目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） **课时安排: 2学时**

第十三讲：基本的图算法 **授课时间 :第十三周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容：（1）图的抽象数据类型：邻接矩阵、邻接表、实现；

（2）广度优先搜索；

（3）深度优先搜索

（4）最短路径问题

教学重点、难点：重点为深度优先搜索、广度优先搜索

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

第十三讲 基本的图算法

1. 图的概念

1.1 图的定义

定义1: 有序三元组$G=(V,E)$称为一个图,如果:
[1] $V=\{v_1,v_2,\ldots,v_3,v_4\}$是有限非空集，$V$称为顶点集, 其中的元素叫图G的顶点.
[2] E称为边集，其中的元素叫图G的边.
[3] P是从边集E到顶点集V中的有序或无序的元素 偶对构成集合的映射，称为关联函数.

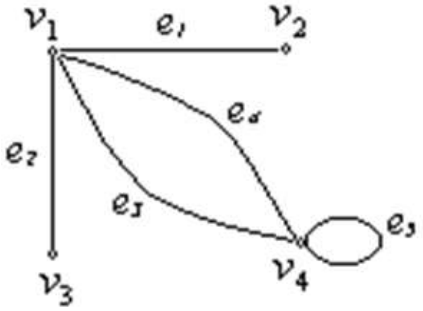
例1 设 $G=(V,E,\Psi)$,其中

$$V=\{v_1,v_2,v_3,v_4\}$$

$$E=\{e_1,e_2,e_3,e_4\}$$

$$\Psi(e_1)=v_1v_2,\ \Psi(e_2)=v_1v_3,\ \Psi(e_3)=v_1v_4,\ \Psi(e_4)=v_1v_4,\ \Psi(e_5)=v_4v_4,$$

G的图解如图



定义2: 在图G中，与V中的有序偶(v_i,v_j)对应的边e ,称为图的有向边(或弧),而与V中顶点的无序偶v_iv_j相对应的边e，称为图的无向边.每一条边都是无向边的图，叫无向图;每一条边都是有向边的图，称为有向图;既有无向边又有有向边的图称为混合图.

定义3: 若将图G的每一条边e都对应一个实数$w(e)$,则称$w(e)$为边的权，并称图G为赋权图。

规定用记号v和ε分别表示图的顶点数和边数.

例2 图 2 展示了简单加权有向图的另一示例。正式地，我们可以将该图表示为六个顶点的集合:

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

和 9 条边的集合

$$E = \left\{ \begin{array}{l} (V0, V1, 5), (V1, V2, 4), (V2, V3, 9), (V3, V4, 7), (V4, V0, 1), \\ (V0, V5, 2), (V5, V4, 8), (V3, V5, 3), (V5, V2, 1) \end{array} \right\}$$

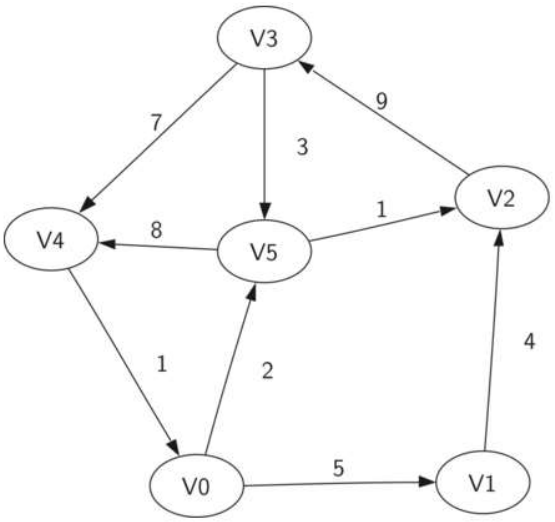


图 2 简单的带权有向图

图 2 中的示例图有助于说明两个其他关键图形术语：

路径： 图中的路径是由边连接的顶点序列。形式上，我们将定义一个路径为 $w_1, w_2, \ldots, w_n$，使得$(wi, wi+1) \in E$, 当 $1 \leq i \leq n-1$。未加权路径长度是路径中的边的数目，具体是 $n-1$ 。加权路径长度是路径中所有边的权重的总和。例如在 Figure 2中，从 V3 到 V1 的路径是顶点序列 $(V3, V4, V0, V1)$。边是 $\{(V3, V4, 7), (V4, V0, 1), (V0, V1, 5)\}$。

循环： 有向图中的循环是在同一顶点开始和结束的路径。例如，在 Figure 2中，路径 $(V5, V2, V3, V5)$是一个循环。没有循环的图形称为非循环图形。没有循环的有向图称为有向无环图或 DAG 。我们将看到，如果问题可以表示为 DAG，我们可以解决几个重要的问题。

常用术语:

(1) 端点相同的边称为环.

(2) 若一对顶点之间有两条以上的边联结，则这些边称为重边.

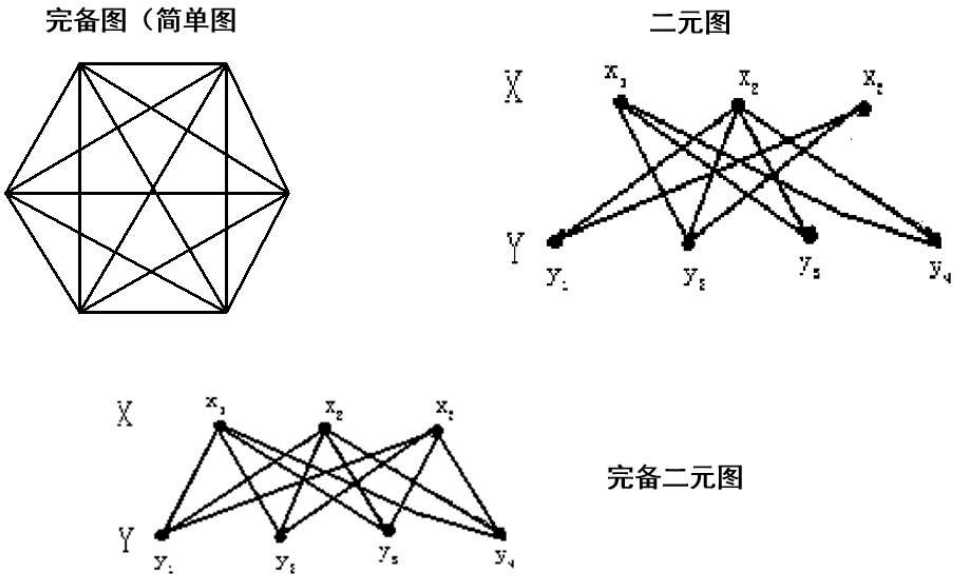
(3) 有边联结的两个顶点称为相邻的顶点，有一个公共端点的边称为相邻的边.

(4) 边和它的端点称为互相关联的.

(5) 既没有环也没有平行边的图，称为简单图.

(6) 任意两顶点都相邻的简单图，称为完备图，记为K_n，其中n为顶点的数目.

(7)若$V = X \bigcup Y, X \bigcap Y = \varphi$，且$X$中任两顶点不相邻，$Y$中任两顶点不相邻，则称$G$为二元图;若$X$中每一顶点皆与$Y$中一切顶点相邻，则$G$称为完备二元图，记为$K(m, r)$,其中$m, n$分别为$X$与$Y$的顶点数目.

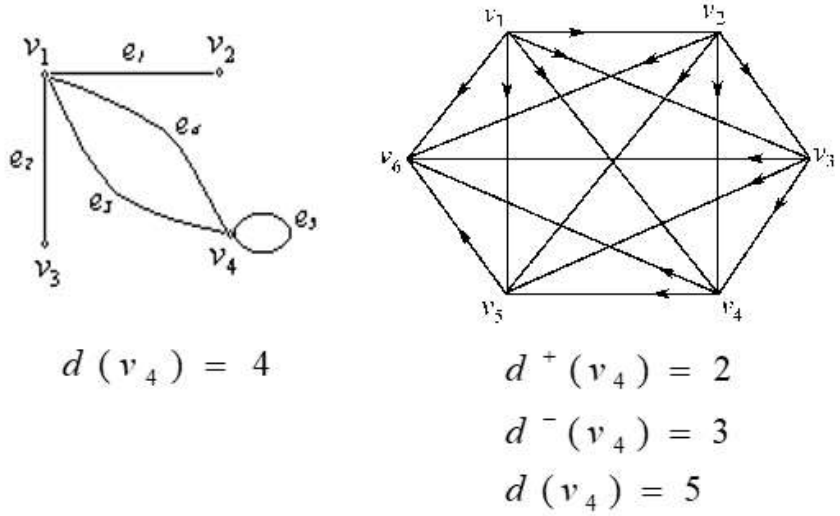


1.2 顶点的次数

定义 4:

(1)在无向图中，与顶点v关联的边的数目(环算两次)称为v的次数，记为$d(v)$.

(2)在有向图中，从顶点v引出的边的数目称为v的出度，记为$d^+(v)$， 从顶点v引入的边的数目称为v的入度，记为$d^-(v)$，$d(v)$=$d^+(v) + d^-(v)$称为v的次数.



$d(v_4) = 4$

$d^+(v_4) = 2$

$d^-(v_4) = 3$

$d(v_4) = 5$

定理 1 $\sum_{v \in V(G)} d(v) = 2\varepsilon(G)$

推论 1 任何图中奇次顶点的总数必为偶数.

例 3 在一次聚会中，认识奇数个人的人数一定是偶数

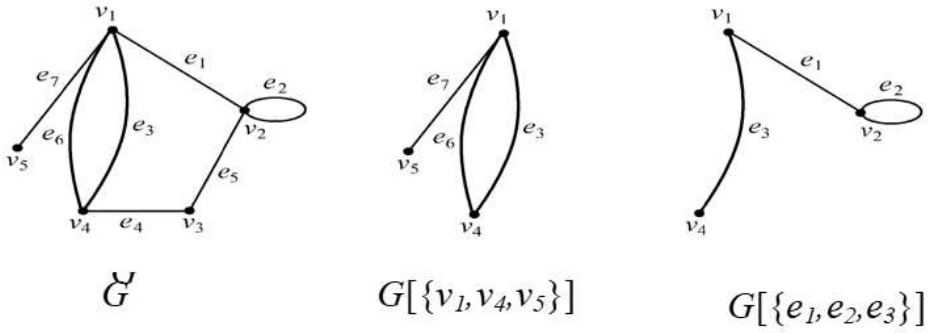
1.3 子图

定义 5: 设图 $G=(V,E,\Psi)$， 设 $G=(V_1,E_1,\Psi_1)$

(1) 若$V_1 \subseteq V, E_1 \subseteq E$， 且当$e \subset E_1$时， $\Psi_1(e)=\Psi(e)$，则称G_1是G的子图.特别的，若$V_1=V$,则G_1称为G的生成子图.

(2) 设$V_1 \subseteq V$,且$V1 \neq \emptyset$，以V为顶点集、两个端点都在V_1.图G的边为边集的图G的子图，称为G的由V_1导出的子图，记为$G[V_1]$.

(3) 设$E_1 \subseteq E$,且$E1 \neq \emptyset$,以E_1为边集,E_1的端点集为顶点集的图G的子图，称为G的由E_1导出的子图，记为$G[E_1]$.



2. 图抽象数据类型

图抽象数据类型（ADT）定义如下:

- Graph() 创建一个新的空图。
- addVertex(vert) 向图中添加一个顶点实例。
- addEdge(fromVert, toVert) 向连接两个顶点的图添加一个新的有向边。
- addEdge(fromVert, toVert, weight) 向连接两个顶点的图添加一个新的加权的有向边。
- getVertex(vertKey) 在图中找到名为 `vertKey` 的顶点。
- getVertices() 返回图中所有顶点的列表。
- in 返回 True 如果 `vertex in graph` 里给定的顶点在图中，否则返回False。

从图的正式定义开始，我们有几种方法可以在 Python 中实现图 ADT。 我们将看到在使用不同的表示来实现上述 ADT 时存在权衡。 有两个众所周知的图形、实现，邻接矩阵 和 邻接表。 我们将解释这两个选项，然后实现一个作为 Python 类。

2.1.邻接矩阵

实现图的最简单的方法之一是使用二维矩阵。在该矩阵实现中，每个行和列表示图中的顶点。存储在行 v 和列 w 的交叉点处的单元中的值表示是否存在从顶点 v 到顶点 w 的边。 当两个顶点通过边连接时，我们说它们是相邻的。 图 3 展示了图 2 中的图的邻接矩阵。单元格中的值表示从顶点 v 到顶点 w 的边的权重。

图 3 邻接矩阵示例

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

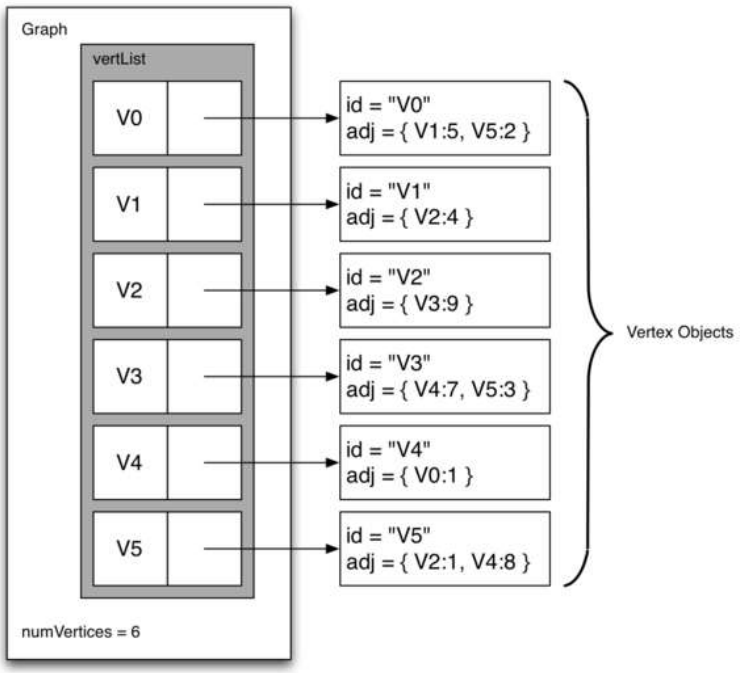
邻接矩阵的优点是简单，对于小图，很容易看到哪些节点连接到其他节点。 然而，注意矩阵中的大多数单元格是空的。 因为大多数单元格是空的，我们说这个矩阵是“稀疏的”。矩阵不是一种非常有效的方式来存储稀疏数据。 事实上，在Python中，你甚至要创建一个如图 3所示的矩阵结构。

当边的数量大时，邻接矩阵是图的良好实现。但是什么是大？填充矩阵需要多少边？ 由于图中每个顶点有一行和一列，填充矩阵所需的边数为$|V|^2$。 当每个顶点连接到每个其他顶点时，矩阵是满的。有几个真实的问题，接近这种连接。 我们在本章中讨论的问题都涉及稀疏连接的图。

