# 致谢

**2015 地空数算课程教材第 7 章翻译小组**

**组长**

刘松吟

**组员**

曹越、汪诗舜、石永祥、韩甲源、段鉴书、赵玖桐、陈哲萌、陈春含、张子玄、庞磊（排名不分先后）

**本文为教材第 7 章**

# 7. 图和图算法

## 7.1. 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
- 了解图如何用于解决不同领域的问题。

在这一章我们学习图。图是比我们上一章学习的树更普遍的结构，事实上你可以认为树是一种特殊的图。图可以被用来表述世界上很多有趣的事情，包括道路系统，城市间航线，因特网的联接，甚至是你完成计算机科学学位所必修的课程顺序。我们在这一章将看到一旦我们很好地表述了某个问题，我们可以使用标准的图算法来解决一些看起来非常困难的问题。

对人类来说，看懂路线图并了解不同地点之间的关系是相当容易的事情，而计算机并没有这样的能力。但是，我们也可以将路线图看作一个图。当我们这样做的时候，可以让计算机为我们做一些有趣的事情。如果你用过地图网站，你就会知道计算机可以找到一个地方到另一个地方最短、最快捷或最简单的路径。

作为学习计算机科学的学生，你可能想知道修完学位所需课程的学习顺序。图就是一个表达课程的先修依赖关系。图 1 展现了在路德学院完成计算机学位必须完成的课程及其顺序。

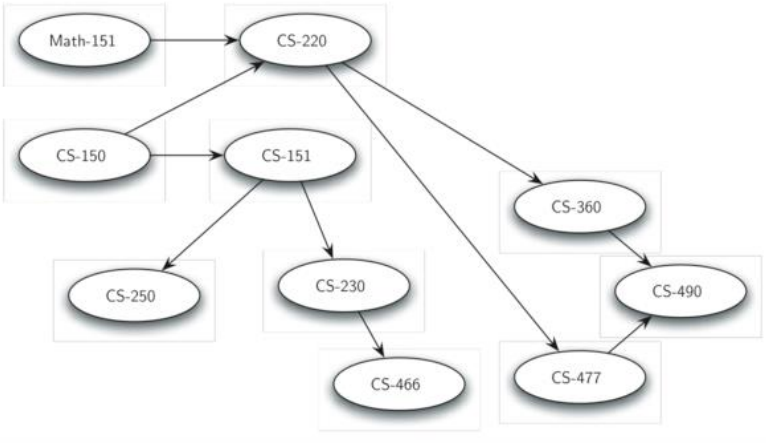


图 1: 计算机科学学位先修依赖关系

# 7.2. 词汇表及定义

看完了图的一些例子，我们可以更正式地定义一个图和它的构成要素。从我们对树的讨论，我们已经知道了一些术语。

**顶点 Vertex**

顶点（也称“节点 node”）是图的基础部分。它具有名称标识“key”。顶点也可以有附加的信息项“playload”。

**边 Edge**

边（也称“弧 arc”）是图的另一个基础组成部分。一条边连接两个顶点来显示两者具有联系。边可以是单向的，也可以是双向的。如果一个图中的边都是单向的，我们就说这个图是“有向图 directed gragh/digragh”。上图所显示的先修依赖图很明显是一个有向图，因为你必须先修某些课程。

**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，一个连接两个城市的道路图中，两个城市之间的距离就可以作为边的权重。

有了这些定义，我们可以正式定义一个图。图可以用 G=(V,E)来表述。对于图 G，V 是顶点的集合，E 是边的集合。每个边是一个元组(v,w)，w,v∈V。我们可以在边元组中加入第三个要素来表述权重。一个子图就是边 e 和顶点 v 的集合，e⊂E and v⊂V。

图 2 展现了另一个简单有向赋权图。这个图是 6 个顶点及 9 条边的集合。

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)}

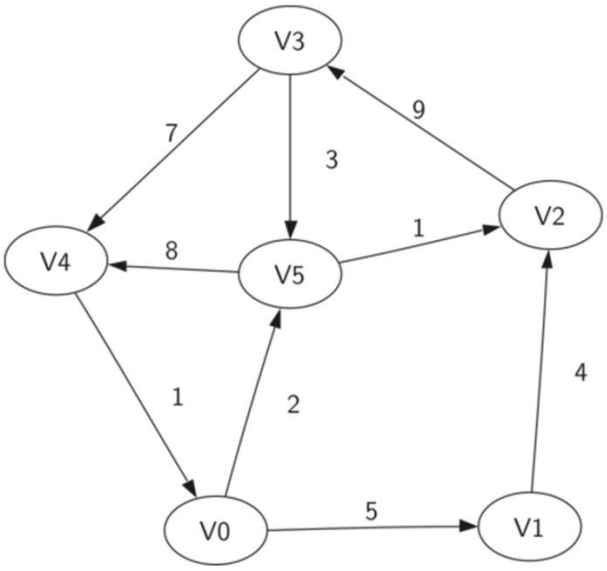

图 2: 有向图的简单例子

图 2 中的图为其他两个图的术语举了例证。

**路径 Path**

图中的路径，是由边依次连接起来的顶点序列。我们将路径定义为 P=(w1, w2, …, wn)，其中对于所有 1<=i<=n-1, (wi, wi+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之和。如图 2 中从 V3 到 V1 的一条路径是顶点序列(v3,v4,v0,v1)，其边为{(v3,v4,7),(v4,v0,1),(v0,v1,5)}。

**圈 Cycle**

有向图里的圈是首尾顶点相同的路径。例如，图 2 里路径(V5,V2,V3,V5)就是一个圈。没有圈的图称为“无圈图 acyclic graph”，没有圈的有向图称为“有向无圈图 directed acyclic graph 或 DAG”。当问题可以被表述成 DAG 时，我们可以解决一些很重要的问题。

# 7.3. 图抽象数据类型

图抽象数据类型（ADT）有如下定义：

- Graph()：创建一个空的图；
- 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

有几种方法可以在 Python 实现图抽象数据结构（ADT），需要在不同的应用中加以权衡。图的实现有两个主要的方法，邻接矩阵 adjacency matrix 和邻接表 adjacency list。我们将解释这两种不同的选择，并作为 Python 的类来实现邻接矩阵。

## 7.4. 邻接矩阵

对图的最直观实现方法是采用二维矩阵。矩阵中的每行和每列都代表图中的顶点。如果顶点 v 到顶点 w 之间有边相连，则值在 v 行和 w 列的矩阵分量中加以体现。当两个顶点通过边来连接，它们就是邻接的。图 3 展现了图 2 中图的邻接矩阵。矩阵分量代表了从顶点 v 到顶点 w 边的权重。

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

图 3:图的邻接矩阵表示

邻接矩阵的优点是简单，对于简单的图来说很容易看出节点之间的联系状态。然而，我们也注意到大部分的矩阵分量是空的，这种情况我们称矩阵“稀疏”。矩阵并不是一个储存稀疏数据的有效途径。事实上，在 Python 里你必须不厌其烦地制造图 3 这样的矩阵结构。

当边的数量庞大时，邻接矩阵是图的一个良好的实现方法。但是我们所说的庞大指的是什么呢？需要多少边来填充矩阵？因为图中每个顶点对应着一行一列，填满矩阵的边的数量是|V|^2。一个充满的矩阵是每个顶点都与其他任何顶点相连。达到这样的连接状态不会产生真正的问题。本章我们要考虑的问题是稀疏图。

## 7.5. 邻接表

邻接列表 adjacency list 可以成为稀疏图的更高效实现方案。在这个实现方法中，我们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。在实现顶点类的方法里，我们使用字典而不是列表。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。图 4 展现了图 2 中图的邻接链表。

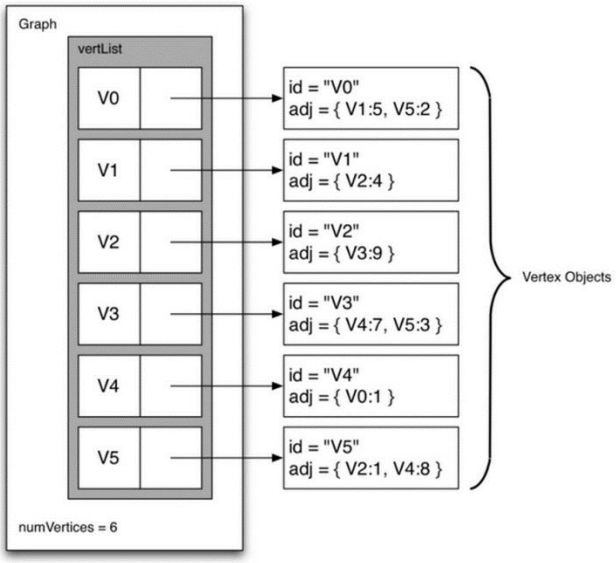


图 4: 图的邻接链表表示

邻接链表实现方法的优点是允许我们紧凑高效地表示一个稀疏图。邻接链表还使我们很容易找到某个顶点所有的连接。

## 7.6. 实现

Phython 中字典的使用使得邻接表的实现很容易。在我们实现图表抽象数据类型时，我们可以创建两个类，Graph 和 Vertex（详见表一和表二）。Graph 保存了包含所有顶点的主表，Vertex 则描绘了图表中顶点的信息。

每一个 Vertex 使用一个字典来记录顶点与顶点间的连接关系和每条连接边的权重，这（个）字典被称作 connectionTo（self. connectionTo）。下面的列表给出了 Vertex 类的代码。构造函数（__init___）简单地初始化了（一般为字符串的）id，和 connectionTo 字典。addNeighbor 方法被用来添加从一个顶点到另一个顶点的连接。getConnections 方法用以返回以 connectionTo 字典中的实例变量所表示的邻接表中的所有顶点。getWeight 方法可以通过一个参数返回顶点与顶点之间的边的权重。

表一

```
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]
```

下表显示出的 Graph 类，包含了一个将顶点名称映射到顶点对象的字典。在图 4 中这个字典对象被阴影灰色框所代表，Graph 也提供了向图中添加顶点和将一个顶点与另一个连接起来的方法。getVertices 方法可以返回图中所有顶点的名称。另外我们可以通过实现__iter__方法简化对特定图中所有顶点对象的迭代。同时，这两种方法允许你通过顶点名称或顶点对象本身去遍历图中的顶点。

表二

```
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())
```

通过使用刚刚定义的 Graph 和 Vertex 类，在图 2 中，我们在下面的 Phython 代码部分可以创建图表。首先，我们创造六个顶点并将其从 0 编码到 5，然后我们显示 Vertex 字典。注意到从 0 到 5 的每一个 key 我们都已经创建了一个 Vertex 实例。接下来，我们添加边来将顶点之间连接起来。最后，用一个嵌套循环核实图表的每条边都被正确地储存了起来。而你在最后的部分应该检查“边”列表的输出并与图 2 比对。

```
>>> 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 )
```

## 7.7. Word Ladder 词梯问题

我们考虑下面这个称作“词梯”的问题以开始我们对图算法的学习。（比如）将单词“FOOL”转变成单词“SAGE”。在词梯问题中，你必须以一次只改变一个字母的方式来逐步转变单词。每一步你都必须将一个单词转变成另一个单词，并且不允许转变成一个不存在的单词。词梯问题是 1878 年由《爱丽丝漫游奇境》的作者 Lewis Carroll 发明出来。下面的单词序列展示出了针对上述问题的一种可能的解决办法。

```
FOOL
POOL
POLL
POLE
PALE
SALE
SAGE
```

词梯问题还有很多的变形。比如你可能被要求以特定数量的步数完成单词转换，或者你需要使用一个特定的单词。在本节内容中，我们感兴趣的是如何算出从开始单词到目标单词所需要的最小转换次数。

既然这一章讨论的是图表，那么自然我们能用一个图算法来解决这个问题。这是我们的纲要：

- 以图的形式描绘出单词之间的关系。
- 利用广度优先搜索的图算法去找到一条从开始单词到目标单词的高效路径。

### 7.7.1. 建立 Word Ladder 图

我们第一个问题是去解决如何将大量单词的集合转变成图。我们想要的是一条始于一个单词而终于另一个与其只有一个字母不同的单词的边。如果我们可以创建这样的图表，那么任何从一个单词到另一个单词的路径都是某个词梯问题的一个解决方案。图一显示了一个由一些单词构成的小图表，可以解决从“FOOL”转变成“SAGE”的词梯问题。注意该图表是无向图表，并且边是没有加权的。

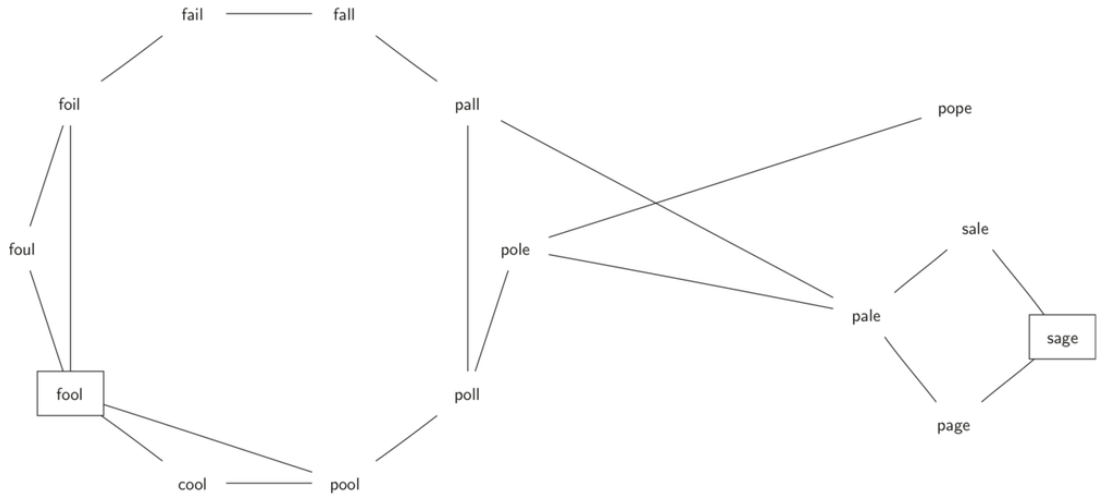


图 1：一个小词梯表

我们可以用几种不同的方法来创建我们用来解决词梯问题的图，。我们先假设我们有一个单词列表，单词都是一样的长度。作为我们的第一步，我们可以为列表里的每个单词在图表中创建一个顶点。而要将这些单词连接起来，我们可以将列表中的每个词与所有其他单词比较。当我们比较的时候，我们想要看到（两者）有多少字母是不同的。如果这两个词只有一个字母不同，我们就可以在图中创建一条它们之间的边。对于少量的单词，这种方法工作得很好；但是假定我们有一个 5110 个单词的列表。大致说来，将列表中的每个词都与所有其他单词比较是一个 O(n2)算法。5110 个单词的 n22600 万还大

我们可以通过使用下面的方法来优化。假设我们有一个巨大的数量的桶，每个桶外都贴有一个四个字母的单词标签，不同的是标签上的单词的一个字母被‘_’(通配符)所代替。例如，考虑图 2，我们就可能会将一个桶贴上“pop_”，当我们处理我们的列表中的每个词，都将其与每个桶比较，使用“_”作为一个通配符，那么“pope”和“pops”都与“pop_”匹配。每次我们找到一个匹配的桶，我们把单词放在桶里。一旦我们所有单词都在适当的桶里，我们便知道，同一个桶里的单词必须要连接起来。

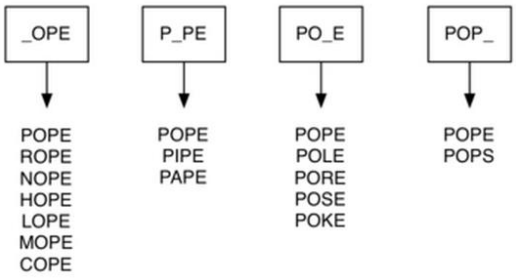


图 2：单词桶：只有一个字母不同的单词的集合

在 Python 中,我们可以通过使用一个字典来实现我们刚刚描述的方案。我们刚刚提到的桶上的标签在我们的字典中是 key。键值 value 存储是一个单词列表。一旦我们有了字典我们就可以创建图形。我们以以图中的每一个单词创建一个顶点开始。然后创建单词（顶点所对应）在上面字典的同一个 key 下的顶点（单词所对应的）之间的边。表 1 显示了构建图所需的 Python 代码。

