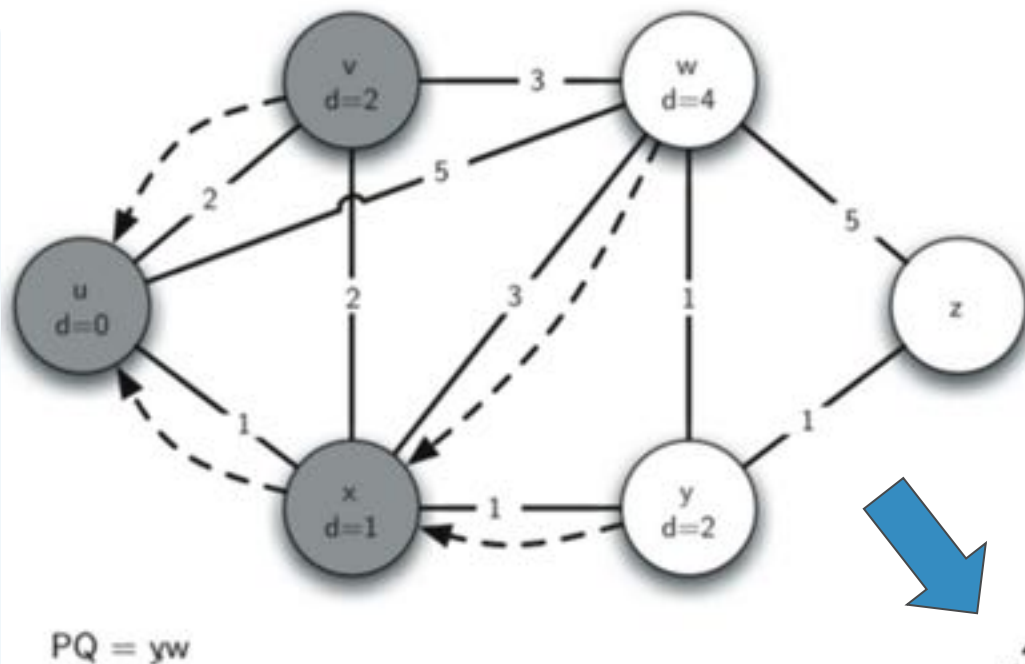
# 最短路径问题：Dijkstra算法实例

数据结构与算法 (Python)



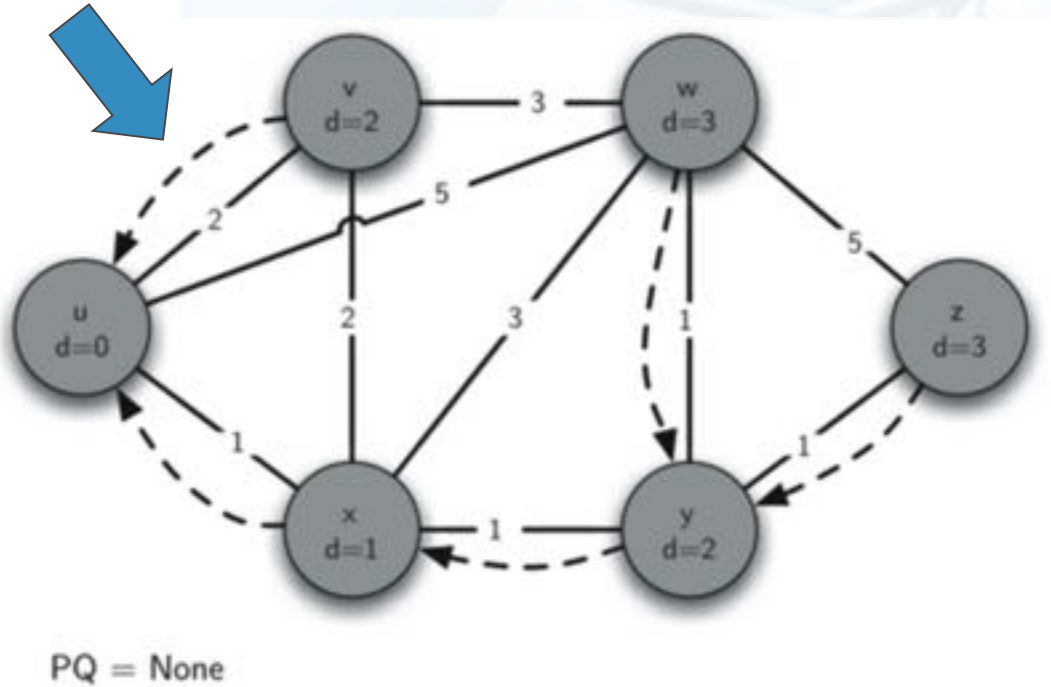
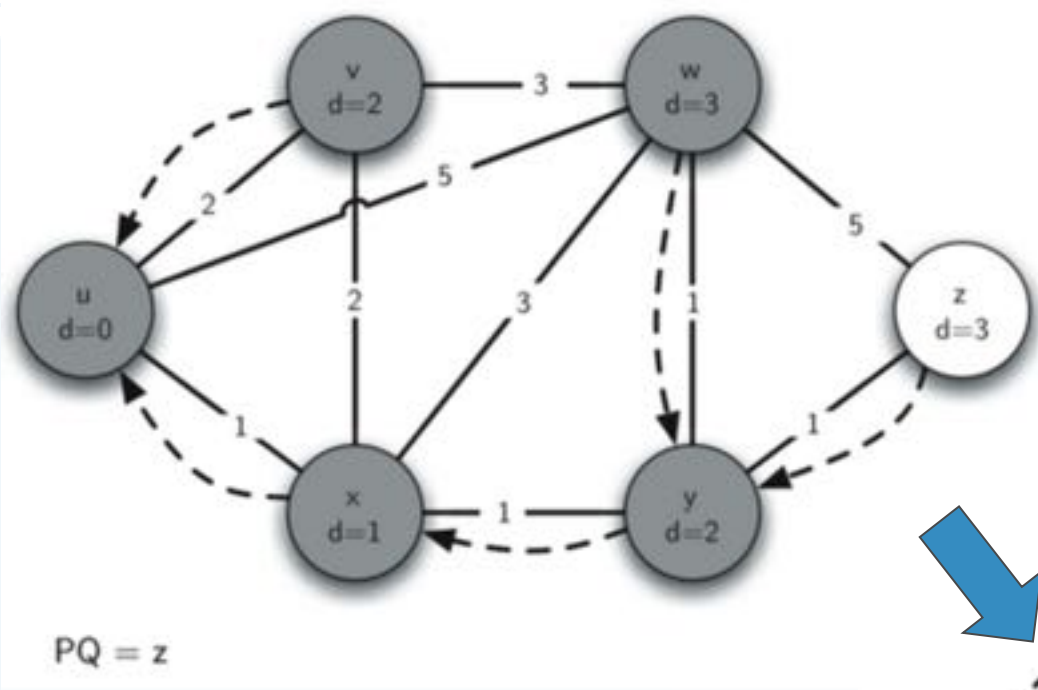
# 最短路径问题：Dijkstra算法实例