2.2.邻接表

实现稀疏连接图的更空间高效的方法是使用邻接表。在邻接表实现中，我们保存Graph 对象中的所有顶点的主列表，然后图中的每个顶点对象维护连接到的其他顶点的列表。 在我们的顶点类的实现中，我们将使用字典而不是列表，其中字典键是顶点，值是权重。图 4 展示了图2中的图的邻接列表示。

图 4 邻接表示例



邻接表实现的优点是它允许我们紧凑地表示稀疏图。 邻接表还允许我们容易找到直接连接到特定顶点的所有链接。

2.3. python代码实现

使用字典，很容易在 Python 中实现邻接表。在我们的 Graph 抽象数据类型的实现中，我们将创建两个类（见 Listing 1和 Listing 2），Graph（保存顶点的主列表）和 Vertex（将表示图中的每个顶点）。

每个顶点使用字典来跟踪它连接的顶点和每个边的权重。这个字典称为connectedTo 。 下面的列表展示了 Vertex 类的代码。构造函数只是初始化 id ， 通常是一个字符串和 connectedTo 字典。 addNeighbor方法用于从这个顶点添加一个连接到另一个。 getConnections 方法返回邻接表中的所有顶点，如 connectedTo 实例变量所示。 getWeight 方法返回从这个顶点到作为参数传递的顶点的边的权重。

Listing 1:

In [16]:

```
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.connecte
dTo])

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

    def getId(self):
        return self.id

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

下一个列表中显示的 Graph 类包含将顶点名称映射到顶点对象的字典。在图 4中，该字典对象由阴影灰色框表示。 Graph 还提供了将顶点添加到图并将一个顶点连接到另一个顶点的方法。getVertices 方法返回图中所有顶点的名称。此外，我们实现了 __iter__ 方法，以便轻松地遍历特定图中的所有顶点对象。 这两种方法允许通过名称或对象本身在图形中的顶点上进行迭代。

Listing 1:

In [17]:

```
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

    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())
```

使用刚才定义的 Graph 和 Vertex 类，下面的 Python 会话创建图 2中的图。首先我们创建 6 个编号为 0 到 5 的顶点。然后我们展示顶点字典。 注意，对于每个键 0 到 5，我们创建了一个顶点的实例。接下来，我们添加将顶点连接在一起的边。 最后，嵌套循环验证图中的每个边缘是否正确存储。 你应该在本会话结束时根据图 2检查边列表的输出是否正确。

In [18]:

```
g = Graph()
for i in range(6):
    g.addVertex(i)
print(g.vertList)
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: <__main__.Vertex object at 0x0000019B41CBC7F0>, 1: <__main__.Ver
tex object at 0x0000019B41CBCA90>, 2: <__main__.Vertex object at 0x0
000019B41CBCAC8>, 3: <__main__.Vertex object at 0x0000019B41CBCB00>,
4: <__main__.Vertex object at 0x0000019B41CBCB38>, 5: <__main__.Vert
ex object at 0x0000019B41CBCB70>}
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```

3. 广度优先搜索

3.1 字梯的问题

让我们从下面的叫字梯的难题开始图算法研究。将单词 “FOOL” 转换为单词 “SAGE”。 在字梯中你通过改变一个字母逐渐发生变化。 在每一步，你必须将一个字变换成另一个字。 字梯益智游戏是刘易斯卡罗尔 1878 年发明的，爱丽丝梦游仙境的作者。下面的单词序列示出了对上述问题的一种可能的解决方案。

FOOL
POOL
POLL
POLE
PALE
SALE
SAGE

有许多关于字梯问题的变种。例如，可能附加了完成转换的特定数量的步骤，或者可能需要使用特定的词。在本节中，我们将计算起始字转换为结束字所需的最小转换次数。

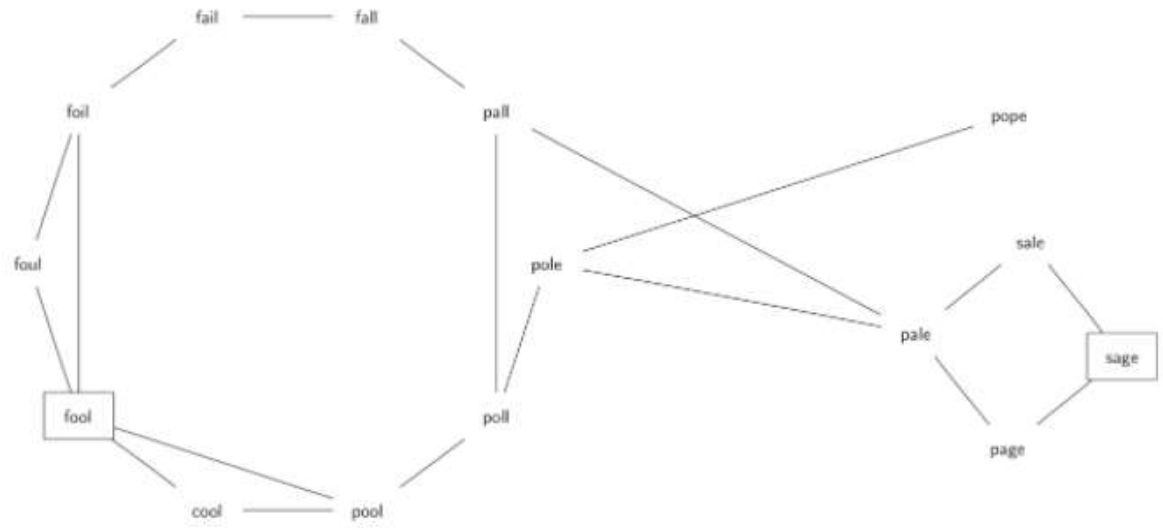
毫不奇怪，因为这一章是图，我们可以使用图算法解决这个问题。 这里是我们需要的步骤：

- 将字之间的关系表示为图。
- 使用称为广度优先搜索的图算法来找到从起始字到结束字的有效路径。

3.2 构建字梯图

我们的第一个问题是弄清楚如何将大量的单词集合转换为图。 如果两个词只有一个字母不同，我们就创建从一个词到另一个词的边。如果我们可以创建这样的图，则从一个词到另一个词的任意路径就是词梯子拼图的解决方案。 Figure 1展示了一些解决 FOOL 到 SAGE 字梯问题的单词的小图。 请注意，图是无向图，边未加权。

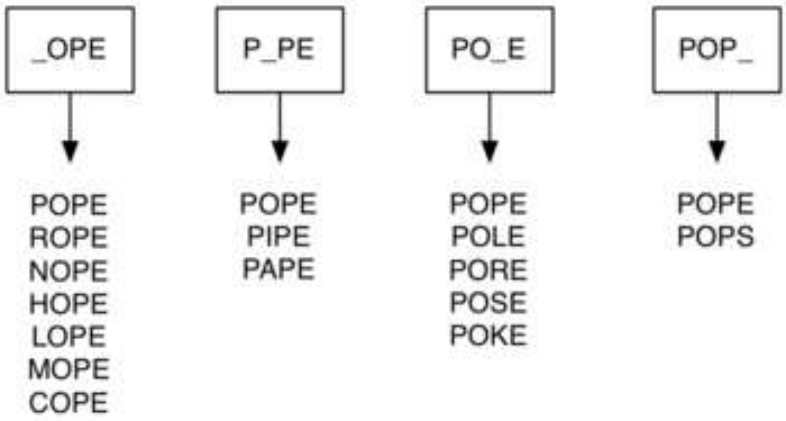
图 5 邻接表示例



我们可以使用几种不同的方法来创建解决这个问题的图。假设我们有一个长度相同的单词列表。作为起点，我们可以在图中为列表中的每个单词创建一个顶点。为了弄清楚如何连接单词，我们可以比较列表中的每个单词。比较时我们看有多少字母是不同的。如果所讨论的两个字只有一个字母不同，我们可以在图中创建它们之间的边。对于小的列表，这种方法会正常工作;然而假设我们有一个 $5,110$ 词的列表。粗略地说，将一个字与列表上的每个其他词进行比较是 $O(n^2)$。对于5110 个词，n^2 是超过2600万的比较。

我们可以通过以下方法做得更好。假设我们有大量的桶，每个桶在外面有一个四个字母的单词，除了标签中的一个字母已经被下划线替代。例如，看 Figure 2，我们可能有一个标记为 “pop” *的桶。当我们处理列表中的每个单词时，我们使用* “\” 作为通配符比较每个桶的单词，所以 “pope” 和 “pops “ 将匹配 ”pop_“。每次我们找到一个匹配的桶，我们就把单词放在那个桶。一旦我们把所有单词放到适当的桶中，就知道桶中的所有单词必须连接。

图 6 邻接表示例



在 Python 中，我们使用字典来实现我们刚才描述的方案。我们刚才描述的桶上的标签是我们字典中的键。该键存储的值是单词列表。 一旦我们建立了字典，我们可以创建图。 我们通过为图中的每个单词创建一个顶点来开始图。 然后，我们在字典中的相同键下找到的所有顶

In [30]:

```
from pythonds.graphs import Graph

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
```

因为这是我们的第一个真实世界图问题，你可能想知道图是如何稀疏？这个问题的四个字母的单词列表是 5,110 字长。 如果我们使用邻接矩阵，则矩阵将具有5,110 * 5,110 = 26,112,100 个格。 由 buildGraph 函数构造的图正好有 53,286 个边，所以矩阵只有 0.20% 的单元格填充！ 这是一个非常稀疏的矩阵。

3.3 实现广度优先搜索

通过构建图，我们现在可以将注意力转向我们将使用的算法来找到字梯问题的最短解。我们将使用的图算法称为“宽度优先搜索”算法。宽度优先搜索（BFS）是用于搜索图的最简单的算法之一。它也作为几个其他重要的图算法的原型，我们将在以后研究。

给定图 G 和起始顶点 s，广度优先搜索通过探索图中的边以找到 G 中的所有顶点，其中存在从 s 开始的路径。通过广度优先搜索，它找到和 s 相距 k 的所有顶点，然后找到距离为 k + 1 的所有顶点。可视化广度优先搜索算法一个好方法是想象它正在建一棵树，一次建一层。广度优先搜索先从其他起始顶点开始添加它的所有子节点，然后再添加其子节点的子节点。

为了跟踪进度，BFS 将每个顶点着色为白色，灰色或黑色。当它们被构造时，所有顶点被初始化为白色。白色顶点是未发现的顶点。当一个顶点最初被发现时它变成灰色的，当 BFS 完全探索完一个顶点时，它被着色为黑色。这意味着一旦顶点变黑色，就没有与它相邻的白色顶点。另一方面，灰色节点可能有与其相邻的一些白色顶点，表示仍有额外的顶点要探索。

下面 Listing 2 中所示的广度优先搜索算法使用我们先前开发的邻接表表示。此外，它使用一个 Queue，一个关键的地方，决定下一个探索的顶点。

此外，BFS 算法使用 Vertex 类的扩展版本。这个新的顶点类添加了三个新的实例变量：distance，predecessor和 color 。这些实例变量中的每一个还具有适当的 getter 和 setter 方法。这个扩展的顶点类代码包含在pythonds包中，但我们不会在这里展示它，因为没有新的需要学习的点。

BFS 从起始顶点开始，颜色从灰色开始，表明它正在被探索。另外两个值，即距离和前导，对于起始顶点分别初始化为 0 和 None 。最后，放到一个队列中。下一步是开始系统地检查队列前面的顶点。我们通过迭代它的邻接表来探索队列前面的每个新节点。当检查邻接表上的每个节点时，检查其颜色。如果它是白色的，顶点是未开发的，有四件事情发生：

1. 新的，未开发的顶点 nbr，被着色为灰色。
2. nbr 的前导被设置为当前节点 currentVert
3. 到 nbr 的距离设置为到 currentVert + 1 的距离
4. nbr 被添加到队列的末尾。 将 nbr 添加到队列的末尾有效地调度此节点以进行进一步探索，但不是直到 currentVert 的邻接表上的所有其他顶点都被探索。

```
from pythonds.graphs import Graph, Vertex from pythonds.basic import Queue

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')
```

让我们看看 bfs 函数如何构造对应于 Figure 1 中的图的广度优先树。开始我们取所有与 `fool` 相邻的节点，并将它们添加到树中。 相邻节点包括 `pool`, `foil`, `foul`, `cool`。 这些节点被添加到新节点的队列以进行扩展。 Figure 3 展示了在此步骤之后树以及队列的状态。

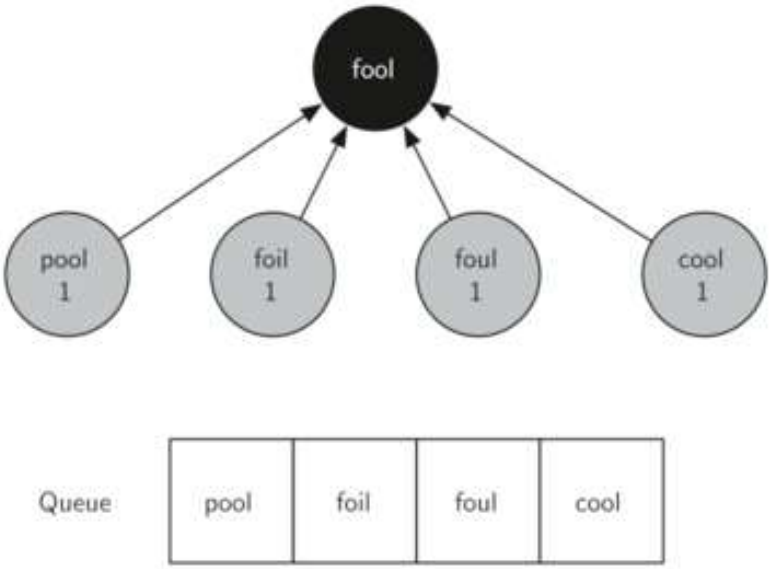


Figure 3

在下一步骤中，bfs 从队列的前面删除下一个节点（`pool`），并对其所有相邻节点重复该过程。 然而，当 bfs 检查节点 `cool` 时，它发现 `cool` 的颜色已经改变为灰色。这表明有一条较短的路径到 `cool`，并且 `cool` 已经在队列上进一步扩展。在检查 `pool` 期间添加到队列的唯一新节点是 `poll`。 树和队列的新状态如 Figure 4所示。