表 1

```
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
```

既然这是我们第一个现实生活中的图的问题,你可能想知道这个图的稀疏程度如何?我们为这个问题准备的 4 个字母的单词列表足足有 5110 个单词那么长。如果我们要使用一个邻接矩阵（二维矩阵）,矩阵会共有 $5110 \times 5110 = 26112100$ 个格子。而由 buildGraph 函数构建出来的图只有 53286 条边,所以矩阵将只有 0.20%的格子被填充了!可见，这个图的确是非常稀疏的。

## 7.7.2. 实现广度优先搜索

在建立了词梯问题的图之后，我们就能把注意力转向寻找最短单词变化序列的算法，这一算法称之为广度优先搜索（BFS）。BFS 是搜索图的最简单的算法之一，同时，他也是我们之后所要学习的一系列其它重要图算法的原型。

已知一个图 G 和它的一个起始顶点 s，广度优先搜索第一步是搜索图 G 中所有从 s 出发可到达顶点的边，其显著的特点是给定距离 k，在搜索达到更远的距离 k+1 的顶点之前，BFS 会找到全部距离为 k 的顶点。如果要将这个 BFS 搜索的过程可视化，那就可以想象为建造一颗以 s 为根的树的过程，一次建造树的一层，同时，广度优先搜索在增加层次之前，会保证将始顶点所有的子顶点都添加在了树中。

为了进一步追踪这一过程，这里的 BFS 算法在搜索过程中，会给每一个顶点上色为白色，灰色或黑色，每一个顶点在被构建时都被默认为白色，在这之后，白色代表的是尚未发现的顶点，灰色代表的是已经发现的顶点，黑色代表已经完成探索的顶点。一旦一个顶点被标识为黑色，这就代表其附近已经没有未探索的顶点，而另一方面，如果一个顶点被标识为了灰色，这就意味着其附近可能还存在着未探索的顶点等待被探索。

表 2 中的广度优先搜索算法使用的是前文出现过的邻接列表来实现的图，此外，它还使用了一个队列来决定下一步应该探索哪一个顶点，我们之后就会知道这一操作有多么重要。

另外，词梯问题的 BFS 算法使用的是扩展版的 Vertex 类，这个新的类中包括三个新的实体变量：距离，前任顶点和颜色，并且这三种实体变量都有相应的取值以及设值的操作函数。扩展部分的代码在 pythonds 的代码包中都有，但我们在这里不作说明，因为它们的实现方法我们之前都已经学过了。

BFS 从起始顶点 s 开始，此时 s 的颜色被设置为灰色，代表它是已经发现的顶点，另外两个参数——距离和前任顶点，对于起始节点 s 初始设置为了 0 和 None。随后，起始节点会被放置在一个队列中，下一步便是系统地探索与队首顶点相连的顶点，这个过程通过队首顶点邻接列表的迭代过程来完成，每检查列表中的一个顶点，便会维护这个顶点的颜色参量，如果颜色是白色的，就说明这个节点尚未被探索，也就会按下述四步操作：

1、 新的未探索的顶点 nbr，标记为灰色
2、 nbr 的前任顶点被设置为当前节点 currentVert
3、 到 nbr 的距离被设置为到当前节点的距离加一
4、 nbr 被加入队尾，这一操作使得 nbr 在当前顶点的邻接列表中的所有顶点被搜索完后，能够进行下一层次的探索操作。

表 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')
```

作为一个例子，我们现在来看看 bfs 函数是如何构建与图 1 所示的图相对应的广度优先树的。首先我们从起始顶点 fool 开始，将与其相连的所有顶点都添加到树中，这些顶点包括 pool，foil，foul，和 cool，并且每当扩展到新的顶点时，原来的顶点就会被加入队列，图 3 所示是这一操作结束后，树和队列的状态。

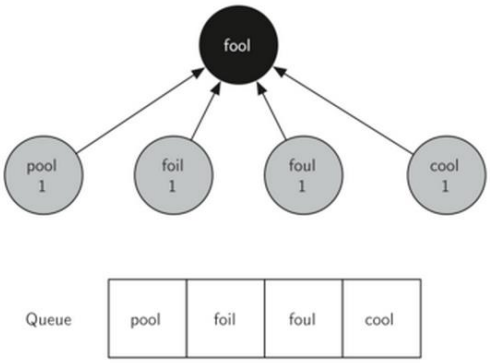


图 3：BFS 的第一步

bfs 函数的下一步是将下一层顶点中位于队首的 pool 从队列中移除，并且对其邻接顶点重复第一步中的操作。然而，当 bfs 函数检查顶点 cool 时，会发现它的颜色已经变为了灰色，这代表它已经被探索过了，并且表明从起始节点到 cool 之间有一条更短的路径已经存在于队列中留待下一步探索。探索顶点 pool 后，唯一新加入队列的顶点是 poll。这一步之后，树和队列的状态如图 4 所示。

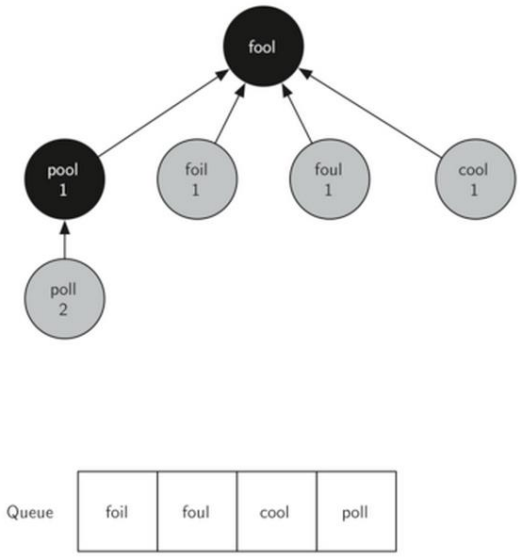


图 4：BFS 的第二步

队列中的下一个顶点是 foil，探索 foil 后唯一加入队列和树中的顶点是 fail；队列中这一层级的另外两个顶点 cool 和 poll 在被探索之后，都没有新的顶点加入树和队列。图 5 展示的是将第二层级的顶点都探索过后，树和队列的状态。

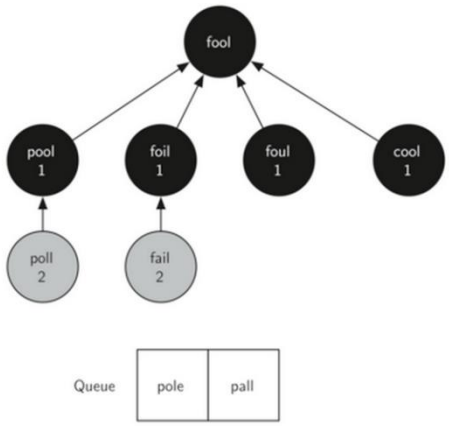


图 5：完成探索一层之后的树

在完成了上述步骤之后，读者最好是自己进一步将之后的步骤都用树和队列的形式描绘出来，以便于对这个过程有更好的理解，图 6 是图 3 中所有顶点都扩展搜索后所得到的最终的 BFS 树。但是 BFS 算法最神奇的地方在于我们不仅解决了我们最开始提出的那个 fool 如何变到 sage 的问题，更是在这过程中还顺便解决了许多别的问题。我们可以从 BFS 树的任意一个节点出发，沿着指向前任节点的箭头返回到树的根节点，从而反演出从任何单词变化到 fool 的最短词梯。Listing 3 是通过指向前任节点的路径打印词梯的函数。

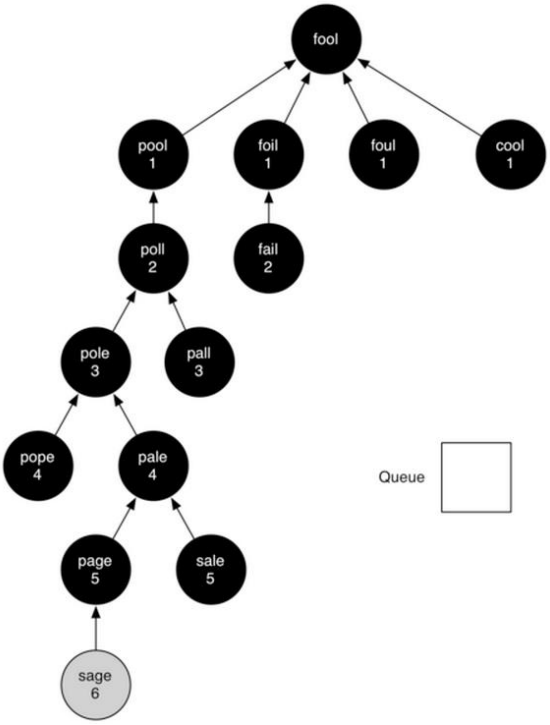


图 6：最终的 BFS 树

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

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

## 7.7.3. 广度优先搜索分析

在我们继续深入研究其它图算法之前，先来分析一下 BFS 算法的时间复杂度。首先我们观察到，当循环被执行时，图中的每个顶点都会被访问一次（也可以从每一个顶点在被探索和加入队列之前都是白色的得出相同的结论），也就是|V|，因而 while 循环拥有 O（V）的复杂度。另外，对于嵌套在 while 语句中的 for 语句，其对于图中的每一条边至多会被执行一次，因为每个顶点只会最多被出队一次，而对于从顶点 u 到顶点 v 的边，只有当顶点 u 出队的时候我们才会检查，所以每条边最多被检查一次，也就是|E|，所以这个循环的复杂度是 O（E）。两者相加，也就是 O（V+E）。

当然，执行 BFS 只是整个过程中的一部分，连接从起始顶点到目标顶点的路径是另一部分，最坏的情况的从起始顶点到目标顶点就是一条长链，在这种情况下遍历所有的顶点的复

杂度就是 O（V）。正常情况下的复杂度是小于|V|的，但我们仍然把 O（V）作为其时间复杂度。

最后，至少对于这个问题而言，我们需要时间来建立最开始的图，现在我们把 buildGraph 函数的分析作为课后练习交由读者完成。

## 7.8. 骑士周游问题

用来描述我们的第二个图算法的经典问题是“骑士周游”。骑士周游问题是在国际象棋棋盘上用‘骑士’这个棋子进行操作。问题的目的是找到一条可以让骑士通过连续的移动遍历棋盘所有格子的路径，每个格子只能走一次。一个这样的走棋序列称为一次“周游”。多年以来，骑士周游问题已经吸引了无数的数学家、国际象棋大师、计算机科学家。在 8×8 的国际象棋棋盘上，合格的“周游”数量上限有 $1.035 \times 10^{35}$ 这么多。然而，走棋过程中死角就更多了。显然这是一个要么耗死大量脑细胞要么占用无数计算资源的问题，或者两者都需要。尽管现在研究人员已经研究了很多不同的算法来解决骑士周游问题，图搜索依旧是最便于理解和编译的算法之一。.下面，我们采用两步走的方案来解决这个问题：

- 将棋盘上合法的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×列 -1）的路径，路径上包含每个顶点恰一次

## 7.8.1. 建立骑士周游图

为了将骑士周游问题表示为图，我们将用到下面两种想法: 将棋盘格可以用图的顶点表示.；骑士合法移动的规则可以用两个顶点间连接的边表示。 （Figure 1 通过一个骑士棋子与与之对应的图的边表示了骑士的合法移动）

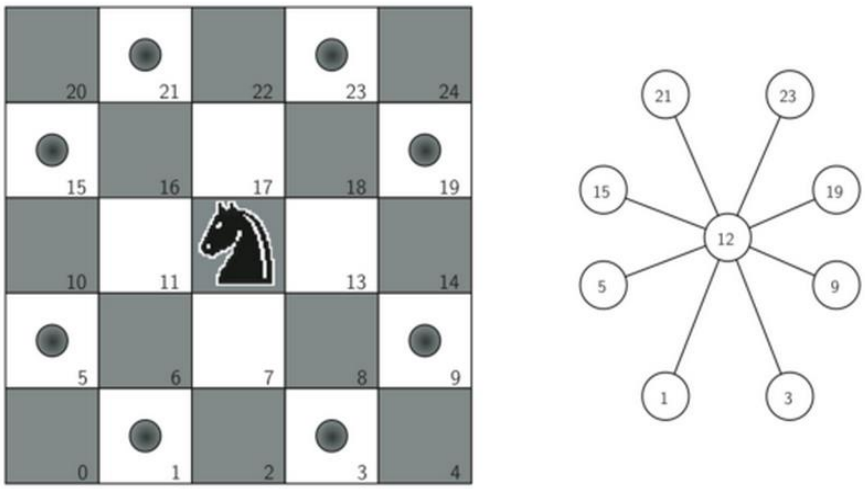


图 1:在 12 这个格子上的骑士的能走的格子，以及与之对应的图

我们需要利用表 1 中的 Python 函数去建立一个 n*n 棋盘对应的完整图。knightGraph 函

数遍历整张棋盘。在每个格子上 knightGraph 函数调用 genLegalMoves 辅助函数来创建一个储存这个格子上骑士所有合法移动的列表。所有的合法移动之后都被转换为图上的边。另一个辅助函数 posToNodeId 将棋盘上位置的行列信息转换成一个线性顶点数，类似于如图 1 所示的线性坐标。

表 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
```

表 2 中的 genLegalMoves 函数获取当期骑士的位置并生成所有的合法走棋步骤。legalCoord 函数确保任何个步骤不会超出棋盘（会导致 IndexError）

表 2

```
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
```

图 2 显示了 8*8 棋盘上所有可能的移动方式，图中只有 336 条边。注意到靠近边的格子比中心的格子连接的边更少。另外，如果这张图全连接则有 4096 条边，比起下面的 336 条，我们的图只占了 8.2%，我们再一次看到这是一张很稀疏的图。

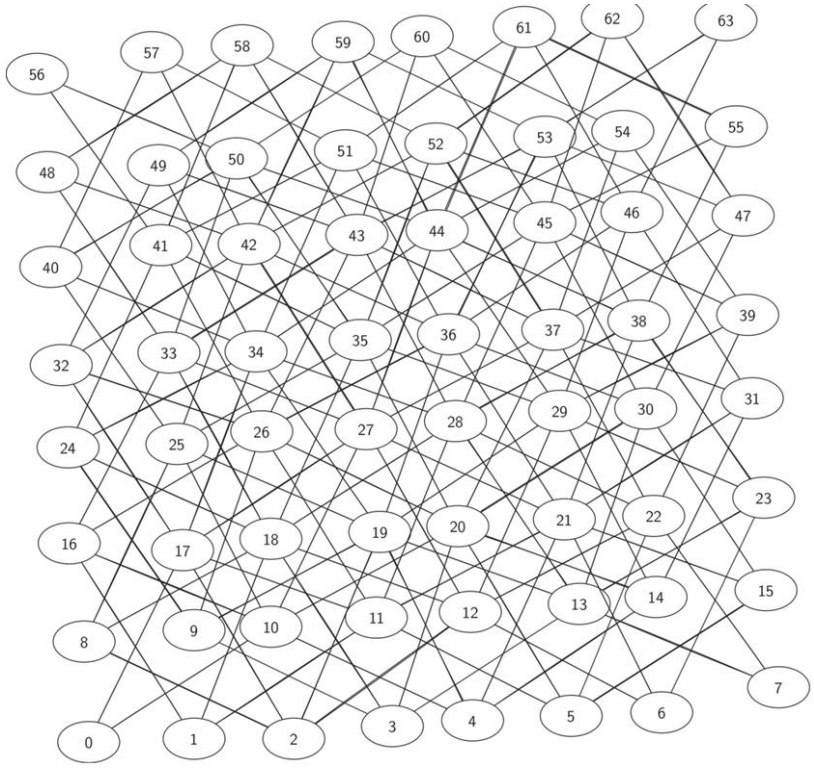


图 2: 8×8 棋盘上骑士的所有合法走棋步骤

## 7.8.2. 实现骑士周游

我们用于解决骑士周游问题的搜索算法是 Depth First Search (DFS)——深度搜索算法。前面讨论的 BFS（广度搜索算法）是一次建立搜索树的一层，而 DFS 算法则是尽可能深的搜索树的一支。这部分我们将介绍两个 DFS 的实现算法。第一个 DFS 算法被我们直接用于解决骑士周游问题，条件是每个顶点仅访问一次。另一个 DFS 算法更为通用，当树建立时允许顶点被重复访问，可作为其它图算法的基础。