数据结构与算法 (Python)



# 最短路径问题：Dijkstra算法实例

数据结构与算法 (Python)



# 最短路径问题：Dijkstra算法

- › 需要注意的是，Dijkstra算法只能处理大于0的权重，如果图中出现负数权重，则算法会陷入无限循环
- › 虽然Dijkstra算法完美解决了带权图的最短路径问题，但实际上Internet的路由器中采用的是其它算法 (☺)
- › 其中最重要的原因是，Dijkstra算法需要具备整个图的数据，但对于Internet的路由器来说，显然无法将整个Internet所有路由器及其连接信息保存在本地，这不仅是数据量的问题，Internet动态变化的特性也使得保存全图缺乏现实性。
- › 路由器的选径算法（或“路由算法”）对于互联网极其重要，有兴趣可以进一步参考“距离向量路由算法”。

<http://baike.baidu.com/view/1227645.htm>

# 最短路径问题：Dijkstra算法分析

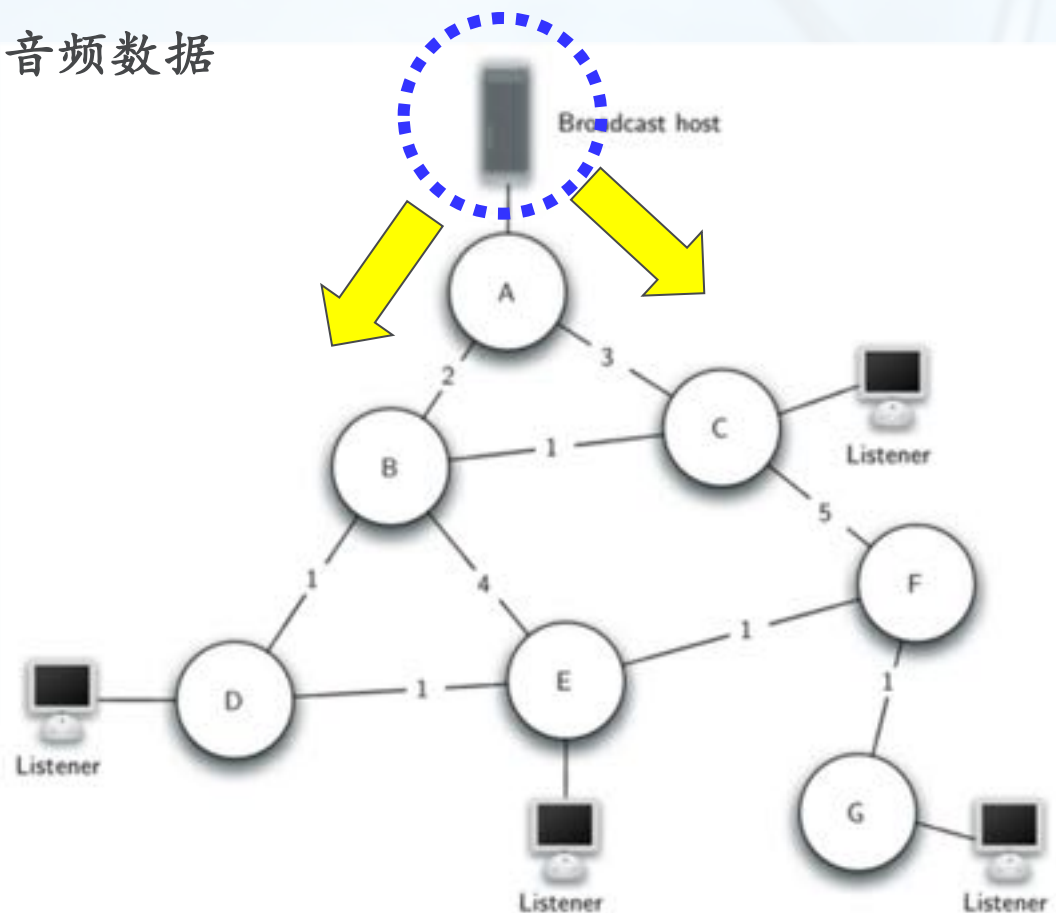
- › 最后，我们对Dijkstra算法的运行时间进行分析
- › 首先，将所有顶点加入优先队列并**建堆**，时间复杂度为 $O(|V|)$
- › 其次，每个顶点仅出队1次，每次**delMin**花费 $O(\log|V|)$ ，一共就是 $O(|V|\log|V|)$
- › 另外，每个边关联到的顶点会做一次**decreaseKey**操作（ $O(\log|V|)$ ），一共是 $O(|E|\log|V|)$
- › 三个加在一起，数量级就是 $O((|V|+|E|)\log|V|)$

