

# 数据结构与算法 (Python)

## 08/图及算法

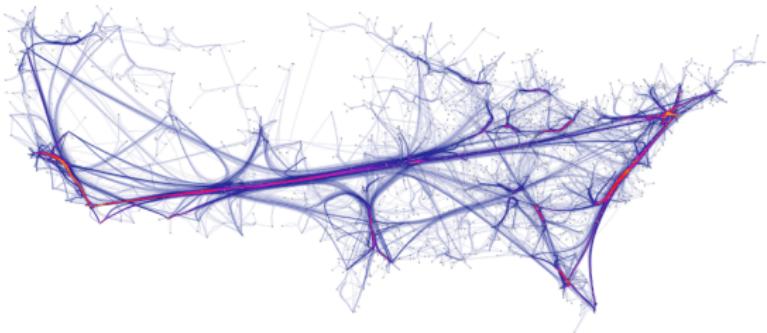
谢正茂 webg@PKU-Mail

计算机学院数据所

May 27, 2025

# 目录

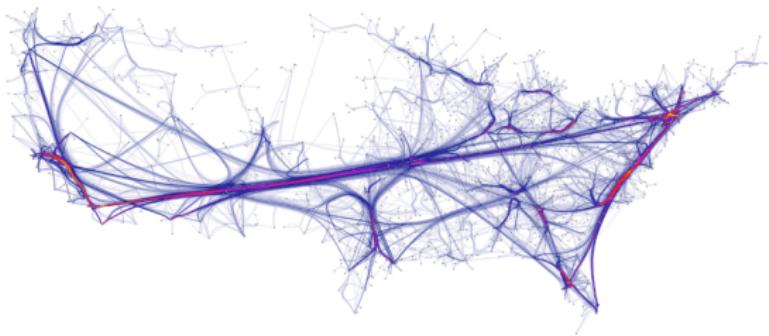
- 本章目标
- 图抽象数据类型及实现
- Word Ladder 词梯问题
- 骑士周游问题
- 拓扑排序和强连通分支
- 最短路径问题
- Prim 最小生成树算法



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

# 目录

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

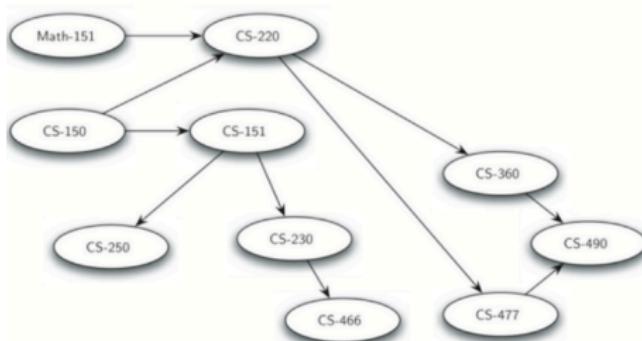


# 图 Graph 的概念

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

# 图 Graph 的概念

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

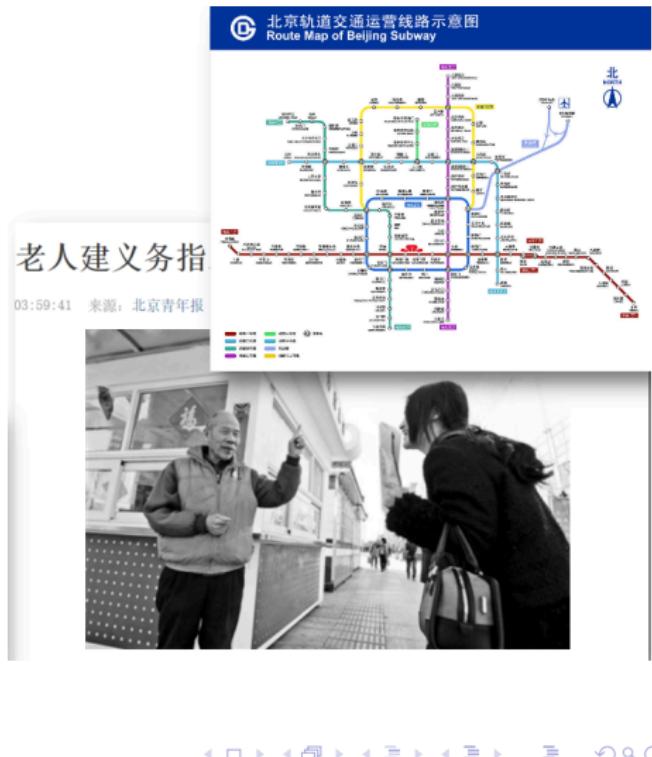


# 图 Graph 的概念

- 计算科学对图的研究相对透彻
- 一些看起来很艰深的问题，如果能够抽象成图，往往能够得到很好的解决
  - 如果用图来表示地图，就可以解决很多对地图很熟悉甚至专业的人才能解决的问题，甚至能超越；
  - 互联网是由成千上万的计算机所连接起来的复杂网络，也可以通过图算法来确定计算机之间达成通讯的最短、最快或者最有效的路径。
  - 大学课程之间的先修依赖关系，也适合用图来表示
  - 生活中大量的事情适合用图来表示

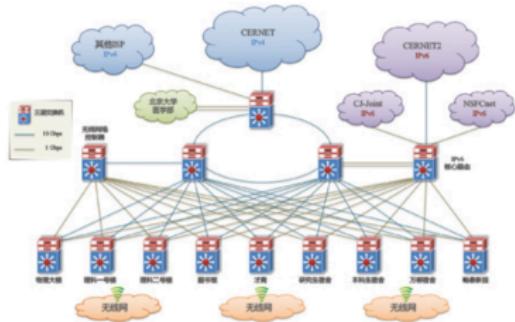
# 公共交通中的图结构

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



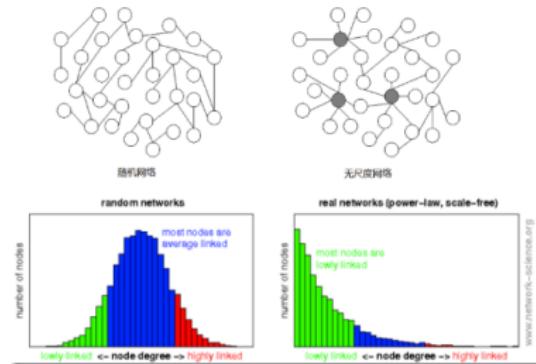
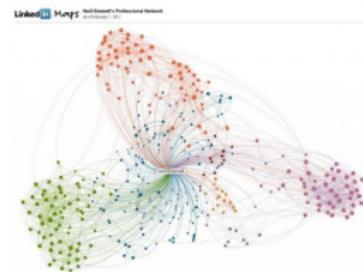
# 互联网中的图结构

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



# 社交网络：六度分隔理论

- 世界上任何两个人之间通过最多 6 个人即可建立联系
  - 互联网社交网络的兴起将每个人联系到一起
- 在社会中有 20% 擅长交往的人，建立了 80% 的连接
  - 区别于随机网络，保证了六度分隔的成立
  - 引出了无尺度网络的研究
  - 现实中的复杂网络多属于无尺度网络



# 术语表

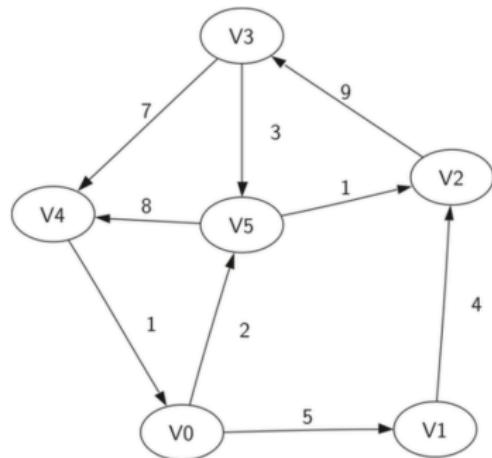
- 顶点 Vertex
  - 也称“结点 Node”，是图的基本组成部分，顶点具有名称标识 Key，也可以携带数据项 payload。
- 边 Edge
  - 也称“弧 Arc”，是图的另一个基本组成部分，作为 2 个顶点之间关系的表示，边连接两个顶点；边可以是单向 one-way 或者双向 two-way 的，如果一个图中的所有边都是单向的，就称这个图为“有向图 directed graph/digraph”。
  - 如何判断一条边是单向还是双向：看边两端结点的地位是不是对称/对等的
  - 一条有向边与它两个顶点的关系：一为连出，一为连入。
- 度 Degree
  - 连接某个顶点边的数量，称为该顶点的“度”。如果是有向图的话，“度”又分为“入度”与“出度”。
  - 如果每个顶点的度都为  $k$ ，则该图被称为  $k$ - 正则图。
- 权重 Weight
  - 为了表达从一个顶点到另一个顶点的“代价”，可以给边赋权；例如公交网络中两个站点之间的“距离”、“通行时间”和“票价”都可以作为权重。

