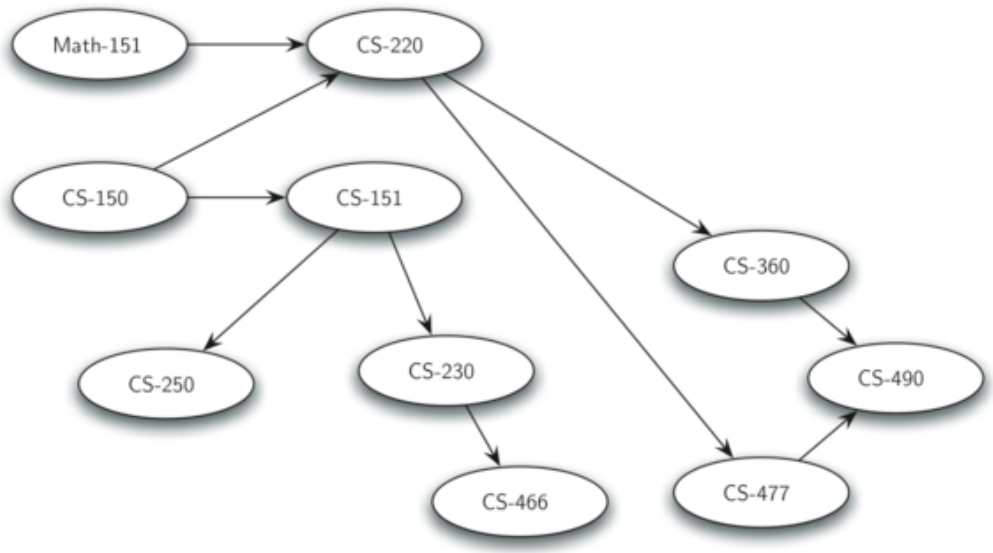
7.1 目标

- 学习图的概念和使用方式。
- 以多种方式实现抽象数据类型Gragh。
- 了解图在解决多种问题时的应用。

本章研究图像。图像是比上一章学习的树更加通用的结构，实际上可以认为树是一种特殊的图像。图像也可以用来表示真实世界中的很多东西，比如公路系统，航线系统，网路系统甚至是完成计算机科学学位所必须的课程序列。本章读者会看到，一旦给某个问题以合适的表示，利用一些图像算法便可以轻松将看起来很困难的问题解决。

对人类来说，看懂道路地图并理解不同地点之间的关系并不困难，但是计算机并没有这种能力。然而，可以将道路系统看作是1个图像，如此一来便可以让计算机做一些有趣的事情了。比如在互联网地图中，计算机可以找到从某个地方到另一个地方的最近、最快或者最简单的路线。

作为计算机科学的学生，读者可能对获得学位所需要的课程比较感兴趣。图像可以很好地表达课程之间的依赖关系。图1便是1个例子。



../_images/CS-Prereqs.png

7.2 术语和定义

顶点（vertex）
顶点（也称为节点**node**）是图像的基础构件。它可以有名称，即键。顶点也可以有其它额外信息，即负载。
边（Edge）
边有时也被称为弧（arc），是图像的另一种基础构件。边将2个顶点连接起来以表示这2者之间存在某种关系。边可以是单向的也可以是双向的。当某个图像中的所有边都是单向的时候，便称该图像是**有向图（directed gragh/digrah）**。上示的课程依赖图显示是有向图，因为必须先完成某些课程后才能继续其它某些课程。
权重（weight）
为了表示从某个顶点到另一个顶点所需的代价，可以顶点进行**赋权（weighted）**。比如说公路图中某个城市连接到另一个城市，边上的权重可以表示为两个城市之间的距离。

有了以上基本概念，便可以来对图像作一定义了。图像可以表示为G，其中G=(V,E)。对于图像G，V是顶点的集合，E是边的集合。每个边都是一个元组（v,w），其中$w, v \in V$。可以向边元组中加入第3个元素来表示权重。子图s是顶

点的集合v和边的集合e，使得$e \subset E$且$v \subset V$。

图2给出了1个简单的赋权有向图作为例子。该图像可以用集合严格地表示：

$$V = V0, V1, V2, V3, V4, V5$$

$$E = \{(v0, v1, 5), (v1, v2, 4), (v2, v3, 9), (v3, v4, 7), (v4, v0, 1), (v0, v5, 2), (v5, v4, 8), (v3, v5, 3), (v5, v2, 1)\}$$

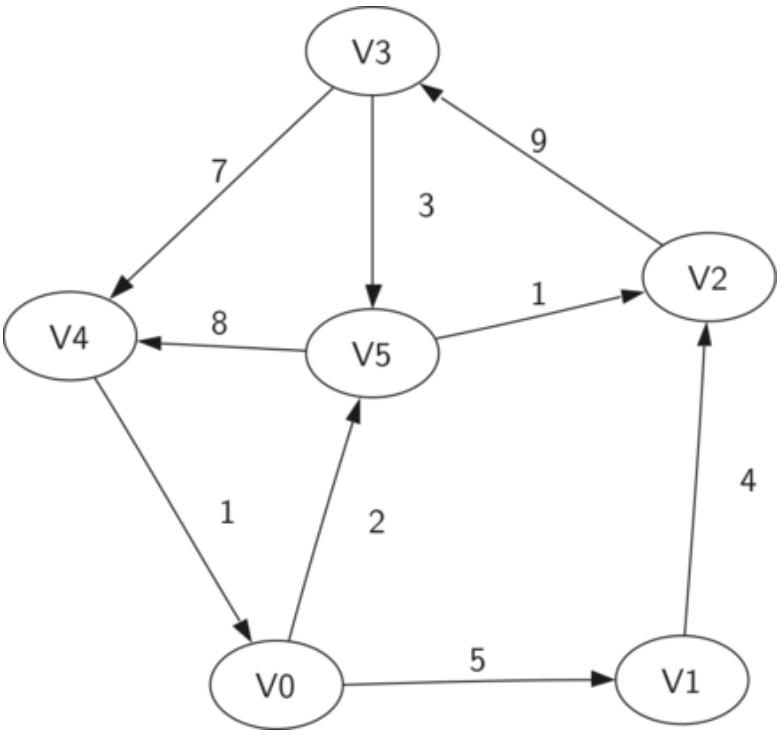


图2中的例子演示了图像的另外2个关键术语。

路径（path）

路径是由边连接起来的顶点的序列。一般地，将路径定义为 $w_1, w_2, \ldots, w_n$ 使得 $(w_i, w_{i+1}) \in E$ 对所有 $1 \leq i \leq n-1$ 成立。无权路径长度是路径中边的个数，即n-1。权重路径长度是路径中边的权重之和。比如图2中，从V3到V1的路径是顶点序列 $(V3, V4, V0, V1)$ 而边为 $(v3, v4, 7), (v4, v0, 1), (v0, v1, 5)(v3, v4, 7), (v4, v0, 1), (v0, v1, 5)$。

环（cycle）

有向图中的环是首尾顶点相同的路径。比如说，图2中路径 $(V5, V2, V3, V5)(V5, V2, V3, V5)$ 就是1个环。没有环的图被称为**无环图（acyclic graph）**。没有环的有向图被称为**有向无环图（directed acyclic graph，DAG）**。接下来读者会看到，一些很重要的问题可以通过表示为DAG获得解决。

7.3 抽象数据类型：Graph

抽象数据类型Gragh被定义为：

- Graph()生成一个新的空图。
- addVertex(vert) 将Vertex的1个实例添加至图。
- addEdge(fromVert,toVert) 向图添加1个连接2个顶点的有向边。
- addEdge(fromVert,toVert,weight)向图添加1个连接2个顶点的赋权有向边。
- getVertex(vertKey) 找到途中以vertKey命名的顶点。
- getVertices() 返回途中所有顶点的列表。

- in 对于vertex in graph这种形式的语句，若给定的vertex在途中，返回True，反之则False。

有了图像的正式定义，在Python中可以用多种方式实现，下面来看看使用不同方式实现该ADT的优劣与差别。有两个著名的图的实现，即**邻接矩阵（adjacency matrix）和邻接表（adjacency list）**，本书将对这两种都进行解释，并对其中1中以Python类的方式实现。

7.4 邻接矩阵（adjacency matrix）##

实现图的最简单方式之一是使用二维矩阵。在二维矩阵中，每行每列的都代表了图中的1个顶点。如果顶点v到顶点w有边，行v和列w的交叉点单元格会存储值。若两个顶点由边连接，便称这两个顶点是**邻接**的。图3给出了图2的邻接矩阵。单元格中存储的值是v到w的权重。

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

../_images/adjMat.png

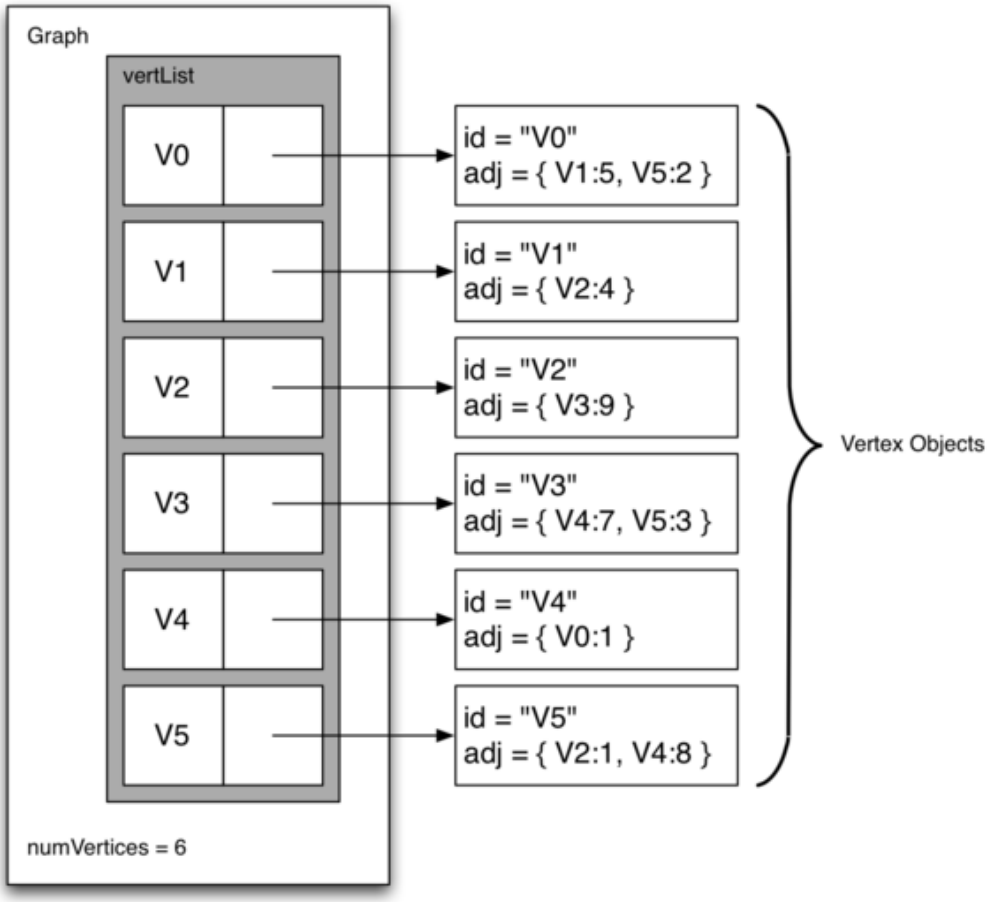
邻接矩阵的优势是简便，并且对于小图来说，很容易看出哪些节点是相连的。然而，注意到矩阵中大多数单元格都是空。因为大多数单元格都是空的，因此称该矩阵是**稀疏（sparse）**的。矩阵并不能高效地存储这种稀疏数据。实际上，在Python中像图3这种矩阵结构都很麻烦。

邻接矩阵对于边树较高的图来说是不错的实现。“较高”是什么意思？到底需要多少个边才能填满该矩阵？因此为每行、每列都对应个顶点，填满矩阵需要的顶点数是$|V|_2$。当每个顶点都与另外每一个顶点相连时，则该矩阵被填满了。实践中很少会遇到以这种方式连接的问题。本章研究的所有问题都是稀疏图。

7.5 邻接表（adjacency list）

邻接表可以用更少的空间实现稀疏图。在邻接表的实现中，维护一个包含图对象所有顶点的主列表，然后图对象中各个顶点对象都维护一个该对象连接的其它顶点对象的列表。在Vertex类的实现中，将使用字典而不是列表，其中字典

的键为顶点，值为权重。图4给出了图2的邻接表实现示意图。



邻接表的有时在于它可以紧凑地表示稀疏图，利用稀疏图也可以轻松地找到某个顶点所有的连接情况。

7.6 实现

使用字典可以很容易地在Python中实现邻接表。在本书的实现中会创建两个类（代码1、代码2），Graph保存顶点的祝列表，而Vertex即图中每个顶点。

每个Vertex都使用字典来保存其连接的对象以及每条边的权重。该字典被命名为connectedTo。以下代码便是Vertex类的实现。构造器只是初始化了可以是字符串的id以及connectedTo字典。addNeighbor方法用来添加从该顶点到另一顶点的连接。getConnections方法返回当前节点connectedTo实例变量保存的邻接表中的所有顶点。getWeight方法返回从该节点到另一个作为参数传入的节点的边的权重。

代码1

```
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]
```

Graph类，如下面的代码所示，保存了一个字典，将顶点名映射到顶点对象。在图4中，该字典对象表示为灰色阴影框。Graph类也听过了方法用于添加顶点和连接顶点。此外，实现了__iter__方法来使得可以在图中对所有顶点对象进行迭代。这两种方法可以用来实现按名称或者是对象本身来对图中顶点的迭代。

代码2

```
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())
```

使用以上定义的Graph和Vertex类，以下代码边可以表示图2。首先生成6个节点编号0-5，查看顶点字典，注意到对0-5的键都创建了1个Vertex实例。然后，将连接了顶点的边集合到一起。最后，使用嵌套循环确定图中的每个边都正确存储了。读者应该将输出结果与图2作一对比。

```
>>> g = Graph()
>>> for i in range(6):
...     g.addVertex(i)
>>> g.vertList
{0: <adjGraph.Vertex instance at 0x41e18>,
 1: <adjGraph.Vertex instance at 0x7f2b0>,
 2: <adjGraph.Vertex instance at 0x7f288>,
 3: <adjGraph.Vertex instance at 0x7f350>,
 4: <adjGraph.Vertex instance at 0x7f328>,
 5: <adjGraph.Vertex instance at 0x7f300>}
>>> 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

以词梯问题开始图算法的研究。将单词"FOOL"转化为"SAGE"。按词梯问题的规则，1次只能修改1个字母，在每1步都必须将单词转换为另一个单词，不允许转换为非单词。词梯问题是由*爱丽丝梦游仙境*的作者lewis Carroll于1878年提出的。以下单词序列是刚才那个问题的一个可行解。