图的深度优先探索很适合用于发现一条包含 63 条边的路径。我们会看到当 DFS 走到一条死路（没有任何可能的合法移动方式）时它会沿着树返回到有路可走的顶点。
函数 knightTour 需要四个传递参量：n，当前树的深度；path，访问到这个节点前所有已访问的点的列表；u，我们希望探索的点；limit，搜索总深度限制。该函数递归使用。当 knightTour 被调用时，首先检查基础状态：如果 path 包含有 64 个节点，返回 True 表示已经找到一条可周游的路径；如果 path 还不够长，我们选择一个新节点，（then 更改状态参量），调用自身，继续探索

DFS 算法需要使用“颜色”来标注图中哪些节点已经被访问过了。未访问的节点标注为

‘white’，访问过的标注为‘gray’。如果某个节点的全部邻居节点都被访问够而没有达到访问全部 64 个节点的目标，我们就到了一条死路。这时我们需要回溯。回溯机制在我们从 knightTour 返回 False 时启动（即递归的终止条件）。在 BFS 里面我们用 queue（队列）来标记要访问的节点，而在 DFS 里由于我们使用了递归，也即默认使用了 Stack（栈）来实现我们的回溯机制。当我们返回 False 结束 knightTour 函数的调用时（line11），我们依然在 while 循环里面并进行下一个节点的搜索。

表 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
```

我们来运行一个骑士周游的问题的简单实例。（每一步见下图）。在这个例子里面我们模拟 getConnections 路径（line6）。我们以调用 knightTour(0,path,A,6)开始。

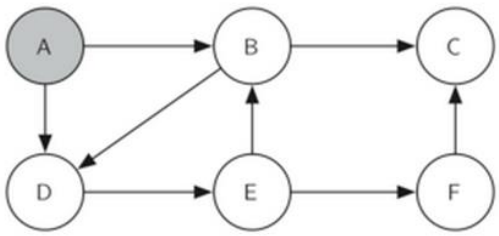


图 3: 以图 3 的 A 节点开始。连接到 A 的节点是 B,D，B 在字母顺序上在 D 前，算法选择 B 来展开下一级（图 4）。

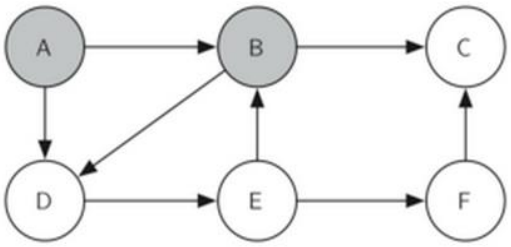


图 4: Explore B

B 相邻的是 C 和 D，所以 knightTour 先选择探索 C 节点。

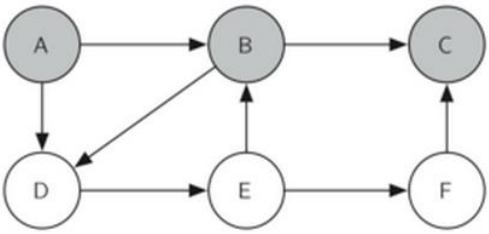


图 5: 节点 C 是一条死胡同，没有相邻的白色未被探索的节点。因此 在这一节点上，我们将 C 的颜色改回为白色（同时返回）。knightTour 的调用返回值自然为 False。

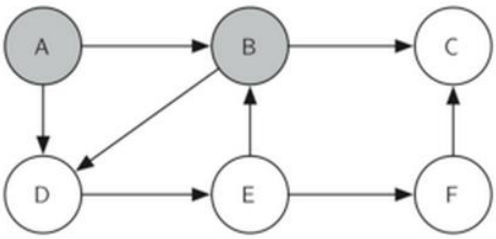


图 6:从递归调用回溯到顶点 B。 探索列表中的下一个节点是节点 D，所以 knightTour 函数开始节点 D 进行递归调用

从节点 D 开始，knightTour 可以连续进行递归调用，直到我们再次到节点 C（图 8，图 9，图 10）。

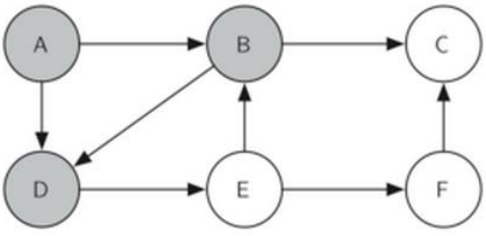


图 7: Explore D

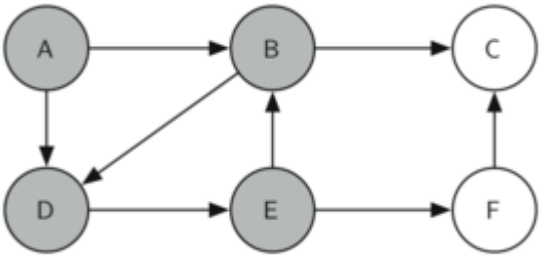


图 8: Explore E

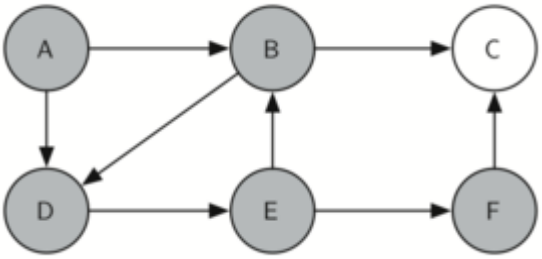


图 9: Explore F

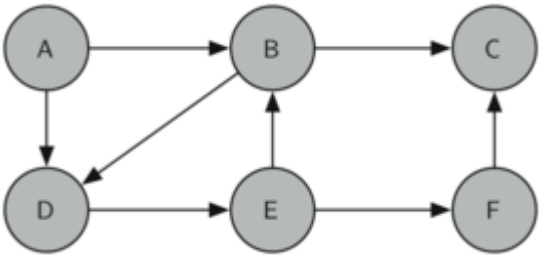


图 10: 完成。当我们到达节点 C 时 N >=limit，所以我们知道我们已用尽图形的所有节点。这样，我们可以返回 true，以表明我们已经完成了一次完整的骑士周游。 在我们返回的列表中，路径具是[A，B，D，E，F，C]，这是我们遍历图形并只对每个节点只访问一次的一个解。

图 11 展示了一个 8*8 棋盘上的一个完整周游图。其实还有很多种可能，有些是对称的。而通过一些对函数的修饰，你也能得到一个开始中止于同一格子的周游路线

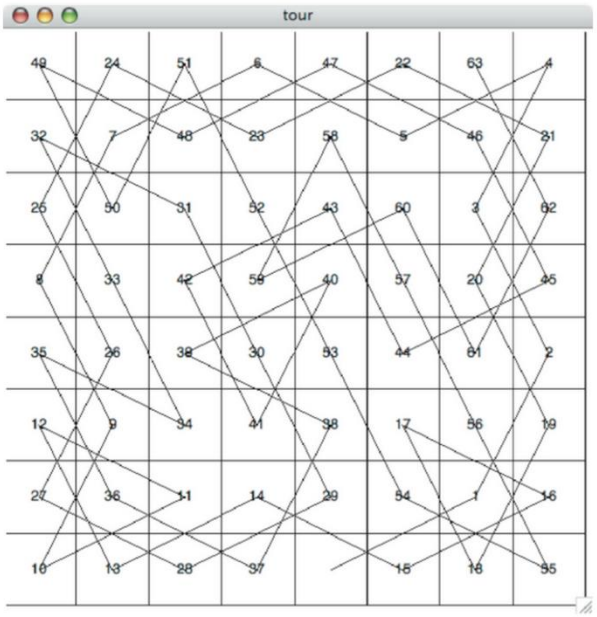


图 11: A Complete Tour of the Board

## 7.8.3. 骑士周游分析

在我们实现通用的深度优先搜索前，还有最后一个关于骑士周游问题的事情需要考虑，这就是算法的性能。特别地，骑士周游问题高度依赖于你选择顶点搜索先后次序的方法。例如，在 5×5 的棋盘上，一个相当快的电脑可以在 1.5 秒内找到一个周游路径。但如果你在 8×8 的棋盘上找呢？这样的话，根据你电脑运行速度的不同，你可能需要等半小时才能得到结果。原因就是我们当前实现的骑士周游算法是一个时间复杂度为指数 $O(k^N)$的算法，其中 N 是棋盘格的数目，k 是一个小的常数。图 12 可以帮助我们形象化地感受这一点。树的根节点代表搜索的起始点，从这一点开始，算法检查了骑士每一个可能的移动位置。就像我们之前注意到的一样，骑士可移动位置的多少取决于骑士在棋盘中的位置。在角上骑士只有两个合法的移动位置，在与角相邻的方格中有三个合法的移动位置，在棋盘的中间则有八个。图 13 展示了棋盘上每个方格中可能移动位置的个数。在树的第二层对于我们正在探索的位置又有 2 到 8 个可能的移动位置。可能的需要检查的位置数与搜索树的节点数一样多。

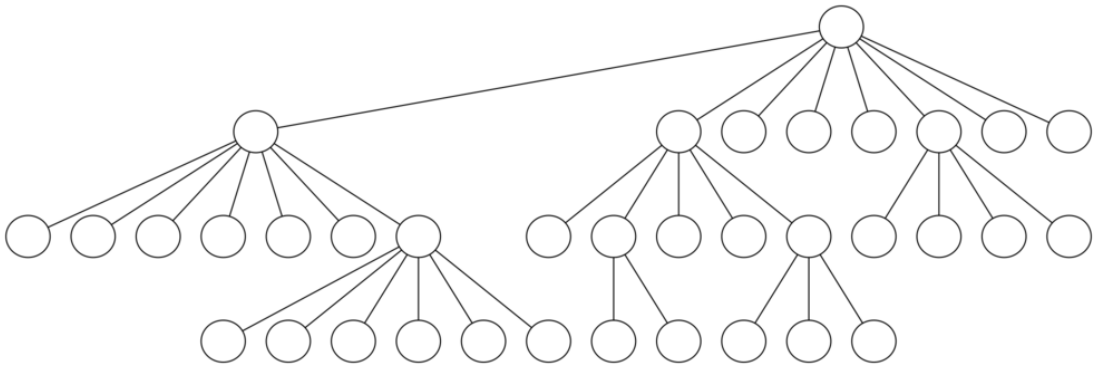

图 12，一个骑士周游问题的搜索树。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

图 13 棋盘上每个方格的可能移动数目。

我们早已知道 N 层高的二叉树的节点总数是 $2^{N+1}$-1，对于一个最多有八个子节点而不

是两个的树来说这个数目会大很多。因为每一个节点的分枝因子是可变的，我们可以通过节点的平均分枝因子来估计节点总数。需要注意的是这个算法是指数增长的：$k^{N+1}1$，k 是棋盘上的平均分枝因子。让我们看看它增长得多块！对于一个 5×5 的棋盘，这个树将会有 25 层深，或者将第一层深度设为 0 使 N=24。平均分枝因子 k=3.8，所以这颗搜索树的节点总数是 $3.8^{25}$-1，或者说 $3.12\times10^{14}$。对于 6×6 的棋盘，k=4.4，总节点数为 $1.5\times10^{23}$，而对于一个常规的 8×8 棋盘，k=5.25，总节点数为 $1.3\times10^{46}$。当然，由于这个问题有很多种结果，我们不用探索每一个节点，但是即使我们只探索需要探索的那一部分节点，对于这个问题仅仅是常数乘子的缩减，并没有改变这个问题指数增长的现状。我们将把这个留作练习，看看你是否将 k 表达成棋盘宽度的函数。

幸运的是，有一种方法可以将 8×8 的棋盘的骑士周游问题提速到不到一秒内跑完。在下面的一段代码中我们展示了可以提速骑士周游问题的算法。这个叫 orderByAvail 的函数（见代码 4）将会被用在我们之前展示的代码中的 u.getConnections 的调用中。在 orderByAvail 函数中至关重要的语句在第十行。这一行保证了我们下一步选择有最少可能移动位置的顶点去走。你可能认为这是起反作用的：为什么不选择那些有最多可能移动位置的节点呢？你可以在程序中排序后插入 resList.reverse()这一行并且跑一下你的程序来尝试一下这种方法。

选择有最多可能移动位置的节点作为下一个顶点的问题在于骑士将倾向于在周游的早期访问棋盘中间的方格。当这样的事情发生的时候骑士将容易被困在棋盘的一边而不能到达棋盘另一边未被访问的方格。另一方面，首先访问那些有最少移动步骤的方格首先围绕着棋盘的边缘访问。进而保证骑士早早访问那些不容易到达的角落，并且在需要的时候通过中间的方格跳跃着穿过棋盘。利用这种先验的知识来改进算法性能的做法，称作为“启发式规则”。人类每天应用启发式规则来做决定，启发式搜索经常被用在人工智能领域。这个启发式规则算法以 H. C. Warnsdorff 的名字来命名，被叫做 Warnsdorff 算法，他在 1823 年发表了自己的想法。

表 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]
```

## 7.8.4. 通用深度优先搜索

骑士周游问题是深度优先搜索中的一个特殊案例，它是以创建深度最深并无分支的优先搜索树为目标。事实上，更一般的深度优先搜索更容易实现。它的目标是尽可能深地搜索，连接图中尽可能多的顶点以及在必要时建立分支。

在深度优先搜索中建立不止一个树也是有可能的。当深度优先搜索算法建立一组树时，我们称之为深度优先森林。与广度优先搜索相同，深度优先搜索也使用前驱链接来创建树。此外，深度优先搜索会使用两个附加的 Vertex 类的实例变量。这两个新的实例变量是“发现时间”和“结束时间”。“发现时间”记录某个顶点第一次出现前算法的操作步数。完成时间则是某个顶点被标记为黑色之前算法的操作步数。观察算法后我们会发现顶点的“发现时间”和完成时间提供了一些我们在后续算法中可以使用的有趣性质。

我们的深度优先搜索的代码如列表 5 所示。由于 dfs 函数和它的辅助函数 dfsvisit 两个函数使用一个变量以在 dfsvisit 的调用过程中保持对时间的记录，我们选择将这段代码作为一个继承于 Graph 类的方法类。这个操作通过添加一个时间实例变量和 dfs 与 dfsvisit 两个方法拓展了图类。观察 11 行你会发现 dfs 算法在图调用 dfsvisit 算法访问白色顶点时会遍历所有的顶点。我们遍历所有顶点而不是简单地从某个选定起始顶点开始搜索是为了保证图中所有顶点都被考虑到并且没有顶点在深度优先森林中被遗漏。在 self 中考察一个顶点的申明或许看起来不同寻常，但是要记住在这例子中 self 是 DFSGraph 类的一个实例并且遍历一个图类的实例中所有的顶点是自然而然的。