# 图的定义

- 一个图  $G$  可以定义为  $G=(V, E)$ 
  - 其中  $V$  是顶点的集合
  - $E$  是边的集合,  $E$  中的每条边  $e=(v, w)$ ,  $v$  和  $w$  都是  $V$  中的顶点;
  - 如果是赋权图, 则可以在  $e$  中添加第三个权重分量
  - 子图 subgraph:  $V$  和  $E$  的子集
- 赋权图的例子: 6 个顶点及 9 条边
  - 有向图
  - 权重为整数

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5\}$$

$$E = \left\{ (v_0, v_1, 5), (v_1, v_2, 4), (v_2, v_3, 9), (v_3, v_4, 7), (v_4, v_0, 1), (v_0, v_5, 2), (v_5, v_4, 8), (v_3, v_5, 3), (v_5, v_2, 1) \right\}$$



# 术语表

## ● 路径 Path

- 图中的路径，是由边依次连接起来的顶点序列；
- $P=(w_1, w_2, \dots, w_n)$ ，其中对于所有  $1 \leq i \leq n-1$ ,  $(w_i, w_{i+1})$  属于  $E$ ；
- 无权路径的长度为边的数量，等于  $n-1$ ；带权路径的长度为所有边权重的和；
- 如，前页中的一条路径  $(v3, v4, v0, v1)$ ，其边为  $(v3, v4, 7), (v4, v0, 1), (v0, v1, 5)$  长度为 13

## ● 圈 (环)Cycle

- 圈是首尾顶点相同的路径，如前页中  $(v5, v2, v3, v5)$  是一个圈
- 如果一个图中不存在任何圈，则称作“无圈图 acyclic graph”
- 无圈的有向图称作“有向无圈图 directed acyclic graph: DAG”
- 后面我们经常看到一个问题能表示成 DAG，然后用图算法很好地解决

## ● 思考：树是一种什么性质的图？是 DAG 么？

# 抽象数据类型：ADT Graph(有向图)

- 抽象数据类型 ADT Graph 定义如下：
  - `Graph()`: 创建一个空的图；
  - `addVertex(vert)`: 将一个顶点 `Vertex` 对象加入图中
  - `addEdge(fromVert, toVert)`: 添加一条有向边
  - `addEdge(fromVert, toVert, weight)`: 添加一条带权的有向边
  - `getVertex(vertKey)`: 查找名称为 `vertKey` 的顶点
  - `getVertices()`: 返回图中所有顶点列表
  - `in`: 按照 `vert in graph` 的语句形式，返回顶点是否存在图中 True/False
- ADT Graph 的实现方法有两种主要形式：
  - 邻接矩阵 `adjacency matrix` 和邻接表 `adjacency list`
  - 两种方法各有优劣，需要在不同的应用中加以选择

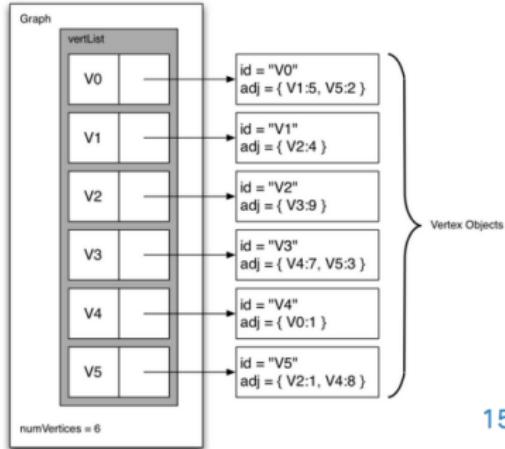
# 邻接矩阵 Adjacency Matrix

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

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

# 邻接表 Adjacency List

- 邻接表 adjacency list 可以成为稀疏图的更高效实现方案
  - 维护一个包含所有顶点的主列表 (master list, 可以用 dict 实现)
  - 主列表中的每个顶点，再关联一个与自身有边连接的所有顶点的列表
    - 在 Python 实现的 Vertex 类中，可以采用字典来保存顶点列表
    - 字典中的 key 对应顶点标识，而 value 则可以保存顶点连接边的权重
- 邻接列表法的存储空间紧凑高效，容易获得邻居顶点及连接边的信息
- 有向图中通常只记录“连出边”或“连入边”(二选一)。缺省为前者，后者特称为“入边邻接表”或“逆邻接表”。



# ADT Graph 的实现

- 包括两个类 VertBase 和 Graph
  - Graph 保存了包含所有顶点的主表 master list
  - VertBase 则包含了顶点信息，以及顶点连接边的信息
  - VertBase:  
connectedTo,  
connectedFrom?
- 参考代码：  
[github:course/graph.py](https://github.com/course/graph.py)

```
1 class VertBase:  
2     """  
3         有向带权图的顶点  
4     """  
5     def __init__(self, key):  
6         self.id = key  
7         self.connectedTo = {}  
8         self.inDegree = self.outDegree = 0  
9  
10    def addNeighbor(self, nbr, weight=1):  
11        """  
12            添加从当前节点到nbr节点的有向边  
13            添加重复边会更新其权值，不影响节点的出入度。  
14        """  
15        if nbr not in self.connectedTo:  
16            self.outDegree += 1  
17            nbr.inDegree += 1  
18            self.connectedTo[nbr] = weight  
19  
20    def delNeighbor(self, nbr):  
21        """  
22            删除有向边  
23        """  
24        if nbr in self.connectedTo:  
25            self.outDegree -= 1  
26            nbr.inDegree -= 1  
27            del(self.connectedTo[nbr])  
28  
29    def __str__(self):  
30        return f"{self.id}({self.inDegree}:{self.outDegree})"  
31  
32    __repr__ = __str__  
33  
34    def getConnections(self):  
35        """  
36            取得所有指向的节点  
37        """  
38        return self.connectedTo.keys()  
39  
40    def getId(self):  
41        return self.id  
42  
43    def getWeight(self, nbr):  
44        """  
45            取得指向某节点边的权重  
46        """  
47        return self.connectedTo[nbr]
```

# ADT Graph 的实现

- 新加顶点 →
- 通过key查找顶点 →

```
class Graph:  
    def __init__(self):  
        self.vertList = {}  
        self.numVertices = 0  
  
    def addVertex(self, key):  
        self.numVertices = self.numVertices + 1  
        newVertex = Vertex(key)  
        self.vertList[key] = newVertex  
        return newVertex  
  
    def getVertex(self, n):  
        if n in self.vertList:  
            return self.vertList[n]  
        else:  
            return None  
  
    def __contains__(self, n):  
        return n in self.vertList
```

# ADT Graph 的实现

- 先加顶点再加边
- 在顶点的出边表中记录了有向边的目标顶点和权重
- 用顶点对象作 dict 键值，是否 hashable？

```
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())
```