```
FOOL
POOL
```

```
POLL
POLE
PALE
SALE
SAGE
```

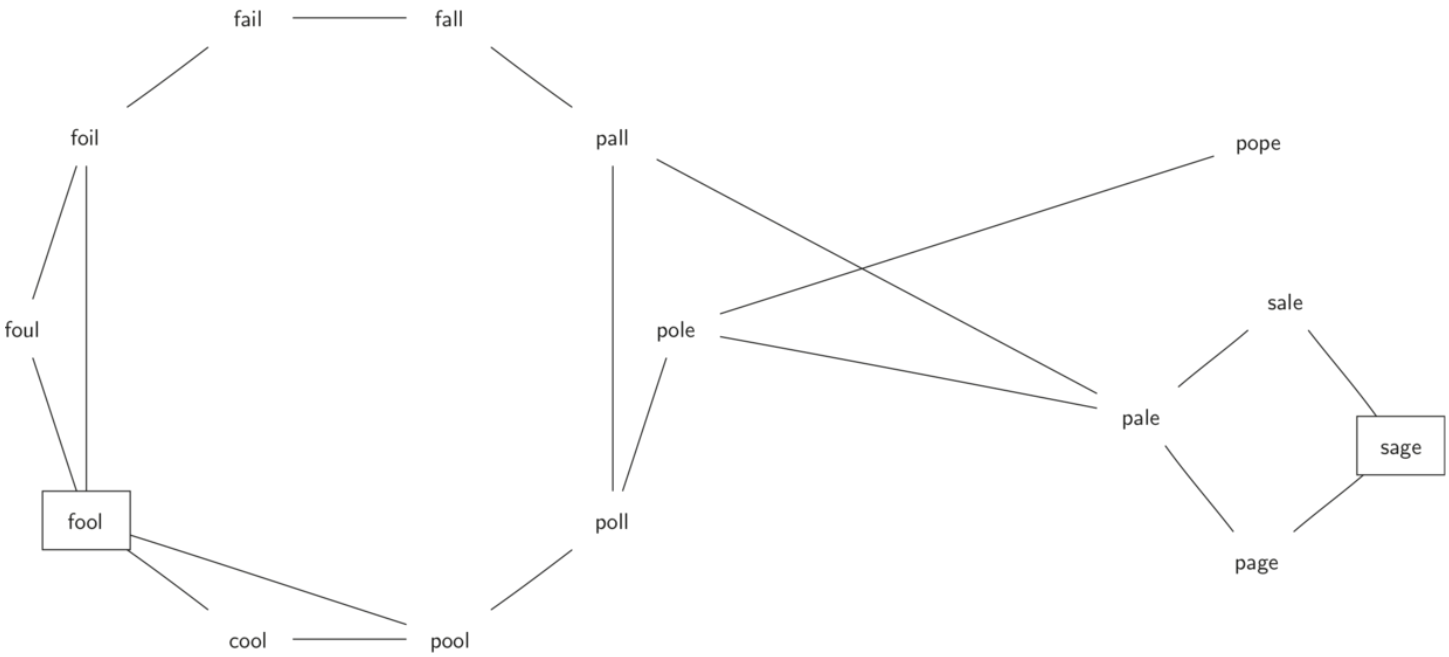
词梯问题有许多变体。比如读者可能遇到要求在给定步数来完成转换的版本，或是必须用给点单词的版本。本节仅关注计算出完成转换所需要的最小修改次数。

当然，因为本章内容是图，因此可以用图算法来将其解决。以下是纲要：

- 将单词之间的关系重新表示为图。
- 使用**广度优先搜索算法（breadth first search**来找到从起始单词到目标单词的最短路径。

7.8 建立词梯图

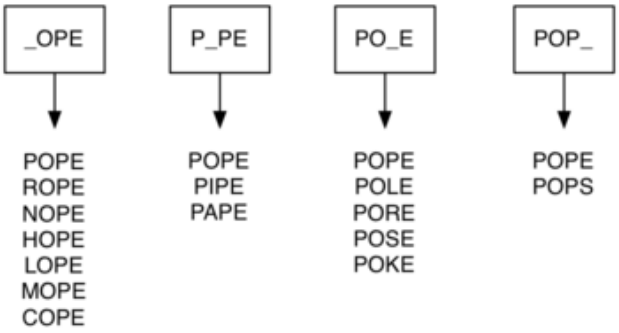
首先要解决的问题是如何将单词列表转换为图。对于边来说，需要保证它们连接的是差且仅差了1个字母的单词。如果生成这种图，那么从一个单词出发的任意路径都是该词梯问题的解。图1给出了1个小的由单词构成的图，它可以用来解决FOOL到SAGE的词梯问题。注意，该图是无向图且无权重。



../_images/wordgraph.png

可以使用很多方法来生成这个图。首先假设有1组长度相同的单词，从起点出发，为列表中的每一个单词创建1个顶点。为了确定如何将这些单词连接起来，可以将列表中的每个单词与其它的单词一一进行比较，在比较时确定有多少个字母不同。如果两个字母仅相差1个字母，便可以在图中为它们两创建1条边。单词数较少时，这个方法还是可行的。但是假设有5110个单词的话（粗略估计一一进行对比需要$O(n^2)$的时间复杂度），需要进行2500万次对比。

使用以下方法可以对其进行优化。假设有大量桶，每个桶外面都贴着1个4字母单词，但是字母中有1个被下划线替代。比如说，对于图2这种情况，可能会有个桶被贴上"POP_"。当在列表中对每个单词进行处理时，将该单词与每个桶进行对比，将"_"用作通配符，因此"POPE"和"POPS"都与"POP_"匹配。每当找到1个匹配的桶时，都将该单词放进去。当所有的单词都放在了正确的桶里时，可以确定同一桶里的单词是相连的。



../_images/wordbuckets.png

在Python中，可以通过字典实现上述方案。桶的标签作为字典的键，键对应的值为单词列表。首先为各个单词在图中创建1个顶点。然后为处于字典同一键下的单词之间创建边，如代码1所示。

```
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
```

这是第一个跟现实生活相关的图问题，读者可能会好奇该图的稀疏程度。对于该问题，准备的单词列表有5510个。若果使用邻接矩阵，矩阵将会有5110*5110个单元格，而使用buildGraph函数的图仅有53286条边，因此仅有0.20%的单元格被填充了，这实际上是一个很稀疏的矩阵了。

7.9 实现宽度优先搜索（breadth first search）

创建好了图，接下来便可以研究解决字梯问题的最优算法了。这里使用的算法被称为**宽度优先搜索（breadth first search,BFS）**。BFS是用于图搜索的最简单的算法之一。它也是之后将研究的其它几种重要的图算法的原型。

给定图G以及起始顶点s，宽度优先算法将对图内的边进行搜索以找到G中所有与s有路径相连的顶点。宽度优先搜索的优势在于，在找到与s距离为k+1的所有顶点之前，它会先找出所有距离为k的顶点。BFS的运行过程可以想象为生成树的过程，每次都生成树的一层。BFS在对子孙顶点进行搜索前，先将初始顶点的子顶点加入进来。

为了跟踪运行过程，BFS将顶点染成白色，灰色或黑色。所有的顶点都被初始为白色。白色顶点是未被探索的顶点。当某个顶点被初次探索时，将其染成灰色，当BFS完成对某个顶点的探索时，将其染成黑色。这意味着，对于黑色的顶点，它是没有白色节点与之邻接的。灰色节点，有可能会存在白色节点与之邻接，需要进一步探索。

此外，BFS算法使用了Vertex类的改进版。这种新的顶点类新增了3种实例变量，即distance,predecessor以及颜色。每个实例变量都有正确的getter和setter方法。这里就不作展示了，因为只是加了3个实例变量而已。

BFS从起始顶点s开始，将start染成灰色来表示当前正在对其进行搜索。对于起始顶点，另外两个值，distance和predecessor分别被初始化为0和None。最后，start被放入1个Queue中。下一步便是系统地对队列前部的顶点进行搜索。所谓搜索，即是对队列前部的某1个节点的邻接列表作迭代。每当对邻接列表中的节点作处理时，先对其颜色进行判断。如果是白色的，则该节点尚未搜索过，于是执行以下4步操作：

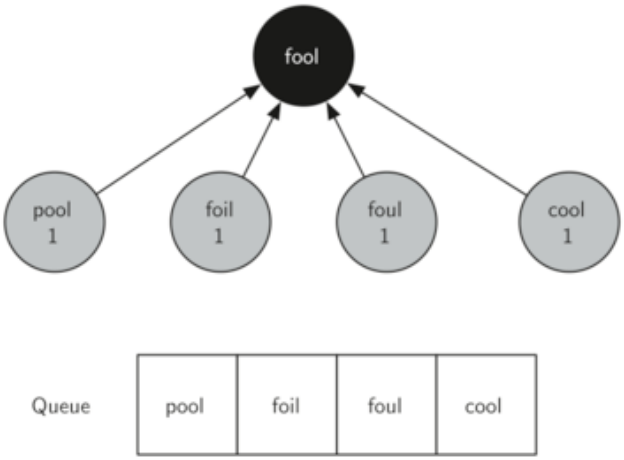
1. 将新的，未搜索的顶点nbr染成灰色。
2. 将nbr的predecessor设置为当前节点currentVert。
3. 到nbr的距离设为currentVert + 1。
4. 将nbr放入队尾。将nbr放在队尾使得该节点只有在currentVert的邻接列表中的所有其它顶点都被搜索过了才会继续被搜索。

代码2

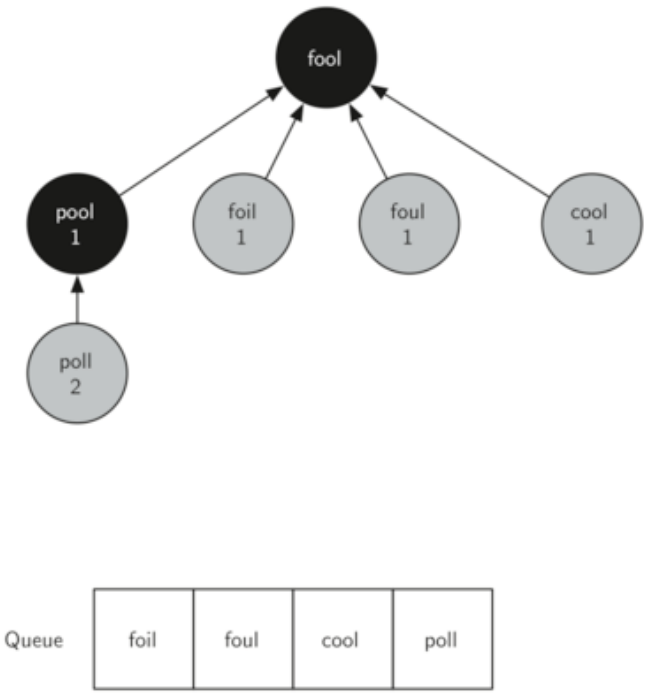
```
from pythonds.graphs import Graph, Vertex
from pythonds.basic import Queue