# 最小生成树

## 最后一个图算法，涉及到在互联网中网游设计者和Internet收音机所面临的问题：信息广播问题

网游需要让所有玩家获知其他玩家所在的位置

收音机则需要让所有收听用户获取直播的音频数据

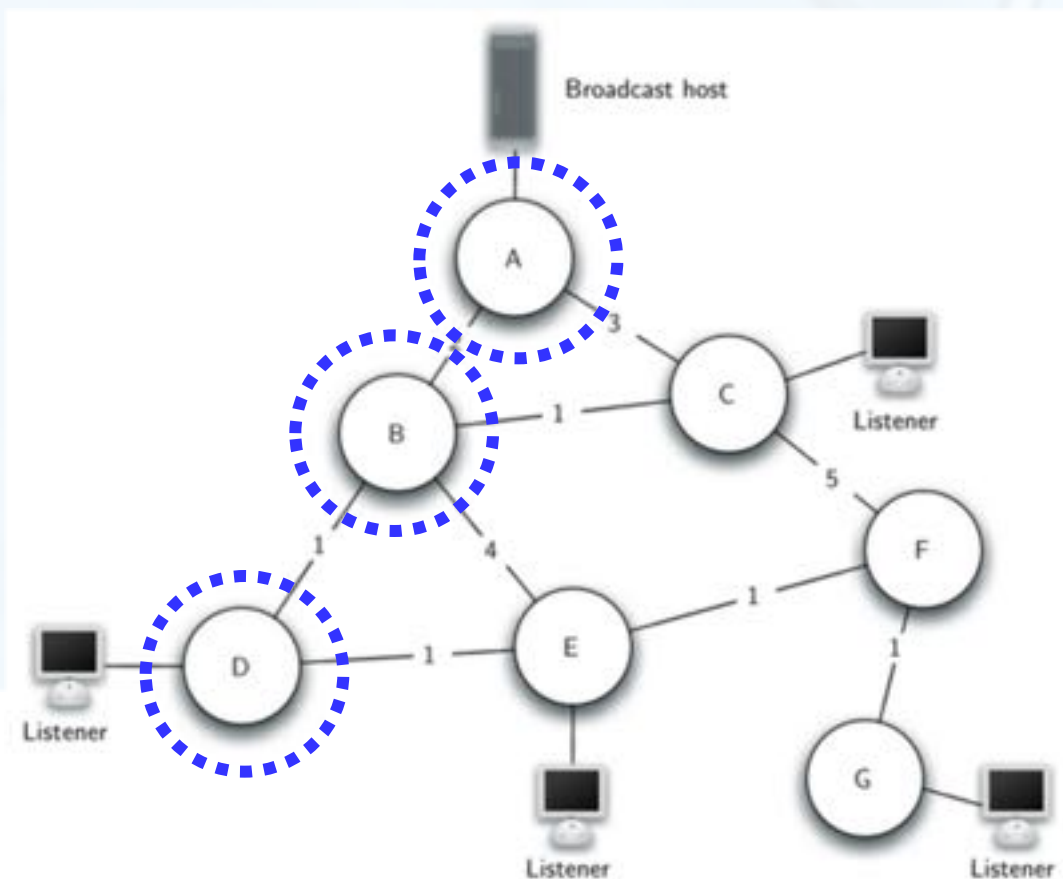


# 信息广播问题：单播解法

- › 信息广播问题最简单的解法是由广播源维护一个收听者的列表，将每条消息向每个收听者发送一次

如图，每条消息会被发送4次，每个消息都采用最短路径算法到达收听者

- › 路由器A会处理4次相同消息；
- › 路由器C仅会处理1次；
- › 而路由器B/D位于其它3个
- › 收听者的最短路径上，则
- › 各会处理转发3次相同消息；
- › 所以，会产生许许多多额外流量



# 信息广播问题：洪水解法

- › **信息广播问题的暴力解法，是将每条消息在路由器间散布出去，所有的路由器都将收到的消息转发到自己相邻的路由器和收听者**  
显然，如果没有任何限制，这个方法将造成网络洪水灾难，很多路由器和收听者会不断重复收到相同的消息，永不停止！
- › **所以，洪水解法还会给每条消息附加一个生命值（TTL:Time To Live），初始设置为从消息源到最远的收听者的距离；**
- › **每个路由器收到一条消息，如果其TTL值大于0，则将TTL减少1，再转发出去**  
如果TTL等于0了，则就直接抛弃这个消息。
- › **TTL的设置防止了灾难发生，但这种洪水解法显然比前述的单播方法所产生的流量还要大。**

# 信息广播问题：最小生成树

- 信息广播问题的最优解法，依赖于路由器关系图上选取具有最小权重的生成树 ( minimum weight spanning tree )