不存在的顶点先添加

调用起始顶点的方法  
添加邻接边

# ADT Graph 的实现：示例

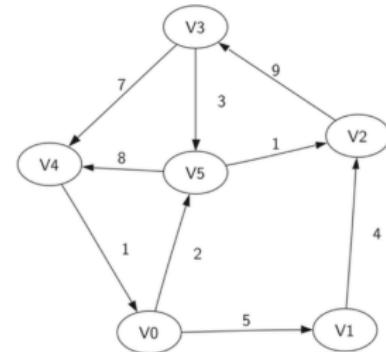
```
>>> g= Graph()
>>> for i in range(6):
    g.addVertex(i)

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

# ADT Graph 的实现：示例

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

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



# 目录

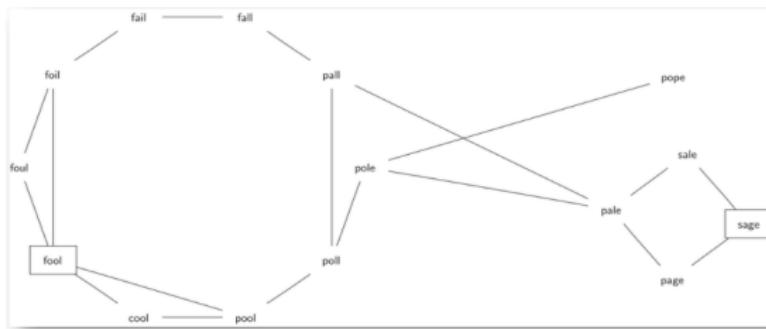
- 本章目标
- 图抽象数据类型及实现
- **Word Ladder 词梯问题**
- 骑士周游问题
- 拓扑排序和强连通分支
- 最短路径问题
- Prim 最小生成树算法



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

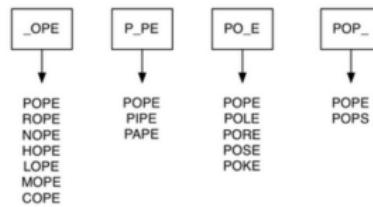
# 词梯问题：构建单词关系图

- 首先是如何将大量的单词集放到图中
  - 将单词作为顶点的标识 Key
  - 如果两个单词之间仅相差 1 个字母，就在它们之间设一条边
  - 这样，在两个单词之间的任意一条路径，就是词梯问题的一个解
- 下图是从 FOOL 到 SAGE 的词梯解，所用的图是无向图，边没有权重，对等关系
  - A 和 B 相差一个字母，必然 B 和 A 也相差一个字母
  - FOOL 到 SAGE 的每条路径都是一个解



# 词梯问题：构建单词关系图-边的发现

- 单词关系图可以通过不同的算法来构建，以 4 个字母的单词表为例，单词表 `vocabulary.txt`，共 3993 个。
  - 首先是将所有单词作为顶点加入图中，再设法建立顶点之间的边
- 建立边的最直接算法，是对每个顶点（单词），与其它所有单词进行比较，如果相差仅 1 个字母，则建立一条边
  - 时间复杂度是  $O(n^2)$ ，对于所有 4 个字母的 3993 个单词，需要超过 1547 万次比较
- 改进的算法是创建大量的桶，每个桶可以存放若干单词，桶的标记是去掉 1 个字母，以通配符“\_”占空的单词，所有匹配标记的单词都放到这个桶里，所有单词就位后，再在同一个桶的单词之间建立边即可。
  - 最多有多少个桶？以空间换时间。



# 词梯问题：构建单词关系图

- 可以采用 Python 字典来建立桶， course/wordLadder.py

```
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
```

4字母单词可以属于4个桶

同一个桶的单词之间建立边

# 词梯问题：构建单词关系图-图的存储

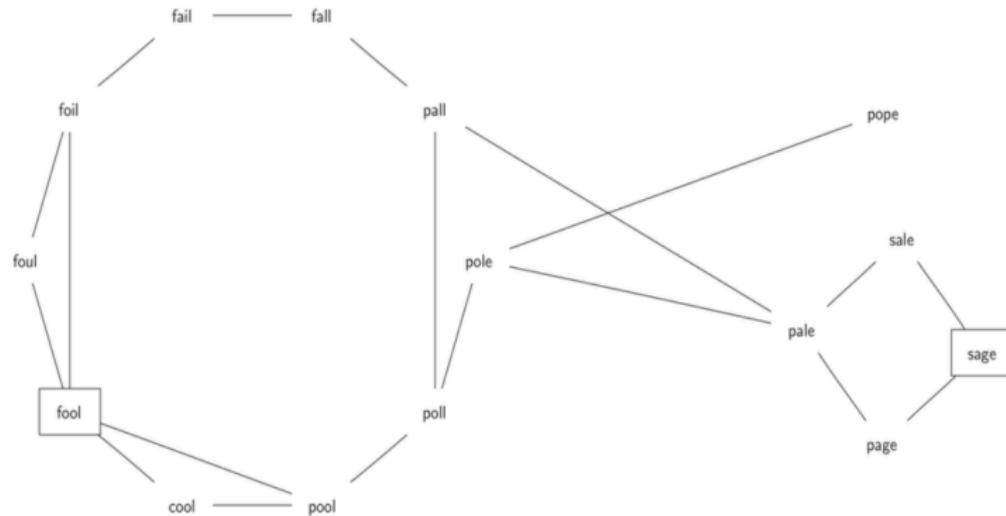
- 样例数据文件 `vocabulary.txt` 包含了 3,933 个 4 字母的单词
- 如果采用邻接矩阵表示这个单词关系图，则需要 1,547 万个矩阵单元
  - $3,933 * 3,933 = 15,468,489$
  - 而单词关系图总计有 42,600 条边，仅仅达到矩阵单元数量的 0.27%
- 单词关系图是一个非常稀疏的图
- 猜想与验证：这个单词关系图会是一个随机网络吗？

# 实现广度优先搜索 (Breadth First Search)

- 在单词关系图建立完成以后，需要继续在图中寻找词梯问题的最短序列。
- 需要用到“广度优先搜索 Breadth First Search:BFS”算法对单词关系图进行搜索
- BFS 是搜索图的最简单算法之一，也是其它一些重要的图算法的基础
- 给定图 G，以及开始搜索的起始顶点 s
  - BFS 搜索所有从 s 可到达顶点的边
  - 而且在达到更远的距离  $k+1$  的顶点之前，BFS 会找到全部距离为 k 的顶点
  - 可以想象为构建一棵树的过程：以 s 为根，从顶部向下逐步增加层次
  - 广度优先搜索能保证在增加层次之前，添加了所有兄弟结点到树中

# BFS 算法过程

- 我们从 FOOL 开始搜索



# BFS 算法过程

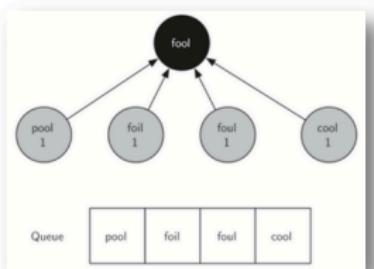
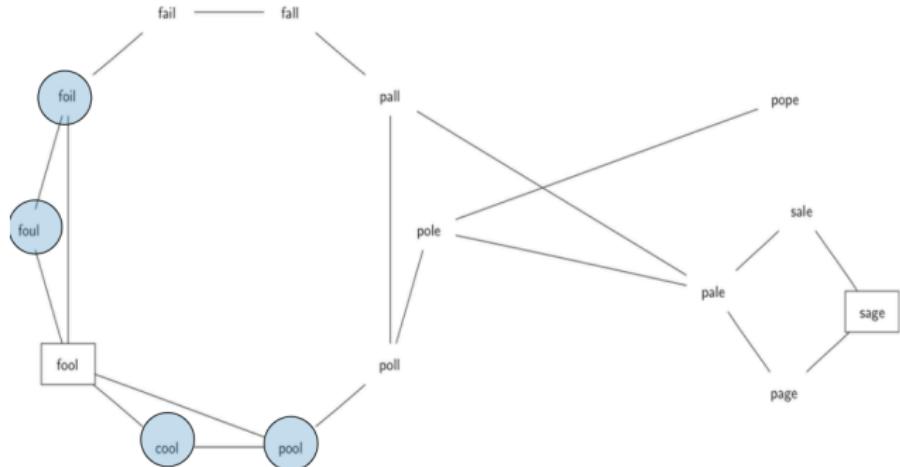
- 为了跟踪顶点的加入过程，并避免重复顶点，要为顶点增加 3 个属性
  - 距离 **distance**: 从起始顶点到此顶点的路径长度；
  - 前驱顶点 **pred**: 可追溯到起始顶点的反向路径；
  - 颜色 **color**: 标识了此顶点是尚未发现（白色）、已经发现（灰色）、还是已经完成探索（黑色）
- 还需要用一个队列 **Queue** 来对已发现的顶点进行排列，决定下一个要探索的顶点（队首顶点）
- 前驱顶点的设定，使搜索过程得到了一个“树”型结构。