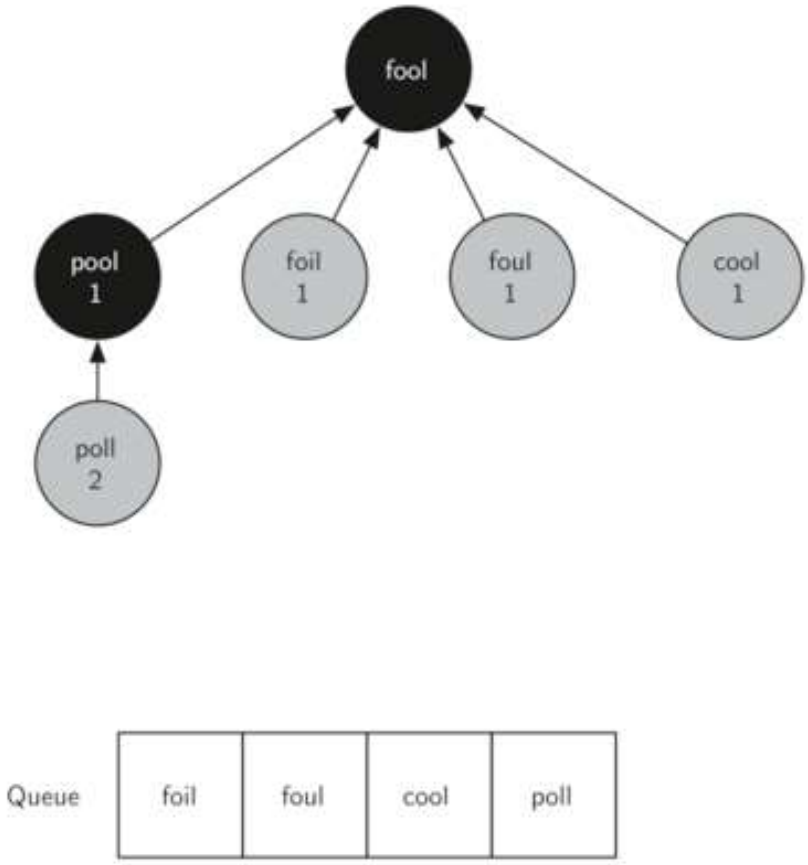
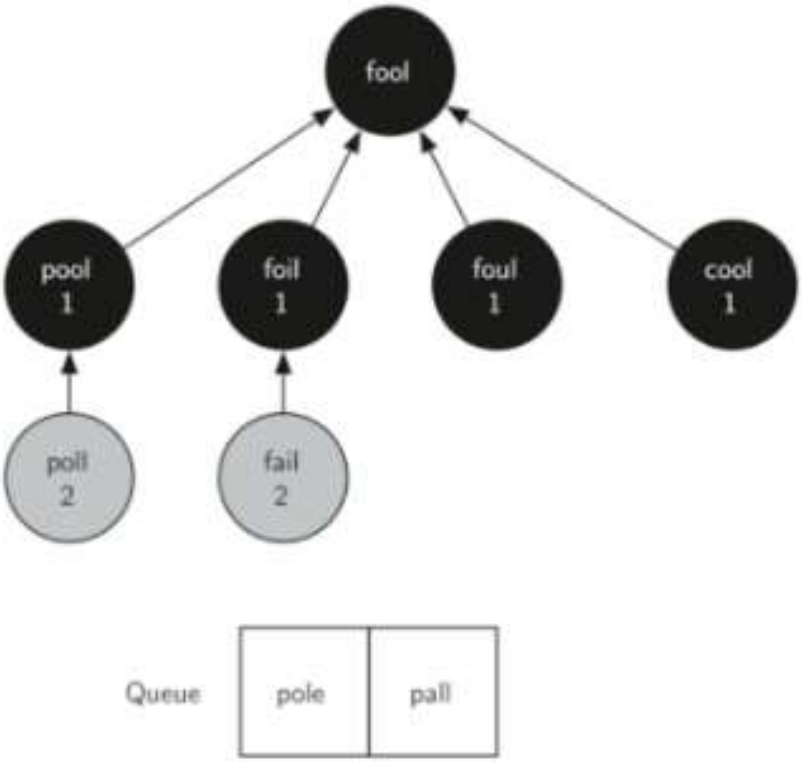
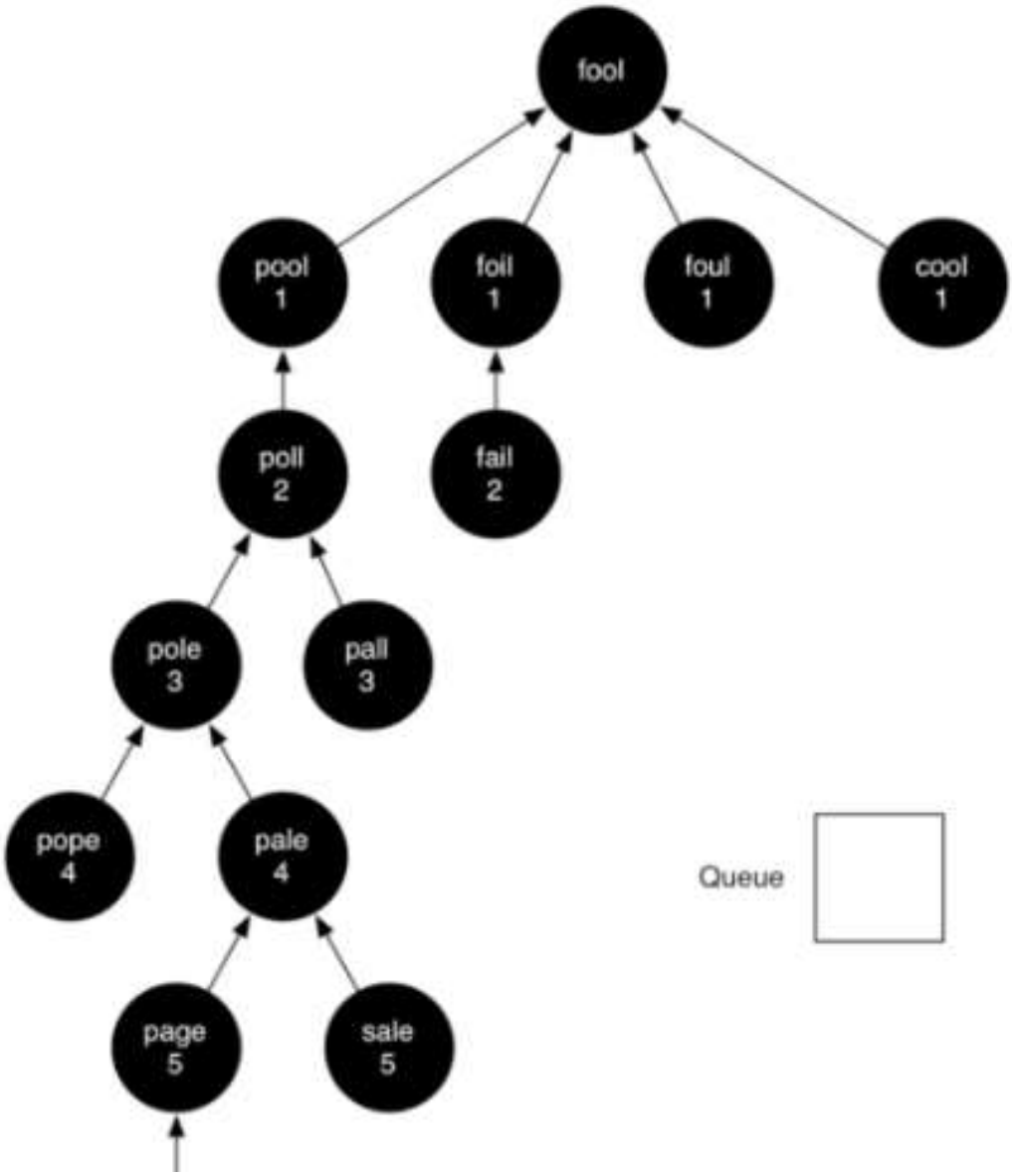


Figure 4

队列上的下一个顶点是 `foil`。 `foil` 可以添加到树中的唯一新节点是 `fail`。 当 bfs 继续处理队列时，接下来的两个节点都不向队列或树添加新内容。 Figure 5 展示了在树的第二级上展开所有顶点之后的树和队列。


fool
pool
1
foil
1
foul
1
cool
1
poll
2
fail
2

Queue
pole
pall


fool
pool
1
foil
1
foul
1
cool
1
poll
2
fail
2
pole
3
pall
3
pope
4
pale
4
page
5
sale
5
Queue


```
def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())

traverse(g.getVertex('sage'))
```

3.4 广度优先搜索分析

在继续使用其他图算法之前，让我们分析广度优先搜索算法的运行时性能。首先要观察的是，对于图中的每个顶点 $|V|$ 最多执行一次 while 循环。因为一个顶点必须是白色，才能被检查和添加到队列。这给出了用于 while 循环的 $O(V)$。嵌套在 while 内部的 for 循环对于图中的每个边执行最多一次，$|E|$。原因是每个顶点最多被出列一次，并且仅当节点 u 出队时，我们才检查从节点 u 到节点 v 的边。这给出了用于 for 循环的 $O(E)$ 。组合这两个环路给出了 $O(V+E)$。

当然做广度优先搜索只是任务的一部分。从起始节点到目标节点的链接之后是任务的另一部分。最糟糕的情况是，如果图是单个长链。在这种情况下，遍历所有顶点将是 $O(V)$。正常情况将是 $|V|$ 的一小部分但我们仍然写 $O(V)$。

最后，至少对于这个问题，存在构建初始图形所需的时间。我们把 buildGraph 函数的分析作为一个练习。

3.5 广度优先搜索的完整程序

广度优先搜索和深度优先搜索是图遍历的两种算法，广度和深度的区别在于对节点的遍历顺序不同。广度优先算法的遍历顺序是由近及远，先看到的节点先遍历。接下来使用python实现广度优先搜索并找到最短路径：

In [1]:

```
from collections import deque
from collections import namedtuple


def bfs(start_node, end_node, graph):  # 开始节点  目标节点 图字典
    node = namedtuple('node', 'name, from_node')    # 使用namedtuple定义节点，用于存储前置节点
    search_queue = deque()  # 使用双端队列，这里当作队列使用，根据先进先出获取下一个遍历的节点
    name_search = deque()   # 存储队列中已有的节点名称
    visited = {}            # 存储已经访问过的节点

    search_queue.append(node(start_node, None))  # 填入初始节点，从队列后面加入
    name_search.append(start_node)               # 填入初始节点名称
    path = []               # 用户回溯路径
    path_len = 0            # 路径长度

    print('开始搜索...')
    while search_queue:     # 只要搜索队列中有数据就一直遍历下去
        print('待遍历节点: ', name_search)
        current_node = search_queue.popleft()  # 从队列前边获取节点，即先进先出，这是BFS的核心
        name_search.popleft()                  # 将名称也相应弹出
        if current_node.name not in visited:   # 当前节点是否被访问过
            print('当前节点: ', current_node.name, end=' | ')
            if current_node.name == end_node:  # 退出条件，找到了目标节点，接下来执行路径回溯和长度计算
                pre_node = current_node        # 路径回溯的关键在于每个节点中存储的前置节点
                while True:                    # 开启循环直到找到开始节点
                    if pre_node.name == start_node:  # 退出条件：前置节点为开始节点
                        path.append(start_node)      # 退出前将开始节点也加入路径，保证路径的完整性
                        break
                    else:
                        path.append(pre_node.name)   # 不断将前置节点名称加入路径
                        pre_node = visited[pre_node.from_node]  # 取出前置节点的前置节点，依次类推
                path_len = len(path) - 1       # 获得完整路径后，长度即为节点个数-1
                break
            else:
                visited[current_node.name] = current_node   # 如果没有找到目标节点，将节点设为已访问，并将相邻节点加入搜索队列，继续找下去
                for node_name in graph[current_node.name]:  # 遍历相邻节点，判断相
```

```
        # 邻节点是否已经在搜索队列
                    if node_name not in name_search:          # 如果相邻节点不在搜索队列则进行添加
                        search_queue.append(node(node_name, current_node.name))
                        name_search.append(node_name)
    print('搜索完毕,最短路径为:', path[::-1], "长度为:", path_len)  # 打印搜索结果


if __name__ == "__main__":

    graph = dict()      # 使用字典表示有向图
    graph[1] = [3, 2]
    graph[2] = [5]
    graph[3] = [4, 7]
    graph[4] = [6]
    graph[5] = [6]
    graph[6] = [8]
    graph[7] = [8]
    graph[8] = []
    bfs(1, 8, graph)    # 执行搜索
```

```
开始搜索...
待遍历节点:  deque([1])
当前节点:  1 | 待遍历节点:  deque([3, 2])
当前节点:  3 | 待遍历节点:  deque([2, 4, 7])
当前节点:  2 | 待遍历节点:  deque([4, 7, 5])
当前节点:  4 | 待遍历节点:  deque([7, 5, 6])
当前节点:  7 | 待遍历节点:  deque([5, 6, 8])
当前节点:  5 | 待遍历节点:  deque([6, 8])
当前节点:  6 | 待遍历节点:  deque([8])
当前节点:  8 | 搜索完毕,最短路径为: [1, 3, 7, 8] 长度为: 3
```

4. 深度优先搜索

4.1 骑士之旅

另一个经典问题，我们可以用来说明第二个通用图算法称为 “骑士之旅”。骑士之旅图是在一个棋盘上用一个棋子当骑士玩。图的目的是找到一系列的动作，让骑士访问板上的每格一次。一个这样的序列被称为“旅游”。骑士的旅游难题已经吸引了象棋玩家，数学家和计算机科学家多年。一个 8×8 棋盘的可能的游览次数的上限为 1.305×10^{35} ;然而，还有更多可能的死胡同。显然，这是一个需要脑力，计算能力，或两者都需要的问题。

虽然研究人员已经研究了许多不同的算法来解决骑士的旅游问题，图搜索是最容易理解的程序之一。再次，我们将使用两个主要步骤解决问题：

- 表示骑士在棋盘上作为图的动作。
- 使用图算法来查找长度为 $rows \times columns - 1$ 的路径，其中图上的每个顶点都被访问一次。

4.2 构建骑士之旅图

为了将骑士的旅游问题表示为图，我们将使用以下两个点：棋盘上的每个正方形可以表示为图形中的一个节点。 骑士的每个合法移动可以表示为图形中的边。 Figure 1 展示了骑士的移动以及图中的对应边。