生成树：拥有图中所有的顶点和最少数量的边，以保持连通的子图。

- 图 $G(V,E)$ 的最小生成树 $T$ ，定义为包含所有顶点 $V$ ，以及 $E$ 的无圈子集，并且边权重之和最小。

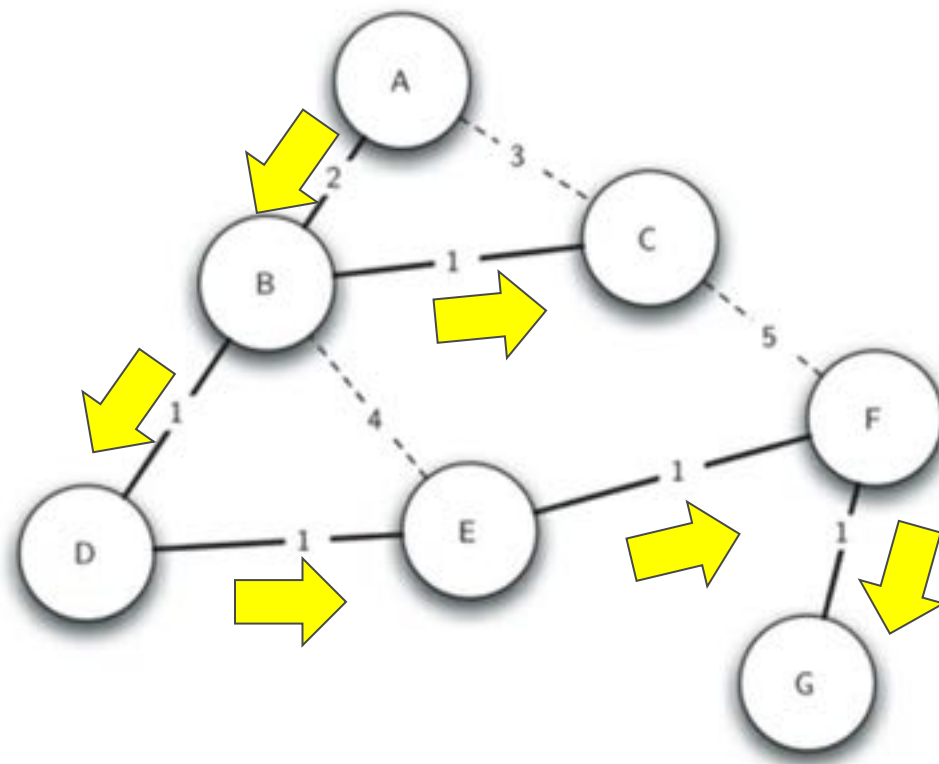
- 图为一个最小生成树