列表 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)
```

虽然我们对 bfs 的实现只考虑拥有一条可以指回起始顶点的路径，但是创建一个广度优先森林来表示图中所有顶点对之间的最短路径是有可能的。我们将此留作练习。在我们接下来的两个算法中我们会看见为何保持对深度优先森林的追踪是重要的。

dfsvisit 方法以一个叫做 startVertex 的单一顶点开始并尽可能深地探索所有相邻白色顶点。如果你仔细观察 dfsvisit 的代码并将其与广度优先搜索进行比较，你应当注意到 dfsvisit 算法除了在内部循环的最后一行外与 bfs 几乎相同。dfsvisit 递归调用自身以继续对更深层次的探索，而 bfs 将顶点添加到一个队列中用以后续探索。需要注意的是，在 bfs 使用队列的地方，dfsvisit 使用的是栈。你虽然在代码中看不见栈的形式，但是它暗含在递归调用 dfsvisit 中。

以下插图说明了深度优先搜索算法对一个简单图的操作。这些图中，虚线表示边已经检查，但在边的另一端的顶点已经被添加到深度优先树中。在代码中这个测试是通过检查另一端的顶点的颜色是否是非白色的来实现的。

搜索从图中的 A 顶点开始（图 14）。由于所有的顶点在开始时都是白色的所以算法首先访问 A 顶点。访问顶点的第一步操作是将其颜色设置为灰色以表明这个顶点已被探索过并且将“发现时间”设置为 1。由于 A 顶点拥有两个相邻顶点（B、D）并且这两个顶点都需要被访问，所以我们随性地决定我们按照字母表顺序依次访问相邻顶点。

接下来访问 B 顶点（图 15），然后它的颜色被设置为灰色并且它的“发现时间”被设置为 2。由于 B 顶点同样有两个顶点（C、D），所以我们按照字母表顺序接着访问 C 顶点。

在访问 C 顶点（图 16）的过程中我们到达了树的一枝的末端。在将 C 顶点涂为灰色并将“发现时间”设置为 3 后，算法认为 C 顶点没有相邻顶点。这意味着我们完成了对 C 顶点搜索并且我们可以将其涂为黑色并设置“结束时间”为 4。你能在图 17 中看见这一阶段的以上叙述。

由于 C 顶点在一枝的末端，所以我们返回 B 顶点并继续探索 B 顶点的相邻顶点。由于 B 顶点的唯一相邻顶点是 D，所以我们现在访问 D 顶点（图 18）并从此继续我们的搜索。 D 顶点迅速将我们带向 E 顶点。E 顶点有 B 和 F 两个相邻顶点。通常我们我们会按照字母表顺序探索这些相邻顶点，但由于 B 顶点已经被标记为灰色，所以算法识别出它不应该访问会导致算法陷入死循环的 B 顶点。所以探索接着列表中的下一个顶点 F 进行。（图 20）

F 顶点只有一个相邻顶点 C，但由于 C 顶点已经被标记为黑色，不能继续探索，并且算法也到达了树的另一枝的末端。从此开始，你会看见算法一路运算返回初始顶点、设置”结束时间”并设置顶点颜色为黑色的操作（图 21~图 25）。

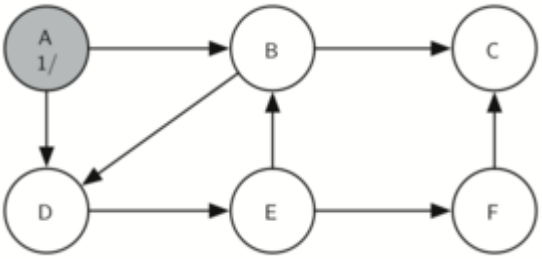


图 14：建立深度优先搜索树 10

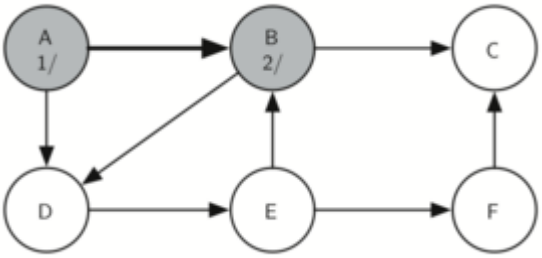


图 15：建立深度优先搜索树 11

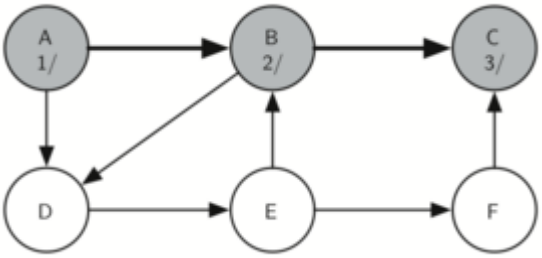


图 16：建立深度优先搜索树 12

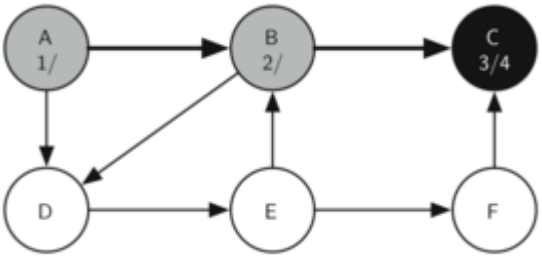


图 17：建立深度优先搜索树 13

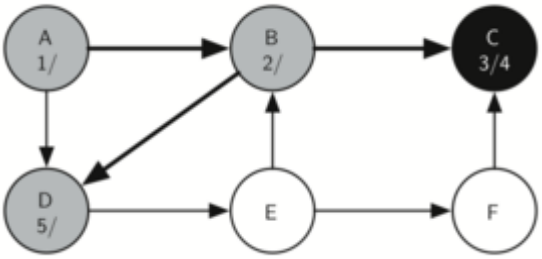


图 18：建立深度优先搜索树 14

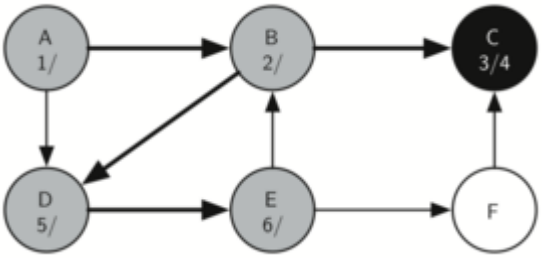


图 19：建立深度优先搜索树 15

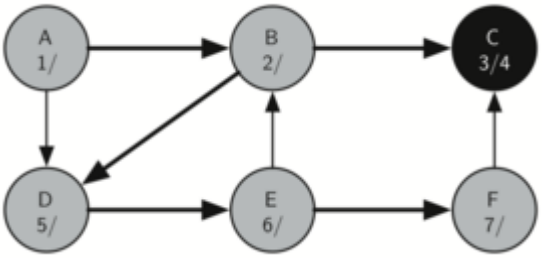


图 20：建立深度优先搜索树 16

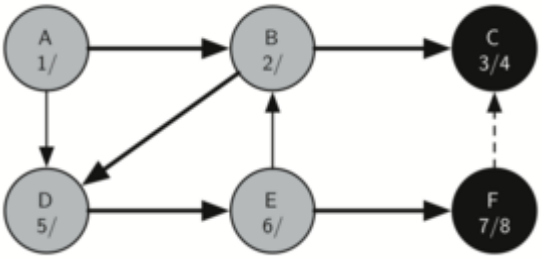


图 21：建立深度优先搜索树 17

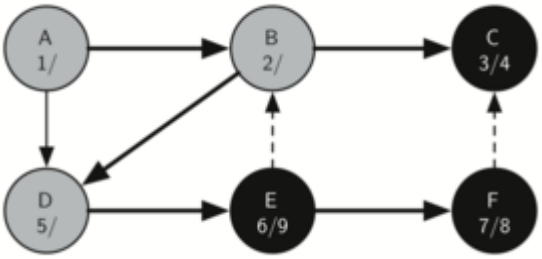


图 22：建立深度优先搜索树 18

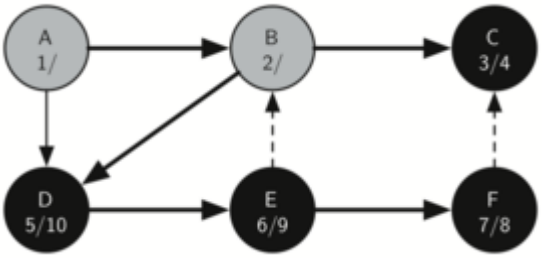


图 23：建立深度优先搜索树 19

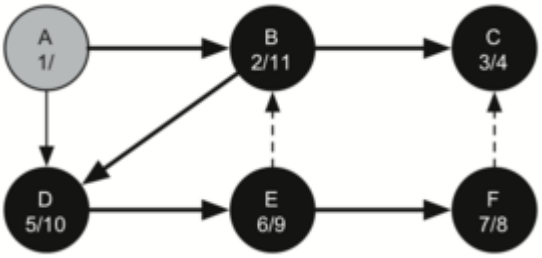


图 24：建立深度优先搜索树 20

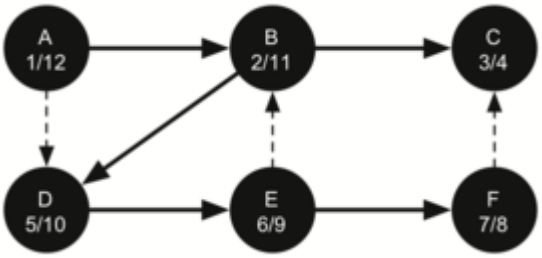


图 25：建立深度优先搜索树 21

开始时间和“结束时间”展示了每个顶点的名为括号性质的性质。这个性质意味着深度优先树中一个特定顶点的所有的子顶点拥有与它们的父顶点相比更晚的”发现时间”和比更早的“结束时间”。图 26 展示了深度优先搜索算法建立的树。

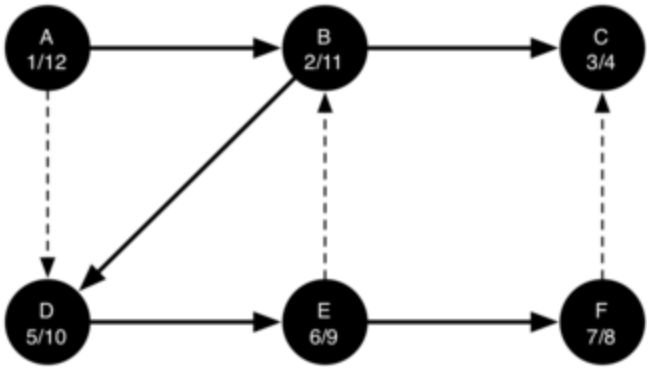

图 26：最终深度优先搜索树

## 7.8.5. 深度优先搜索分析

对通常的深度优先搜索的运行时间复杂度分析如下。由于 dfs 和 dfsvisit 在图中的每个顶点被执行了一次，不考虑 dfsvisit 中的影响的情况下 dfs 中的循环耗时为 O(V)。在 dfsvisit 中，循环语句在当前顶点的相邻顶点列表中的每一条边被执行了一次。由于 dfsvisit 只在顶点为白色的情况下递归调用，循环语句将对每条边只运行一次即时间复杂度最大为 O(E)。所以深度优先森林的总时间复杂度为 O(V+E)。

# 7.9. 拓扑排序

几乎所有问题都可以被转化为图问题，为了说明这一点，让我们来思考一个困难的问题——做松饼。食谱十分简单：鸡蛋 1 个，松饼粉 1 杯，油 1 汤匙，牛奶 3~4 杯。制作松饼需要先预热平底锅，将所有原料混合均匀后，将原料舀入加热好的锅中。待松饼浆开始冒泡时，将松饼翻面，煎至底部变为金黄色。在吃松饼之前你也可以加热一些甜酱汁作为佐料。图 27 以图表的形式说明了这一过程。

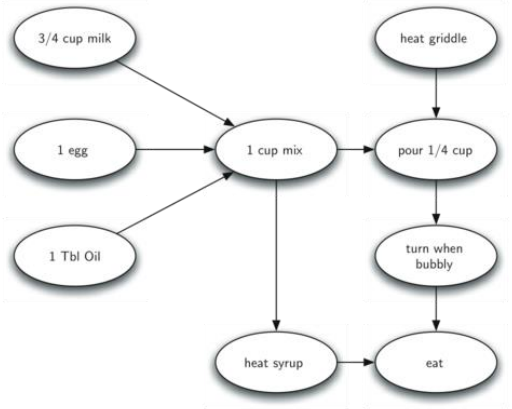

图 27：制作松饼的步骤

制作松饼的困难之处在于第一步做什么。如图 27 所示，你可以从加热平底锅开始，也

可以从向松饼粉中加入原料开始。为了在进行制作松饼的每一个步骤时都有一个清晰的顺序，我们可以~~求~~借助于一个叫拓扑排序的图算法。

拓扑排序可以将一个有向无圈图（DAG）转化只含有它顶点的线性序列。DAG 在应用中往往可以起到指示事件优先级的作用~~。~~，而制作松饼只是其中一个例子~~。~~。像这样的例子还有：制定软件工程日程~~，~~、制作最优化数据库访问优先级的表格~~，~~、矩阵相乘等等。

拓扑排序是深度优先搜索（DFS）的一个简单却有效的应用。拓扑排序的算法遵从以下法则：
调用 DFS 解决图问题，选择 DFS 的主要原因是~~要~~为了计算每一个顶点的完成时间。
按完成时间降序将事件顶点存储在一个~~序列~~列表中。将列表作为拓扑排序的结果返回。
图 28 展示了图 26 中所说的做松饼问题所构建成的 DFS 森林。

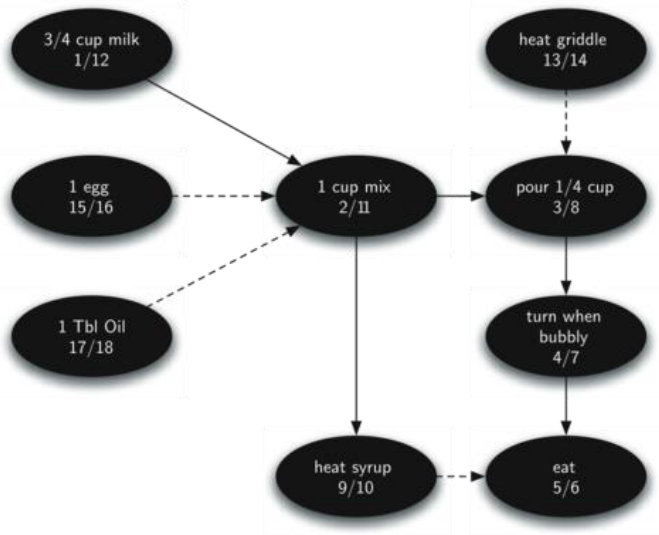