```
140 #对 Vertex 进行扩充,
141 #增加了 distance, color, pred
142 #和 discovery, finish 属性,
143 #从而对 BFS/DFS 搜索算法进行支持。
144 import sys
145 class Vertex(VertexBase):
146     def __init__(self, key):
147         super().__init__(key)
148         self.color = 'white'
149         self.distance = sys.maxsize
150         self.pred = None
151         """
152         self.pred = []
153         前驱可能有多个值，用一个列表来存放。
154         这样可以通过 BFS 算法找出所有的最短路径。
155         """
156         self.discovery = None
157         self.finish = None
158
159     def setDiscovery(self, t):
160         self.discovery = t
161
162     def setFinish(self, t):
163         self.finish = t
164
165     def getDistance(self):
166         return self.distance
167
168     def setDistance(self, dist):
169         self.distance = dist
170
171     def setPred(self, pred):
172         self.pred = pred
173
174     def getPred(self):
175         return self.pred
176
177     def getColor(self):
178         return self.color
179
180     def setColor(self, color):
181         self.color = color
```

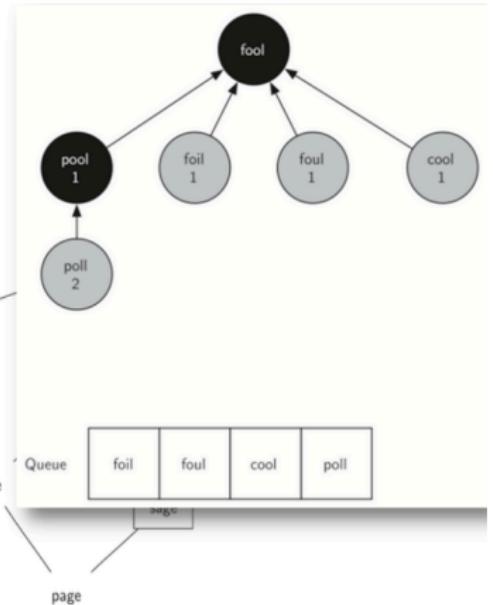
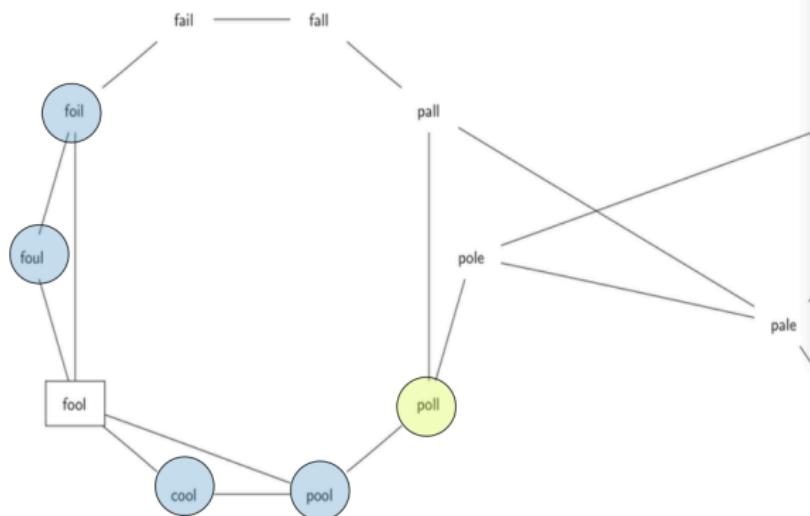
# BFS 算法过程

- 从起始顶点  $s$  开始，作为刚发现的顶点，标注为灰色，距离为 0，前驱为  $\text{None}$ ，加入队列，接下来是个循环迭代过程：
  - 从队首取出一个顶点作为当前顶点；
  - 遍历当前顶点的邻接顶点，如果是尚未发现的白色顶点，则将其颜色改为灰色（已发现），距离增加 1，前驱顶点为当前顶点，加入到队列中
  - 遍历完成后，将当前顶点设置为黑色（已探索过），循环回到步骤 1

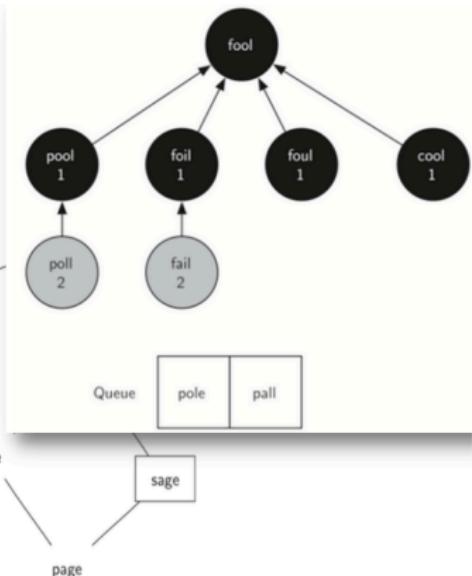
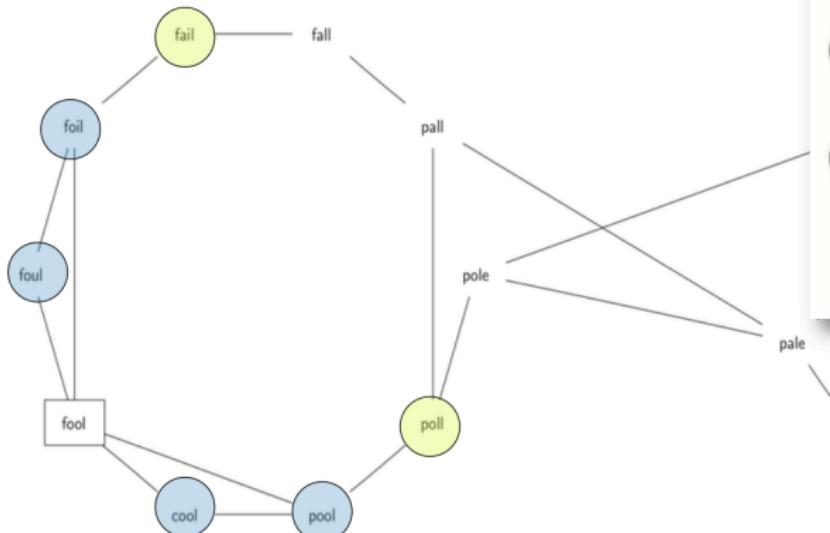
# BFS 算法示例运行



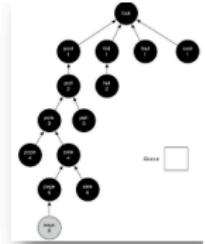
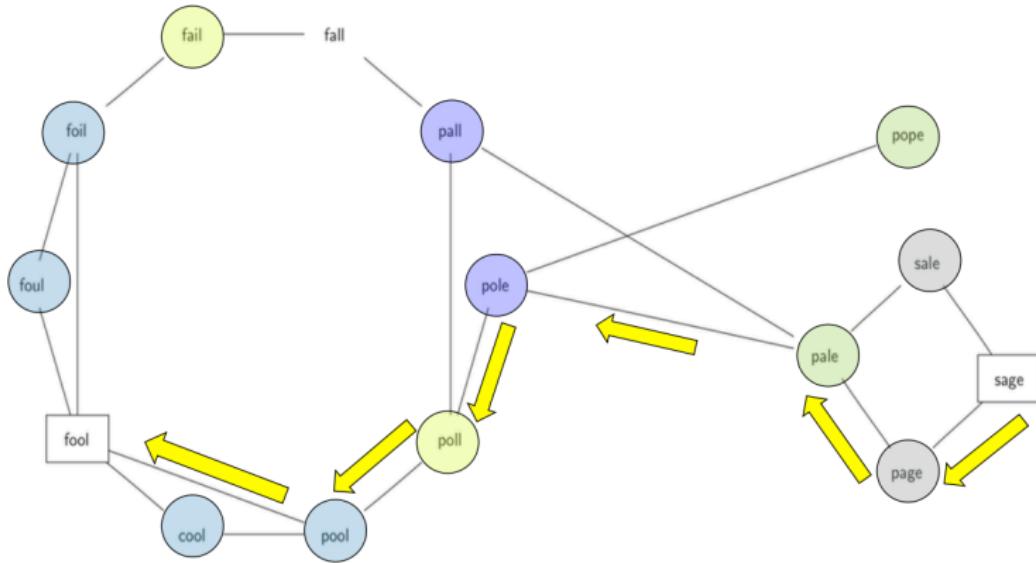
# BFS 算法示例运行



# BFS 算法示例运行



# BFS 算法示例运行



# BFS 算法代码

- course/bfs.py