- 这样信息广播就只需要从A开始

- 沿着树的路径层次向下传播

- 就可以达到每个路由器只需要

- 处理1次消息，同时总费用最小。

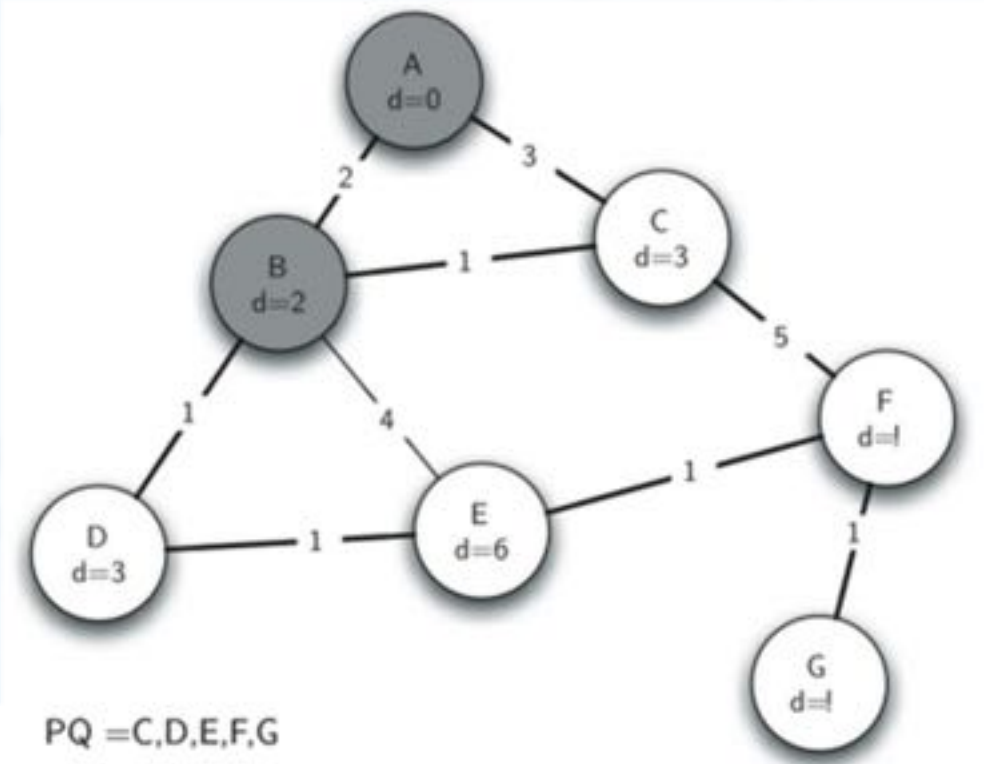
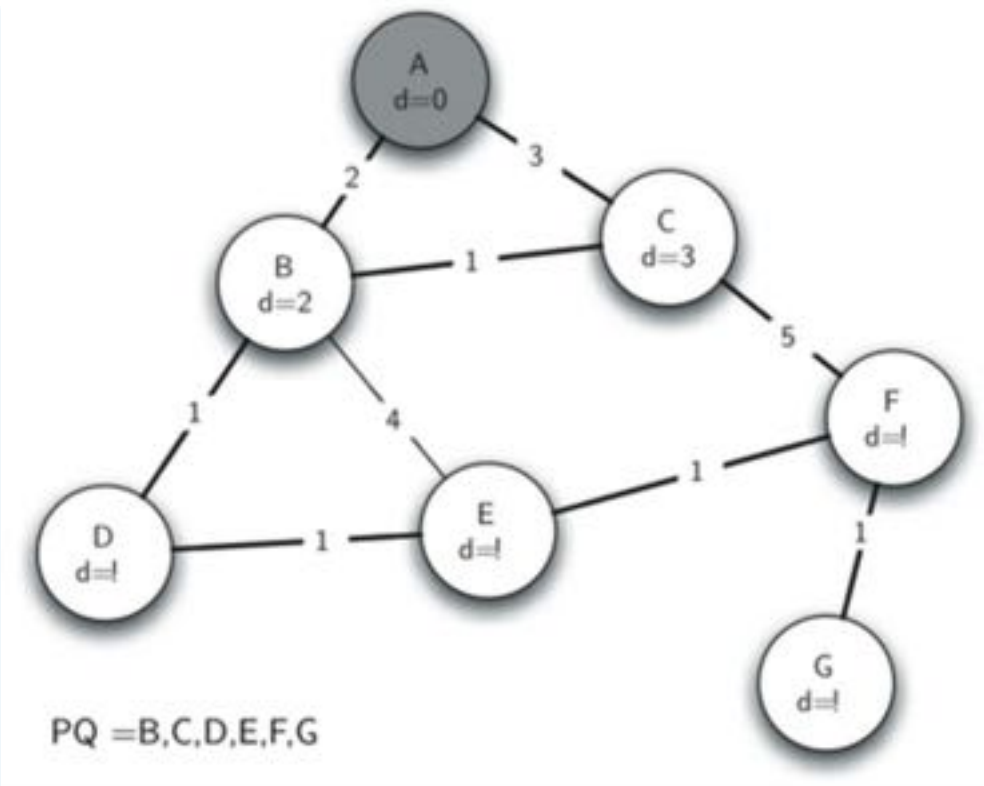