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')
```

现在来仔细研究下bfs函数是如何对图1构件宽度优先树的。从FOOL出发，将所有与FOOL邻接的节点连接到该树上。临近节点包括POOL，FOIL，FOUL和COOL。每个节点都放入了待搜索待新节点。图3给出了该操作中的树及完

成后的队列。

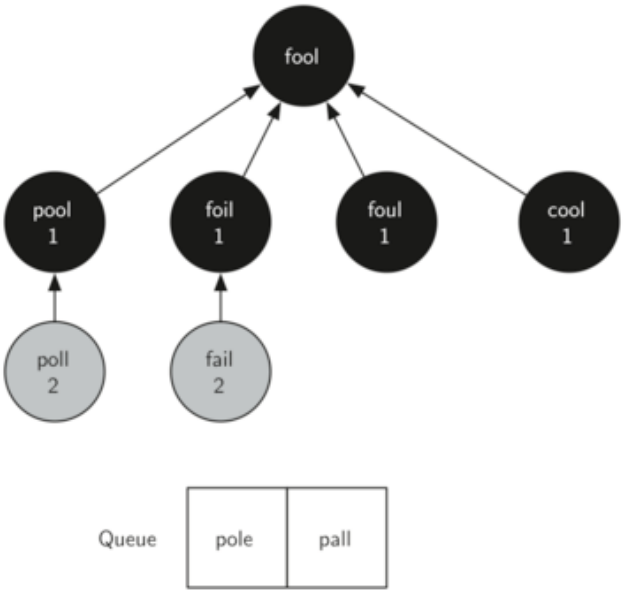


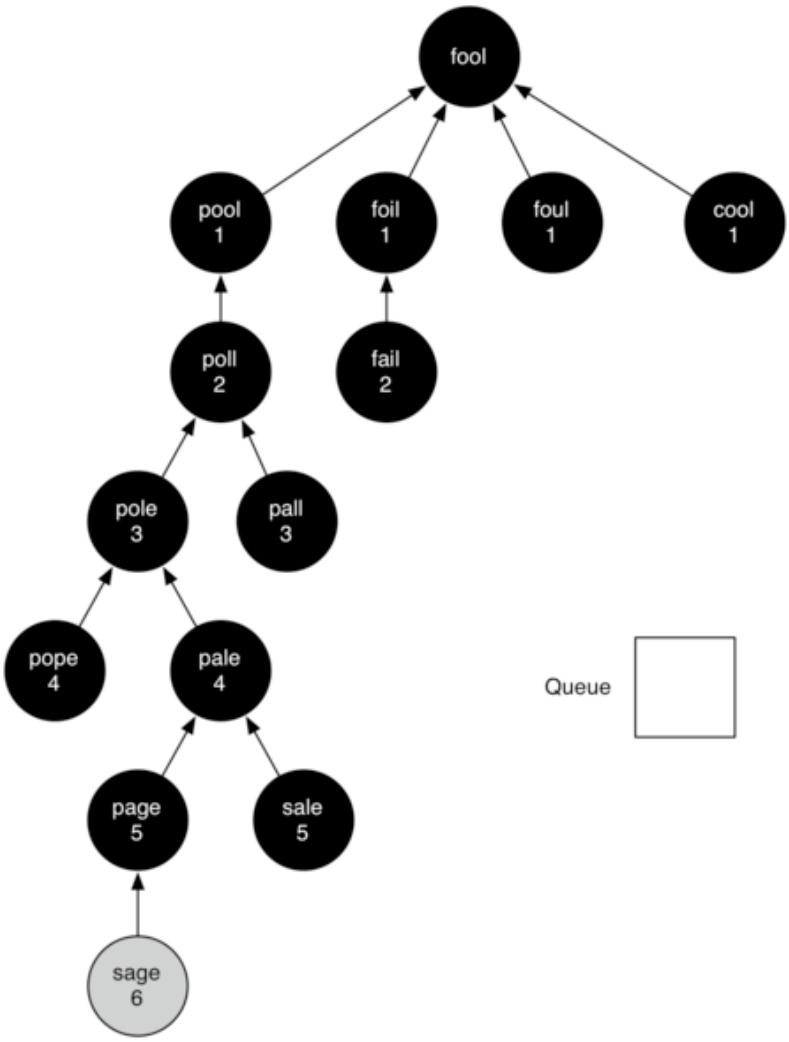
接下来，bfs从队首去掉了下一个节点(POOL)，并对其所有邻接节点重复该操作。然而，当bfs检查到节点COOL时，它发现COOL的颜色已经是灰色了。这意味着有更短的路径到达COOL，并且COOL已经留在队列中以待进一步搜索。在对POOL进行监测时唯一新加入队列的是POLL。新状态如图4所示。



../_images/bfs2.png

队列中的下一个顶点是FOIL。FOIL可以加入树的节点是FAIL，当bfs方法继续处理队列时，接下来两个节点都没有向树或者队列加入新的节点。图5展示的是对树的第二层所有顶点都探索完成了后的树和队列。





读者应该自己动手过一遍该算法，以对该算法获得更好的理解。图6是图3中已完成所有顶点的搜索后的最终宽度优先搜索树。宽度优先搜索的惊人之处在于，它不仅解决了初始的FOOL-SAGE问题，也顺便解决了许多其它问题。从宽度优先搜索树中的任一节点出发，沿着父节点的箭头返回到根节点，便可以得到该单词变为FOOL的最短词梯。代码3沿着父节点链打印出了词梯：

代码3

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

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

7.10 宽度优先搜索分析

在继续研究其它图算法前，先来分析一下宽度优先搜索算法的运行性能。首先应当观察到的是，图|V|中的每个顶点在循环中最多被处理一次，因为在进行监测和加入队列前，该顶点必须是白色的。因此整个循环为O(V)。嵌套在while中的for循环对图中的每条边最多执行1次，|E|，因此每个顶点最多出队1次并且只在节点u出队时才检查节点u到节点v的边，该for循环为O(E)。因此两个循环的最终结果是O(V+E)。

当然，执行BFS仅完成了该任务的一部分。从起始节点出发到达目标顶点是该任务的另一部分。最坏情况是，该图是单链的。在这种情况下，遍历所有节点需要O(V)。正常情况下应该是|V|的某个分数，但时间复杂度不会变，仍然是O(V)。

最后，至少对这哥问题而言，生成初始的图也需要时间，其时间复杂度的分析就作为练习了。

7.11 骑士周游问题

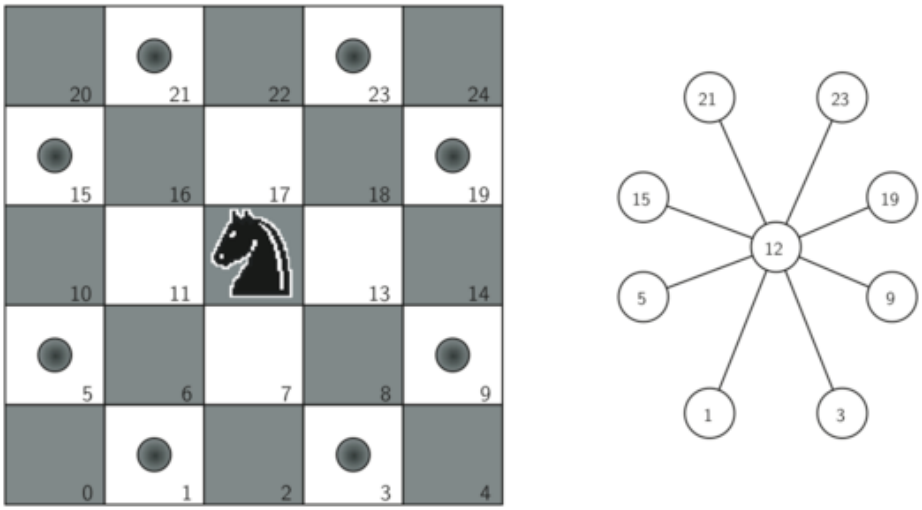
起始周游是可以用来演示第二种常见图算法的经典问题。骑士周游问题是在国际象棋棋盘上仅使用1个棋子，即起始，该问题的目标是找到1个出棋序列使得骑士可以访问各个方格恰好1次。这样的序列被称为周游。起始周游问题多年来吸引了很多棋手、数学家和计算机科学家等。在8*8的棋盘上，可行的序列数的上界为1.305×10^{35}。然而，还有很多时候会走到死棋的情况。当然，还有很多时候会失败。显然这是一个需要些智慧和计算力的问题。

学者已经提出了很多该问题的接发，图搜索是最最容易理解和编程的。同样地，分两步来求解：

- 将骑士在棋盘上符合规则的移动以图表示。
- 使用图算法来找到长度为$rows \times columns - 1$的路径，其中每个顶点都恰好只访问1次。

7.12 生成骑士周游图

为了将骑士周游问题表示为图，使用以下两种思想：棋盘上的每个方格都被表示为图中的一个节点；骑士的每个合法的移动都被标记为图中的边。图1是骑士的1中合规走法，并给出了其对应的在图中的边。



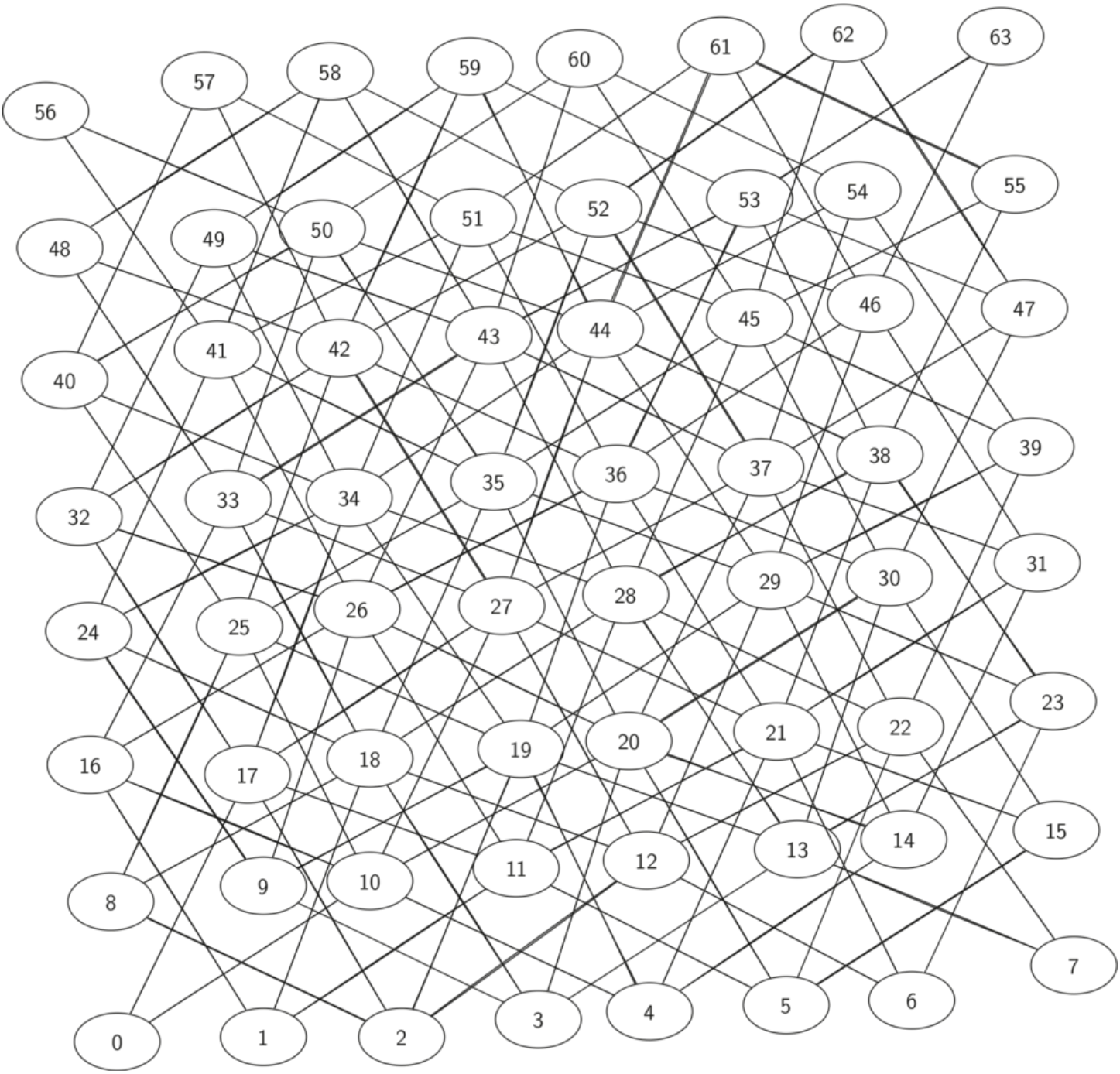
../_images/knightmoves.png

为$n \times n$的棋盘生成图，其Python代码如代码1所示。knightGraph函数对整个棋盘作1次遍历，在棋盘上的每1个方格，kinightGraph函数都调用1个辅助函数genLegalMoves，为棋盘上的该节点处的合法走法生成1个列表。图中的所有合法走法最后都被转换成了边。另一个辅助函数，posToNodeId将棋盘上的1个位置按照其行列转换为线性节点编号，如图1所示。

代码1

```
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
```

代码2中的genLeagalMove函数将以骑士的位置作为参数，并创建了可能的8个走法。legalCoord辅助函数用来保证创建的某个走法是在棋盘内的。该图中有336条边，并且可以发现，棋盘边缘的顶点的合规走法比棋盘内部的顶点少，此外该棋盘是稀疏的，填充率只有8.2%。



../_images/bigknight.png

7.13 实现骑士周游

解决骑士周游问题将使用**深度优先搜索（depth first search）**。上一节讨论的宽度优先搜索算法是一次为搜索树建立一层，而深度优先算法则尽量向枝的深处搜索。本节介绍两种实现DFS的算法。第1个算法是专门用来解决骑士周游问题的，它显式地要求各节点最多被访问1次。第2种实现更加通用，但会允许在生成树时对其某个节点访问不止一次。在后续章节中，基于第2种算法开发了其它的算法。

图的深度优先搜索很适合用来找到1条由63条边构成的路径。当深度搜索算法发现1条死路（图中的某个节点，使得接下来没有合法的移动了），它便会返回，并且朝着有合法移动的顶点的最深处移动。

骑士周游函数接受4个参数：n，搜索树当前的深度；path，到该节点为止已访问过的顶点；u，待搜索节点；limit，路径中的节点数。knightTour函数是递归的。当knightTour函数被调用时，它首先检查约束条件。如果某条路径含有64个顶点，将True从knightTour返回，表示找到了一条可行的周游路径。若该路径长度不够，则继续向更深处搜索——————选择1个新顶点以搜索并递归调用knightTour。

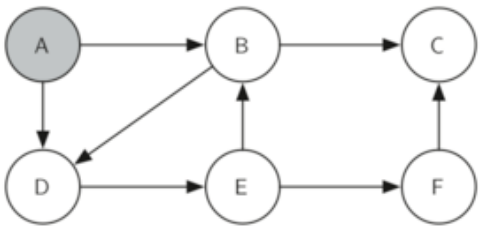
DFS也使用颜色来记录图中哪些节点已经被访问，没有被访问过的顶点被涂为白色，访问过的是灰色。若某一顶点所有的邻接顶点都被搜索过了并且还没有达到64个顶点的长度的话，则说明当前是一条死路，此时必须要进行回溯，这里是通过从knightTour返回False来实现回溯的。在宽度优先搜索中使用队列来记录需要访问的节点。因为深度优先搜索是递归的，因此其实是隐式地使用了栈来进行回溯。当knightTour函数返回False的时候，程序仍处于while循环中，并在nbrList中寻找下一个顶点。

代码3

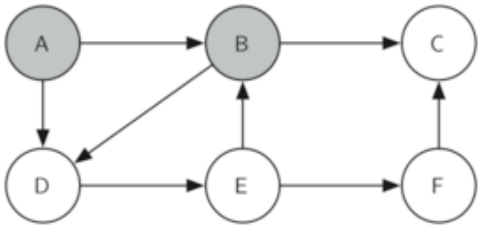
```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
u.setColor('gray')
path.append(u)
if n < limit:
nbrList = list(u.getConnections())
i = 0
done = False
while i < len(nbrList) and not done:
if nbrList[i].getColor() == 'white':
done = knightTour(n+1, path, nbrList[i], limit)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

下面来简单地试运行。读者可以参考下面的图例来观察该搜索的步骤。作为例子，假设getConnections方法返回的节点是按字母排序的。首先调用knightTour(0,path,A,6)。

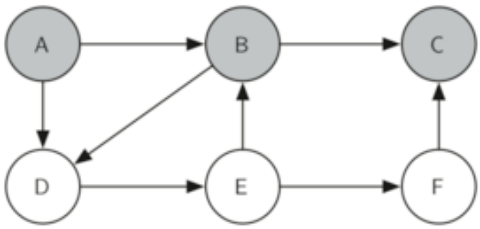
如图3所示，knightTour从节点A开始。A的邻接节点为B和D，由于B在D之前（字母排序），DFS选择B来对下一级进行搜索（图4）。通过递归调用knightTour实现对B的搜索。由于C、D是B的邻接节点，因此knightTour接下来选择C以继续搜索。然而，如图5所示，节点C是一条死路，因为它没有白色的邻接节点。此时，将节点C的颜色改为白色，knightTour返回False。实际上该递归函数的返回导致了回溯到节点B以继续搜索（如图6所示）。节点B中的下一个可探索节点是D，因此knightTour函数继续进行递归调用，直到再次遇到节点C（如图8、9、10）。然而，这次在节点C进行n<limit的测试结果是False，因此可以确认图中所有节点已经被遍历完。此时返回True表示已经找到1条实现周游的路径，返回该列表path，其值[A,B,D,E,F,C]，它即是实现对图中各节点访问恰好1次的顺序。