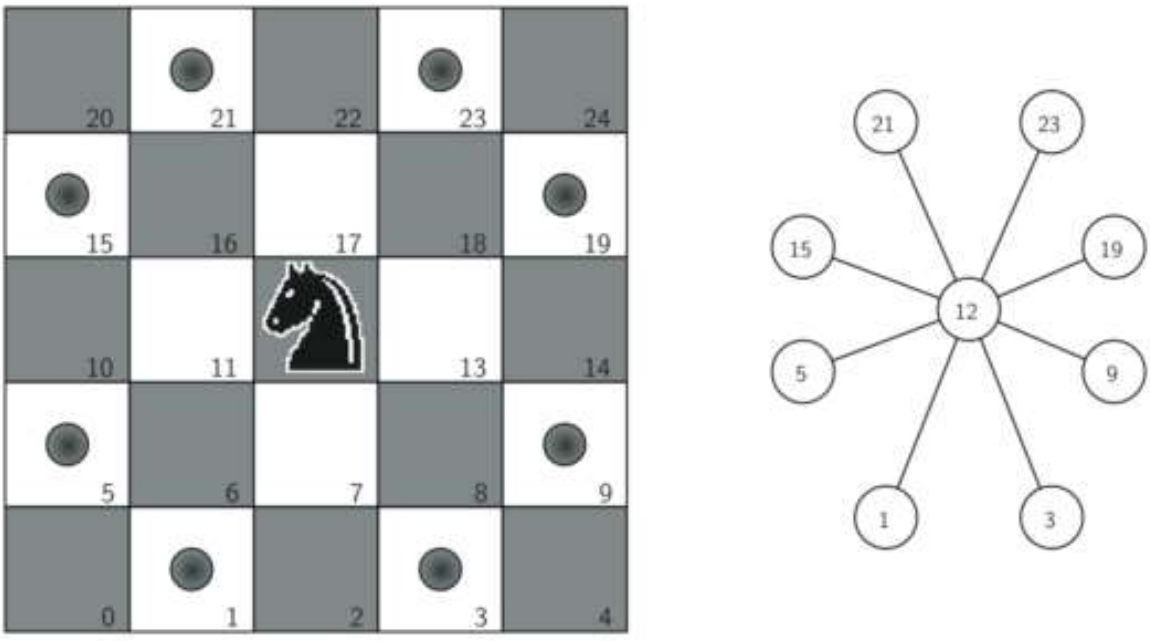


Figure 1

要构建一个 n*n 的完整图，我们可以使用 Listing 1 中所示的 Python 函数。knightGraph 函数在整个板上进行一次遍历。 在板上的每个方块上，knightGraph 函数调用 genLegalMoves ，为板上的位置创建一个移动列表。 所有移动在图形中转换为边。 另一个帮助函数 posToNodeId 按照行和列将板上的位置转换为类似于 Figure 1 所示的顶点数的线性顶点数。

```
from pythonds.graphs import Graph

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, column, board_size):
    return (row * board_size) + column
```

Listing 1

genLegalMoves 函数（Listing 2）使用板上骑士的位置，并生成八个可能移动中的一个。 legalCoord 辅助函数（Listing 2）确保生成的特定移动仍在板上。

```
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
```

Listing 2

Figure 2 展示了一个

8×8

板的可能移动的完整图。图中有正好 336 个边。 注意，与板的边相对应的顶点具有比板中间的顶点更少的连接（移动数）。 再次我们可以看到图的稀疏。 如果图形完全连接，则会有 4,096 个边。 由于只有336 个边，邻接矩阵只有 8.2% 填充率。

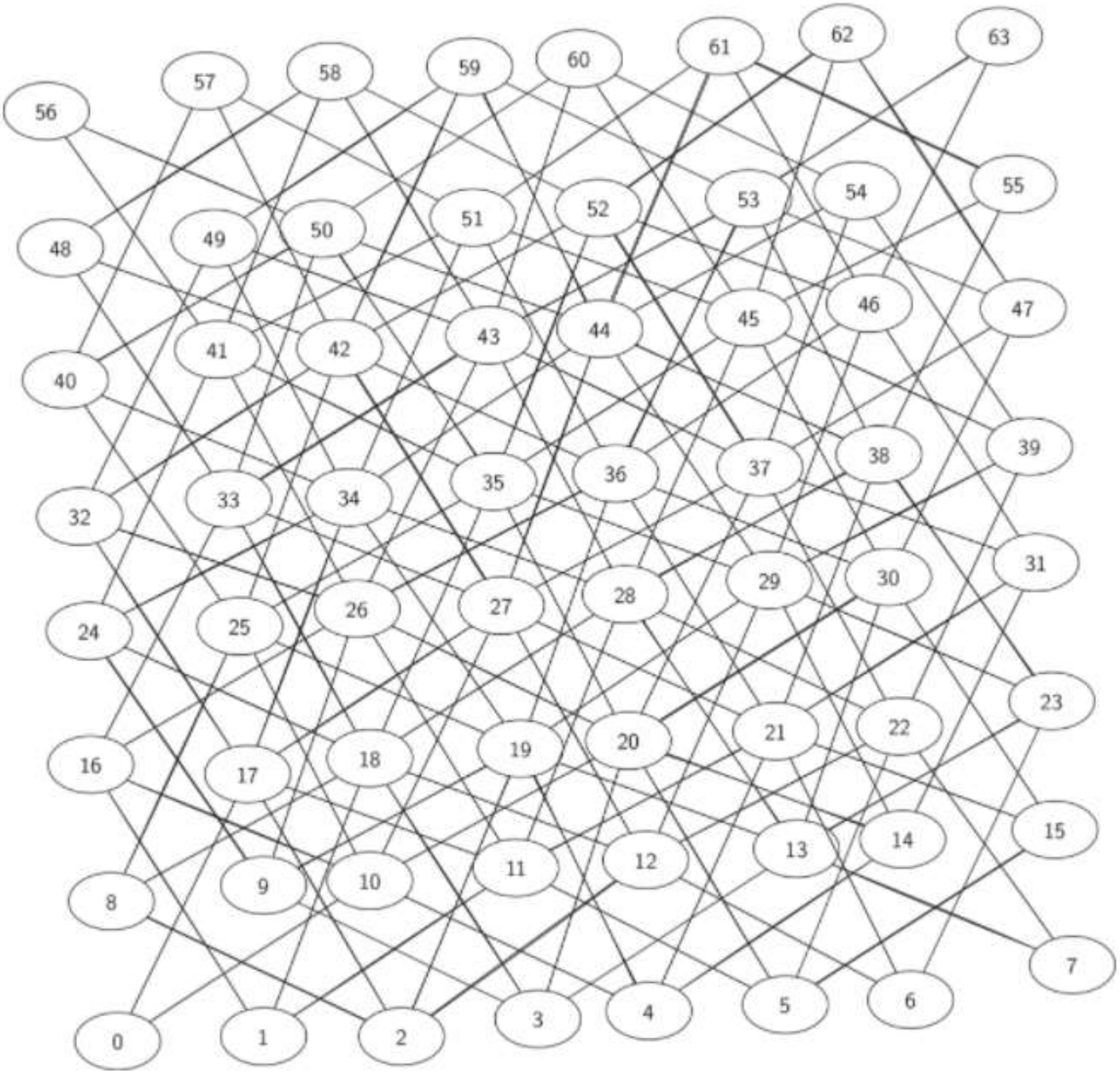


Figure 2

4.3 实现骑士之旅

我们将用来解决骑士旅游问题的搜索算法称为 深度优先搜索（DFS）。尽管在前面部分中讨论的广度优先搜索算法一次建立一个搜索树，但是深度优先搜索通过尽可能深地探索树的一个分支来创建搜索树。在本节中，我们将介绍实现深度优先搜索的两种算法。我们将看到的第一个算法通过明确地禁止一个节点被访问多次来直接解决骑士的旅行问题。第二种实现是更通用的，但是允许在构建树时多次访问节点。第二个版本在后续部分中用于开发其他图形算法。

图的深度优先搜索正是我们需要的，来找到有 63 个边的路径。我们将看到，当深度优先搜索算法找到死角（图中没有可移动的地方）时，它将回到下一个最深的顶点，允许它进行移动。

knightTour 函数有四个参数： n ， 搜索树中的当前深度; path，到此为止访问的顶点的列表; u，图中我们希望探索的顶点; limit 路径中的节点数。 knightTour 函数是递归的。当调用 knightTour 函数时，它首先检查基本情况。如果我们有一个包含 64 个顶点的路径，我们状态为 True 的 knightTour 返回，表示我们找到了一个成功的线路。如果路径不够长，我们继续通过选择一个新的顶点来探索一层，并对这个顶点递归调用knightTour。

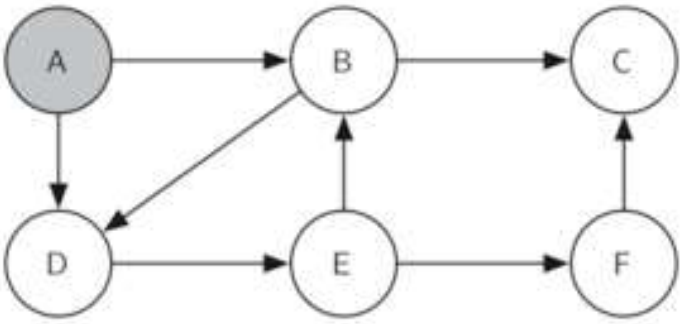
DFS 还使用颜色来跟踪图中的哪些顶点已被访问。未访问的顶点是白色的，访问的顶点是灰色的。如果已经探索了特定顶点的所有邻居，并且我们尚未达到64个顶点的目标长度，我们已经到达死胡同。当我们到达死胡同时，我们必须回溯。当我们从状态为 False 的 knightTour 返回时，发生回溯。在广度优先搜索中，我们使用一个队列来跟踪下一个要访问的顶点。由于深度优先搜索是递归的，我们隐式使用一个栈来帮助我们回溯。当我们从第 11 行的状态为 False 的knightTour 调用返回时，我们保持在 while 循环中，并查看 nbrList 中的下一个顶点。

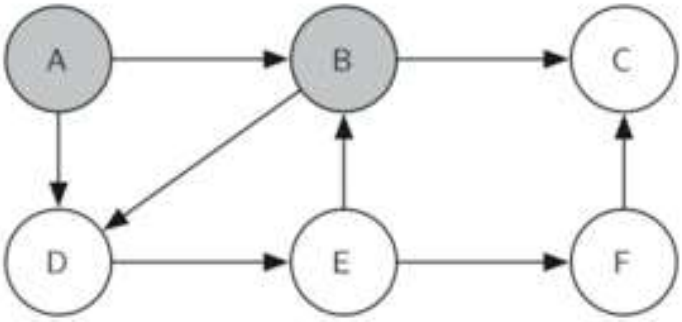
```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
            nbrList = list(u.getConnections())
            i = 0
            done = False
            while i < len(nbrList) and not done:
                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
```

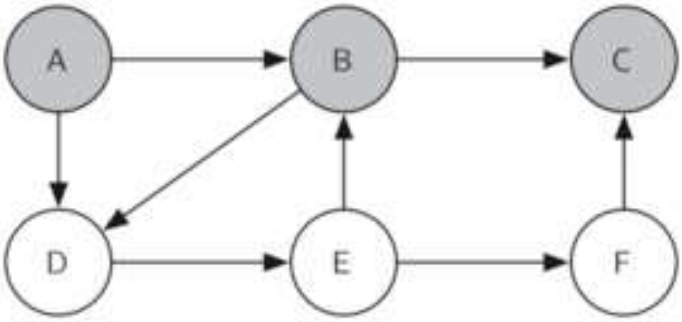
Listing 3

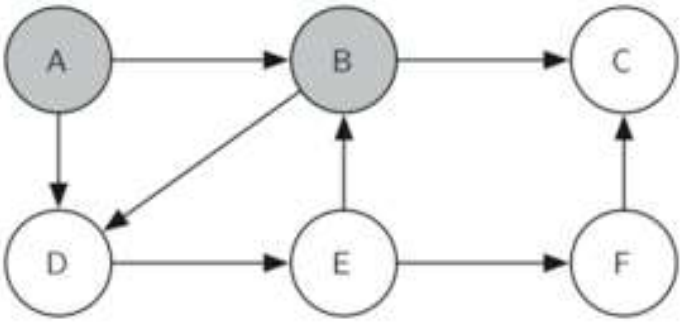
让我们看看一个简单的例子 knightTour 。你可以按照搜索的步骤参考下面的图。对于这个例子，我们假设对第 6 行的 getConnections 方法的调用按字母顺序对节点排序。我们首先调用 knightTour(0, path, A, 6)

Figure 中 knightTour 从节点 A 开始.与 A 相邻的节点是 B 和 D。由于 B 在字母 D 之前，DFS 选择 B 展开下一个，如 Figure 4 所示。当knightTour 被递归调用时，开始从 B 开始探寻。 B 与 C 和 D 相邻，所以 knightTour 选择接下来探索 C。然而，如 Figure 5 所示，节点 C 是没有相邻节点的死胡同。此时，我们将节点 C 的颜色更改为白色。对 knightTour 的调用返回值 False。从递归调用的返回有效地将搜索回溯到顶点B（参见Figure 6）。列表中要探索的下一个顶点是顶点 D，因此 knightTour 使递归调用移动到节点 D（参见 Figure 7）。从顶点 D 开始，knightTour 可以继续进行递归调用，直到我们再次到达节点 C（参见Figure 8，Figure 9 和 Figure 10）。然而，当我们到达节点C时，测试 n <limit 失败，所以我们知道已经耗尽了图中的所有节点。在这一点上，我们可以返回 True，表示我们已经成功地浏览了图。当我们返回列表时，路径具有值 [A，B，D，E，F，C]，这是我们需要遍历图以访问每个节点的顺序。