# 最小生成树：Prim算法

- › 解决最小生成树问题的Prim算法，属于“贪心算法”，即每步都沿着最小权重的边向前搜索。
- › 构造生成树的思路如下：  
如果T还不是生成树，则反复做：
  - 找到一条可以安全添加到树T的边
  - 将边添加到树T
- › “**可以安全添加**”的边，定义为**一端顶点在树中，另一端不在树中的边**，以便保持树的无圈特性

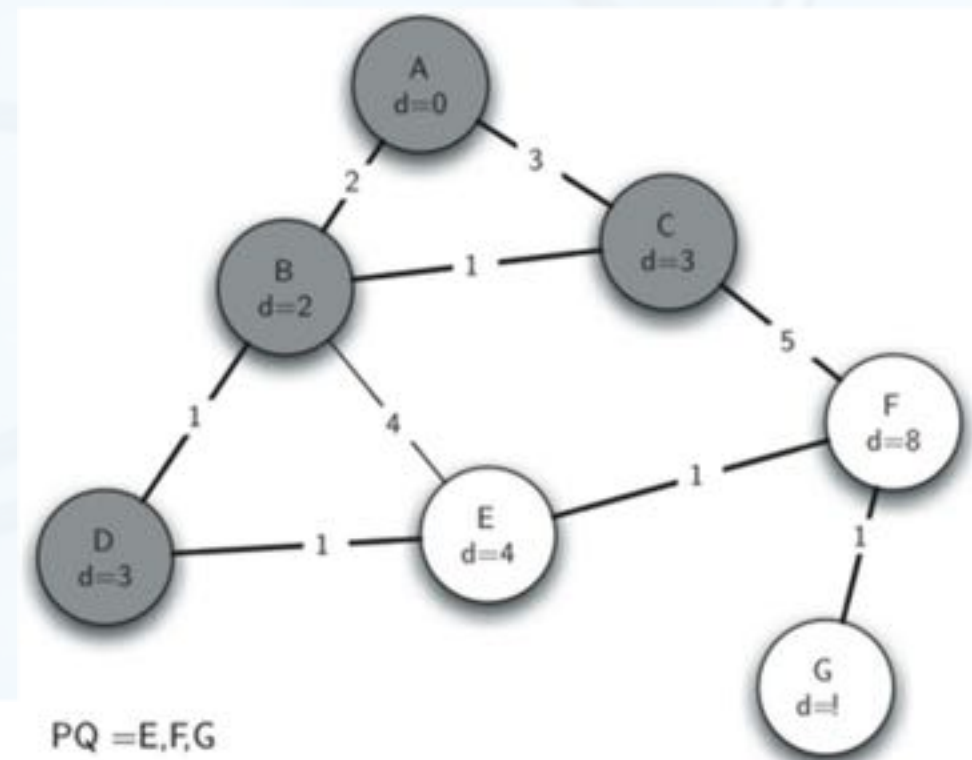
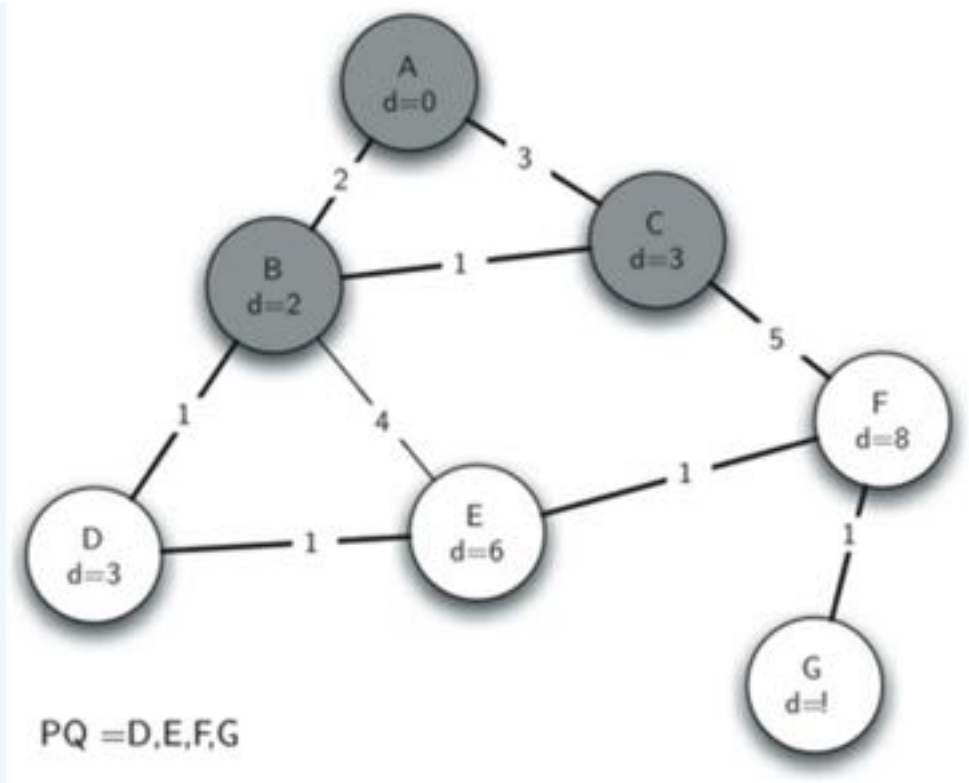
# 最小生成树：prim算法示例