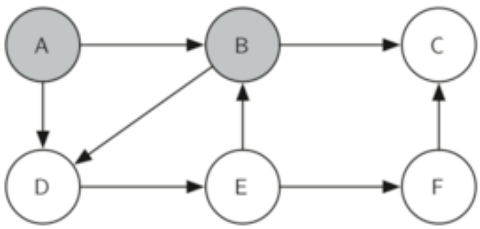
../_images/ktdfsa.png



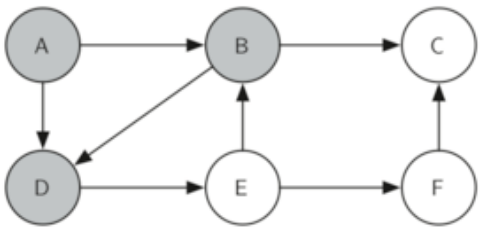
../_images/ktdfsb.png



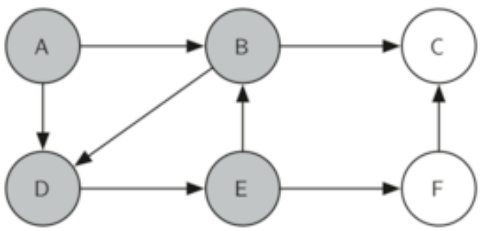
../_images/ktdfsc.png



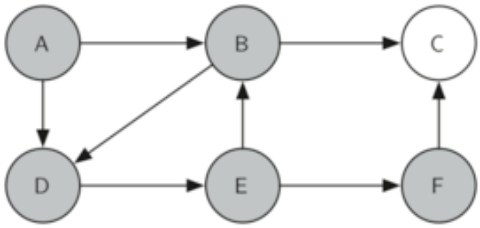
../_images/ktdfsd.png



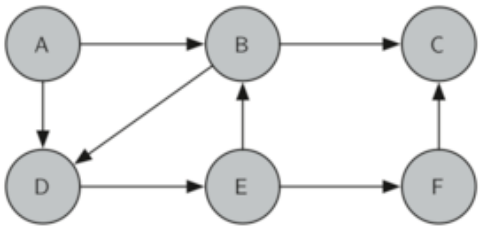
../_images/ktdfse.png



../_images/ktdfsf.png

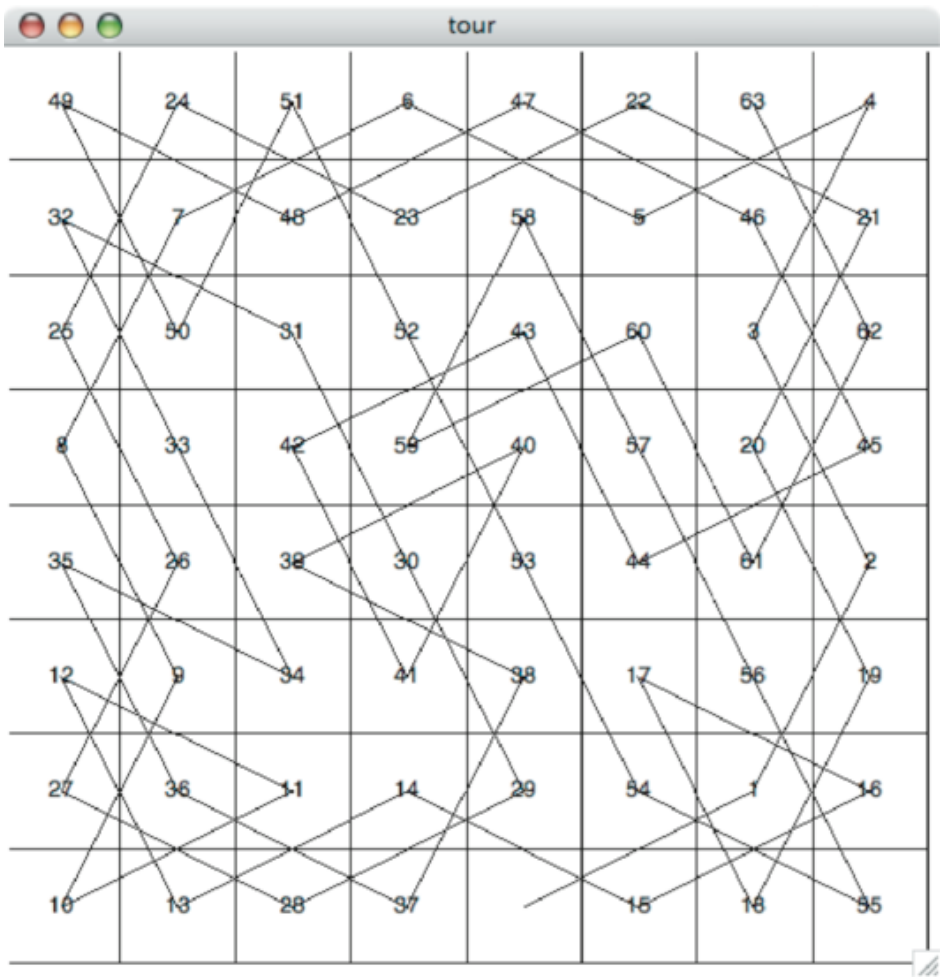


../_images/ktdfsg.png



../_images/ktdfsh.png

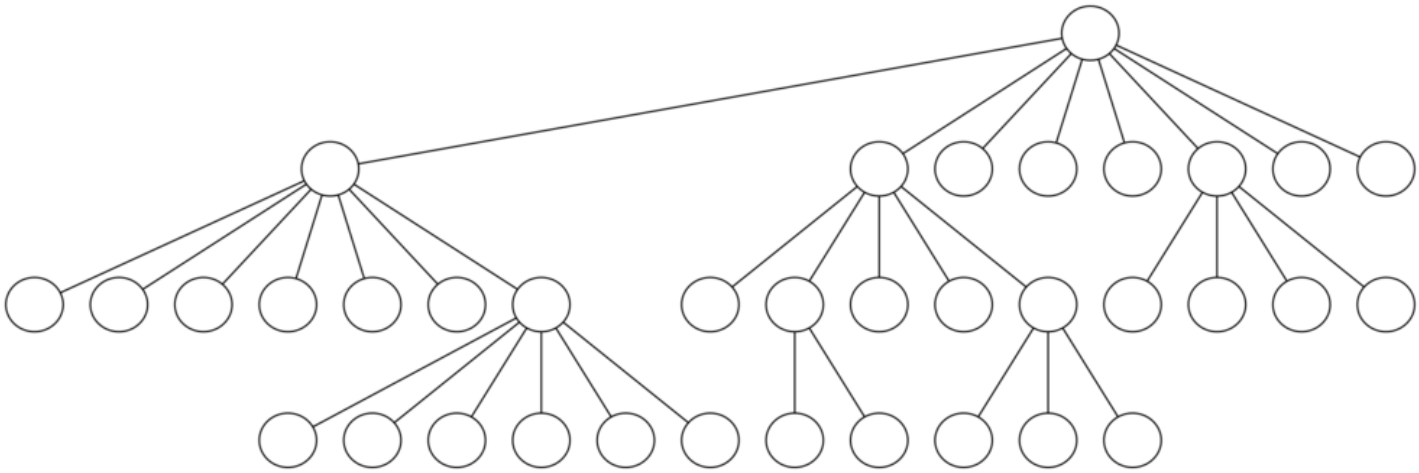
图11是8*8棋盘中的周游例子。可以看出有多种可行的路径，并且其中有一些还是堆成的。改进一下的话，读者也可以获得起始与结束为同一个格子的循环路径。



../_images/completeTour.png

7.14 骑士周游分析

关于骑士周游问题还有1个要点即性能需要讲解，然后再继续介绍深度搜索的通用版本。特别地，knightTour高度依赖于顶点搜索次序的方式。比如说，*在5*5*的棋盘上在还算比较快的电脑上于1.5秒内计算出1条路径，但对于*8*8的棋盘来说又是怎样的？在这种情况下，取决于计算机的速度，有可能需要半小时才能计算出答案。原因是当前实现的骑士周游解法是时间复杂度为$O(k^N)$的算法，其中N是棋盘中方格数，而k是1个小常数。图12以图形的方式进行了阐释。树的根节点代表的是搜索树的起点，从起点出发，算法生成并检查每一个合法的移动。之前也说到过，合法移动数的个数依赖于骑士在棋盘上的位置。在角的时候，仅可能有2个合法的移动；在与角相邻的位置则可能有3个合法的移动；而在棋盘中间的话，可能会有8个。图13给出了棋盘上每个位置可能的合法移动。在树的下一层，同样又有2-8个可能的合法移动。需要检查的位置对应于搜索树中节点个数。



../_images/8arrayTree.png

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

../_images/moveCount.png

读者已经知道，高度为N的二叉树的节点数为 $2^{N+1}-1$ 。子节点树可达8的树的节点数是远远大于二叉树的。由于每个节点的分枝因子是可变的，因此可以用通过平均分枝因子来估计节点数。值得注意的是，该算法是指数级的：$k^{N+1}-1$ ，其中k是棋盘的平均分枝因子。来看看它增长得有多块。对于*5*5的棋盘，树会有25层深（或者说24，若将第1层记为0）。平均分枝因子是k=3.8，因此搜索树中的节点数为：* $3.8^{25}-1$ *或* 3.12×10^{14} *。对于*6*6的，有 1.5×10^{23} 个节点，而对于普通的8*8棋盘，k=5.25，则有 $1.3\times10^{4}6$ 。当然，对于同一个问题可能有多重解法，因此也不必搜索每个单节点，但是这也只能是对节点数取了个分数，事实上也并不会改变其指数级的性质。至于将k表示为棋盘大小的函数，就作为练习了。

幸运的是，有种方法可以将8*8的情况控制在1秒内完成，如下面的代码4所示。其中orderbyAvail函数是在u.getConnections被调用时所使用的函数。行10是orderByAvail中最关键的1行。该行保证了选择的下一步顶点是可能的合法走法最少的那个。读者可能觉得这有些起反作用，为什么不选择可行走法最多的？读者大可以自己试试。

在选择路径的下一个顶点时采用可能合法数最多的那个会导致的问题是，该机制下会倾向于在早期便访问棋盘的中间，这样一来，骑士很容易会困在棋盘的一侧而不能访问棋盘另一侧中未访问的方格。而如果采用合法数最少的那个，则强迫其实首先访问棋盘的边缘，这样一来，骑士一开始便访问了"很难达到"的角落，然后在有必要时利用中间的方格来达到棋盘的另一侧。利用这种知识来加速算法被称为"启发式"算法。人类日常中也是利用启发式规则来作出各种决定的，启发式搜索常常用于AI领域。本例使用的启发式算法被称为Warnsdorff算法。

代码4

```
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]
```

7.15 通用深度搜索

骑士周游是DFS的1个特例，它的目标是生成深度最大且无分枝的树。更通用的深度优先搜索实际上还要简单一点。它的目标是尽可能深地对树进行搜索，连接图中尽可能多的顶点并且在必要时进行分枝。

深度优先搜索甚至可能创建多棵树。当深度优先搜索算法生成了一组树时，便称之为深度优先森林。同宽度优先搜索一样，深度优先搜索在构造时也利用了父节点引用。此外，深度优先搜索在Vertex类中还会使用额外两个实例变量，即发现时间和完成时间。发现时间记录某个顶点第1次出现前的步骤数，而完成时间记录该顶点被涂为黑色时的步骤数。观察算法可以发现，发现时间和完成时间能够提供一些有趣的性质可供后续的一些算法使用。

深度优先搜索如代码5所示。因为两个函数dfs和其辅助函数dfsvisit使用了1个变量来记录在调用dfsvisit时的时间，从而将这部分代码实现为Graph子类的一个方法。这里给出的实现通过增加了1个time实例变量以及dfs、dfsvisit方法将Graph类进行了扩张。仔细研究行11，可以发现dfs方法对图中所有的顶点进行了遍历，并对白色节点调用dfsvisit方法。之所以对所有顶点进行迭代而不从给定起点开始搜索，是为了保证图中所有节点都被考虑了并且在DFS树中没有顶点被一楼。for aVvertex in self可能看起来有点奇怪，但实际上这里的self是DFSGraph的一个实例，对Graph实例中的所有顶点进行迭代是很常规的操作。

代码5

```
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)
```

虽然这里给出的bfs实现只考虑了那些可以通过某条路径回到起点的节点，但是也可以生成表示图中各节点最短路径的BFS森林。这就作为练习了。在后续的2个算法中，可以看出记录DFS森林的重要性。

dfsvisit方法从startVertex这一单节点出发，尽可能地探索所有邻接白色顶点。如果读者仔细地研究dfsvisit的代码并且将其于BFS对比，可以发现dfsvisit算法几乎等同于bfs，除了for循环中的最后1行，dfsvisit递归地调用其本身以向更深的层继续搜索，而bfs将该节点加入队列供以后搜索。有趣的是，bfs使用的是队列，而dfsvisit使用的是栈。当然读者并不能在代码中看到栈，实际上是通过递归调用dfsvisit而使用了栈。

下面的1组图演示了用于小图的深度优先搜索算法的实际运行。在这些图中，虚线表示检查过的边，但边的另一端的节点已经被加入了深度优先树。在代码中，这是通过检查另一节点的颜色为非白色来实现的。

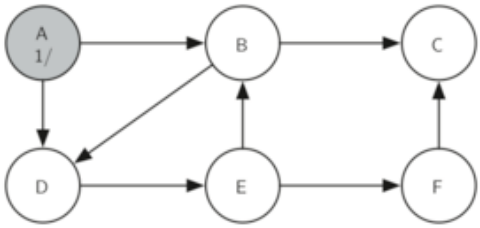
搜索从图中的顶点A（图14）开始。由于在搜索开始时，所有的顶点都是白色的，算法首先访问了顶点A。访问节点的第一步是将该节点的颜色设置为灰色，表示该节点正在被搜索，并且把发现世界设置为1。由于顶点A有两个邻接顶点（B，D），每一个都需要被访问，因此就随意地按字母顺序进行访问了。

接下来访问的是节点B（如图15），因此将其颜色设置为灰色，并将其发现时间设置为2。顶点B与（C，D）邻接，因此按照字母顺序访问节点C。

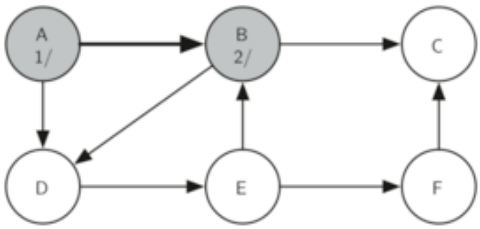
访问顶点C（图16）后来到了树的一个分枝的末端。将节点涂成灰色，将其发现时间设置为3，并且算法也确定了C是没有邻接顶点的，这意味着已经完成了对节点C的搜索，将其涂成黑色，并且将完成时间设置为4。此时的状态如图17所示。