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C
D
E
F


A
B
C
D
E
F

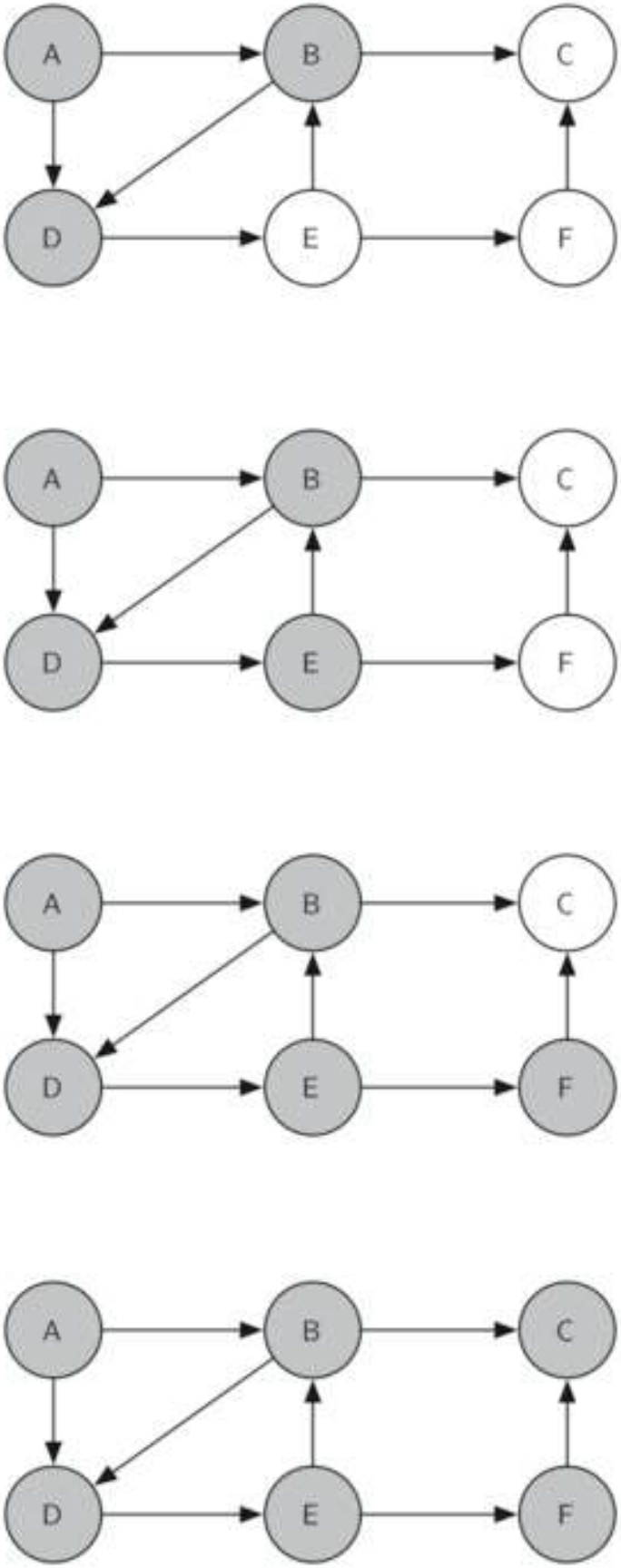
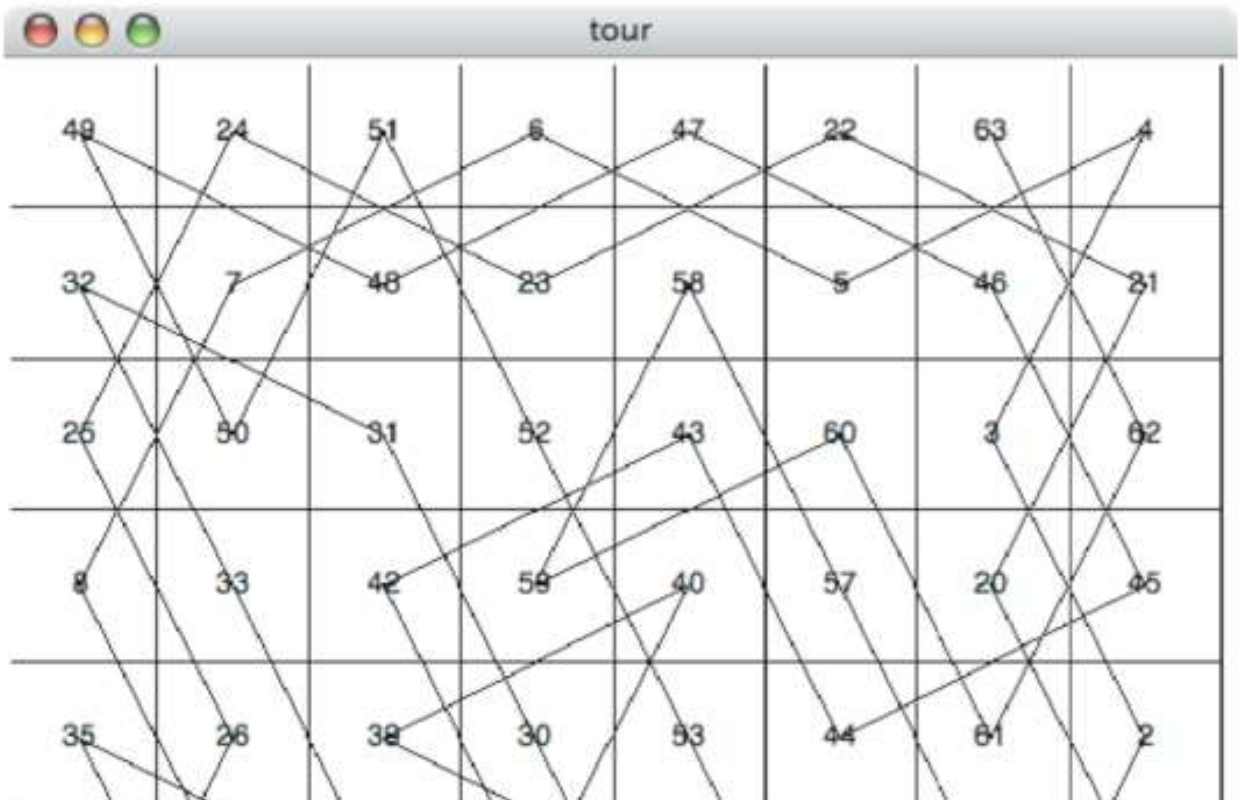


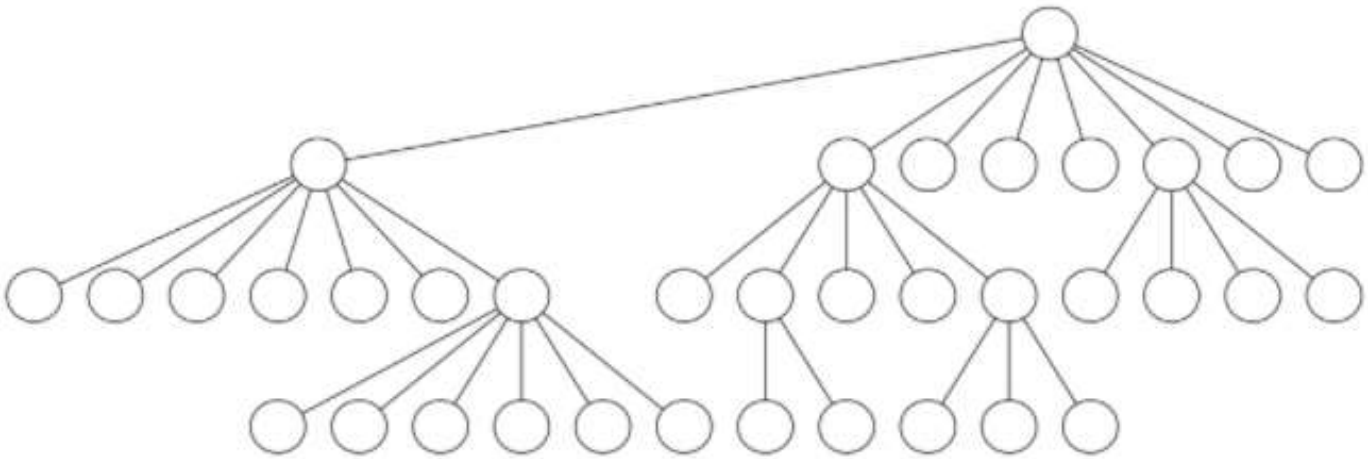
Figure 3-10

Figure 11 展示了一个 8×8 板的完整遍历。有许多可能的路径; 一些是对称的。 通过一些修改，你可以使遍历开始和结束在同一个正方形。



4.4 骑士之旅分析

有最后关于骑士之旅一个有趣的话题，然后我们将继续到深度优先搜索的通用版本。主题是性能。特别是，knightTour 对于你选择下一个要访问的顶点的方法非常敏感。例如，在一个5乘5的板上，你可以在快速计算机上处理路径花费大约1.5秒。但是如果你尝试一个 8×8 的板，会发生什么？在这种情况下，根据计算机的速度，你可能需要等待半小时才能获得结果！这样做的原因是我们到目前为止所实现的骑士之旅问题是大小为 $O(k^N)$ 的指数算法，其中 N 是棋盘上的方格数，k 是小常数。Figure 12 可以帮助我们搞清楚为什么会这样。树的根表示搜索的起点。从那里，算法生成并检查骑士可以做出的每个可能的移动。正如我们之前注意到的，可能的移动次数取决于骑士在板上的位置。在角落只有两个合法的动作，在角落邻近的正方形有三个，在板的中间有八个。Figure 13 展示了板上每个位置可能的移动次数。在树的下一级，再次有 2 到 8 个可能的下一个移动。要检查的可能位置的数量对应于搜索树中的节点的数量。



2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Figure 12-13

我们已经看到，高度 N 的二叉树中的节点数量是 $2^{N+1}-1$。对于具有可以具有多达八个孩子而不是两个节点的树，节点的数量要大得多。因为每个节点的分支因子是可变的，我们可以使用平均分支因子估计节点的数量。重要的是要注意，这个算法是指数：$k^{N+1}-1$，其中 k 是板的平均分支因子。让我们看看这增长有多快！对于 5×5 的板，树将是 25 级深，或者 N = 24，将第一级算为级 0。平均分支因子是 k = 3.8 因此，搜索树中的节点数量是 $3.8^{25}-1$ 或 3.12×10^{14} 。对于 6x6 板，k = 4.4，有 1.5×10^{23} 个节点，对于常规的 8x8 棋盘，k = 5.25 ，有 1.3×10^{46} 。当然，由于问题有多个解决方案，我们不必去探索每个节点，但是我们必须探索的节点的小数部分只是一个不会改变问题的指数性质的常数乘数。我们将把它作为一个练习，看看你是否可以表示k 作为板的大小的函数。

幸运的是有一种方法来加速八乘八的情况，使其在一秒钟内运行完成。在下面的列表中，我们将展示加速 knightTour 的代码。这个函数（见Listing 4），被称为 orderbyAvail 将被用来代替上面代码中对 u.getConnections 的调用。orderByAvail 函数中的关键是第 10 行。此行确保我们选择具有最少可用移动的下一个顶点。你可能认为这具有相反效果; 为什么不选择具有最多可用移动的节点？你可以通过运行该程序并在排序后插入行resList.reverse() 来尝试该方法。

使用具有最多可用移动的顶点作为路径上的下一个顶点的问题是，它倾向于让骑士在游览中早访问中间的方格。当这种情况发生时，骑士很容易陷入板的一侧，在那里它不能到达在板的另一侧的未访问的方格。另一方面，访问具有最少可用移动的方块首先推动骑士访问围绕板的边缘的方块。这确保了骑士能够尽早地访问难以到达的角落，并且只有在必要时才使用中间的方块跳过棋盘。利用这种知识加速算法被称为启发式。人类每天都使用启发式来帮助做出决策，启发式搜索通常用于人工智能领域。这个特定的启发式称为 Warnsdorff 算法，由 H. C. Warnsdorff 命名，他在 1823 年发表了他的算法。

```
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]
```

4.5 通用深度优先搜索

骑士之旅是深度优先搜索的特殊情况，其目的是创建最深的第一棵树，没有任何分支。更一般的深度优先搜索实际上更容易。它的目标是尽可能深的搜索，在图中连接尽可能多的节点，并在必要时创建分支。

甚至可能的是，深度优先搜索将创建多于一个树。当深度优先搜索算法创建一组树时，我们称之为深度优先森林。与广度优先搜索一样，我们的深度优先搜索使用前导链接来构造树。此外，深度优先搜索将在顶点类中使用两个附加的实例变量。新实例变量是发现和完成时间。发现时间跟踪首次遇到顶点之前的步骤数。完成时间是顶点着色为黑色之前的步骤数。正如我们看到的算法，节点的发现和完成时间提供了一些有趣的属性，我们可以在以后的算法中使用。

我们深度优先搜索的代码如 Listing 5 所示。由于 dfs 和它的辅助函数`dfsvisit` 这两个函数使用一个变量来跟踪调用 `dfsvisit` 的时间，所以我们选择将代码实现为继承自 `Graph` 类。此实现通过添加时间实例变量和两个方法 `dfs` 和 `dfsvisit`来扩展 `Graph` 类。看看第 11 行，你会注意到，`dfs` 方法在调用 `dfsvisit` 的图中所有的顶点迭代，这些节点是白色的。我们迭代所有节点而不是简单地从所选择的起始节点进行搜索的原因是为了确保图中的所有节点都被考虑到，没有顶点从深度优先森林中被遗漏。`for aVertex in self` 语句可能看起来不寻常，但请记住，在这种情况下，`self`是 `DFSGraph` 类的一个实例，遍历实例中的所有顶点是一件自然的事情。

```
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)
```

Listing 5

虽然我们 bfs 的实现只对有一条路径回到开始的路径的节点感兴趣，但是有可能创建一个宽度优先森林，其表示图中的所有节点之间的最短路径。我们把这作为一个练习。在接下来的两个算法中，我们将看到为什么跟踪深度优先森林的深度很重要。

dfsvisit 方法从名为 startVertex 的单个顶点开始，并尽可能深地探查所有相邻的白色顶点。如果仔细查看 dfsvisit 的代码并将其与广度优先搜索进行比较，应该注意的是，dfsvisit 算法几乎与 bfs 相同，除了在内部 for 循环的最后一行，dfsvisit 将自行递归调用以继续在更深的级别搜索，而 bfs 将节点添加到队列以供稍后探查。有趣的是，bfs 使用队列，dfsvisit 使用栈。你在代码中没有看到栈，但是它在 dfsvisit 的递归调用中是隐含的。

以下图的序列展示了针对小图的深度优先搜索算法。在这些图中，虚线指示检查的边，但是在边的另一端的节点已经被添加到深度优先树。在代码中，通过检查另一个节点的颜色是非白色的。

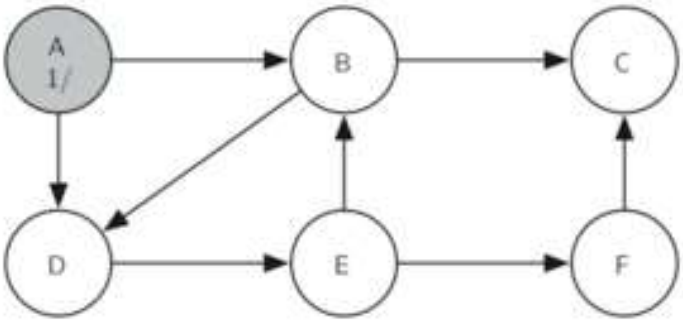
搜索从图的顶点 A 开始（Figure 14）。由于所有顶点在搜索开始时都是白色的，所以算法访问顶点 A。访问顶点的第一步是将颜色设置为灰色，这表示正在探索顶点，并且将发现时间设置为1，由于顶点 A 具有两个相邻的顶点（B，D），因此每个顶点也需要被访问。我们将做出任意决定，我们将按字母顺序访问相邻顶点。

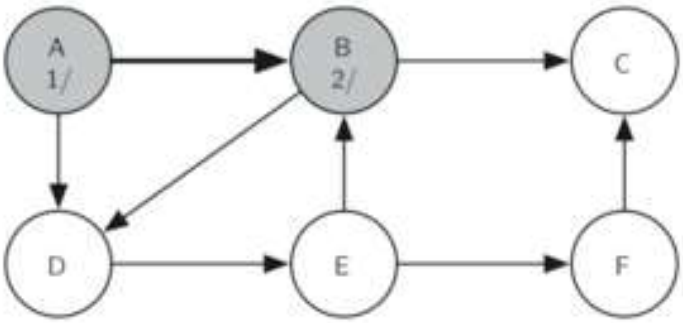
接下来访问顶点B（Figure 15），因此其颜色设置为灰色并且其发现时间被设置为 2。顶点 B 也与两个其他节点（C，D）相邻，因此我们将遵循字母顺序和访问节点 C 接下来。