图 28：对做松饼的图进行深度优先搜索的结果

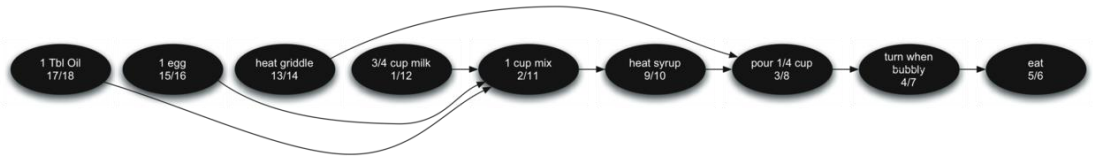


图 29：用 DAG 表示拓扑排序的结果

## 7.10. 强连通分支

本章剩余部分我们将关注一些巨大的图，用来学习一些其他算法。这些图产生于互联网上的主机之间的连接和网页之间的链接。我们将从这些网页开始。

像 Google 和 Bing 这样的搜索引擎，它们从网页构成的巨大有向图中搜索事实。为了将互联网转换为图，我们将页看作顶点，页上的超链接看作联接顶点的边。图 30 显示了图的一个非常小的部分，这部分开始于 Luther 大学的计算科学主页，通过网页之间的链接产生。

当然，这张图可以相当巨大，所以我们限制网站到 CS 主页的连接距离最大为 10。

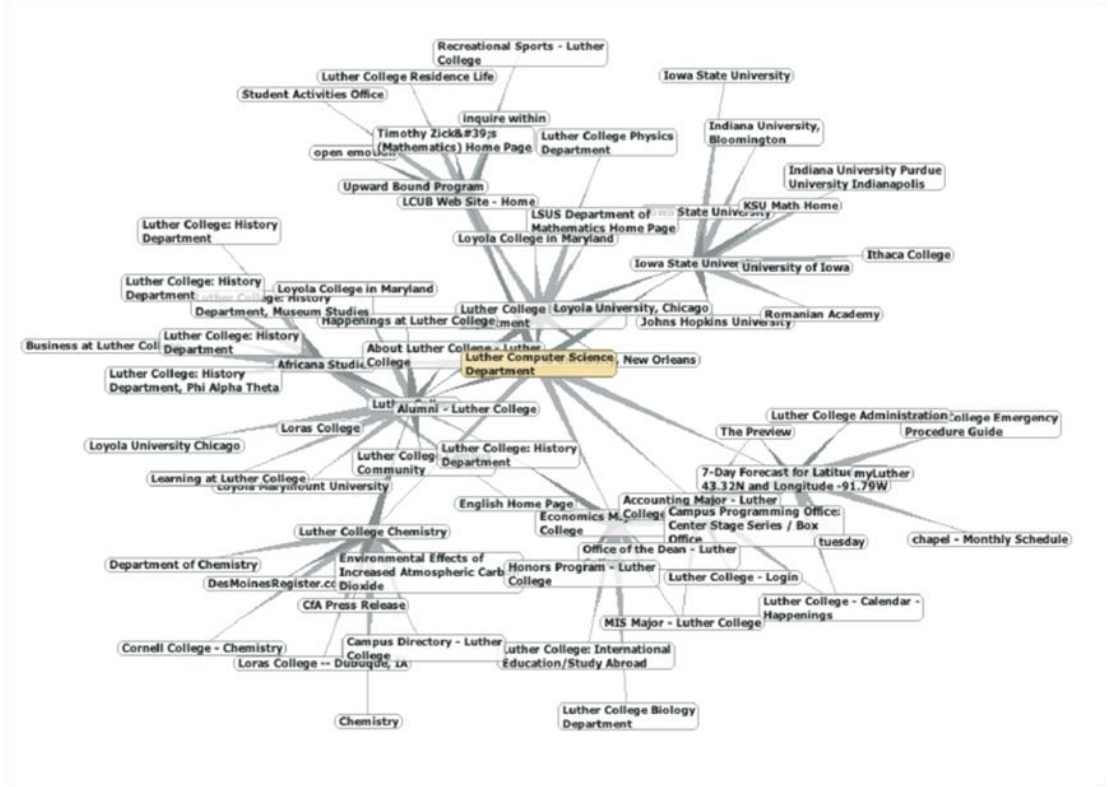


图 30：由源自 Luther 大学计算科学主页的链接形成的图

如果你学习了图 30 你也许会观察到一些有意思的现象。首先你也许会注意图中许多其他的网站是 Luther 大学的一些网站。其次，你也许会注意有一些链接指向了爱荷华的其他一些大学。第三，你可能会发现有一些链接到其他的人文艺术学院。你可能会得出结论，这些将同一水平的网站聚合在一起的网页有一些潜在的结构。

一种帮助发现图中高度连通的顶点形成的串的图算法被称为强连通分支算法（SCC）。我们正式定义一张图 G 中的一个强连通分支，C，是最大的顶点子集 $C \subset V$，其中每一对顶点 $v,w \in C$ 我们均有一个路径从 v 到 w 和从 w 到 v 的路径。图 31 显示了一个有着三个强连通分支的图。其中的强连通分支由不同的阴影区分开。

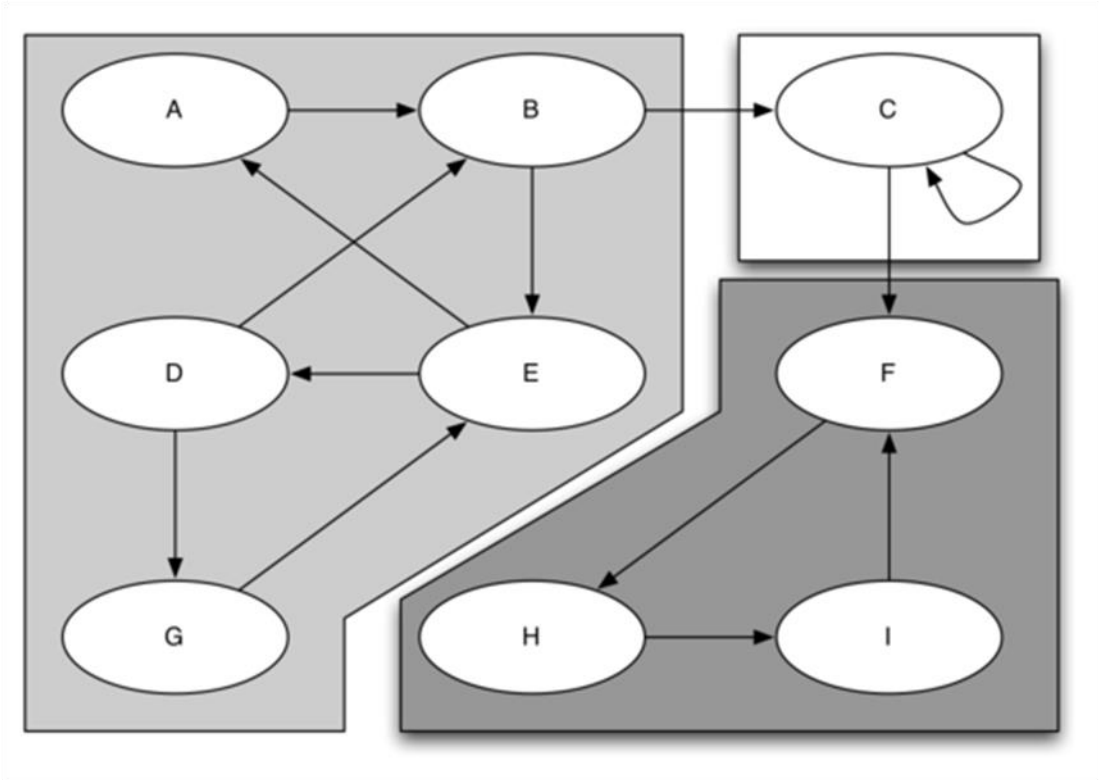


图 31：一张有三个强连通分支的有向图

当强连通分支被区分后我们可以将一个分支中的顶点合并为单个更大的顶点从而简化图。图 31 的简化形式见图 32。

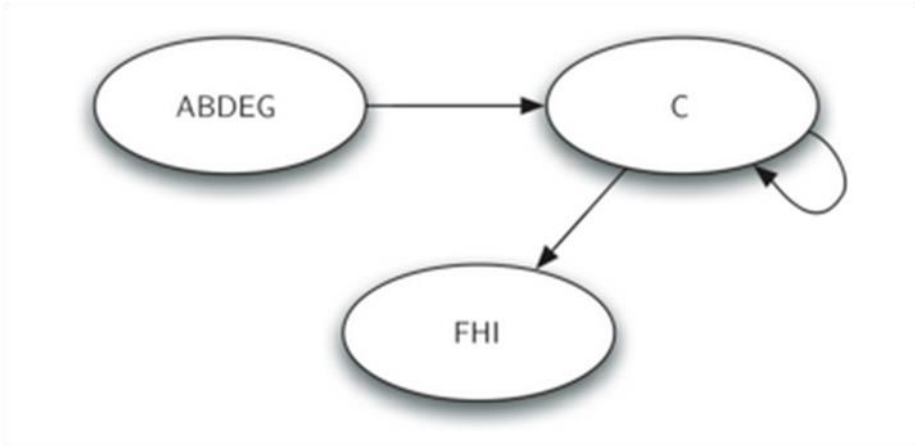


图 32：简化图

我们将再一次看到我们可以利用深度优先搜索开发强有力的算法。在我们处理 SCC 算法之前我们必须考察另一个定义。图 G 的换置被定义为图 GT 中所有的边已经逆转。也就是说，如果原图有一个从节点 A 至节点 B 的有向边，那么 GT 会包含一个从节点 B 至节点 A 的边。图 33 和 34 分别显示了一张简单图和它的换置图。

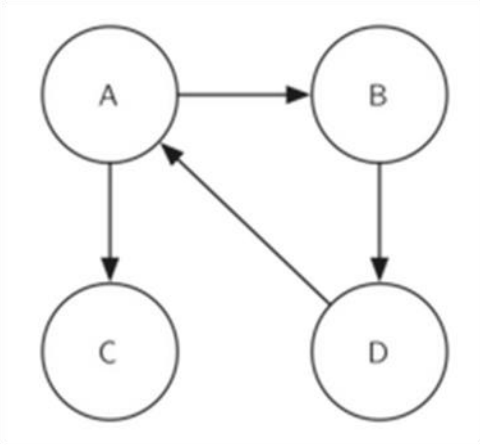


图 33：图 G

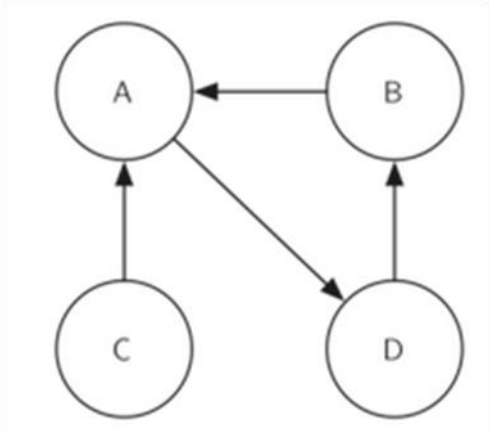


图 34：换置图 GT

再看一遍上图。注意图 33 有两个强连通分支。再看图 34。注意它也有两个强连通分支。

我们现在可以描述该算法，计算整个图中的强连通分支。

1. 用深度优先搜索算法（DFS）为图 G 计算每个顶点的完成时间
2. 计算出 GT
3. 用 DFS 计算图 GT 但在主循环的 DFS 探索每个顶点在减少命令时的完成时间。
4. 在步骤 3 中，森林中每一棵树都是一个强连通分支。输出每一棵树的每一个顶点的地址从而识别其分支。

让我们追溯图 31 中所描述的每步的操作。图 35 显示了对原图 DFS 算法的起始和结束时间。图 36 显示了在转置的图中运行 DFS 的起始和结束的时间。

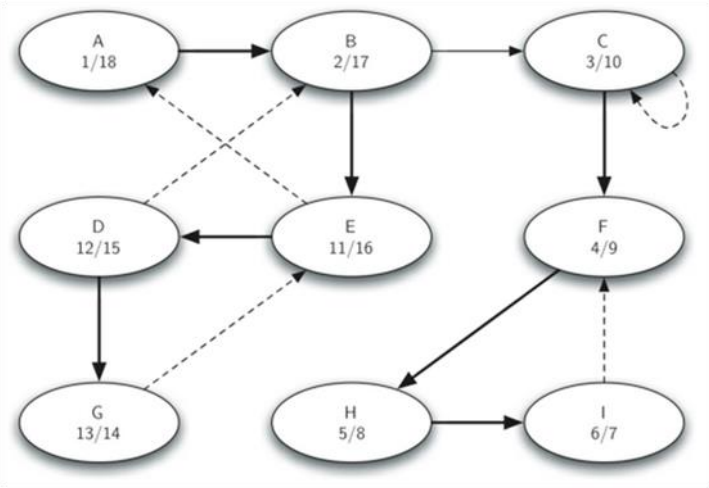


图 35：原图 G 的结束时间

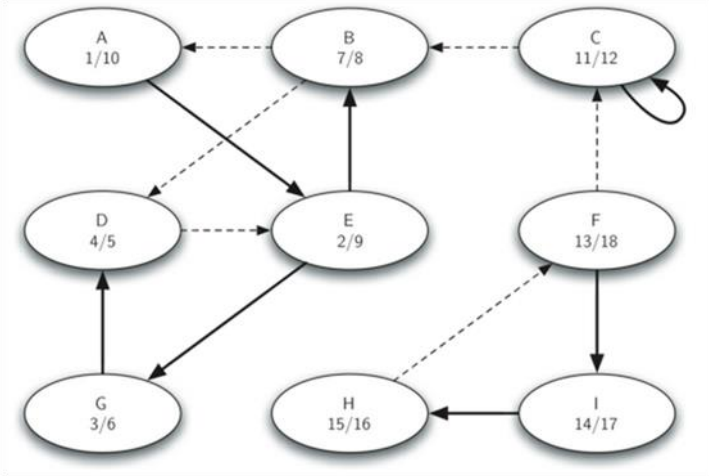


图 36：GT 图的结束时间

最终，图 37 显示了强连通分支算法的步骤 3 中形成的三棵树组成的树林。你会注意到我们并没有提供 SCC 算法的 python 代码，我们将其留为作业。

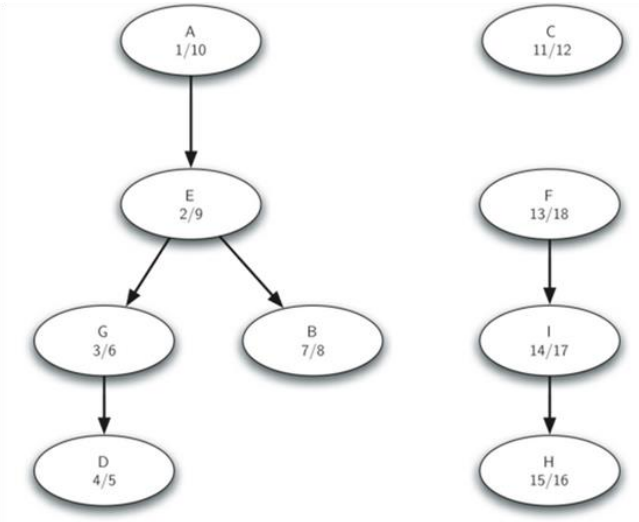


图 37：强连通分支

## 7.11. 最短路径问题

当你浏览网页，发送邮件，或者在校园里另一个地方登录某实验室主机时，电脑后台会运行很多指令，为了将本机的信息传输给另一台电脑。深入研究电脑与电脑之间通过网络的信息传递方式是计算机网络课程的首要课题，然而，我们讨论计算机网络的工作方式只是为了理解另一种十分重要的图运算法。

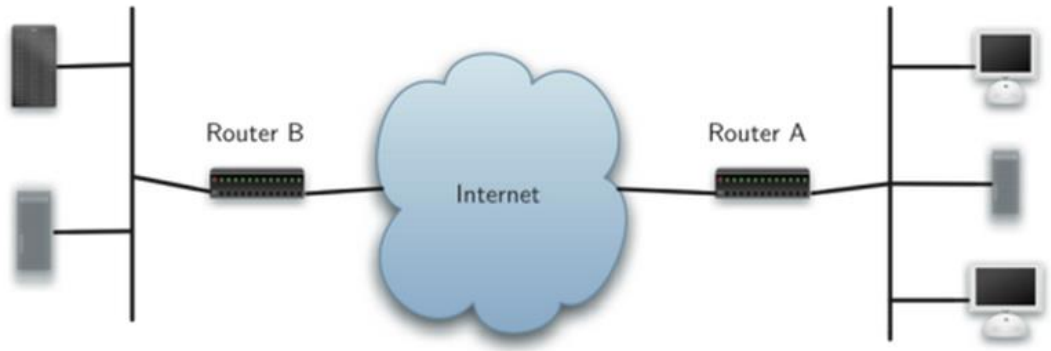


图 1：互联网连接概览

图 1 展示了高度抽象概括后的互联网交流方式。当你使用自己的浏览器向服务器发出一个网页浏览请求时，这个请求必须遍历当地局域网并且通过路由器发送到互联网上，在搜索过互联网后最终到达服务器所在局域网的路由器。然后你所要浏览的网页经由相同的路由器组传送回你的浏览器。图 1 中的云形标记里的“Internet”是路径中其他的路由器，它们共同的任务是把你的信息从一个地方传输到另一个地方。如果你的电脑支持路由跟踪指令，你就可以看到许多路由器在为你服务。以下文档是路由跟踪指令的输出结果，显示了在路德学院的网页服务器和明尼苏达大学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

互联网上每一个路由器都连接着一个或多个其他的路由器。所以如果你在一天中的不同时间段执行路由跟踪指令，很有可能发现你发送的信息在不同的时间经过了不同的路由器。这是因为两个路由器之间的连接是有代价的，这个代价受时间、线路拥挤情况以及许多其他因素的影响。于是，我们认为路由器之间的网络是一张含有有权边的图，这种想法也就十分自然了。

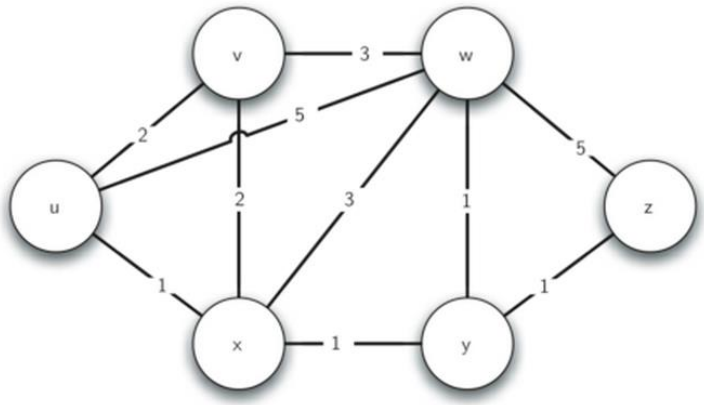


图 2：互联网路由器之间的连接和权值

图 2 展示了一个用有权值图来代表互联网路由器之间相互连接情况的小例子。我们想要解决的问题是，如何找到一条总权值最小的路线，并且可以实现任何信息传递功能。这个问题听起来很熟悉，因为它与我们用广度优先算法解决过的问题很相似，区别只是现在考虑的是整条路线的总权值，而不是路线中的节点数。需要指出的是，如果所有边的权值都相等，则这两个问题是等价的。

## 7.11.1. Dijkstra 算法

我们即将使用的选择加权最短路径的算法被称为“Dijkstra 算法”。Dijkstra 算法是一种迭代算法，以提供从一个确定的开始节点到所有图中其他节点的最短路径，这与广度优先搜索的结果也很类似。

为了追踪从开始节点到每一个结束节点的总权值，我们将在顶点 Vertex 类中使用 dist 作为实例变量。成员 dist 包含从起点到目的节点的最小权值路线的当前总权值。对于图中的每一个节点，算法重复一次，但各顶点重复的先后顺序是由一个优先队列控制的，队列中决定顺序的参量是 dist 值。当节点被创建时 dist 设置成一个很大的数，虽然理论上说应该将其设为正无穷，但实际操作中我们只需要将它设置成比实际问题中可能出现的任何距离都要大的值即可。

Dijkstra 算法的代码如表 1 所示。当算法结束时各个距离都被正确地设置成表中每节点之间的先驱连接。

表 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)
```