数据结构与算法 (Python)



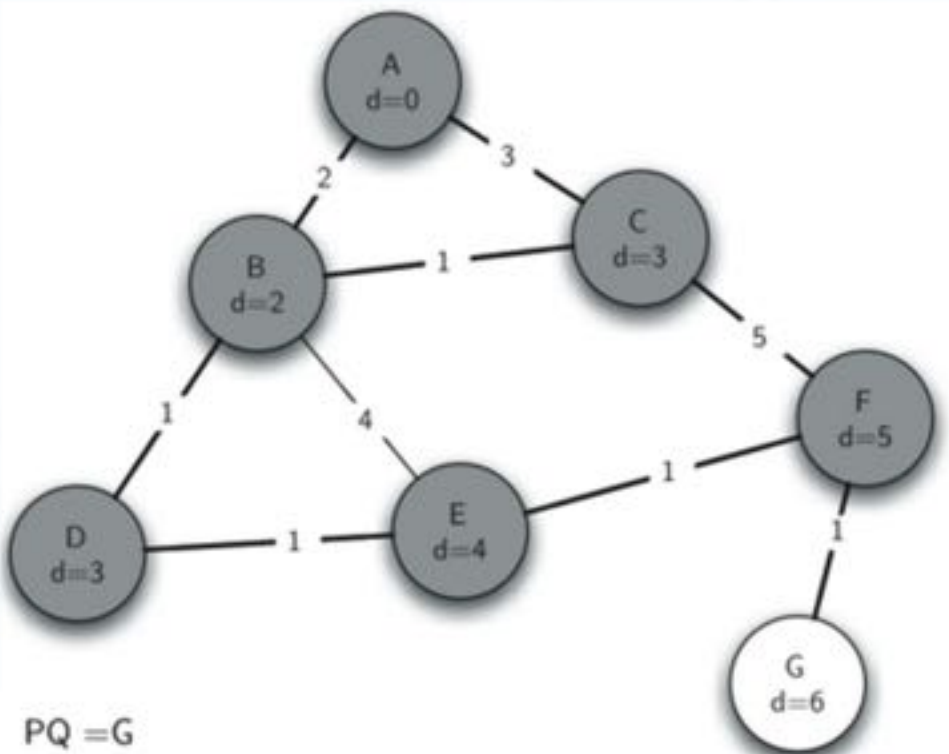
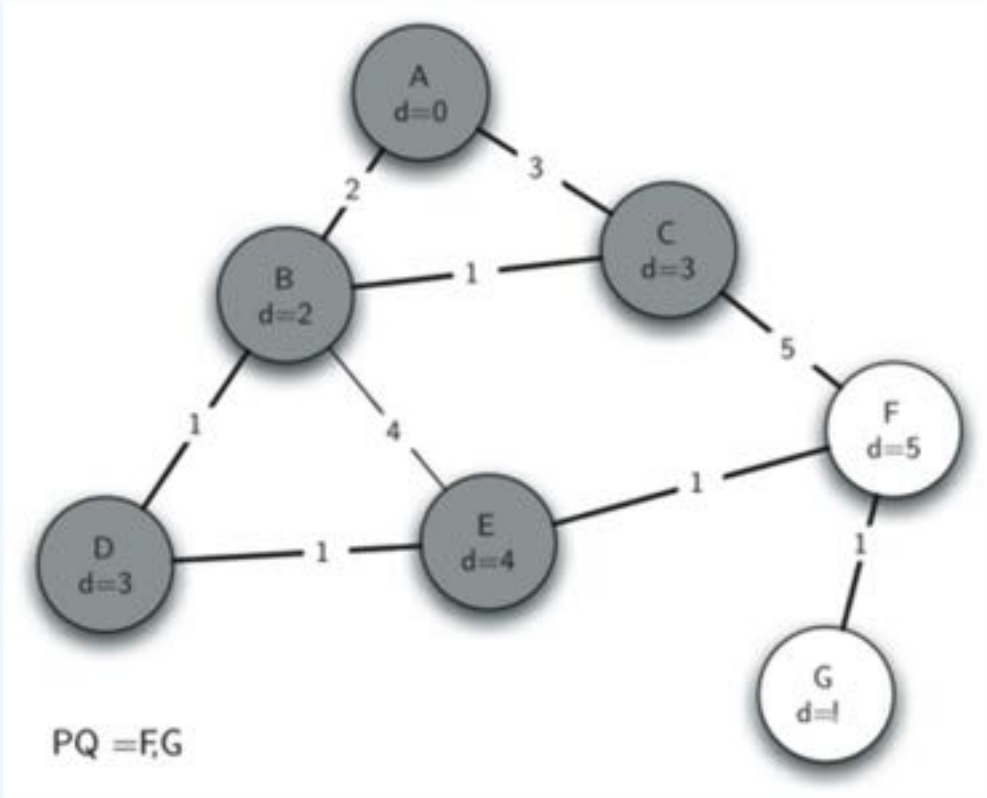
# 最小生成树：prim算法示例