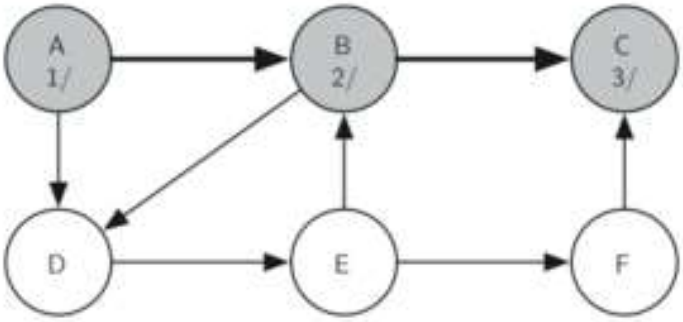
访问顶点 C（Figure16）使我们到树的一个分支的末端。在将节点灰色着色并将其发现时间设置为 3 之后，算法还确定没有与 C 相邻的顶点。这意味着我们完成了对节点 C 的探索，因此我们可以将顶点着色为黑色，并将完成时间设置为 4，在Figure 17 中，可以看到我们的搜索的状态。

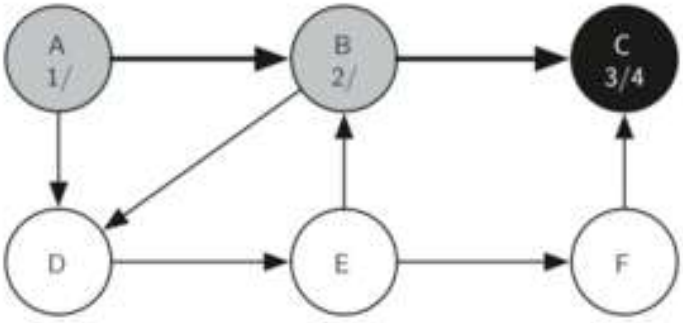
由于顶点 C 是一个分支的结束，我们现在返回到顶点 B，继续探索与 B 相邻的节点。从 B 中探索的唯一额外的顶点是 D，所以我们现在可以访问 D（Figure 18），并继续搜索顶点 D。顶点 D 快速引导我们到顶点 E（Figure 19）。顶点 E 具有两个相邻的顶点 B 和 F 。通常我们将按字母顺序探索这些相邻顶点，但是由于 B 已经是灰色的，所以算法识别出它不应该访问 B，因为这样做会将算法置于循环中！因此，继续探索列表中的下一个顶点，即 F（Figure 20）。

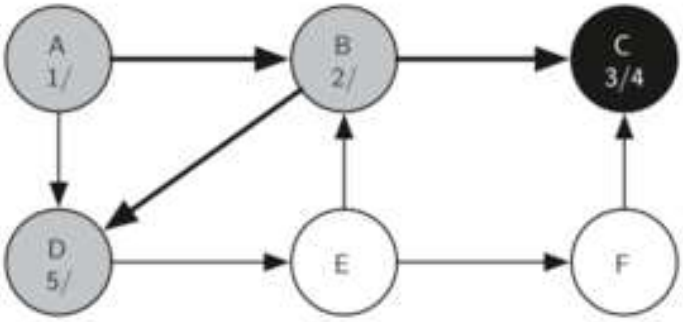
顶点 F 只有一个相邻的顶点 C，但由于 C 是黑色的，没有别的东西可以探索，算法已经到达另一个分支的结束。从这里开始，你将在 Figure 21至 Figure 25中看到算法运行回到第一个节点，设置完成时间和着色顶点为黑色。

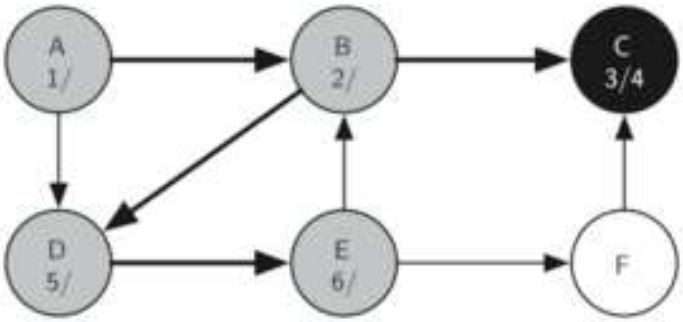

A
1/
B
C
D
E
F

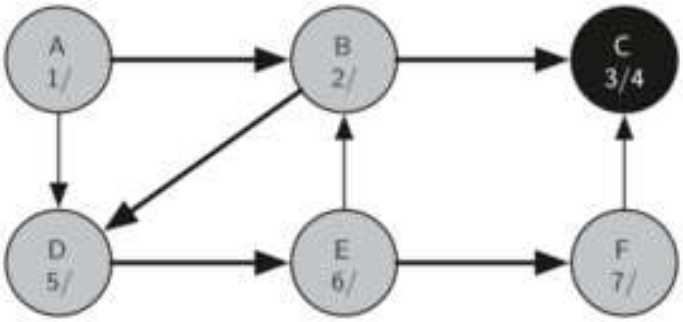

A
1/
B
2/
C
D
E
F

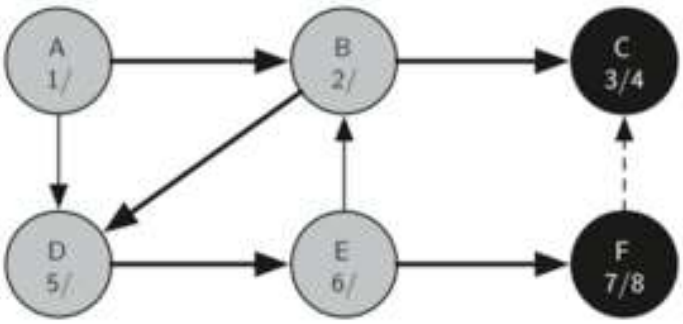

A
1/
B
2/
C
3/
D
E
F


A
1/
B
2/
C
3/4
D
E
F


A
1/
B
2/
C
3/4
D
5/
E
F


A
1/
B
2/
C
3/4
D
5/
E
6/
F


A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/


A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/8

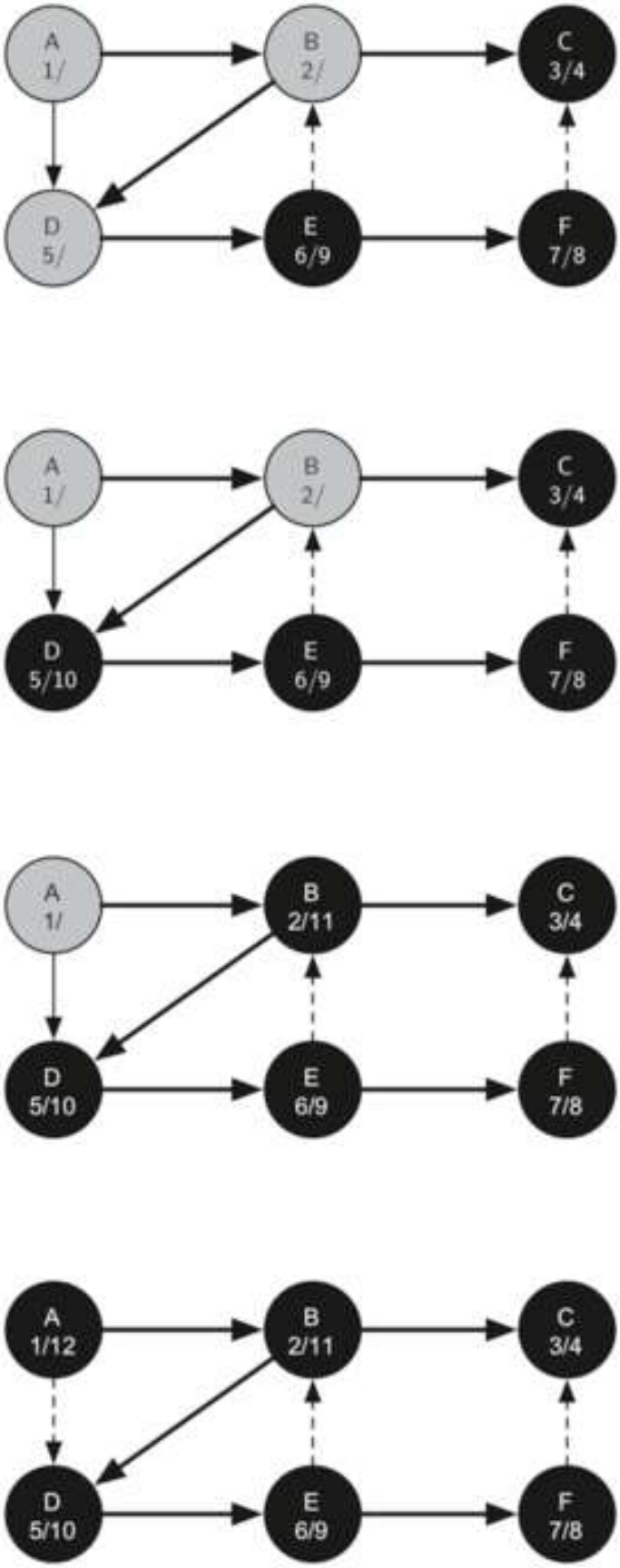
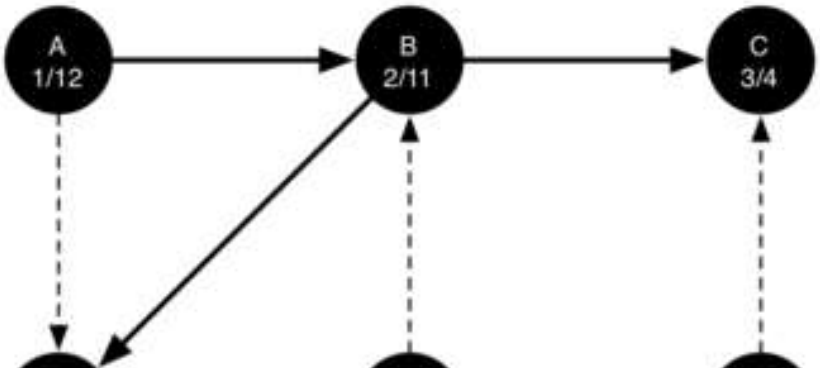

A
1/
B
2/
C
3/4
D
5/
E
6/9
F
7/8
A
1/
B
2/
C
3/4
D
5/10
E
6/9
F
7/8
A
1/
B
2/11
C
3/4
D
5/10
E
6/9
F
7/8
A
1/12
B
2/11
C
3/4
D
5/10
E
6/9
F
7/8

Figure 14-25

每个节点的开始和结束时间展示一个称为 括号属性 的属性。 该属性意味着深度优先树中的特定节点的所有子节点具有比它们的父节点更晚的发现时间和更早的完成时间。 Figure 26 展示了由深度优先搜索算法构造的树。



4.6 深度优先搜索分析

深度优先搜索的一般运行时间如下。 dfs 中的循环都在 $O(V)$ 中运行，不计入`dfsvisit` 中发生的情况，因为它们对图中的每个顶点执行一次。 在`dfsvisit` 中，对当前顶点的邻接表中的每个边执行一次循环。 由于只有当顶点为白色时，`dfsvisit` 才被递归调用，所以循环对图中的每个边或 $O(E)$ 执行最多一次。 因此，深度优先搜索的总时间是 $O(V+E)$。

4.7 深度优先搜索的完整程序

In [8]:

```
from collections import deque
from collections import namedtuple


def bfs(start_node, end_node, graph):
    node = namedtuple('node', 'name, from_node')
    search_stack = deque()  # 这里当作栈使用
    name_search = deque()
    visited = {}

    search_stack.append(node(start_node, None))
    name_search.append(start_node)
    path = []
    path_len = 0

    print('开始搜索...')
    while search_stack:
        print('待遍历节点: ', name_search)
        current_node = search_stack.pop()  # 使用栈模式，即后进先出，这是DFS的核心
        name_search.pop()
        if current_node.name not in visited:
            print('当前节点: ', current_node.name, end=' | ')
            if current_node.name == end_node:
                pre_node = current_node
                while True:
                    if pre_node.name == start_node:
                        path.append(start_node)
                        break
                    else:
                        path.append(pre_node.name)
                        pre_node = visited[pre_node.from_node]
                path_len = len(path) - 1
                break
            else:
                visited[current_node.name] = current_node
                for node_name in graph[current_node.name]:
                    if node_name not in name_search:
                        search_stack.append(node(node_name, current_node.name))
                        name_search.append(node_name)
    print('搜索完毕,路径为:', path[::-1], "长度为:", path_len)  # 这里不再是最短路径，深度优先搜索无法一次给出最短路径


if __name__ == "__main__":

    graph = dict()
```

```
    graph[1] = [3, 2]
    graph[2] = [5]
    graph[3] = [4, 7]
    graph[4] = [6]
    graph[5] = [6]
    graph[6] = [8]
    graph[7] = [8]
    graph[8] = []
    bfs(1, 8, graph)
```

```
开始搜索...
待遍历节点:  deque([1])
当前节点:  1 | 待遍历节点:  deque([3, 2])
当前节点:  2 | 待遍历节点:  deque([3, 5])
当前节点:  5 | 待遍历节点:  deque([3, 6])
当前节点:  6 | 待遍历节点:  deque([3, 8])
当前节点:  8 | 搜索完毕,路径为: [1, 2, 5, 6, 8] 长度为: 4
```

5. 拓扑排序

为了表明计算机科学家可以把任何东西变成一个图问题，让我们考虑做一批煎饼的问题。 菜谱真的很简单：1个鸡蛋，1杯煎饼粉，1汤匙油 和 3/4 杯牛奶。 要制作煎饼，你必须加热炉子，将所有的成分混合在一起，勺子搅拌。 当开始冒泡，你把它们翻过来，直到他们底部变金黄色。 在你吃煎饼之前，你会想要加热一些糖浆。 Figure 27将该过程示为图。