由于顶点C是一个分枝的末端，现在回到顶点B，并继续搜索B的邻接顶点。除了C以外仅剩D了，因此现在访问D（图18），并且从顶点D继续搜索。通过顶点D便来到了顶点E（图19），顶点E有两个邻接顶点，B和F。正常情况下会按照字母顺序来访问邻接顶点，但是由于B已经被涂成灰色了，算法对其进行识别，确认不能访问B，因为这样一来该算法便陷入了无限循环了。因此，算法从顶点F继续（图20）。

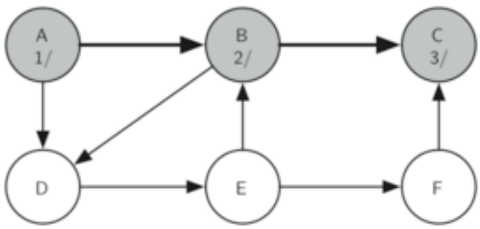
顶点F仅有1个邻接顶点C，但是C是黑色的，因此就不用进行搜索了：算法已经到达了另一分枝的末端。



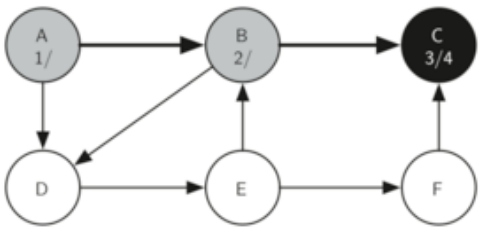
../_images/gendfsa.png



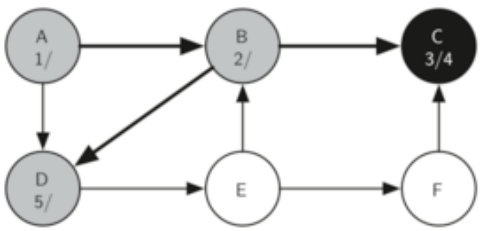
../_images/gendfsb.png



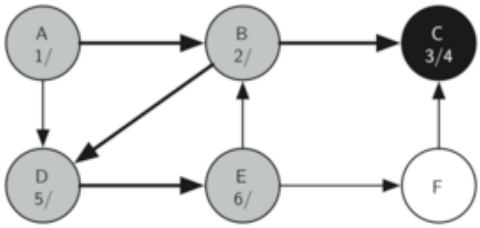
../_images/gendfsc.png



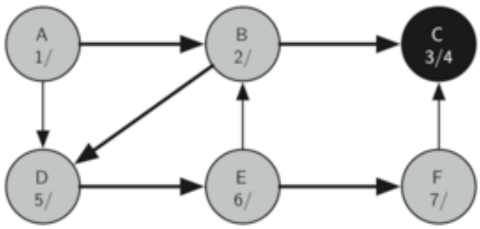
../_images/gendfsd.png



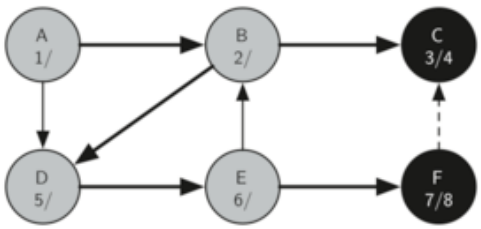
../_images/gendfse.png



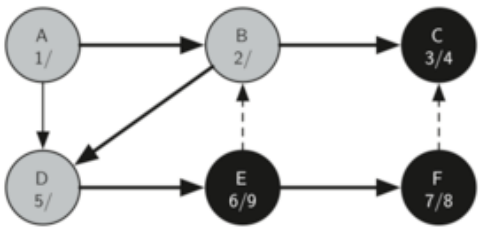
../_images/gendfsf.png



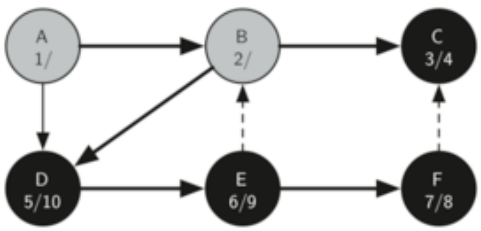
../_images/gendfsg.png



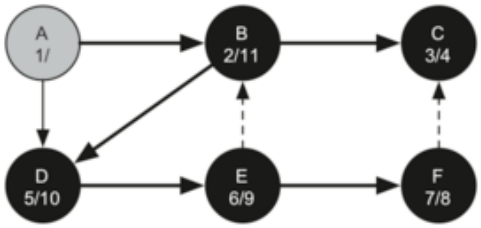
../_images/gendfsh.png



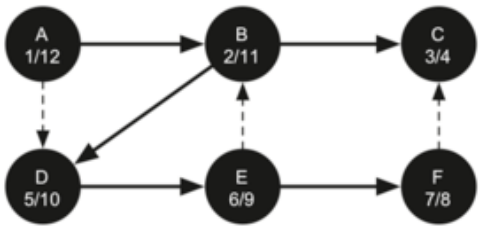
../_images/gendfsi.png



../_images/gendfsj.png



../_images/gendfsk.png



../_images/gendfsl.png

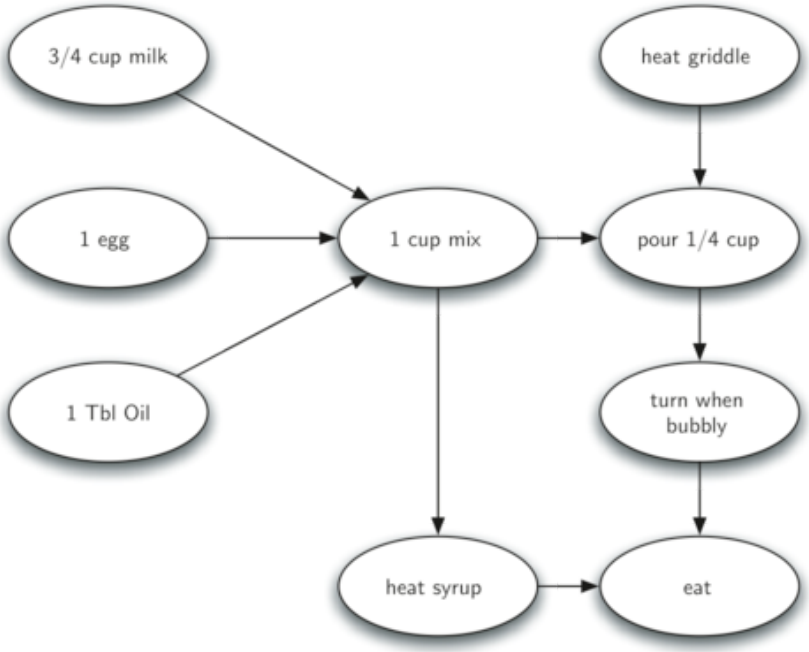
每个节点的发现和完成时间表示了**括号属性（parenthesis property）**，意味着某个节点在深度优先树中的所有子节点的发现时间都比其父节点晚，而结束时间比其父节点早。

7.16 深度优先搜索分析

深度优先搜索一般运行时间分析如下。不考虑dfsvisit的具体情况，dfs中的循环都是O(V）的，因为它们都是遍历图中各顶点。在dfsvisit中，遍历当前顶点的邻接列表中的边。由于dfsvisit仅在顶点为白色时进行递归调用，该循环对图中的每条边最多执行1次，或者说O(E)。因此，DFS的总时间是O(V+E)的。

7.17 拓扑排序

计算机科学家几乎可以把所有问题都转化为图问题，下面以1个复杂问题作演示，即制作煎蛋饼。配方很简单，1个鸡蛋，1杯面粉，1勺油以及$\frac{3}{4}$杯牛奶。为了制作煎饼，必须要加热平底锅，将所有材料混合在一起，然后将混合物用勺子放入热好的锅中。当煎饼开始冒泡时，将其反转过来，煎至底部变为金黄色。在享用煎饼之前，也可以加一些果酱。图27将以上过程表示为图。



../_images/pancakes.png

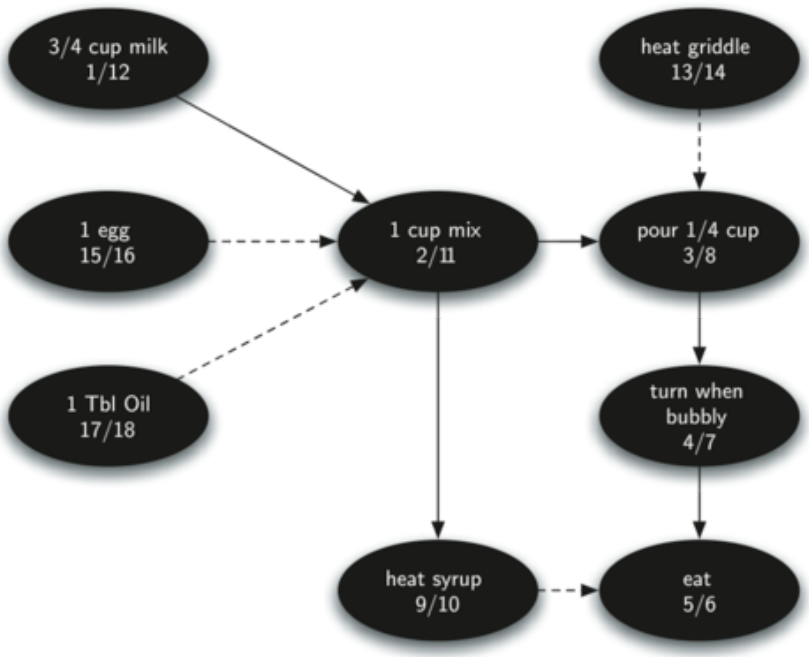
制作煎饼的难点在于确定第一步干啥。如图27所示，可以从加热锅开始，也可以从加入任意原料开始。为了确定制作煎饼的每一个步骤的准确顺序，可以使用图算法中的**拓扑排序（topological sort）**来求解。

拓扑排序将有向无圈图转化为其顶点的线性排列，使得对于含有边(v,w)的图G的排列中，顶点v在顶点w之前。有向无圈图在很多应用程序中用来表示事件的优先级。制作煎饼就是一个例子，另一些例子还有软件工程规划，数据库请求优化优先级以及矩阵乘法等问题。

拓扑排序是深度优先搜索的一种简单而强大的改进。拓扑排序算法如下：

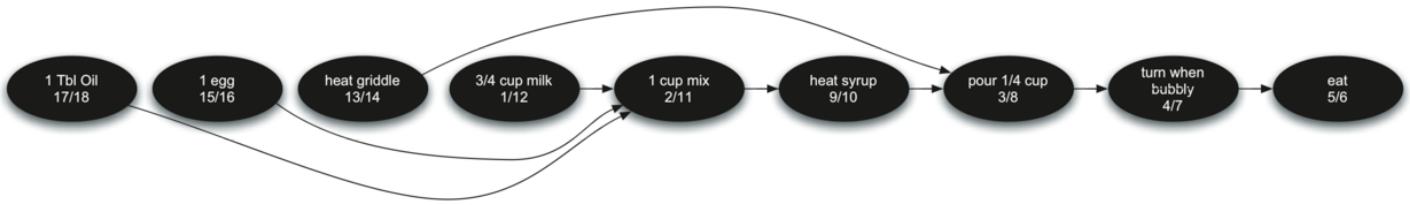
1. 对于某个图g调用dfs(g)。调用深度优先搜索的主要目的是为了计算每个顶点的完成时间。
2. 按照完成时间以降序排列的方式将顶点存储于一个列表中。
3. 返回拓扑排序的降序排列列表。