数据结构与算法 (Python)

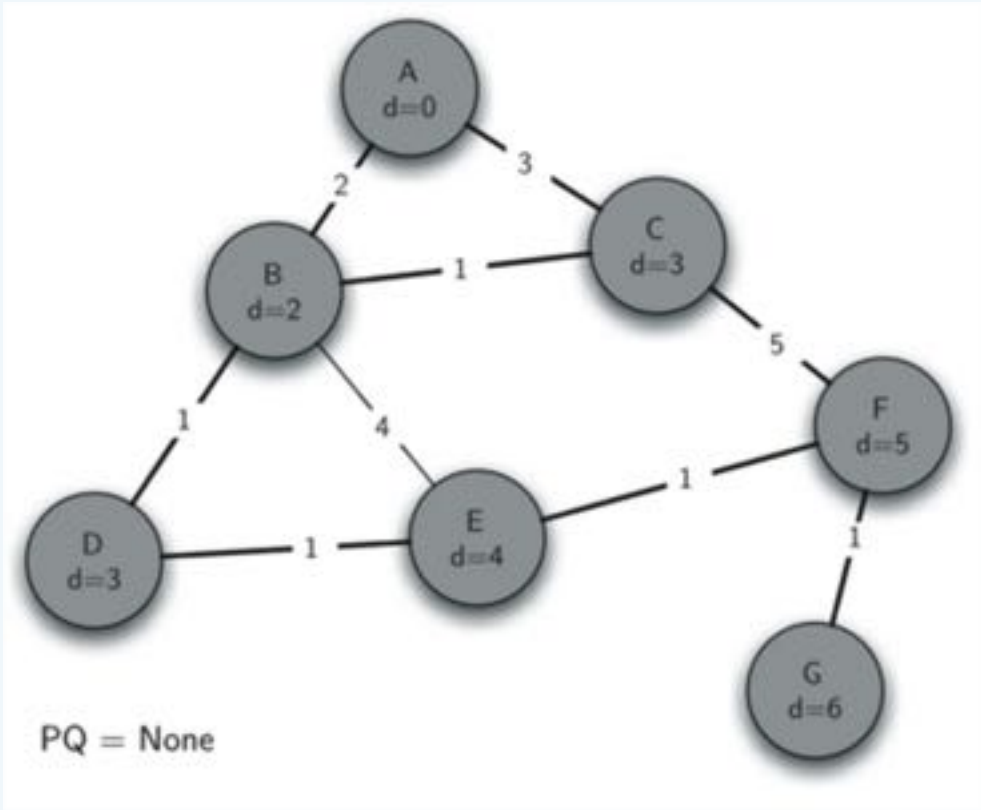


# 最小生成树：prim算法示例

数据结构与算法 (Python)



# 最小生成树：prim算法示例



# 本章总结

› 本章我们学习了图抽象数据类型，以及若干实现方法

› 本章讨论了一些图的算法和应用

广度优先搜索算法BFS，解决无权图的最短路径问题；

带权图的Dijkstra算法；

图的深度优先搜索算法；

用于简化图的强连通分支算法；

用于关联任务排序的拓扑排序算法；

用于广播消息的最小生成树算法。

# 参考文献

## › 从六度分隔到无尺度网络：

<http://www.socialbeta.com/articles/the-wisdom-of-sns-part-one.html>