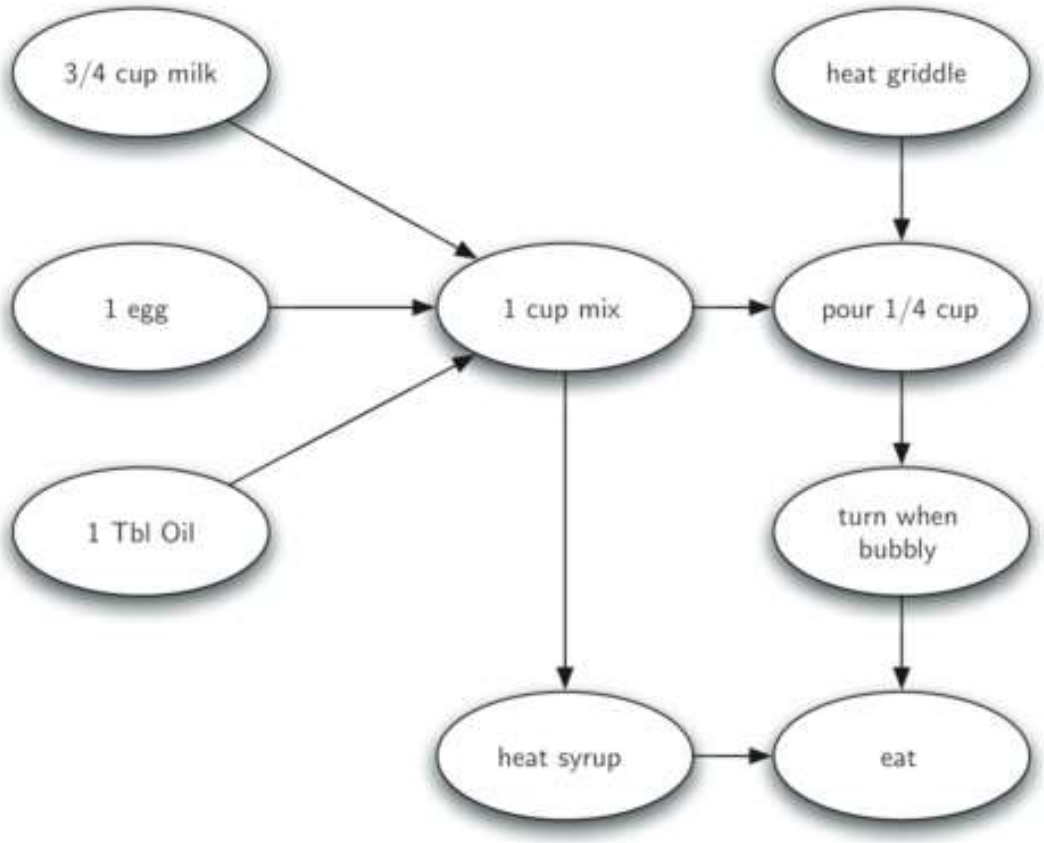


Figure 27

制作煎饼的困难是知道先做什么。从 Figure 27 可以看出，你可以从加热煎饼开始，或通过添加任何成分到煎饼。为了帮助我们决定应该做的每一个步骤的精确顺序，我们转向一个图算法称为 拓扑排序。

拓扑排序采用有向无环图，并且产生所有其顶点的线性排序，使得如果图 G 包含边(v, w)，则顶点 v 在排序中位于顶点 w 之前。定向非循环图在许多应用中使用以指示事件的优先级。制作煎饼只是一个例子;其他示例包括软件项目计划，用于数据库查询的优先图以及乘法矩阵。

拓扑排序是深度优先搜索的简单但有用的改造。拓扑排序的算法如下：

1. 对于某些图 `g` 调用 `dfs(g)`。我们想要调用深度优先搜索的主要原因是计算每个顶点的完成时间。
2. 以完成时间的递减顺序将顶点存储在列表中。

3. 返回有序列表作为拓扑排序的结果。

Figure 28 展示了在 Figure 26 所示的薄煎饼制作图上由 `dfs` 构建的深度优先森林。

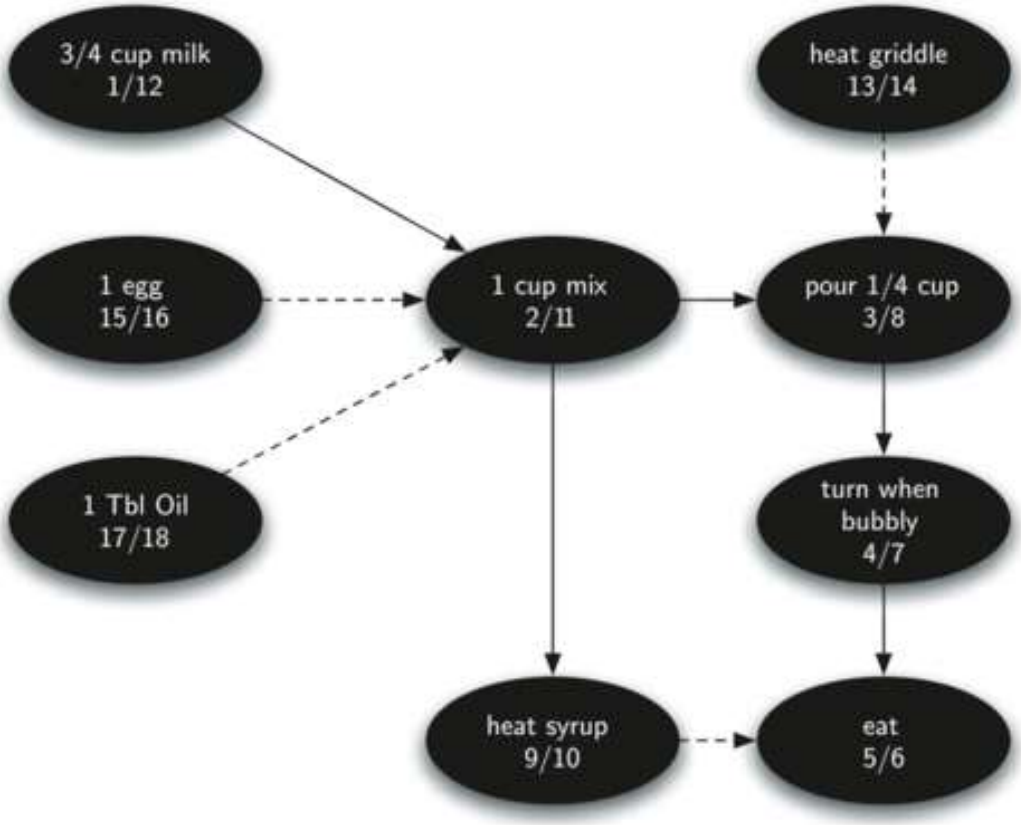


Figure 28

最后，Figure 29 展示了将拓扑排序算法应用于我们的图形的结果。 现在所有的分支已被删除，我们知道确切的做煎饼的步骤顺序。

Figure 29

拓扑排序的定义为：由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。而个人认为，拓扑排序即是在图的基本遍历法上引入了入度的概念并围绕入度来实现的排序方法，拓扑排序与Python多继承中mro规则的排序类似，若想深入研究mro规则的C3算法，不妨了解一下 DAG(有向无环图) 的拓扑排序。

拓扑排序的Python程序

In [10]:

```
#拓扑排序
# 先定义图结构
graph = {
    "A": ["B","C"],
    "B": ["D","E"],
    "C": ["D","E"],
    "D": ["F"],
    "E": ["F"],
    "F": [],
}

def TopologicalSort(G):
    # 创建入度字典
    in_degrees = dict((u, 0) for u in G)
    # 获取每个节点的入度
    for u in G:
        for v in G[u]:
            in_degrees[v] += 1
    # 使用列表作为队列并将入度为0的添加到队列中
    Q = [u for u in G if in_degrees[u] == 0]
    res = []
    # 当队列中有元素时执行
    while Q:
        # 从队列首部取出元素
        u = Q.pop()
        # 将取出的元素存入结果中
        res.append(u)
        # 移除与取出元素相关的指向，即将所有与取出元素相关的元素的入度减少1
        for v in G[u]:
            in_degrees[v] -= 1
            # 若被移除指向的元素入度为0，则添加到队列中
            if in_degrees[v] == 0:
                Q.append(v)
    return res
print(TopologicalSort(graph))
```

['A', 'C', 'B', 'E', 'D', 'F']

6. 强连通分量

在本章的剩余部分，我们将把注意力转向一些非常大的图。我们将用来研究一些附加算法的图，由互联网上的主机之间的连接和网页之间的链接产生的图。 我们将从网页开始。

像 Google 和 Bing 这样的搜索引擎利用了网页上的页面形成非常大的有向图。 为了将万维网变换为图，我们将把一个页面视为一个顶点，并将页面上的超链接作为将一个顶点连接到另一个顶点的边缘。 Figure 30 展示了从 Luther College 的计算机科学主页开始，通过跟踪从一页到下一页的链接产生的图的非常小的部分。当然，这个图可能是巨大的，所以我们把它限制在距离 CS 主页不超过 10 个链接的网站。

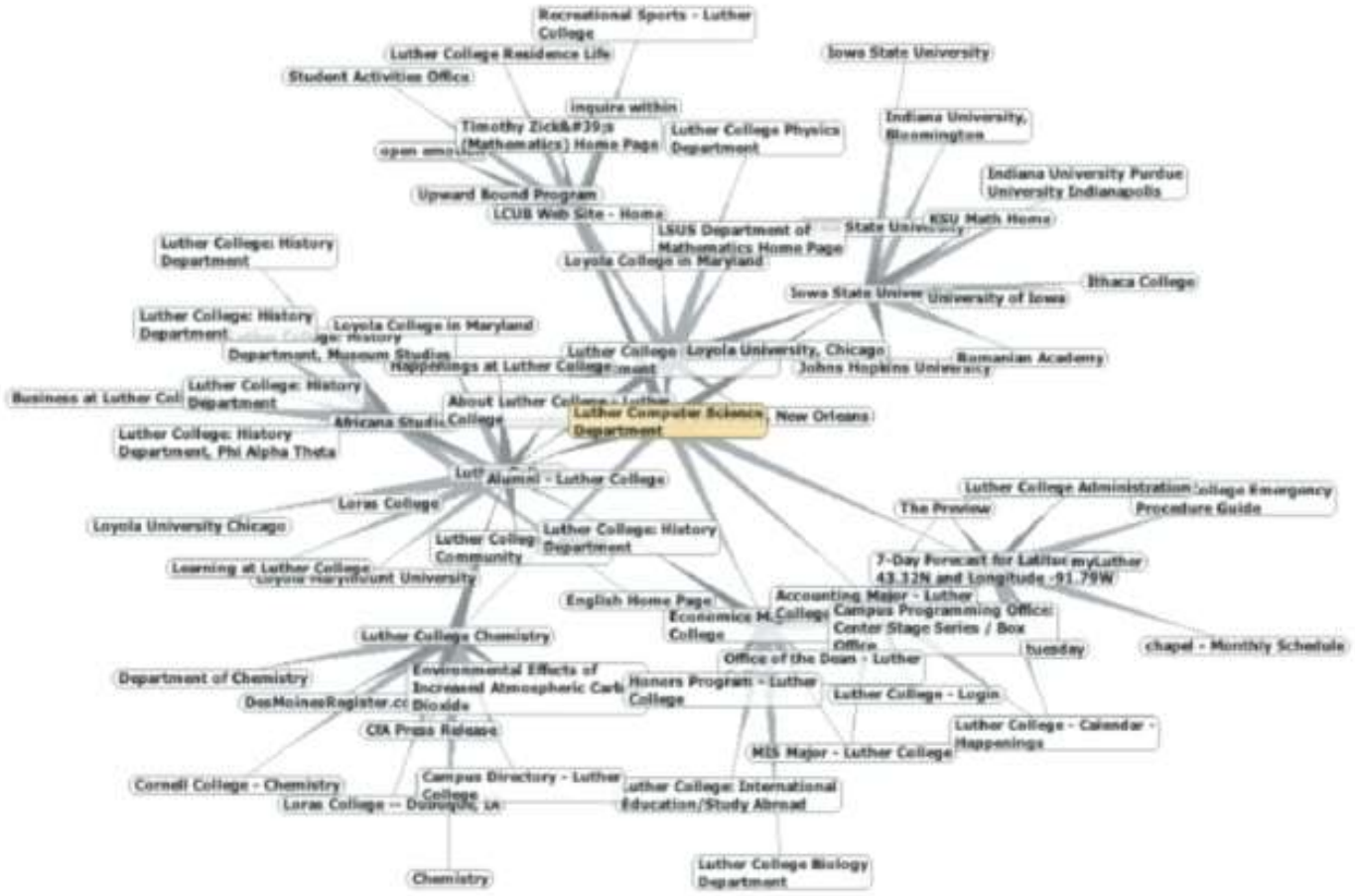


Figure 30

如果你看 Figure 30中的图形，你可能会有一些有趣的观察。首先你可能会注意到，图上的许多其他网站是其他路德学院网站。第二，你可能注意到有几个链接到爱荷华州的其他学院。第三，你可能注意到有几个链接到其他文理学院。你可能会得出这样的结论，网络集群上的网站在一些级别上底层结构类似。

可以帮助找到图中高度互连的顶点的集群的一种图算法被称为强连通分量算法（SCC）。我们正式定义图 G 的强连通分量 C 作为顶点 $C \subset V$ 的最大子集，使得对于每对顶点 $v, w \in C$，我们具有从 v 到 w 的路径和从 w 到 v 的路径。Figure 27 展示了具有三个强连接分量的简单图。强连接分量由不同的阴影区域标识。

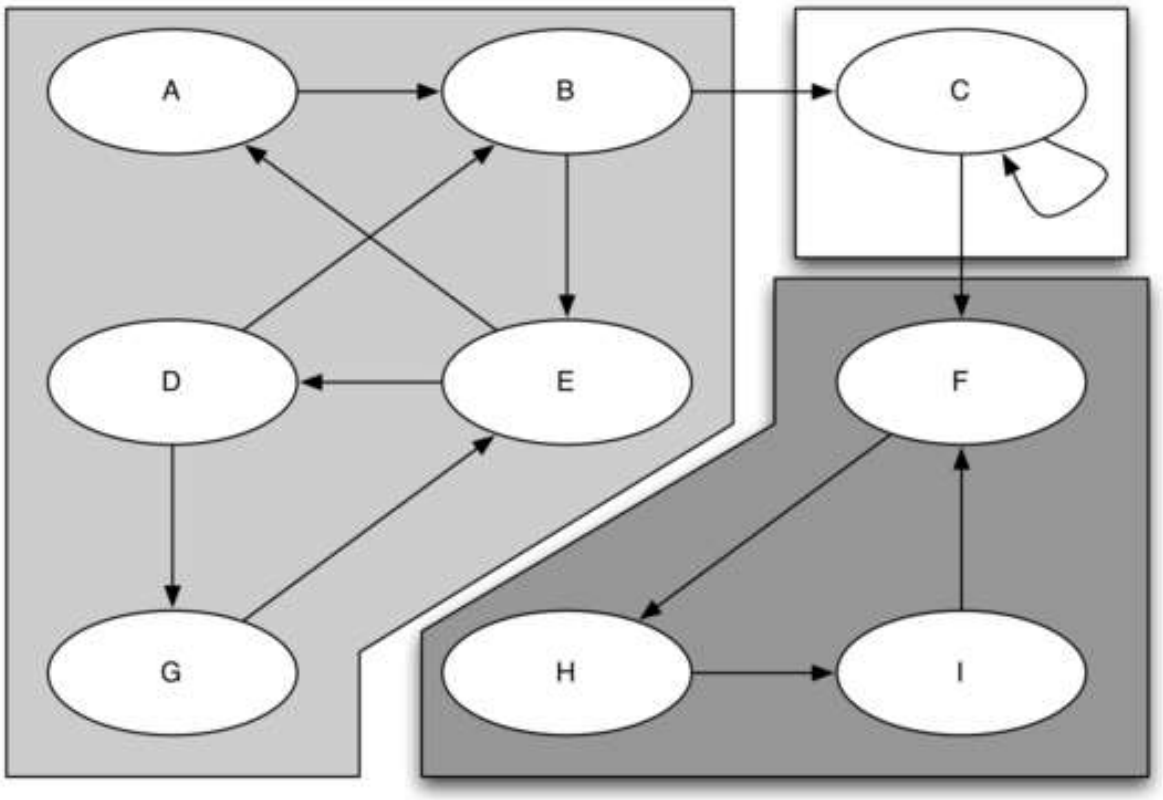


Figure 27

一旦确定了强连通分量，我们就可以通过将一个强连通分量中的所有顶点组合成一个较大的顶点来显示该图的简化视图。 Figure 31中的曲线图的简化版本如 Figure 32所示。