```
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')
```

取队首作为当前顶点

遍历当前顶点邻接顶点

当前顶点设黑色

# BFS 算法代码

- 在 BFS 以 FOOL 为起始顶点，遍历了所有顶点，并为每个顶点着色、设置距离和前驱之后
- 即可以通过一个回途追溯函数来确定 FOOL 到任何单词顶点的最短词梯！

- 思考：FOOL 到其他单词顶点的最短词梯可能有多条，BFS 能得到所有的最短词梯解么？
- 为什么？

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

wordgraph = buildGraph("fourletterwords.txt")
bfs(wordgraph, wordgraph.getVertex('FOOL'))
traverse(wordgraph.getVertex('SAGE'))
#traverse(wordgraph.getVertex('COOL'))
```

# 广度优先搜索算法分析

- BFS 算法主体是两个循环的嵌套：while-for

- while 循环对图中每个顶点访问一次，所以是  $O(|V|)$
- 而嵌套在 while 中的 for，由于每条边只有在其起始顶点  $u$  出队的时候才会被检查一次，而每个顶点最多出队 1 次，所以边最多被检查 1 次，一共是  $O(|E|)$
- 综合起来 BFS 的时间复杂度为  $O(|V|+|E|)$

- 词梯问题还包括两个部分算法

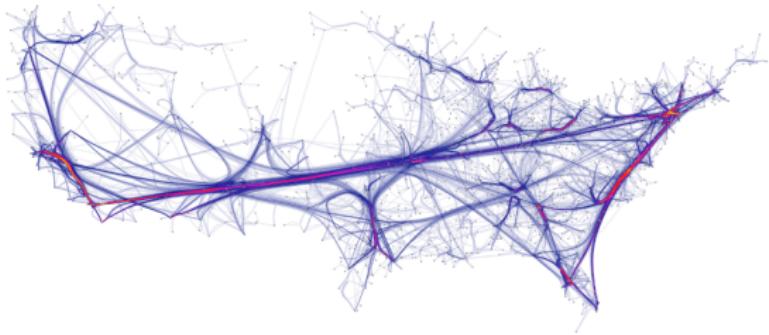
- 建立 BFS 树之后，回溯顶点到起始顶点的过程，最多为  $O(|V|)$
- 创建单词关系图也需要时间，思考：其复杂度为多少？

# 课后思考：广度优先搜索算法（Optional）

- 整理词梯问题完整算法（ADT Graph 实现、buildGraph、BFS、traverse）
- 用嵌套列表作为矩阵来实现 ADT Graph
- BFS 能得到从起始词到其他词最短词梯的所有解么？
  - 如果不能，说明理由；
  - 如果能，在现有 BFS 代码的基础上实现它

# 目录

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



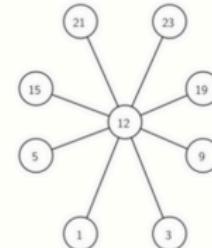
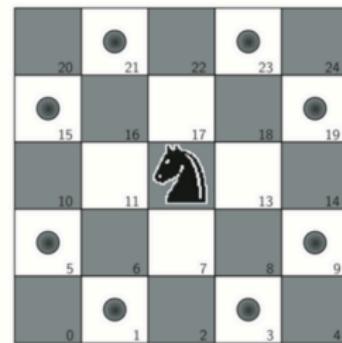
# 骑士周游问题 Knight's Tour Problem

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

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

# 构建骑士周游图

- 将棋盘和走棋步骤构建为图的思路，顶点和边
  - 将棋盘格作为顶点
  - 按照“马走日”规则的合法走棋步骤作为顶点之间的连接边
  - 建立每一个棋盘格的所有合法走棋步骤能够到达的棋盘格关系图



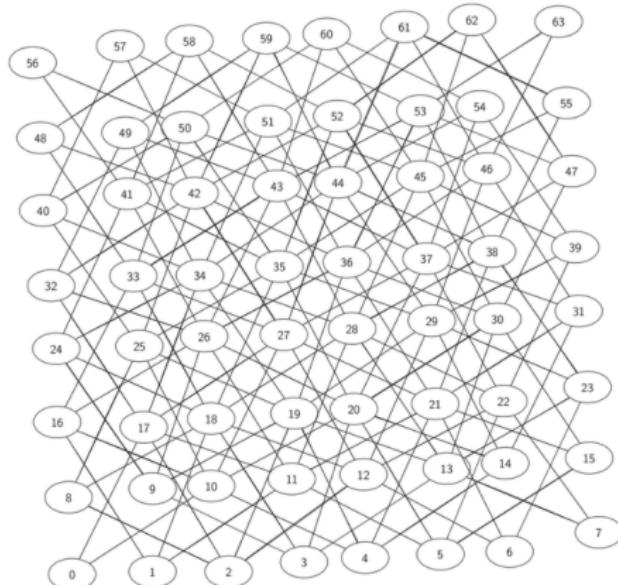
# 构建骑士周游图-代码

- course/knight.py

```
1 from graph import Graph, Vertex
2
3 def knightGraph(bdSize):
4     ktGraph = Graph(Vert=Vertex)
5     for row in range(bdSize):                      #遍历每一行
6         for col in range(bdSize):                  #遍历行上的每一个格子
7             #按照“马走日”，返回下一步可能的位置
8             for e in legalMoves(row, col, bdSize):
9                 ktGraph.addEdge((row, col), e)    #在骑士周游图中为两个格子加一条边
10
11    return ktGraph
12
13 def legalMoves(x, y, bdSize):
14     offsets = [(-1, -2), (-1, 2), (-2, -1), (-2, 1),
15                (1, -2), (1, 2), (2, -1), (2, 1)]    #马走日的8种走法
16
17     for ix, iy in offsets:
18         #检查一下不能走出棋盘
19         if 0 <= x+ix < bdSize and 0 <= y+iy < bdSize:
20             yield(x+ix, y+iy)
```

# 骑士周游图： $8 \times 8$ 棋盘生成的图

- 具有 336 条边，相比起全连接的 4096 条边， $64 \times 64$  的邻接矩阵，仅 8.2%，还是稀疏图



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

- 深度优先搜索解决骑士周游的关键思路在于：

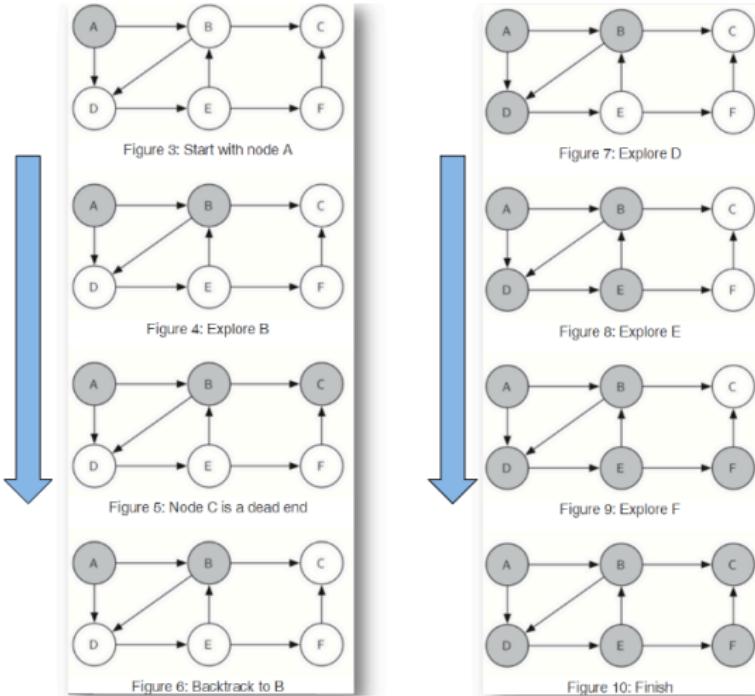
- 把棋盘、马走日都变为图信息，骑士周游变为图搜索问题
- 从起点开始，在图中每走一步给对应的结点标上颜色
- 如果沿着单支深入搜索到无法继续（所有合法移动都已经被走过了）时，而路径长度还没有达到预定值（ $8 \times 8$  棋盘为 63），那么就清除当前结点颜色标记，回退一步，换一个分支继续深入搜索。
- 上面的“回退一步”，也是一种“回溯算法”。