Dijkstra 算法使用了优先队列，回忆一下，在学习树的时候我们曾经基于“堆”结构实现了优先队列。简单的优先队列实现与 Dijkstra 算法中的优先队列实现有一些不同。首先，优先队列 PriorityQueue 类储存了键值对元组，这对于 Dijkstra 算法来说非常重要，因为优先队列中的 key 值必须与图中节点的 key 值匹配。其次，键值对中的值用来决定优先级，也决定了 key 在优先队列中的位置。在这种实现优先队列的方法中，我们把节点间的距离作为优先级，因为我们都知道，在探索下一个节点时，我们总是想要探索有最小距离的节点。第二点不同之处是添加了 decreaseKey 方法。正如你所看到的，当队列中已经存在了到某个节点的距离，而这个距离减小的时候，就将这个节点移到队首。

让我们沿着下面几张图片的顺序理解 Dijkstra 算法的一个应用实例。开始节点是 u，与其相邻的三个节点是 v，w 和 x。因为开始时到 v，w，x 的距离都被初始化成 sys.maxint，从起始节点到这三个节点的代价就是它们的直接代价，所以我们更新这三个节点的代价值。同时，设置每个节点的 predecessor 前驱节点为 u，并且把距离作为 key 值将每个节点都放进优先队列中。算法当前状态见图 3。

下一次 while 循环中我们考察与 x 相邻的节点。之所以先考察 x 是因为 x 的总代价最小，因此被弹到优先队列的顶端。对于 x 节点，我们考察与它相邻的节点 u，v，w，和 y。对于每一个相邻的节点，我们要检查经由 x 节点的当前节点的距离是否小于已知的距离。明显可以看出 y 满足条件，因为它的距离是 sys.maxint。节点 u，v 不满足条件，因为它们各自的固有距离是 0 和 2。然而，我们现在可以知道，经由 x 到 w 的距离小于直接从 u 到 w 的距离。因此需要更新 w 的距离值，并将 w 的前驱节点从 u 改为 x。算法当前状态见图 3。
下一步是检查与 v 相邻的顶点（见图 5）。这一步对于图本身无影响，所以我们转而检查节点 y。此时发现

对于 w 和 z 来说经由 y 代价更小，所以如上段所述调整其距离值和前驱节点（见图 6）。最后检查节点 w 和节点 z（见图 6 和图 8），然而没有发现需要改变的地方，因此优先队列变为空的，Dijkstra 算法结束。

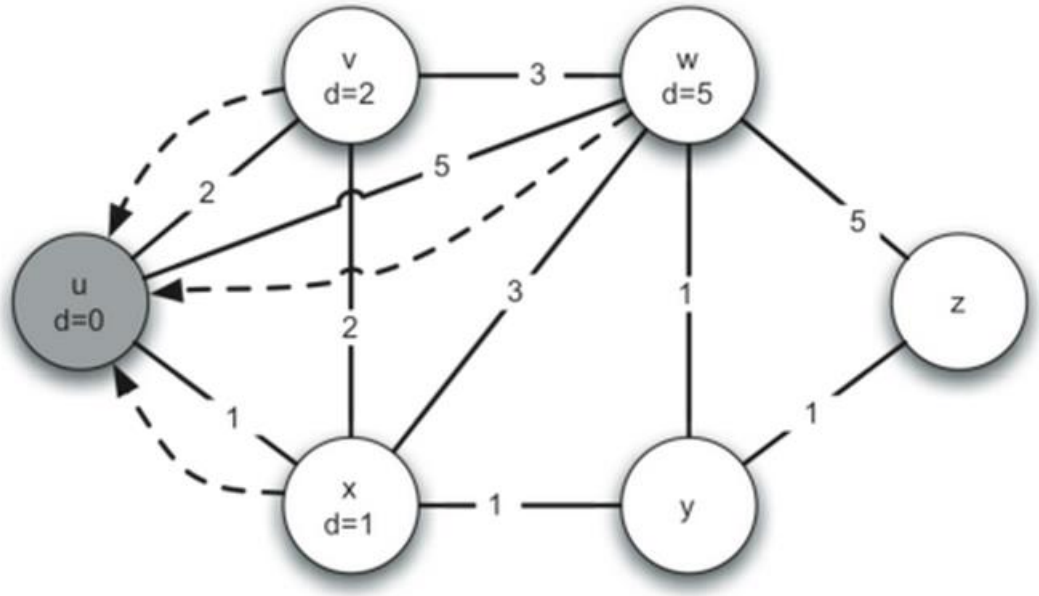


图 3：追踪 Dijkstra 算法

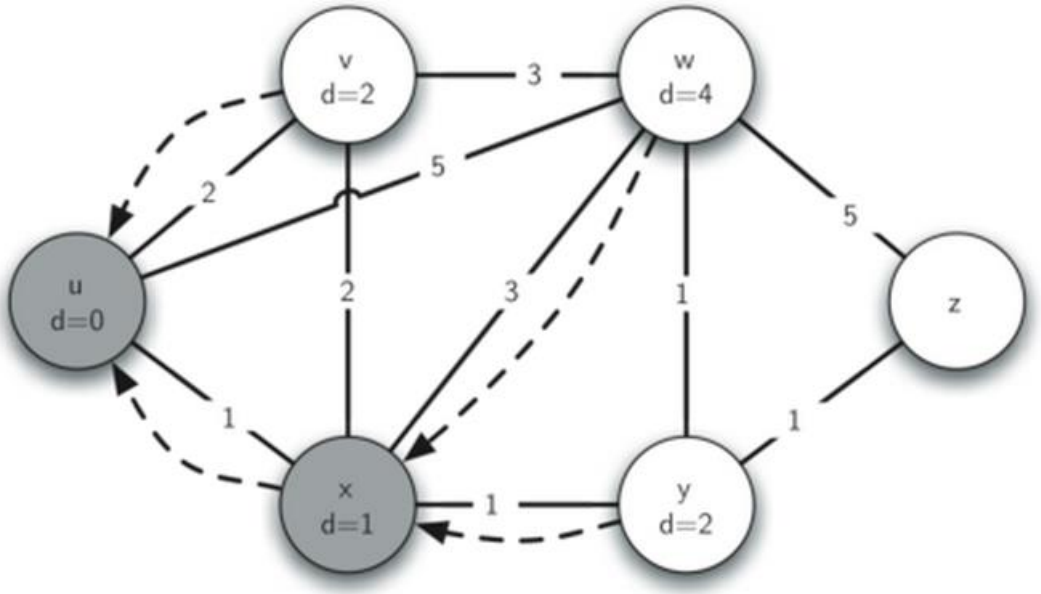


图 4：追踪 Dijkstra 算法

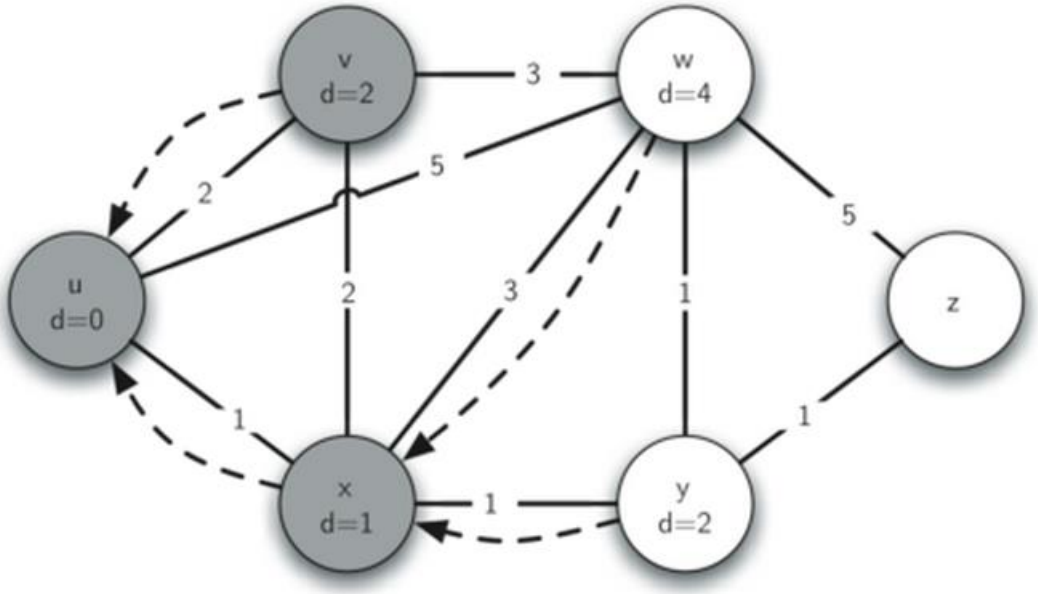


图 5：追踪 Dijkstra 算法

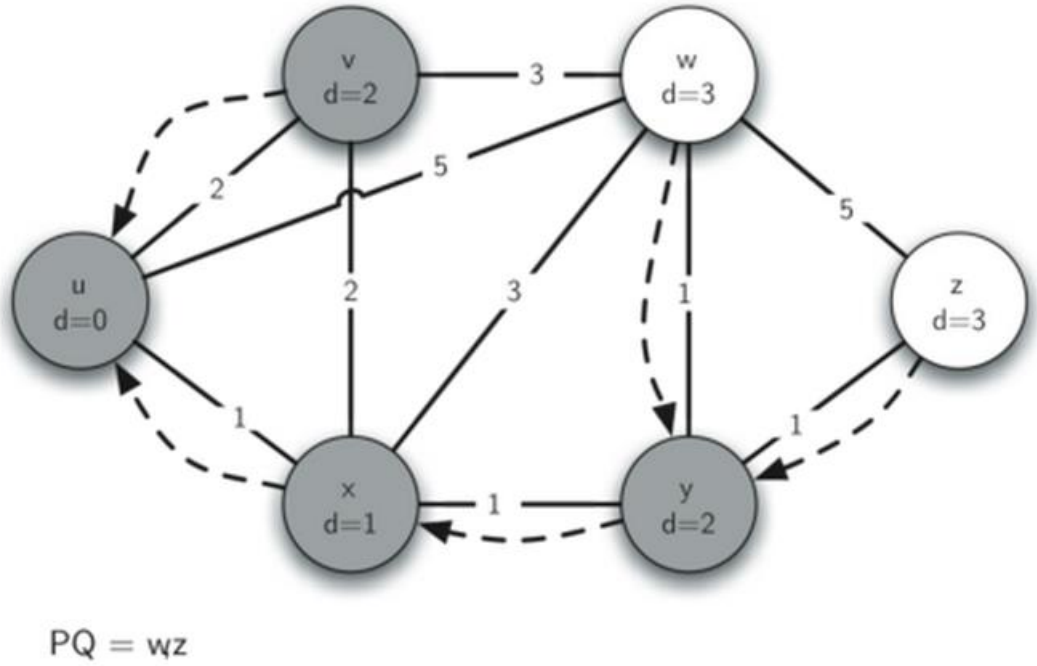


图 6：追踪 Dijkstra 算法

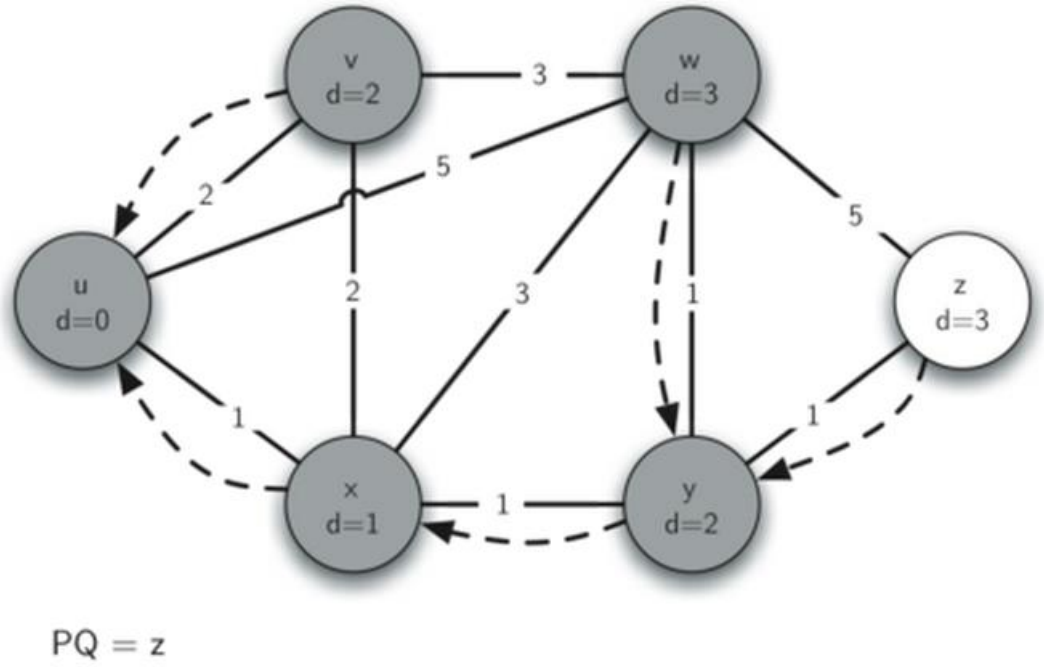


图 7：追踪 Dijkstra 算法

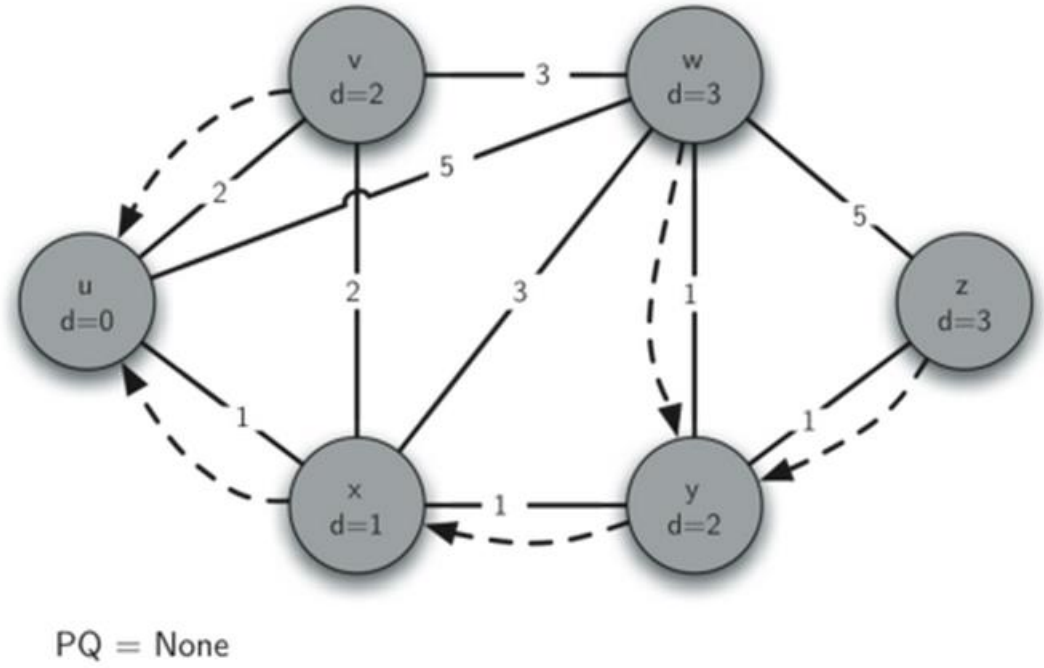


图 8：追踪 Dijkstra 算法

十分重要的一点是，Dijkstra 算法只适用于所有权值都为正数的情况。如果在某条边引入了负的权值，那么整个算法将陷入无限循环。

我们需要注意的是，为了在互联网上传递信息，其他的算法也被用来寻找最短路径。在互联网问题上使用 Dijkstra 算法的一个问题是你必须有完整的图结构，否则算法无法运行。这暗示着每个路由器需要拥有互联网上所有路由器的完整地图。实际操作中这是不可能做到