图28是制作煎饼的图（图26）对应的深度优先树。



../_images/pancakesDFS.png

最后，将拓扑排序算法应用到图中的结果如图29所示。

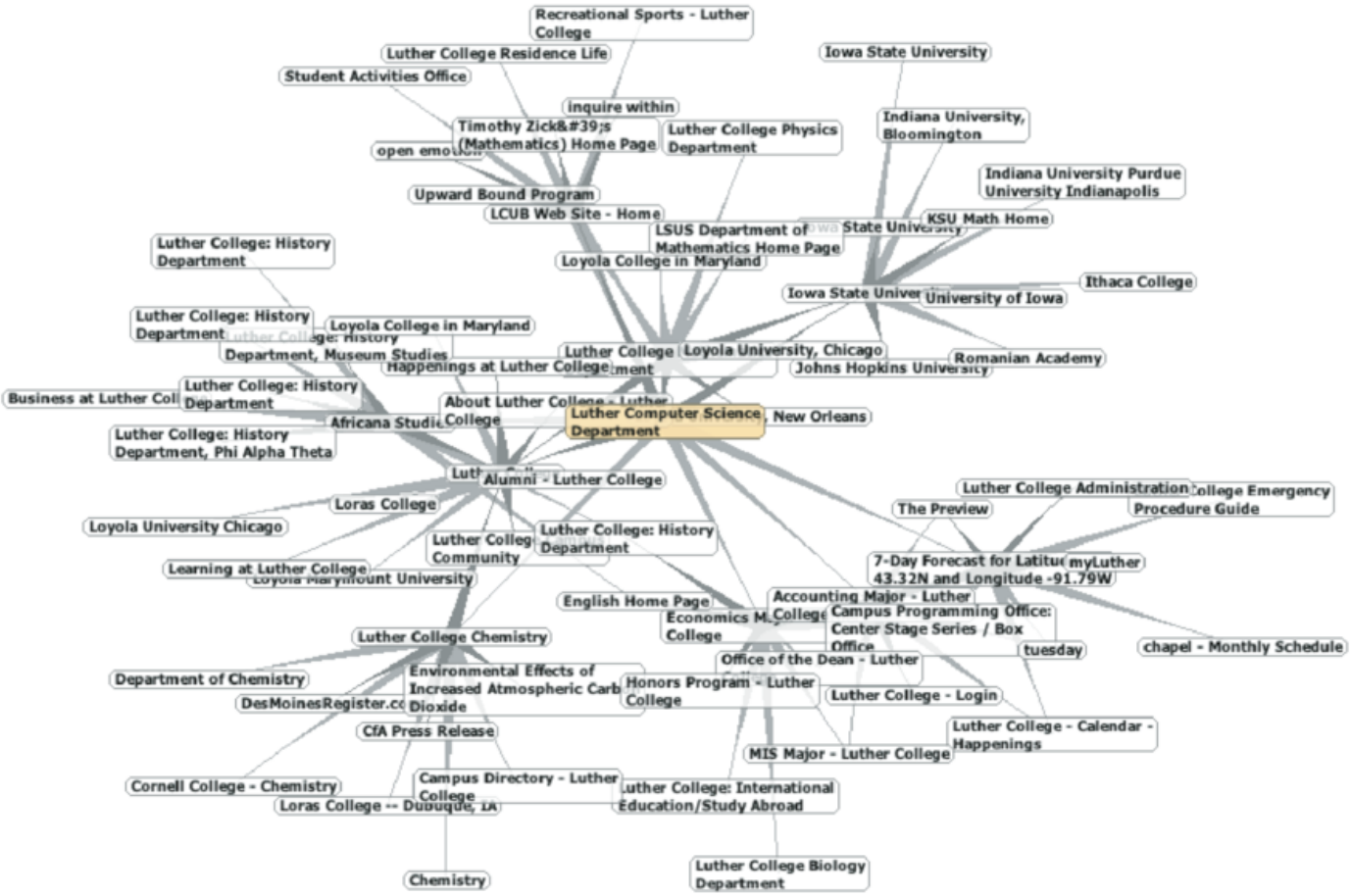


../_images/pancakesTS.png

7.18 强连通分量

本章剩余内容将集中于超大规模图，使用由互联网上主机及网页间的连接构成的图作为学习另外一些图算法的材料。首先是网页。

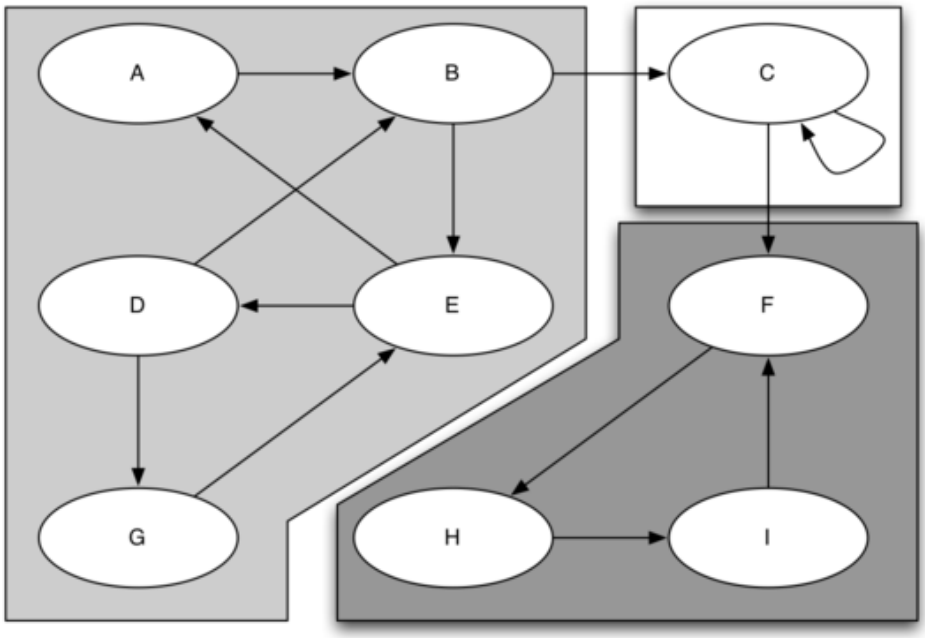
搜索引擎比如Google和Bing都利用了这一特点：互联网上的网页实际上构成了一个超大规模有向图。为了将万维网（World Wide Web）转化为图，可以将网页当作顶点，而网页上的超链接则作为连接顶点的边。图30是以Luther College's Computer Science主页为起点，沿着网页之间的连接生成的图的一小部分。当然，这个图完整版是很大的，因此这里将连接数限制为不超过10个。



../_images/cshome.png

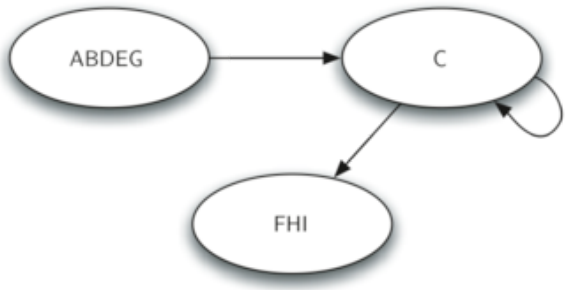
仔细研究图30中的图，读者会发现一些有趣的东西。首先，图中的许多其它网页也是Luther College的网页。第二，有好几个连接指向了Iowa的其它大学。第三，还有些连接指向了其它人文艺术学院。据此可以猜测，或许网页存在某种内在结构使得可以将具有类似结构的网页进行聚类。

强连通分量算法（strongly connected component，SCC）可以用来识别图中强连通顶点中的聚类。定义图G的**强连通分量（记为C）**为：以某种划分方式将顶点集合划分为不含重复元素的、元素数尽量多的子集，并且保证这些子集中的顶点任意两两之间存在路径相通。图27是一个有3个强连通分量的图，不同的强连通分量通过不同的阴影区区分开来。



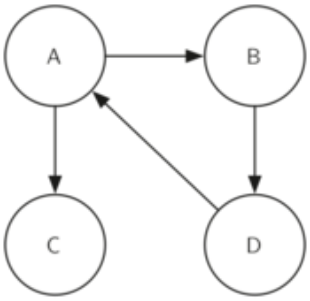
../_images/scc1.png

识别了强连通分量，便可以将强连通分量中的所有顶点表示为单个大顶点来对图进行简化。图31的简化如图32所示。

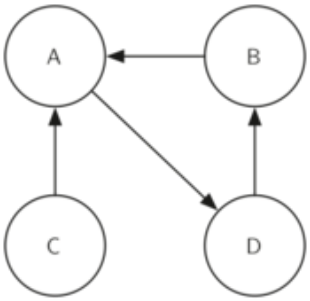


../_images/scc2.png

同样地，这里将再次使用强大而高效的深度搜索算法。在研究SCC算法的主要内容前，必须先研究另一个概念。图G的**转置（transposition）**记为G^T，是指所有边都被转向后形成的图。也就是说，若原始图中节点A到节点B存在有向边，则该图的转置G^T中对应的有向边是节点B到节点A。如图33和图34所示。



../_images/transpose1.png



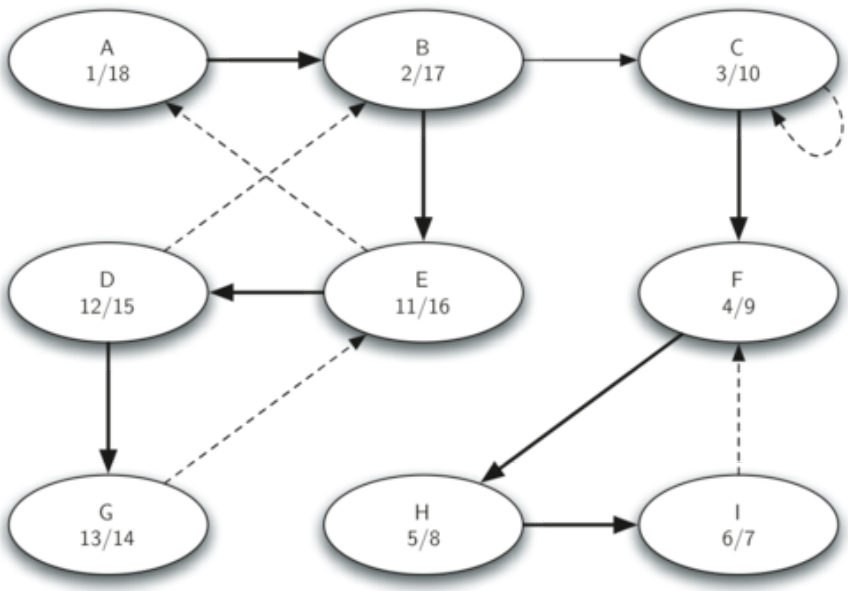
../_images/transpose2.png

再来看看这些图。注意图33中的图有2个强连通分量。而图34中，也有2个强连通分量。

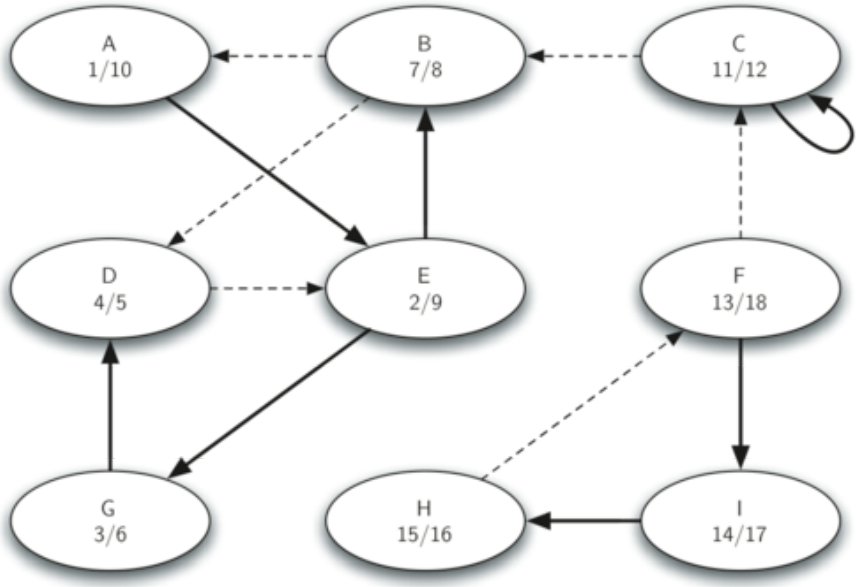
现在来设计用来计算图中强连通分量的算法。

1. 对图G调用dfs来计算每个顶点的结束时间。
2. 计算G^T。
3. 为图G^T调用dfs，但在DFS的主循环中按结束时间降序搜索每个顶点。
4. 在第3步中计算的森林中的每棵树都是1个强连通分量。为森林中的每棵树的每个顶点输出其顶点id，以示区分。

以之前的图31为例，跟踪上述步骤执行结果。图35是通过DFS算法计算的原图的发现和结束时间。图36是在转置图上运行DFS后计算出的发现和结束时间。

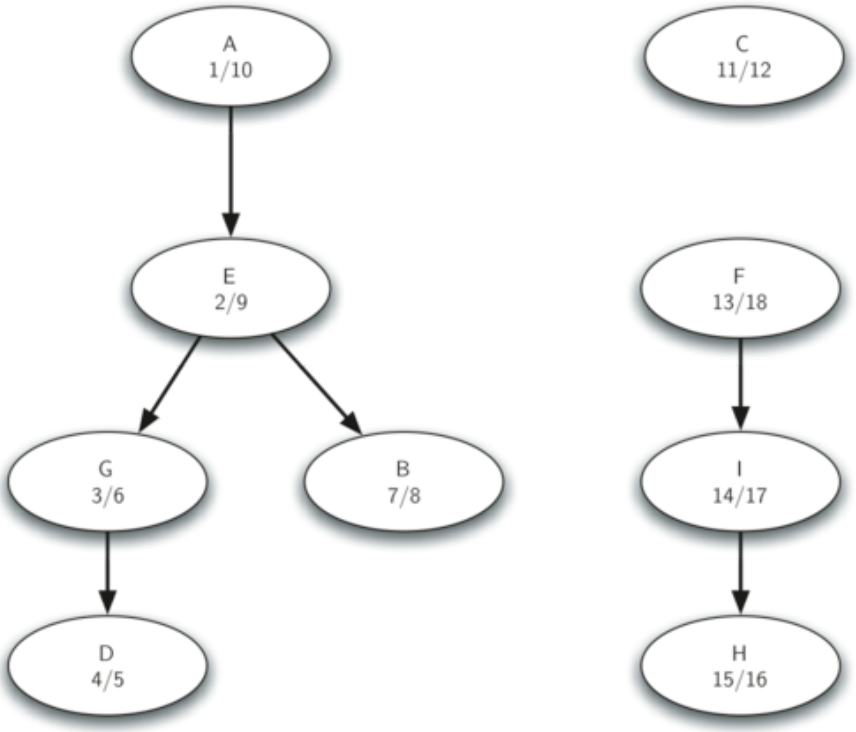


../_images/scc1a.png



../_images/scc1b.png

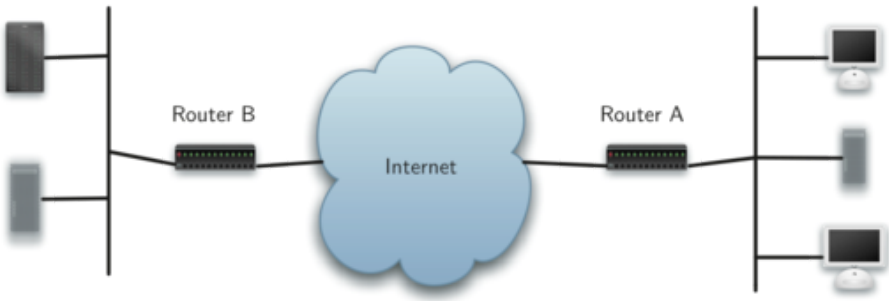
最后，图37给出了强连通分量算法中第3步生成的3颗树。SCC算法的代码就作为练习了。



../_images/sccforest.png

7.19 最短路径问题

当读者在网上冲浪（？），发送邮件，或者从校内另一个地方登陆实验室电脑时，为了将本地计算机的信息传送到另一台计算机，其实是要做很多事情的。对在计算机之间通过网络传递信息的方式的深入研究是计算机网络课程不可缺少的首要课题。但是在这里对计算机网络知识仅要求足以理解另一个非常重要的图算法即可。

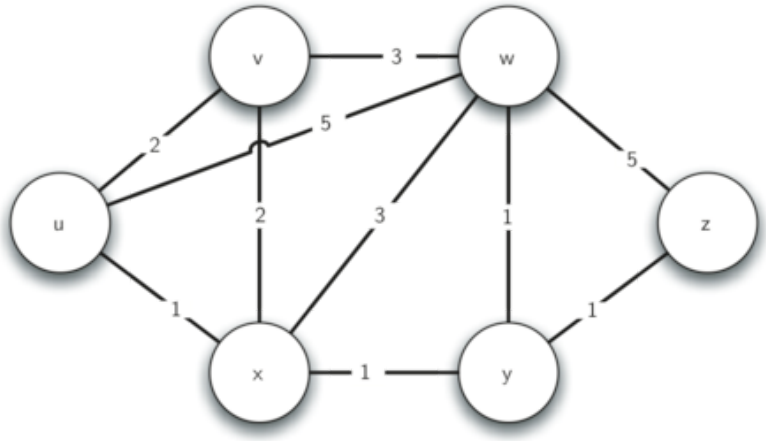


../_images/Internet.png

图1是对计算机通信方式的高度抽象概括。当使用浏览器向服务器请求网页时，该请求必须通过路由器穿过本地网络到达互联网，然后该请求在互联网上进行传送并最终抵达服务器所在的本地网络的路由器，所请求的网页再通过相同的一系列路由器组传回正在使用的浏览器。在图1中云的标签"Inernet"是一些其它的路由器，这些路由器用来实现信息传输。若读者电脑支持traceroute命令，可以看到实际上有很多路由器的。下面的文本可以看出，该traceroute命令结果显示在Luther College和University of Minnesota的邮件服务器之间有13个路由器。

```
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!)

Routers from One Host to the Next over the Internet
```