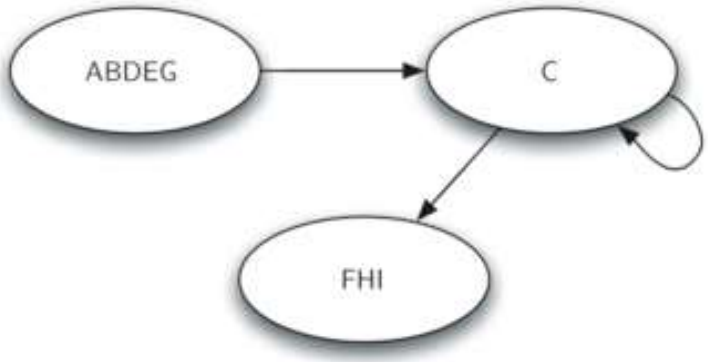
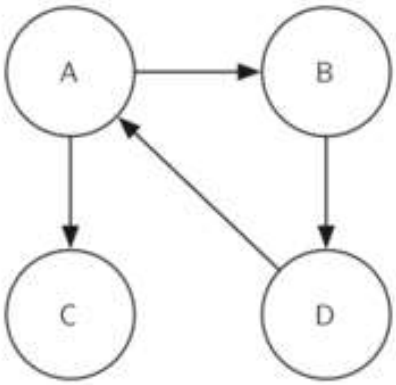


Figure 32

我们再次看到，我们可以通过使用深度优先搜索来创建一个非常强大和高效的算法。 在我们处理主 SCC 算法之前，我们必须考虑另一个定义。 图 G 的转置被定义为图 G^T ，其中图中的所有边已经反转。 也就是说，如果在原始图中存在从节点 A 到节点 B 的有向边，则 G^T 将包含从节点 B 到节点 A 的边。Figure 33和 Figure 34 展示了简单图及其变换。



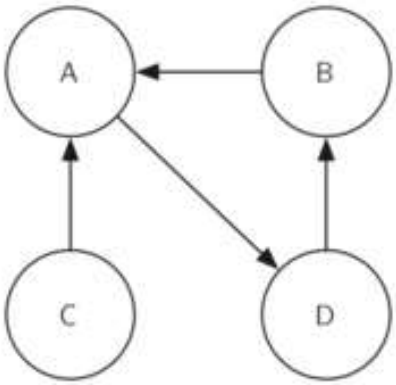


Figure 33-34

再看看数字。 请注意，Figure 33中的图形有两个强连通分量。 现在看看Figure 34。注意它也有两个强连通分量。

我们现在可以描述用于计算图的强连通分量的算法。

1. 调用 dfs 为图 G 计算每个顶点的完成时间。
2. 计算 G^T 。
3. 为图 G^T 调用 dfs，但在 DFS 的主循环中，以完成时间的递减顺序探查每个顶点。
4. 在步骤 3 中计算的森林中的每个树是强连通分量。输出森林中每个树中每个顶点的顶点标识组件。

让我们在 Figure 31中的示例图上跟踪上述步骤的操作。Figure 35 展示了由 DFS 算法为原始图计算的开始和结束时间。 Figure 36 展示了通过在转置图上运行 DFS 计算的开始和结束时间。

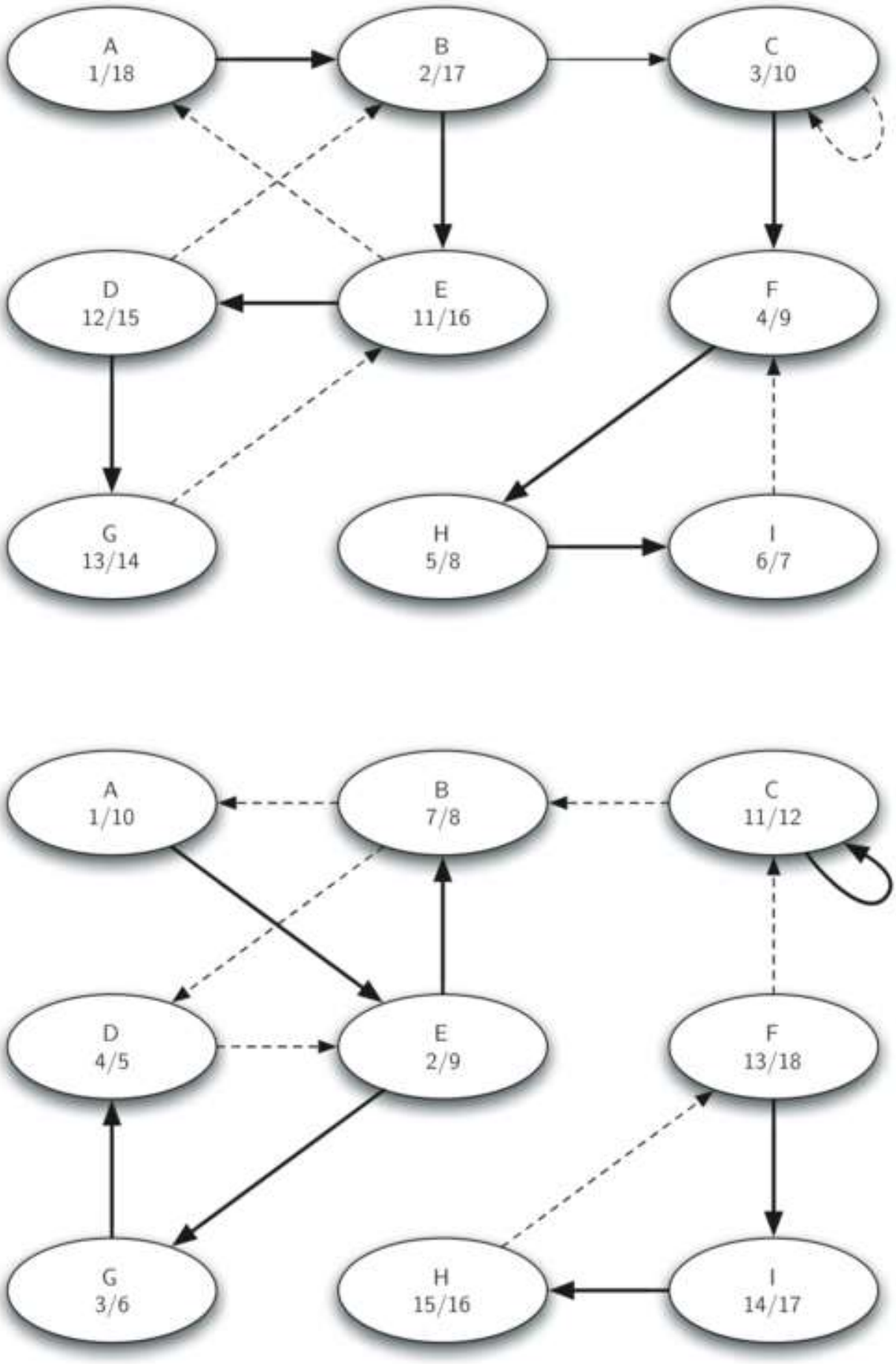

A
1/18
B
2/17
C
3/10
D
12/15
E
11/16
F
4/9
G
13/14
H
5/8
I
6/7
A
1/10
B
7/8
C
11/12
D
4/5
E
2/9
F
13/18
G
3/6
H
15/16
I
14/17

Figure 36

最后，Figure 37 展示了在强连通分量算法的步骤 3 中产生的三棵树的森林。 你会注意到，我们不为你提供 SCC 算法的 Python 代码，我们将此程序作为练习。

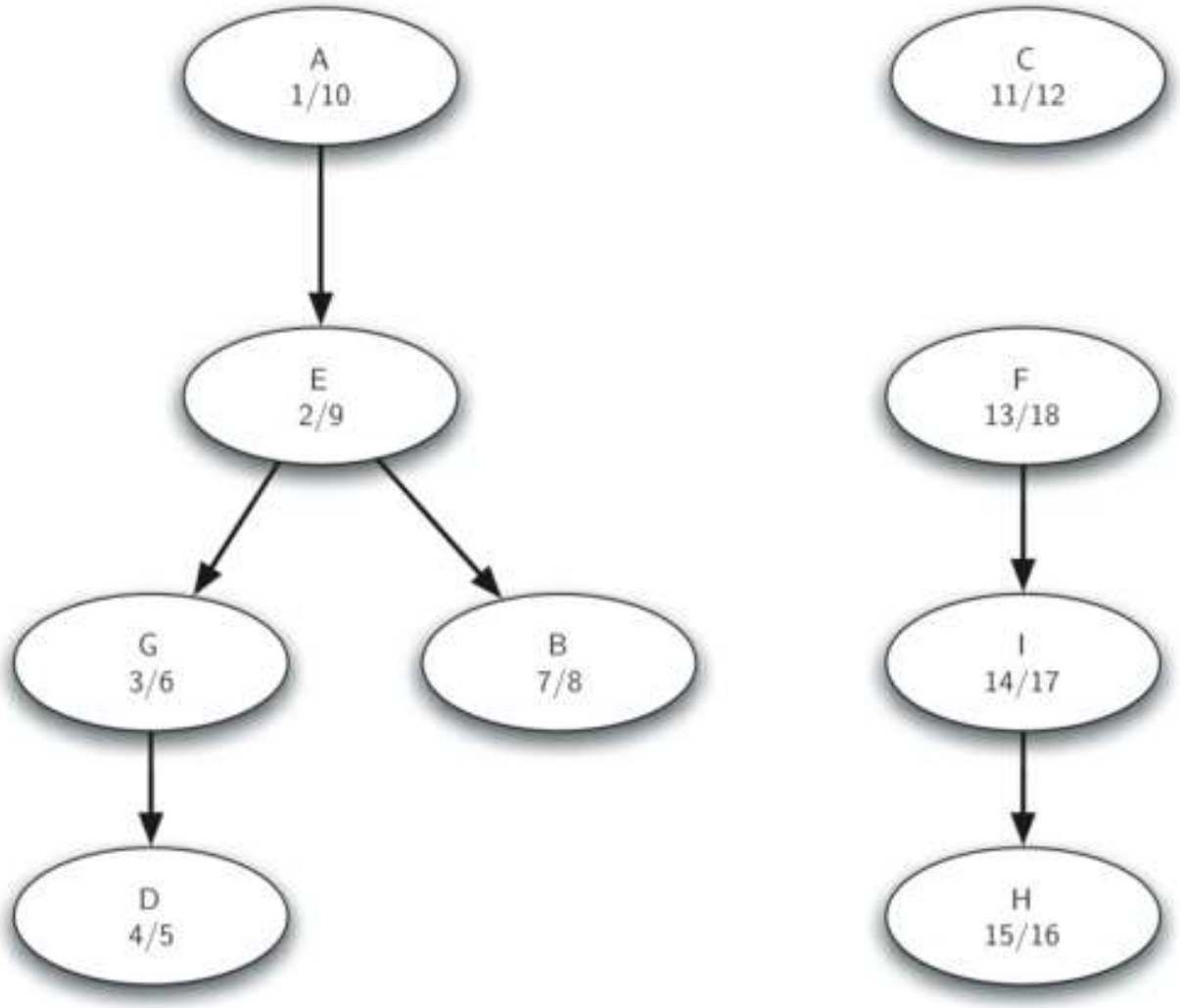


Figure 37

强连通分量Python程序：

In [12]:

```
class Graph:
    def __init__(self):
        self.V = []

class Vertex:
    def __init__(self, x):
        self.key = x
        self.color = 'white'
        self.d = 10000
        self.f = 10000
        self.pi = None
        self.adj = []

class Solution:
    def Dfs(self, G):
        for u in G.V:
            u.color = 'white'
            u.pi = None
        global time
        time = 0
        for u in G.V:
            if u.color == 'white':
                list=[u]
                self.DfsVisit(G, u, list)
                print(''.join([i.key for i in list]))


    def DfsVisit(self, G, u, list):
        global time
        time = time + 1
        u.d = time
        u.color = 'gray'
        for v in u.adj:
            if v.color == 'white':
                list.append(v)
                v.pi = u
                self.DfsVisit(G, v, list)
        u.color = 'black'
        time = time + 1
        u.f = time

    def GraphTransposition(self, G):
        for u in G.V:
            u.adj = (u.adj,[])

        for u in G.V:
```

```
            for v in u.adj[0]:
                v.adj[1].append(u)

        for u in G.V:
            u.adj = u.adj[1]

        return G

    def StronglyConnectedComponents(self, G):
        self.Dfs(G)
        G_Transposition = self.GraphTransposition(G)
        G_Transposition.V.sort(key=lambda v: v.f, reverse=True)
        self.Dfs(G_Transposition)



if __name__ == '__main__':
    a,b,c,d,e,f,g,h = [Vertex(i) for i in ['a','b','c','d','e','f','g','h']]

    a.adj = [b]
    b.adj = [c,e,f]
    c.adj = [d,g]
    d.adj = [c,h]
    e.adj = [a,f]
    f.adj = [g]
    g.adj = [f,h]
    h.adj = [h]

    G = Graph()
    G.V = [a,b,c,d,e,f,g,h]

    m = Solution()
    m.StronglyConnectedComponents(G)
```

```
abcdhgfe
aeb
cd
gf
h
```

引用及参考：

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.

[2] https://www.cnblogs.com/Peter2014/p/10682833.html (https://www.cnblogs.com/Peter2014/p/10682833.html)

[3] https://www.cnblogs.com/zuoyuan/p/4343148.html (https://www.cnblogs.com/zuoyuan/p/4343148.html)

[4] https://blog.csdn.net/qq_37738656/article/details/83316863 (https://blog.csdn.net/qq_37738656/article/details/83316863)

课后练习

1. 写出深度优先搜索和广度优先搜索的Python代码或 C语言代码。

2. 针对图三的邻接矩阵开展进行优先搜索，写出相应python程序。

3. 推导出拓扑排序算法的时间复杂度；推导出强连通分量算法的时间复杂度。

讨论、思考题、作业：

参考资料（含参考书、文献等）：算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。