的，因此其他的算法允许路由器在传递信息的过程中发现新的图结构。这种算法被称为“距离向量”路由算法。

## 7.11.2. Dijkstra 算法分析

最后，进行 Dijkstra 算法的时间复杂度分析。首先，由于我们最初把图中的每一个节点都加入了优先队列，所以建立优先队列的时间复杂度为 O(V)。建立完队列之后，while 循环对于每个节点都会执行一次。因为所有顶点在一开始就都被加入，并且仅当循环执行完后才会被移出。循环体内部 delMin 操作的时间复杂度为 O（Vlog（V））。For 循环对于图中的每条边都会执行一次，在循环体内部 decreaseKey 操作的时间复杂度为 O（Elog（V））。所以结合起来，总的时间复杂度为 O（(V+E) log（V））。

## 7.12. Prim 最小生成树算法

为了我们的最后一个图算法让我们来考虑一个在线游戏设计者和无线电网络开发者面对的问题。这个问题是他们想要高效地传递信息给那些可能正在收听的人。这在游戏中非常重要以便于每一名玩家都可以了解到其他玩家的实时位置。这在无线电网联络中也很重要以便于每一名接入频道的的听众可以得到他们需要的全部数据去重现他们正在收听的歌曲。图 9 向我们阐明了这个广播问题。

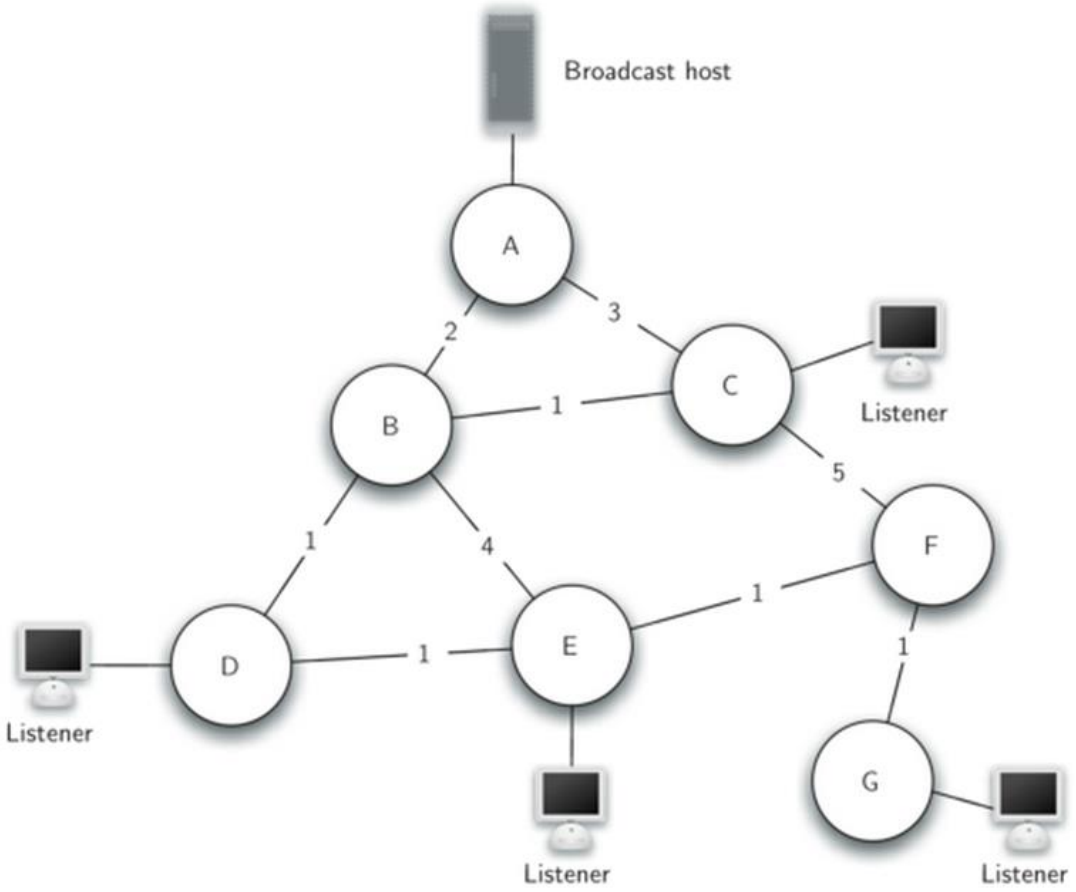


图 9：广播问题

关于这一问题有一些无脑的强制解决办法，所以让我们先来看一看这些解决办法来帮助我们更好的理解广播问题。这也会在我们解决这一问题时让你更加欣赏我们给出的解决方法。在开始的时候，广播主持有一些所有听众都需要接收的信息。最简单的解决方式是广播主持有一个所有听众的名单然后向每一名听众逐一发送信息。在图 9 中我们展现了一个小的网络包括一个广播和一些听众。用第一种方式，每一段信息需要发送四次。假设使用的是最短路径，让我们来看看每一个路由器需要处理多少次相同的信息。

每一段从广播发出的信息都经过了路由器 A，因此 A 看到了每一段信息的四次拷贝。路由器 C 只看到了面向它的听众的信息的一次拷贝。然而路由器 B 和 D 会看到每一条信息的三次拷贝因为路由器 B 和 D 在通往听众 1,2,3 的最短路径上。当你考虑到播音主持每秒要发送上百条信息给一个收音机，这就会成为一个巨大的额外负担。

一个无脑的强制解决办法是播音主持每份信息只发送一份然后让路由器来把它们分类。在这种情形下，最简单的解决方式是一种叫做无控制流动的方法。这种方式根据安排工作。每段信息的开始时间(ttl)值设置为大于或者等于播音主持到距他最远的收听者之间距离的一些数。每一个路由器复制一份信息然后把信息传递给所有与它相邻的路由器。当信息传递时 ttl 降低。每一个路由器持续地向与它相邻的所有路由器传递信息拷贝知道 ttl 值变为 0.显然无控制流动的方法会比我们的第一种方法产生更多的不必要的信息。

这种解决方式属于最小重量生成树的一种。形式上我们给图 G=(V,E)定义了如下的最小生成树 T。T 是一个连接 V 中所有顶点的 E 的非循环子集。T 中边界重量的和是最小化的。

图 10 展示了一个简化版的广播图并且着重突出了在图中形成最小生成树的边界。现在为了解决我们的广播问题，广播主持向网络中简要发送了一份播音信息的拷贝。每一个路由器向作为生成树一部分的任意相邻路由器发送信息，包括刚刚向它发送信息的相邻路由器。在例子中 A 向 B 发送信息。B 向 D 和 C 发送信息。D 向 E 发送信息，E 向 F 发送信息，F 向 G 发送信息。没有路由器会重复接收同一份信息，并且所有对其感兴趣的听众都能够看到这一信息。

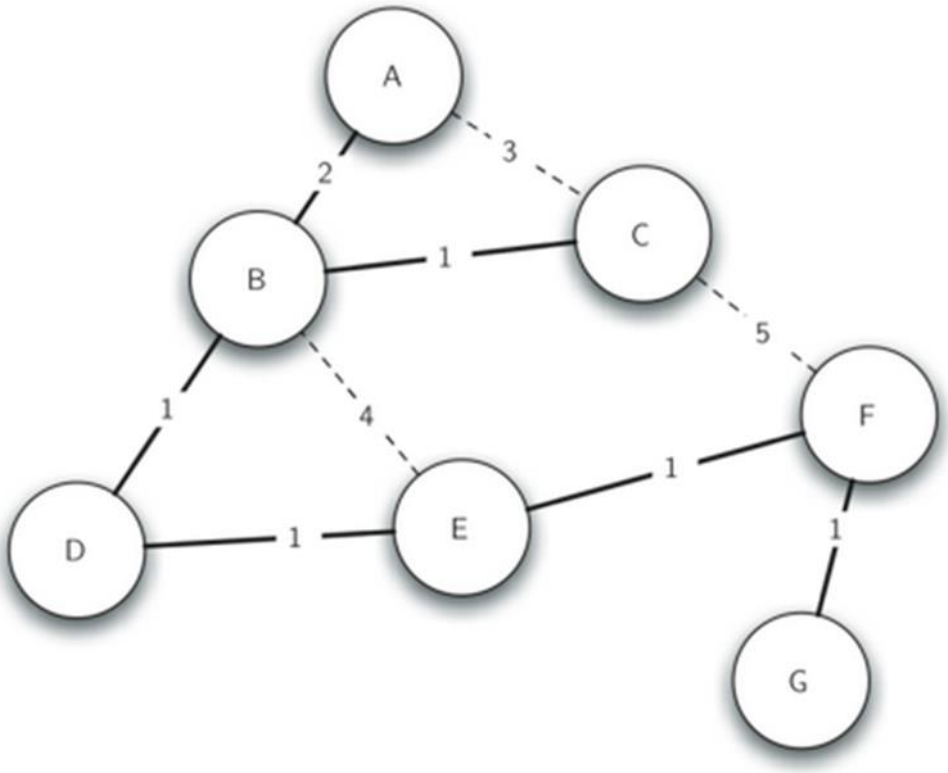


图 10：广播图的最小生成树

我们将要用来解决这一问题的算法叫做 Prim 算法。Prim 算法隶属于“贪心算法”一类因为每一步我们都会选择最简单的下一步。在这种情形下最简单的下一步将会沿着最低重量的边缘。

我们的最后一步是开发 Prim 算法。

最基本的想法是构建一个如下的生成树：

```
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
```

步骤中的窍门是指引我们去“找到一条安全的边界。”我们像定义任意边界那样定义安全边界，即连接在生成树中的一个顶点和不在生成树中的一个顶点。这确保了这个树总会保持一个树的形状不会产生循环。

执行 Prim 算法的 Python 代码在 Listing2 中展示。Prim 算法与 Diikstra 算法相似因为他们都用优先队列去选择下一个顶点添加到图中。

表 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) \
                  + currentVert.getDistance()
          if nextVert in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

接下来的这些图（图 11 到图 17）展示了算法在样品树上面的运行。我们将顶点 A 作为起始点。到其他任意点的距离被初始化为无穷。看着 A 的相邻点我们可以更新顶点 B 和 C 的两个距离因为 A 到 B 和 C 的距离小于无穷。这将 B 和 C 已到了优先队列的前端。更新之前 B 和 C 的连接通过将它们指向 A。需要着重强调的是我们还没有将 B 和 C 添加到生成树中。一个点在没有被移出优先队列之前是不会被添加到生成树中的。

因为 B 有着我们看向 B 的下一个顶点的最小距离。检查 B 的相邻顶点我们发现 D 和 E 可以被更新。D 和 E 都能够得到新的距离值并且他们的之前连接可以被更新。移至优先队列的下一个点我们找到了 C。优先队列中唯一与 C 相邻的点是 F，然后我们可以更新到 F 的距离并且调整 F 在优先队列中的位置。

现在我们寻找点 D 的临近点。我们发现我们可以更新 E 并且将到 E 的距离从 6 减到 4. 当我们这么做时我们改变了从 E 指回 D 的连接，然后准备把它植入生成树的不同位置。剩下的就是像你期待的那样算法开始，将每一个新的点加入树中。