# 骑士周游算法代码

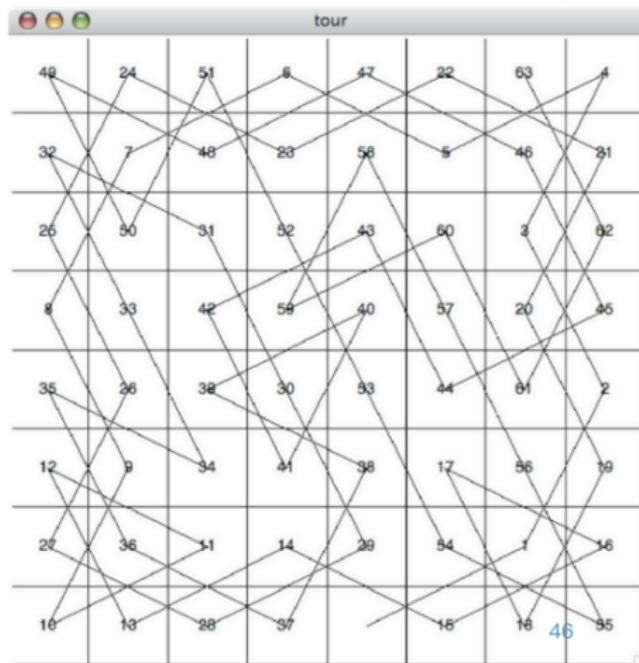
- 棋盘信息已经被转化成了图。

```
20 def knightTour(n, path, u, limit):
21     """
22     n:层次； path:路径； u:当前顶点； limit:搜索总深度
23     """
24     u.setColor('grey')
25     path.append(u)                                     #当前顶点涂色并加入路径
26     if n < limit:
27         nbrList = list(u.getConnections())           #对所有的合法移动依次深入
28         for nbr in nbrList:
29             #选择“白色”未经深入的点，层次加一，递归深入
30             if nbr.getColor() == 'white' and knightTour(n+1, path, nbr, limit):
31                 return True
32             else:                                       #所有的“下一步”都试了走不通
33                 path.pop()                           #回退途径
34                 u.setColor('white')                  #改回颜色
35                 return False                      #回到上一层，尝试下一个顶点
36     else:                                           # n==limit, 基本结束条件
37         return True
38
39 if __name__ == "__main__":
40     bd_size = 5
41     g = knightGraph(bd_size)
42     path=[]
43     start = g.getVertex((1, 1))
44     if knightTour(0, path, start, bd_size**2 - 1):
45         for i, v in enumerate(path):
46             print(i, v.id)
47     else:
48         print("knight tour failed!")
```

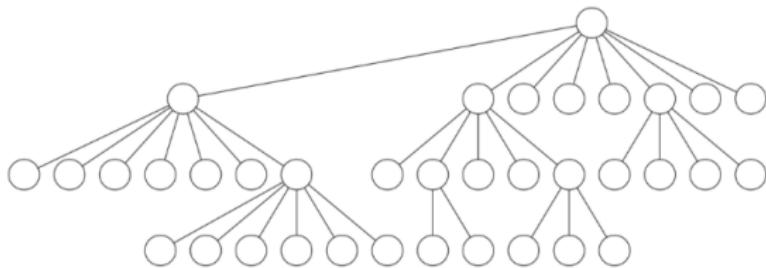
# 深度优先搜索-示例



# 骑士周游问题：一个解



- 上述算法 `knightTour` 是通过递归实现的，每走一步递归深度加一
- 算法性能高度依赖于深入顶点 `nbrList` 的搜索次序：
  - 就  $5 \times 5$  的棋盘而言，大约 1.5 秒可以得到一个周游路径
  - 但  $8 \times 8$  的棋盘上，则需要半个小时以上的时间才能得到一个解！
- 目前实现的算法，其复杂度为  $O(k^n)$ ，其中  $n$  是棋盘格数目， $k$  是每一步的走法数。这是一个指数时间复杂度的算法！其搜索过程表现为一个层次为  $n$  的树



# 骑士周游算法改进

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

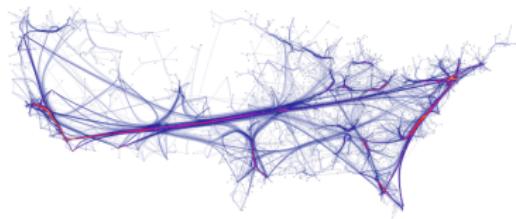
```
def orderByAvail(n):  
    resList = []  
    for v in n.getConnections():  
        if v.getColor() == 'white':  
            c = 0  
            for w in v.getConnections():  
                if w.getColor() == 'white':  
                    c = c + 1  
            resList.append((c,v))  
    resList.sort(key=lambda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

# 骑士周游算法改进

- 这个改进算法被特别以发明者名字命名：Warnsdorff 算法
  - <https://github.com/douglassquirrel/warnsdorff.git>
  - 建立根据地，稳扎稳打、避免孤军深入（150\*150 的棋盘）
- 采用先验的知识来改进算法性能的做法，称作为“启发式规则 heuristic”
  - 启发式规则经常用于人工智能领域；
  - 可以有效地减小搜索范围、更快达到目标等等；
  - 如棋类程序算法，会预先存入棋谱、布阵口诀、高手习惯等“启发式规则”，能够在最短时间内从海量的棋局落子点搜索树中定位最佳落子。
    - 例如：黑白棋中的“金角银边”口诀，指导程序优先占边角位置等等
- 常见的 DFS 优化策略：剪枝，从死路上提前返回。
  - openjudge/24375: 小木棍，有巨大的优化空间

# 目录

- 本章目标
- 图抽象数据类型及实现
- Word Ladder 词梯问题 (BFS)
- 深度优先搜索 (DFS)
  - 骑士周游问题 (无分枝搜索)
  - 通用的深度优先搜索
- 拓扑排序和强连通分支
- 最短路径问题
- Prim 最小生成树算法



# 通用的深度优先搜索

- 骑士周游问题是一种特殊的对图进行深度优先搜索，其目的是建立一个**没有分支的**最深的深度优先树
- 一般的深度优先搜索实际上更为容易，其目标是在图上进行尽量深的搜索，连接尽量多的顶点，必要时**可以进行分支**（创建了树）
  - 有时候深度优先搜索会创建多棵树，称为“深度优先森林”
  - 深度优先搜索同样要用到顶点的“前驱”属性，来构建树或森林
  - 另外还要设置“发现时间”和“结束时间”两个属性
    - 前者是算法在第几步访问到这个顶点（设置灰色）
    - 后者则是算法在第几步完成了此顶点的探索（设置黑色）
  - 这两个新属性对后面的图算法很重要
- 本质上是希望在搜索过程中到达图的每一个顶点，完成图的“遍历”；以后的其他各种工作可以在 DFS 的框架上完成
- 如骑士周游中的“深搜”不同，不需要回撤操作

# 通用的深度优先搜索-图的实现

- **pythonds** 模块，教材提供（也可以自己实现 **Graph**）
  - 源代码：<https://github.com/bnmnntp/pythonds.git>
  - \$ pip install pythonds
- 我们把带有 **DFS** 算法的图实现为 **Graph** 的子类
  - 顶点 **Vertex** 增加了成员 **Discovery** 及 **Finish**
  - 图 **Graph** 增加了成员 **time** 用于记录算法执行的步骤数目
  - **BFS** 采用队列存储待访问顶点，**DFS** 则是通过递归调用，隐式使用了栈

# 通用的深度优先搜索算法代码