互联网上的每个路由器都连接着1个或多个路由器，因此如果多次运行traceroute命令，有可能会发现每次信息都是通过不同的路由器转发的。这是因为在任意对路由器间的连接都是有消耗的，取决于流量，当前实际，以及其它很多因素。现在将路由器网路表示为权重图应该说是很顺理成章了。



../_images/routeGraph.png

一个小权重图例子如图2所示。需要解决的问题是找到一条总权重最小的路径来传递信息，这与宽度优先搜索有些类似，不过这里要考虑的是总权重而不是节点的总数，如果各节点权重一致，那就跟宽度优先搜索一样了。

7.20 ## Dijkstra算法

这里用来确定最短路径的算法是Dijkstra算法。Dijkstar算法是一种迭代算法，用来计算从某个起始节点到另一个节点的最短路径。这也与宽度优先搜索的结果比较类似。

为了跟踪从起始节点到每一个目的地的总权重，在Vertex类中使用了dist实例变量。dist保存了从起点到目的地最小权重路径的当前总权重值。算法将图中的每个顶点迭代1次，然而对顶点进行迭代的顺序是通过优先队列控制的。确定对象在优先队列中顺序的值是dist。dist被初始化为一个非常大的值。

从理论上来说，可以将dist设置为无限大，但实际上设置为当前问题的某个上界值即可。

Dijkstar算法如代码1所示。当算法运行完毕时，图中各个顶点的dist都被设置为到祖先节点的路径（predecessor links，这里的predecessor应是指出发点）。

代码1

```
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)
```

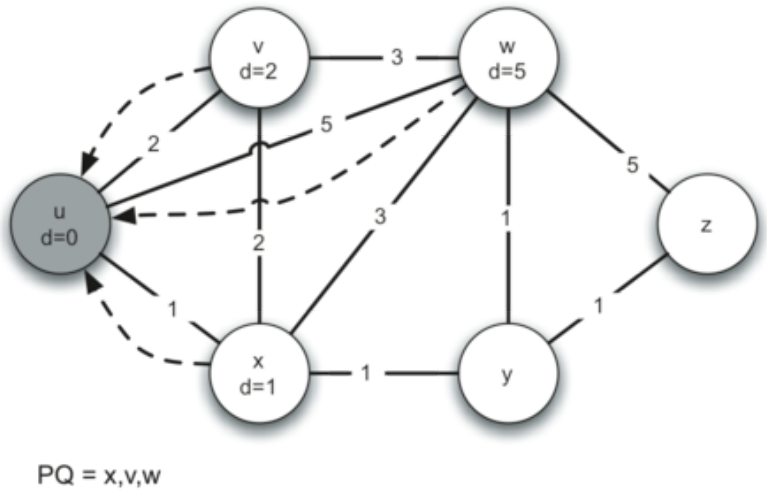
Dijjkstra算法使用了优先队列，读者可能还记得优先队列是基于第三章实现的堆的。用于Dijkstra算法的实现和第三章中的简单实现有相当差别。首先，PriorityQueue类存储的键-值对组成的元组。对于Dijkstra来说这是很重要的，因为优先队列中的键必须与图中的键对应。其次，决定优先级的是值，从而也决定了键在优先队列中的位置。在这种实现中，将到该顶点的距离作为优先级，因为当搜索下一个顶点时，应当搜索距离最小的顶点。

第二个不同之处在于，decreaseKey方法的添加。当队列中的某个顶点的dist减小的时候，调用该方法，使得该顶点向队首移动。

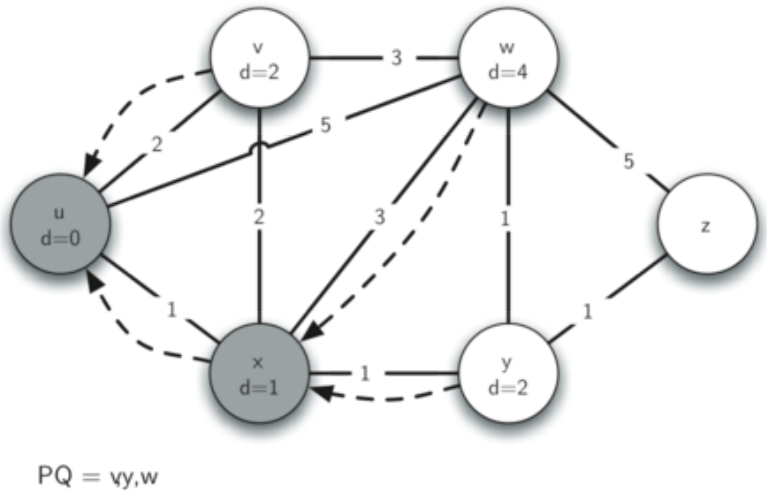
作为示范，下面的一系列图演示了Dijkstra算法的应用。从顶点u开始，u的3个邻接顶点为v,w,x，由于v,w和x都被初始化为sys.maxint，到达它们的代价即是它们的有向权重，更新到达这三个节点的代价值。将每个节点的祖先节点设置为u并且将每个节点加入优先队列中。使用距离作为优先队列的键，算法当前状态如图3所示。

在while循环的下一次迭代中，检测x的邻接顶点，之所以是x，是因为它的总代价最小，于是被弹到了队首。在x，它的邻接顶点为u,v,w,y。对每个邻接节点，检测从x到该节点的距离是否比上1个已知dist小。显然y满足条件，因为它的dist为sys.maxint，而不是u或者v，因为它们的距离分别是0和2。然而，现在可以确认到从x到w的距离比u到w的距离小。据此，更新w的dist值并且将其祖先节点从u改为x。当前状态如图4所示。

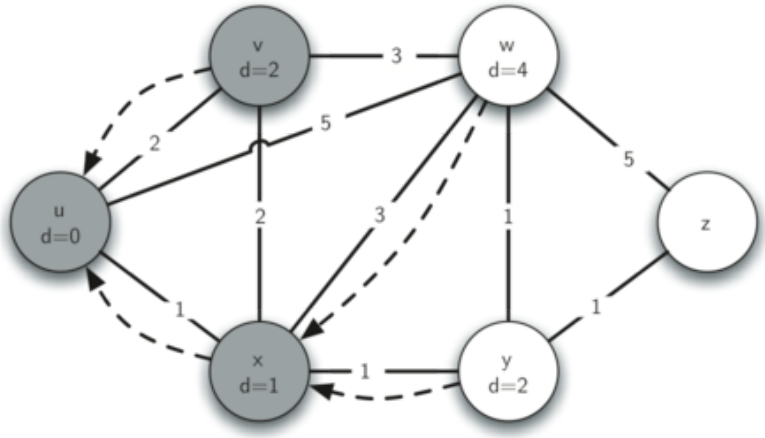
下一步是研究v的邻接节点（如图5）。在该步骤中并没有对图进行修改，因此移动至节点y。在节点y（图6）发现从它到w和z的消耗更小，因此对应地修改dist值和祖先连接。最后检查节点w和z（如图6、8所示），由于没发现需要作修改的地方，因此优先队列为空，Dijkstra算法结束。



../_images/dijkstraa.png

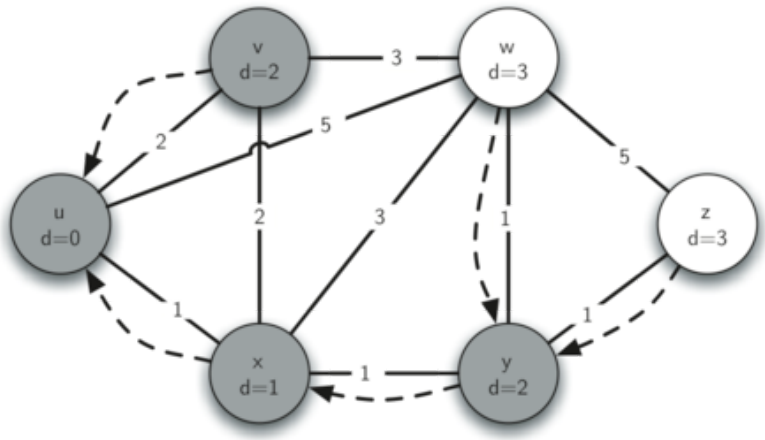


../_images/dijkstrab.png



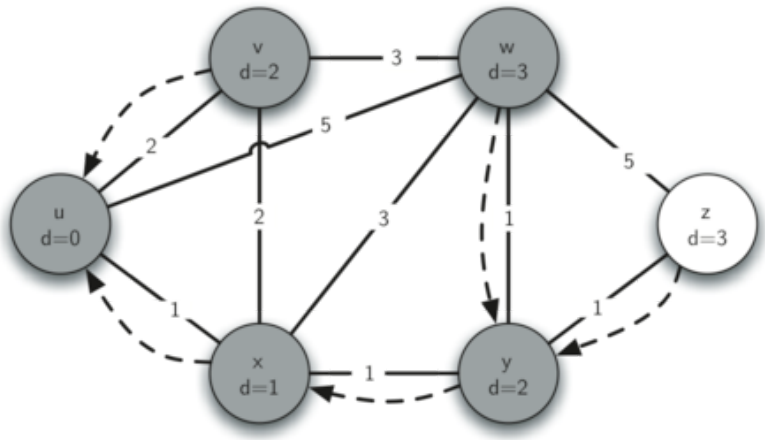
PQ = yw

../_images/dijkstrac.png



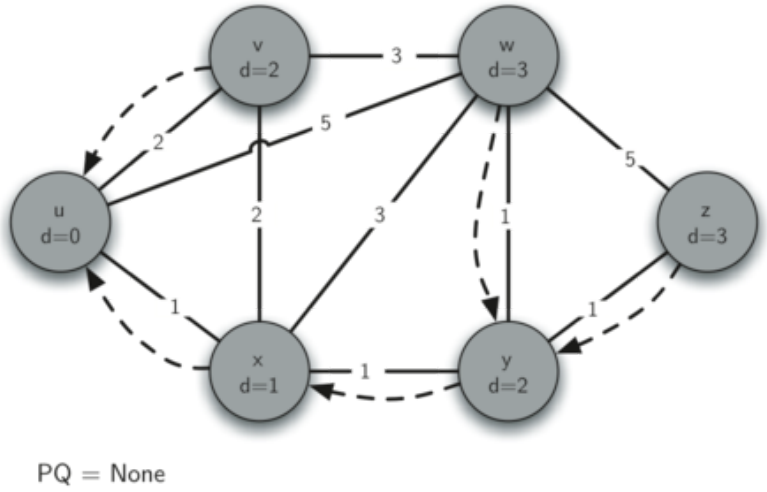
PQ = wz

../_images/dijkstrad.png



PQ = z

../_images/dijkstrae.png



../_images/dijkstraf.png

值得注意的是，Dijkstra算法只有当权重为正时有效，读者应当明白如果在某条边上引入了负值，那么该算法将陷入无限循环。

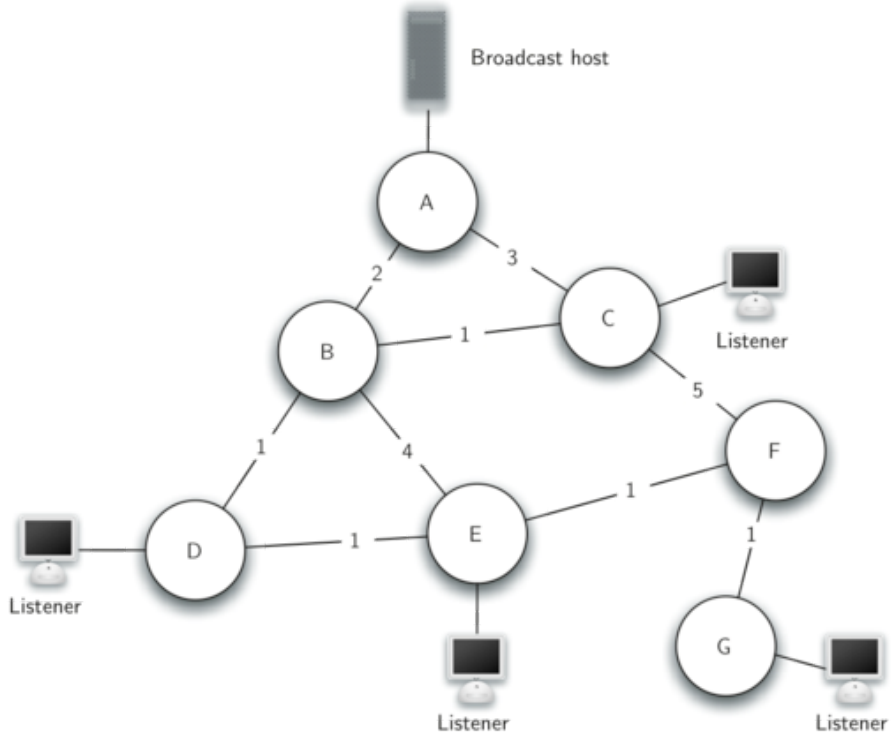
为了在互联网上传输信息，还有其它算法也可用来寻找最短路径。对互联网问题使用Dijkstra算法的问题在于必须要给出完整的图以保证算法的运行。这意味着，图中的每个路由器都要连接到互联网上所有路由器，在实践中当然不是这么作的，该算法的另一些变体可以让路由器在传输的过程中不断地探索图，比如说距离向量路由算法。

7.21 Dijkstra算法分析

现在分析一下Dijkstra算法的运行时间。首先可以注意到，生成优先队列需要O(V)的时间，因为在初始化时将图中的每个顶点都加入到了优先队列中。构造完成队列后，对每个顶点都执行1次while循环，因为顶点都是在一开始就加入然后在完成该循环后被移除。在循环内，每个对delMin的调用时间复杂度都是$O(\log V)$，时间复杂度为$O(\log V)$。综合循环内的那部分以及delMin，一共是$O(V \log(V))$。for循环对图中的每个边执行1次，在for循环内decreaseKey的调用是$O(E \log(V))$。因此总的时间复杂度为$O((V + E) \log(V))$。