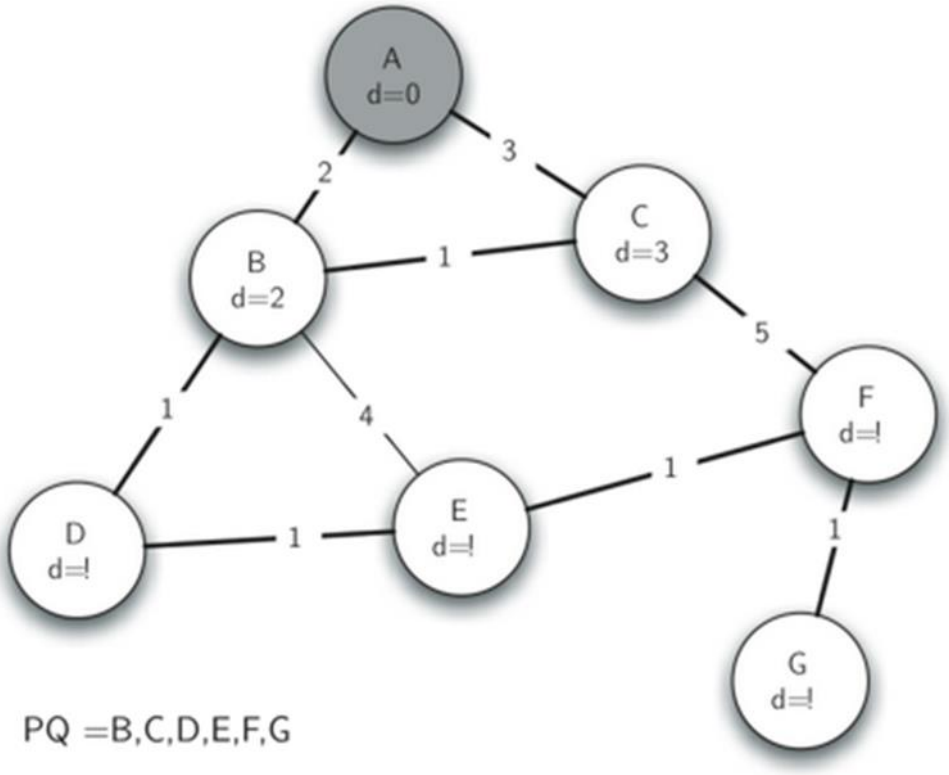


图 11：描绘 Prim 算法

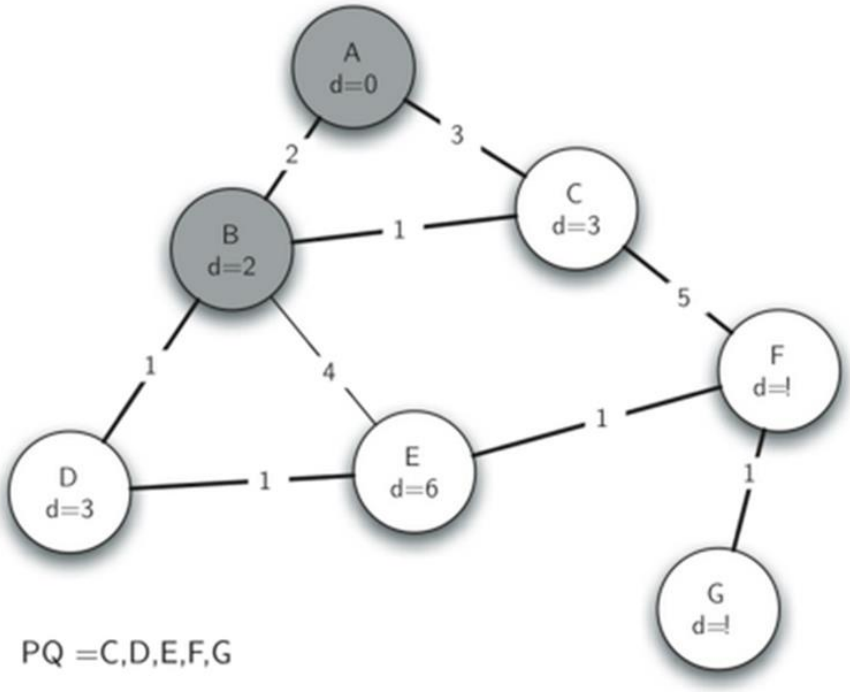


图 12：描绘 Prim 算法

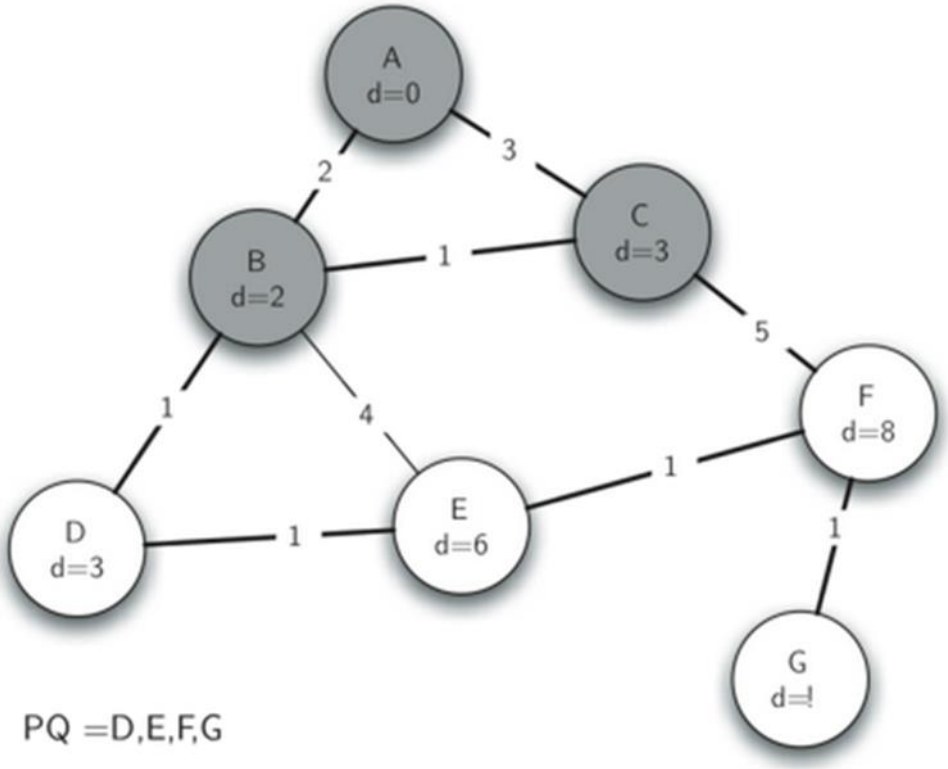


图 13：描绘 Prim 算法

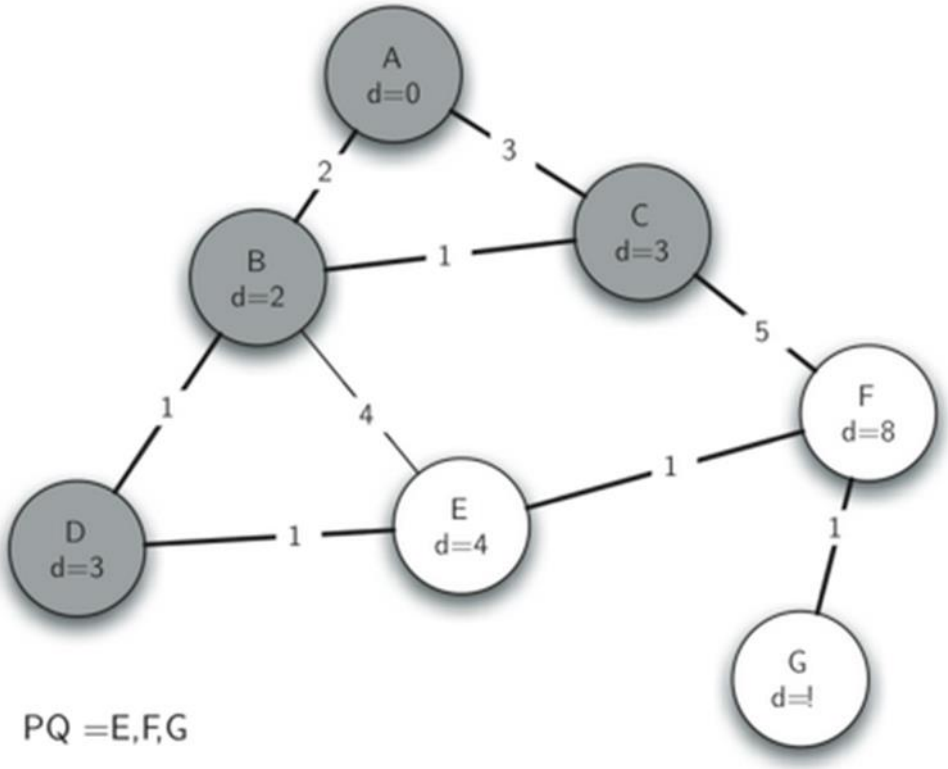


图 14：描绘 Prim 算法

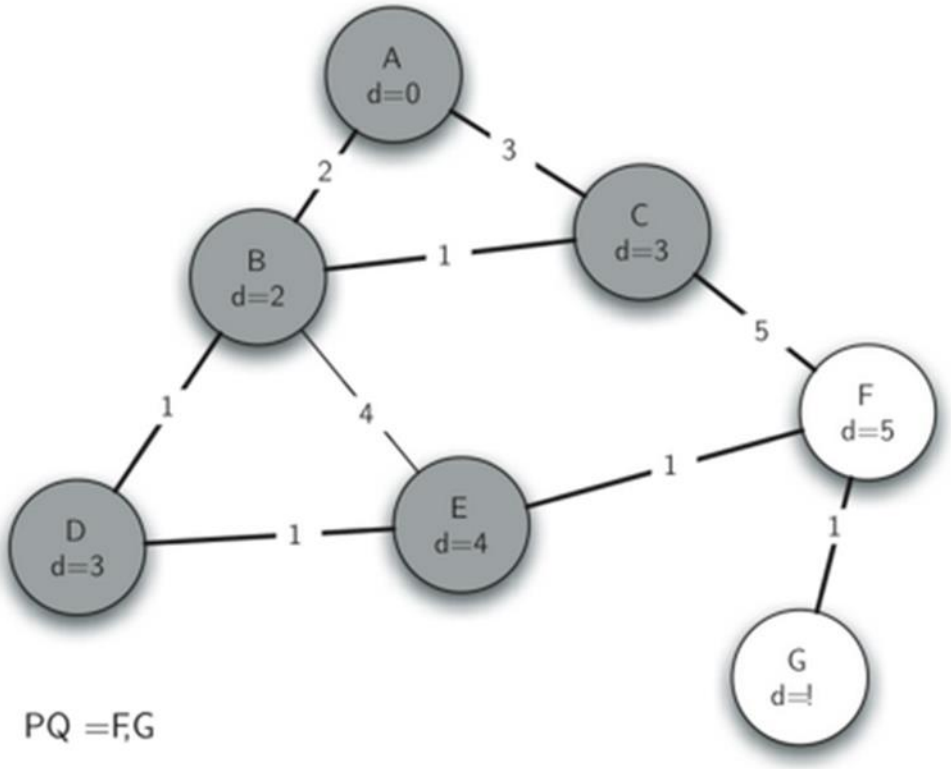


图 15：描绘 Prim 算法

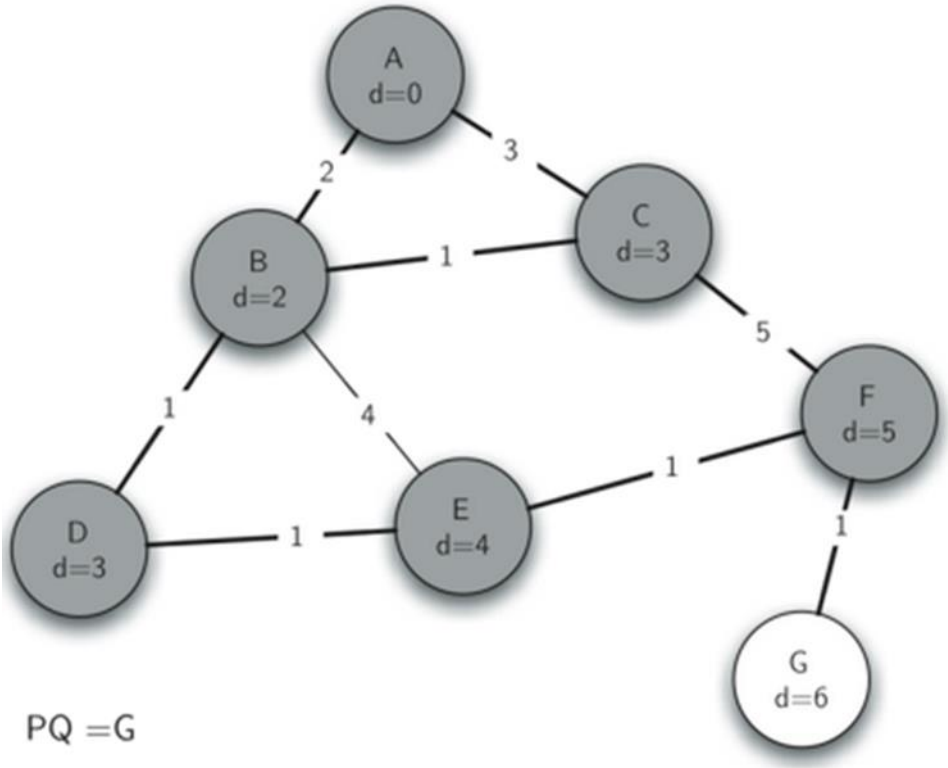


图 16：描绘 Prim 算法

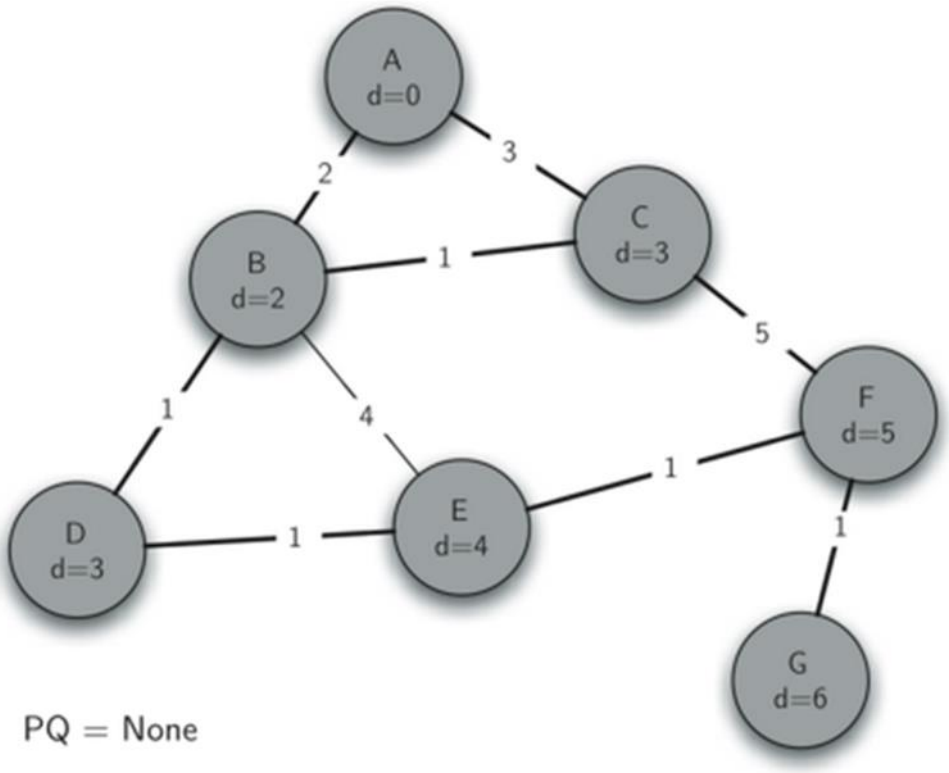


图 17:描绘 Prim 算法

## 7.13. 小结

本章我们学习了图抽象数据类型，以及若干实现方法。假如我们可以把一个原始问题转化成可以被表示成图的事物，我们就可以用图来解决许多问题。特别地，我们看到图在以下领域可以解决很多问题：

- ·广度优先搜索算法 BFS，解决无权图的最短路径问题
- ·带权图的最短路径算法 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
- ·用于排序任务的拓扑排序算法
- ·用于广播消息的最小生成树算法

## 7.14. 关键词

| | | |
|---|---|---|
| 无圈图 | 邻接表 | 邻接矩阵 |
| 邻近的 | 广度优先搜索 (BFS) | 圈 |
| 有圈图 | 有向无圈图(DAG) | 深度优先森林 |
| 深度优先搜索 (DFS) | 有向图 | 有向无圈图 (DAG) |

| | | |
|---|---|---|
| 有向图 | 边的权重 | 边 |
| 括号性质 | 路径 | 最短路径 |
| 生成树 | 强连通分支 (SCC) | 拓扑排序 & 不受限洪水 |
| 顶点 | 权重 | |

## 7.15. 问题讨论

1、画出与下示邻接矩阵对应的图。

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

2、画出与下列表格中的边对应的图。

| 起点 | 终点 | 权重 |
|---|---|---|
| 1 | 2 | 10 |
| 1 | 3 | 15 |
| 1 | 6 | 5 |
| 2 | 3 | 7 |
| 3 | 4 | 7 |
| 3 | 6 | 10 |
| 4 | 5 | 7 |
| 6 | 4 | 5 |
| 5 | 6 | 13 |

3、去掉权重，在上个问题的图中实现深度优先搜索。

4、buildGraph 函数的大 O 时间复杂度是多少。

5、导出拓扑排序算法的大 O 时间复杂度。

6、导出强连通分支算法的大 O 时间复杂度。

7、展示将 Dijkstra 算法应用在上述图时的每一步。

8、用 Prim 算法，找到上述图中的具有最小权重的生成树。

9、画出关系图来解释你发一封邮件所需的步骤。在你的图中执行一次拓扑排序。

10、导出表达骑士周游问题执行时间的指数的底的表达式。

11、解释为什么一般的深度优先搜索算法不适合解决骑士周游问题。

12、Prim 最小生成树算法的大 O 时间复杂度是多少。

## 7.16.编程练习

1. 修改深度优先搜索函数产生一个拓扑排序。
2. 修改深度优先搜索生成强连通分支。
3. 写出图的转置方法。
4. 使用广度优先搜索，写一个算法来确定从每个顶点到其他所有顶点的最短路径。这就是所谓的对最短路径问题。
5. 用广度优先搜索修改递归章节的迷宫程序，找到走出迷宫的最短路径。
6. 写一个程序来解决以下问题：你有两个罐子，一个是 4 加仑、另一个是 3 加仑。每个罐子上都没有标记。有一个泵，可以用来为罐子满上水。你如何使用 4 加仑的水罐得到两加仑的水？
7. 总结上述问题，你解决方案的参数包括每一个罐子的大小和最后留在较大罐子里的水量。
8. 写一个程序来解决以下问题：三个传教士和三个食人族到河边，发现一艘船一次只能载两人。每个人都要过河来继续旅程。然而，如果任意一边的岸上的食人族数量超过传教士，传教士将被吃掉。找到使每个人都安全地到达河的另一边的所有路径。