```
1 from graph import Graph, Vertex
2 class DFSGraph(Graph):
3     def __init__(self, other = None):
4         super().__init__(Vertex, other)
5         self.time = 0                      # 不是物理时间，而是算法执行步数
6
7     def dfs(self):
8         for aVertex in self:
9             aVertex.setColor('white')    # 颜色初始化
10            aVertex.setPred(None)
11        for aVertex in self:          # 从每一个顶点开始遍历
12            if aVertex.getColor() == 'white':
13                self.dfsvisit(aVertex) # 建立一个以aVertex为根的树
14                                # 如果有多棵树，则形成森林
15    def dfsvisit(self, startVertex):
16        startVertex.setColor('gray')
17        self.time += 1               # 记录算法的步数
18        startVertex.setDiscovery(self.time)
19        for nextVertex in startVertex.getConnections():
20            if nextVertex.getColor() == 'white':
21                nextVertex.setPred(startVertex)
22                self.dfsvisit(nextVertex) # 深度优先递归访问
23        startVertex.setColor('black')
24        self.time += 1
25        startVertex.setFinish(self.time)
```

- course/DFSGraph.py

- 上面的 `dfsvisit` 用递归方式实现，尝试把它改写为非递归的。

# 通用的深度优先搜索算法：示例

- 与骑士周游中的无分枝 DFS<sup>[48]</sup> 比较，对于顶点 C 的处理不同

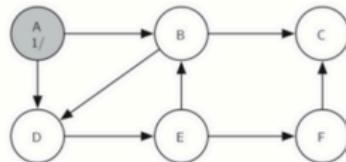


Figure 14: Constructing the Depth First Search Tree-10

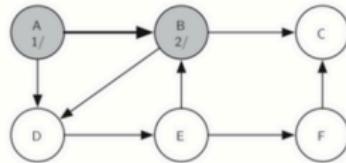


Figure 15: Constructing the Depth First Search Tree-11

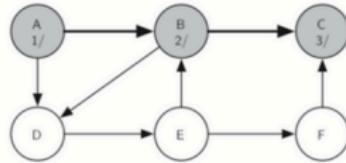


Figure 16: Constructing the Depth First Search Tree-12

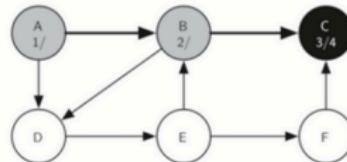


Figure 17: Constructing the Depth First Search Tree-13

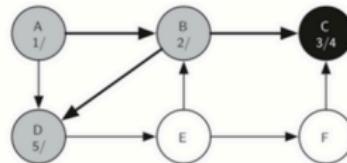


Figure 18: Constructing the Depth First Search Tree-14

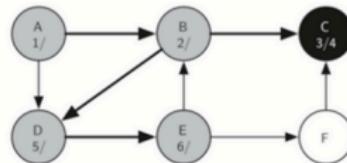


Figure 19: Constructing the Depth First Search Tree-15

# 通用的深度优先搜索算法：示例

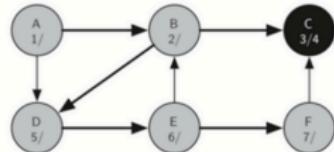


Figure 20: Constructing the Depth First Search Tree-16

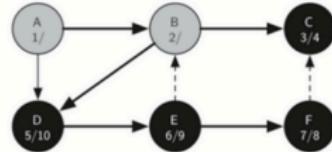


Figure 23: Constructing the Depth First Search Tree-19

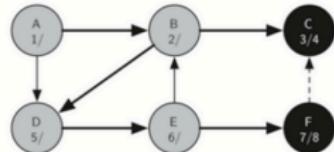


Figure 21: Constructing the Depth First Search Tree-17

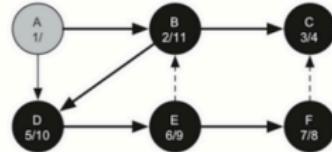


Figure 24: Constructing the Depth First Search Tree-20

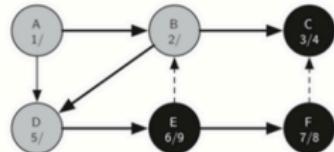


Figure 22: Constructing the Depth First Search Tree-18

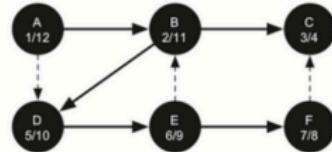


Figure 25: Constructing the Depth First Search Tree-21

# 通用的深度优先搜索算法：分析

- 如果当前顶点的所有邻居都已经被访问过，把它标记为访问“完成”，而不是像骑士周游 DFS 那样把它标记为白色并回撤。
- DFS 构建的树，其顶点上的“发现时间”和“结束时间”属性，具有类似括号的性质：即一个顶点的“发现时间”总小于所有子顶点的“发现时间”，而“结束时间”则大于所有子顶点的“结束时间”
  - 比子顶点更早被发现，更晚被结束探索
- `dfsvisit` 函数会访问所有它能够到达的结点，并把它们放在同一棵子树中
- DFS 运行时间同样也包括了两方面：
  - `dfs` 函数中有两个循环，每个都是  $|V|$  次，所以是  $O(|V|)$
  - 而 `dfsvisit` 函数中的循环则是对当前顶点所连接的顶点进行，而且仅有在顶点为白色的情况下才进行递归调用，所以对每条边来说只会运行一步，所以是  $O(|E|)$
  - 加起来就是和 BFS 一样的  $O(|V|+|E|)$
  - Pitfall: `dfs` 在循环中调用了 `dfsvisit` 啊，为什么不是乘积？

# 深度优先搜索树（林）的性质

通用的深度优先搜索算法被其他算法广泛采用，我们需要先讨论它的一些性质：

- 从  $u$  点到  $v$  点的路径  $P=(u,x,\dots,y,v)$  记为：  $u \rightarrow v$
- 定义两个顶点的大小：

$$\forall u, v \in G, u < v \equiv u.fin < v.fin \quad (1)$$

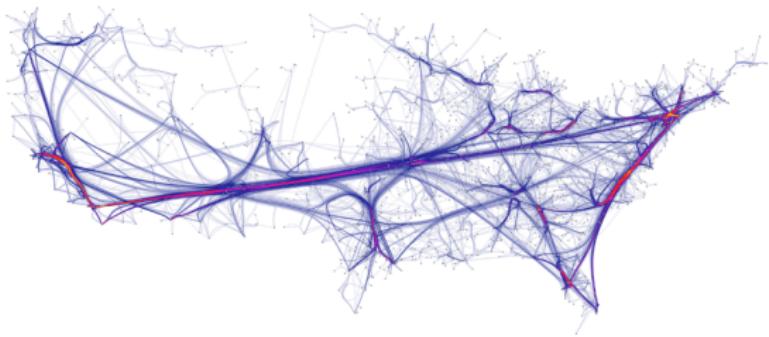
- 每棵子树所有结点的  $fin$  是连续的，可以定义两棵不相交子树的大小：

$$t1 < t2 \equiv \forall u \in t1, v \in t2, u < v \quad (2)$$

- 如果  $t1 < t2, u \in t1, v \in t2$ , 则  $u \rightarrow v$  不存在
- 如果  $u < v$ , 且在同一颗树中, 则只有两种可能：
  - $v$  是  $u$  的祖先
  - $v$  和  $u$  有共同祖先  $t$ ,  $u, v$  分别属于  $t$  的子树  $t1, t2$ , 且  $t1 < t2$

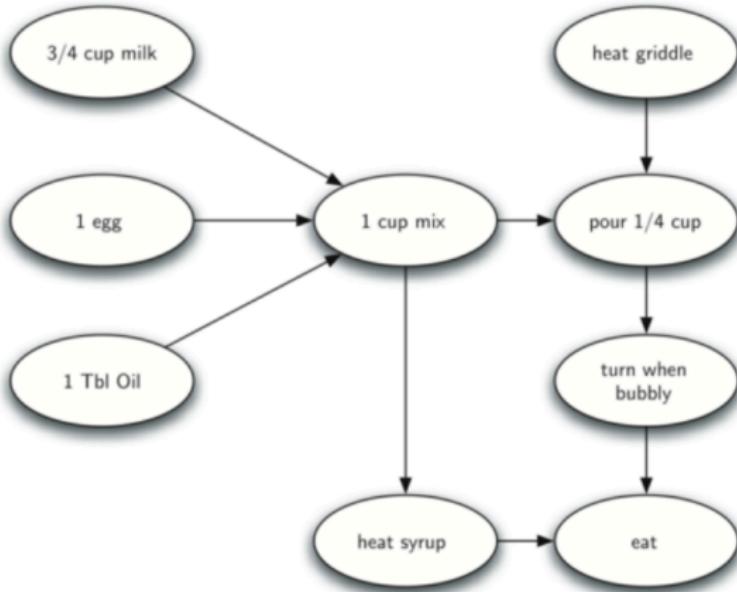
# 目录

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



# 拓扑排序 Topological Sort

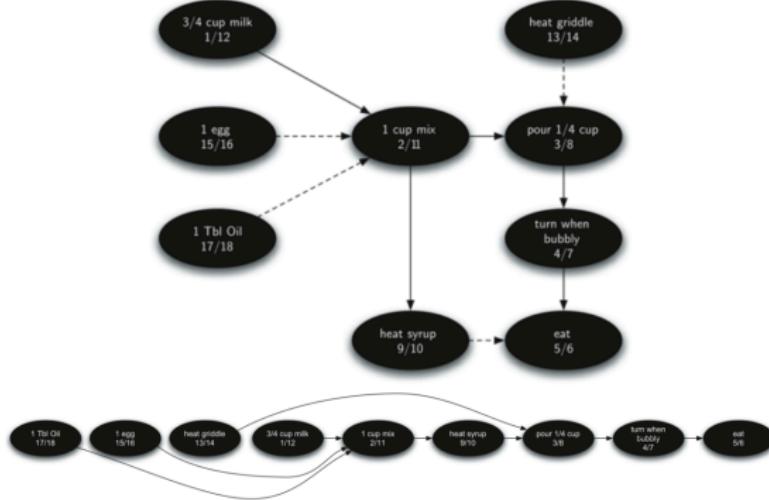
- 很多问题都可以转化为图，利用图算法来解决
- 例如早餐吃薄煎饼的过程
  - 以动作为顶点
  - 以先后次序为有向边
- 问题是整个过程而言
  - 如果一个人独自做，
  - 所有动作的先后次序？
- 从加料开始？
  - 还是从加热烤盘开始？
- 思考：整个过程次序



# 拓扑排序 Topological Sort

- 从工作流程图得到工作次序线性排列的算法，称为“拓扑排序”
- 拓扑排序输入一个 DAG（有向无圈图），输出顶点的线性序列，使得任意两个顶点  $v, w$ ，如果  $G$  中有  $(v, w)$  边，在线性序列中  $v$  就出现在  $w$  之前。
- 该算法广泛应用在依赖事件的排期上，除了吃早餐过程之外，还可以用在项目管理、数据库查询优化和矩阵乘法的次序优化上
- 拓扑排序可以采用通用的 DFS 很好地实现：
  - 对 DAG 图调用 DFS 算法，以得到每个顶点的“结束时间”
  - 按照每个顶点的“结束时间”从大到小排序
  - 输出这个次序下的顶点列表
- DFS 拓扑排序算法的证明，利用深度优先搜索树的性质 [60]
  - 反证法：假设存在  $u < v$ ，且  $G$  中存在  $(u, v)$  边。
- 课后练习：整理出骑士周游算法、通用 DFS 以及拓扑排序算法的代码

# 拓扑排序：示例



- 如果输入的图中有“圈”会发生什么？算法失效。