7.22 Prim生成树算法

作为图算法的结尾，来考虑一个在线游戏设计者和互联网广播提供商面对的一个问题，即如何将高效地将信息传输给正在监听的人。对于游戏来说这是很重要的，如此才能知道游戏中其他人的当前位置。而对于互联网广播来说这也很重要，如此用户端才能获取还原歌曲所必须的所有数据，如图9所示。



../_images/bcast1.png

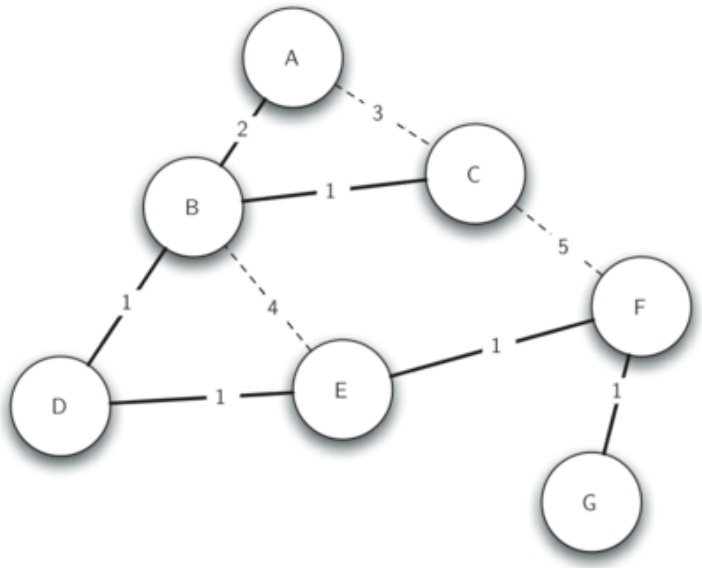
该问题当然有暴力法可以来解决，先来研究一下它们以便于更好地理解广播问题，同时也会使得读者对本书给出的解法感到赞赏。首先，广播主机有一些信息要发送给所有收听者。最简单的方法是，广播主机保存一个所有收听者的列表，然后分别对其发送信息。如图9所示，一个小网络中存在一个广播主机和收听者。使用第一种方法的话，每条信息都要发送4遍。假定使用的是损耗最低路径，来研究一些每个路由器要处理相同的信息多少次。

来自广播主机的所有信息都要经过路由器A，因此A对于每条信息都要处理4次。路由器C只用为它对应的收听者处理1次。然而，B和D对于每条信息都要处理3次，因为B和D是在通向收听者1，2，3的损耗最低路径上。考虑到广播主机每秒必须发送数百条消息，那么如此一来便会导致巨大的流量开销。

一种暴力解法是，广播主机对于每条消息只发送1条，然后由路由器来对其分类。在这种情况下，最简解是一种叫做**无控泛洪（uncontrolled flooding）的方法。该泛洪策略步骤如下。每条消息都被设置了一个存活时间（time-to-live value）**，该值大于等于广播主机与最远收听者间的边数。每个路由器都会收到一份消息，然后将该消息传送给*所有*它的邻接路由器。传输过程中，每经过1个路由器ttl值减1直到为0。很明显，无控泛洪会比第一种方法生成还要多的无用消息。

本书给出的解法基于最小权重生成树。以下给除正式定义：对于图G=(V,E)，最小权重生成树T是E的一个无圈子集且连接了V中所有的顶点。这样可以保证，T中所有边的总权重是最低的。

图10是广播的一个简化示意图，并突出显示了图中的最小生成树的边。现在来求解广播问题，广播主机只用向网络中发送一条消息，每个路由器将该信息传输到其位于生成树上的邻接顶点，并且排除已发送过消息的邻接顶点。每个路由器最多只发送1条消息，并且相关的所有收听者都会收到一份消息。



../_images/mst1.png

使用Prim算法来解决该问题。Prim算法属于贪心算法，因为每一步都选择了局部最优，在本例中即是选用权重最低的边。最后一步是开发Prim算法。

构建生成树的基本思想如下：

```
While T is not yet a spanning tree
   Find an edge that is safe to add to the tree
   Add the new edge to T
```

难点在于如何"找到安全边（Find an edge that is safe）"。将安全边定义为由生成树中的某个顶点连接至另一个尚不存在于生成树的顶点的边。这可以保证生成树始终保持为树的形状，因此是无圈的。

Prim算法的Python实现如代码2所示。Prim算法与Dijkstra算法有相似之处，它们都使用了优先队列来选择下一个顶点以添加至图中。

代码2

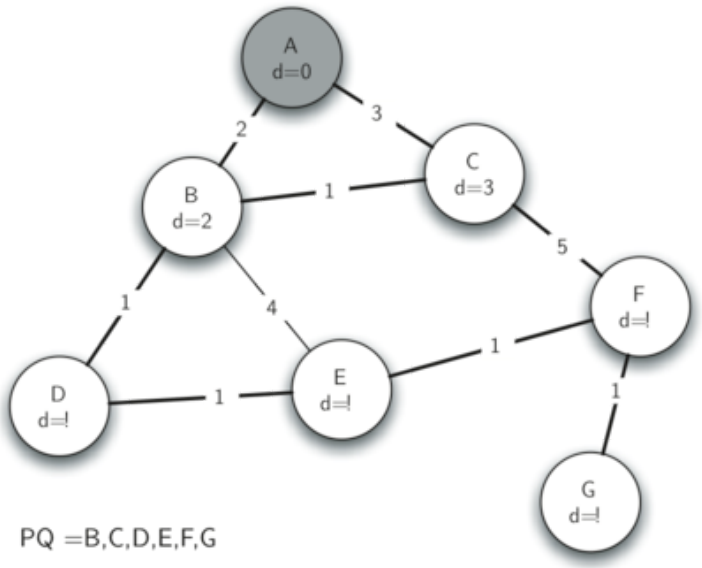
```
from pythonds.graphs import PriorityQueue, Graph, Vertex

def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert)
          if nextVert in pq and newCost<nextVert.getDistance():
```

```
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(nextVert,newCost)
```

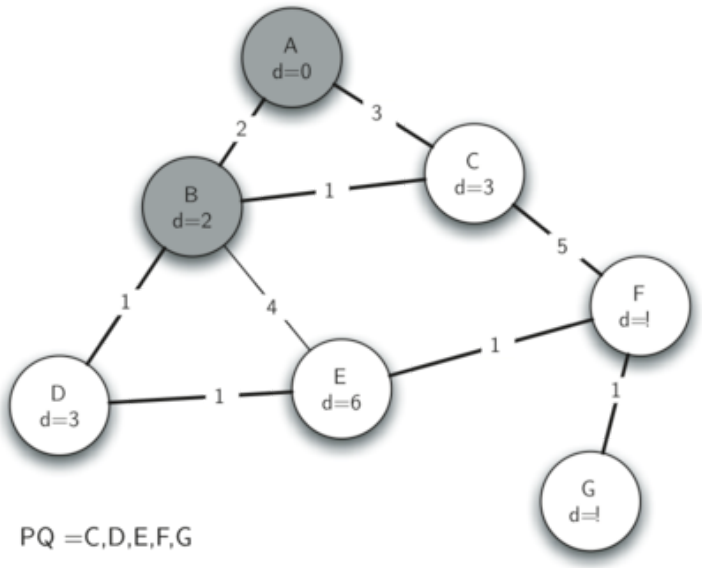
下图（图11到图17）演示了该算法的运行。以顶点A作为起点，其它任意顶点的dist都被设置为无穷大，对A，可以为B和C更新其dist值，因为通过A到B、C的距离并不是无穷，于是将B和C移动到队首，更新B和C的祖先连接为指向A的引用。值得注意的是，到此为止还没有正式地将B和C加入生成树中，当节点从优先队列中移除时，才认为其加入了生成树。

由于B的距离值最小，接下来便研究B，检查B的邻接节点，发现了D和E，于是更新D和E的dist值以及predecessor link，在优先队列中继续往下走找到的是C。C的邻接顶点中仍在优先队列中的是F，因此更新F的dist，然后调整F在优先队列中的位置。

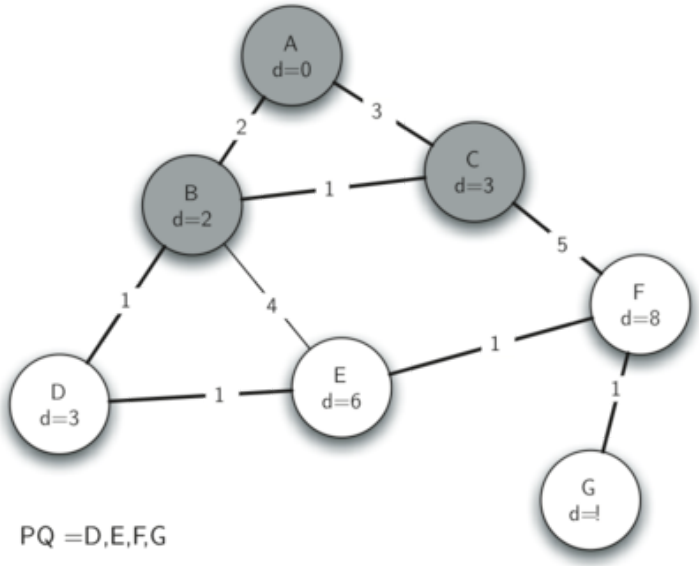
现在检查节点D的邻接顶点，发现应该将E的dist从6更新为4，将E的predecessor link修改为指向D，这样便使得它可供移入生成树中的另一个位置。算法的其余部分如预期一样，不断将新的节点加入到树中。



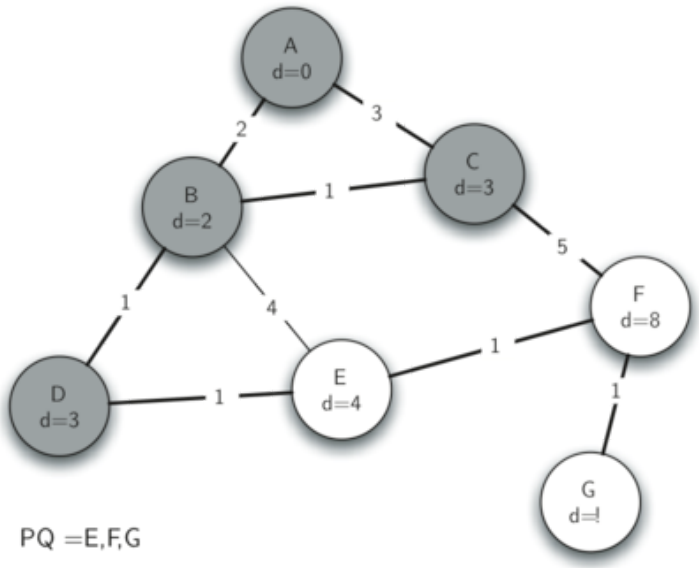
../_images/prima.png



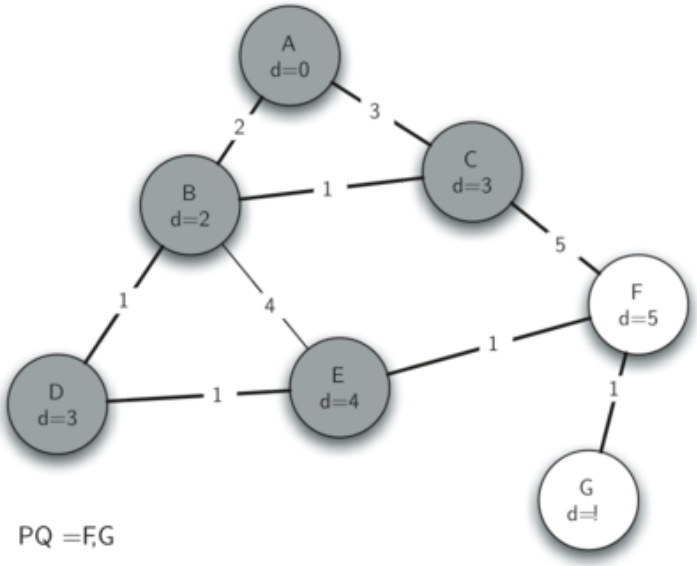
../_images/primb.png



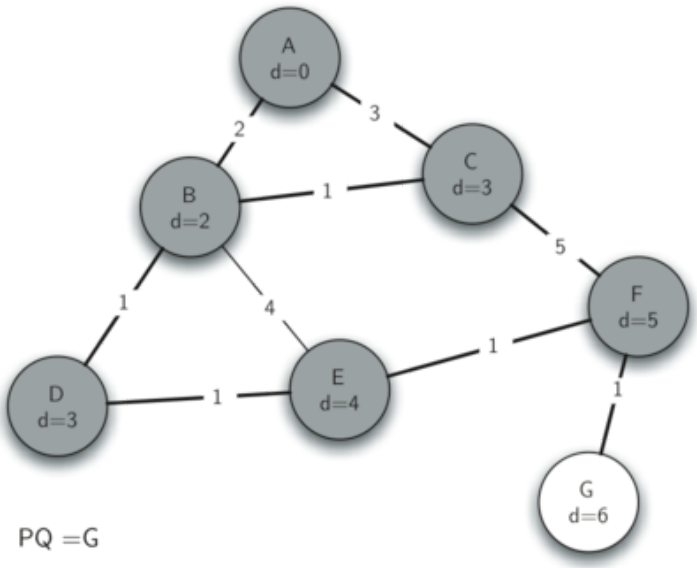
../_images/primc.png



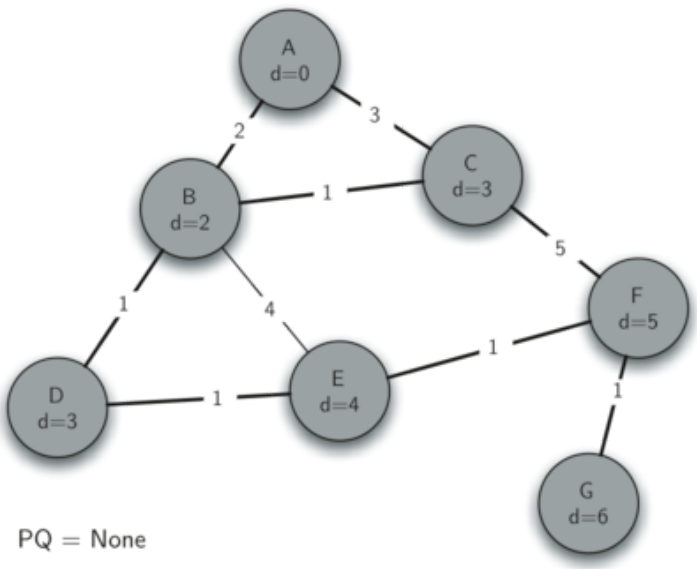
../_images/primd.png



../_images/prime.png



../_images/primf.png




../_images/primg.png