```
+++ b/dsa2020/code/DFSGraph.py
@@ -20,6 +20,9 @@ class DFSGraph(Graph):
    if nextVertex.getColor() == 'white':
        nextVertex.setPred(startVertex)
        self.dfsvisit(nextVertex) #深度优先递归访问
+       elif nextVertex.getColor() == 'gray':
+           raise ValueError("Ring detected")
+
```

```
27 if __name__ == "__main__":
28     g = DFSGraph()
29     #把做煎饼的步骤导入到图中
30     g.addEdge("milk", "mix")
31     g.addEdge("egg", "mix")
32     g.addEdge("oil", "mix")
33     g.addEdge("mix", "syrup")
34     g.addEdge("mix", "pour")
35     g.addEdge("pour", "turn")
36     g.addEdge("turn", "eat")
37     g.addEdge("eat", "syrup")
38     g.addEdge("syrup", "eat")
39
40     g.dfs()
41     vertices = [vert for vert in g]
42     vertices.sort(key=lambda x:x.fin,
43                   reverse=True)
44     for vert in vertices:
45         print(vert.getId())
```

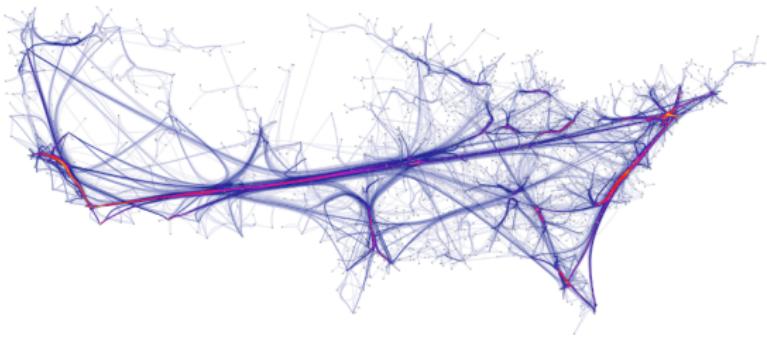
# 拓扑排序 II

- 根据顶点的入度进行拓扑排序
- 图中入度为“零”的顶点不需“依赖”其他顶点，从图中取出排序
- 更新余下顶点的入度，重复上一过程，直至所有顶点都被取出
- 如果还有顶点没被排序，但又找不到“零入度”顶点，即说明图中存在“圈”

```
1 def topSort(g):
2     result = []
3     while g.numVertices > 0:
4         zeroIn = [vert for vert in g if vert.inDegree == 0]
5         if len(zeroIn) == 0:
6             raise ringFound
7         for vert in zeroIn:
8             result.append(vert.getId())
9             g.delVertex(vert)
10    return result
```

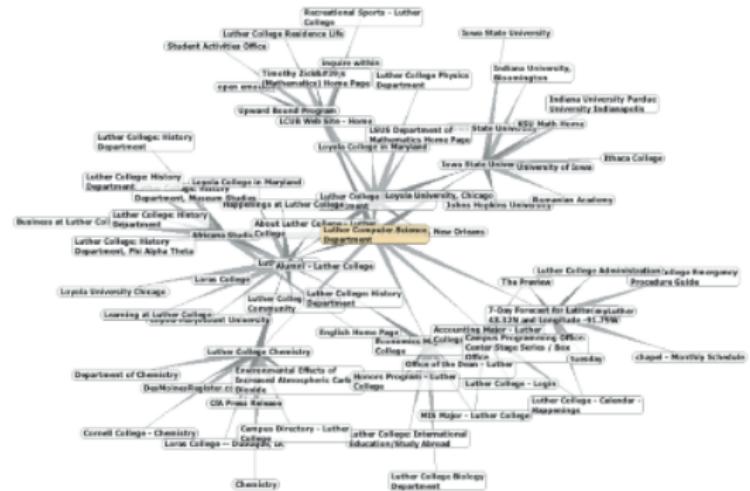
# 目录

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



## 强连通分支-关联结点的聚集

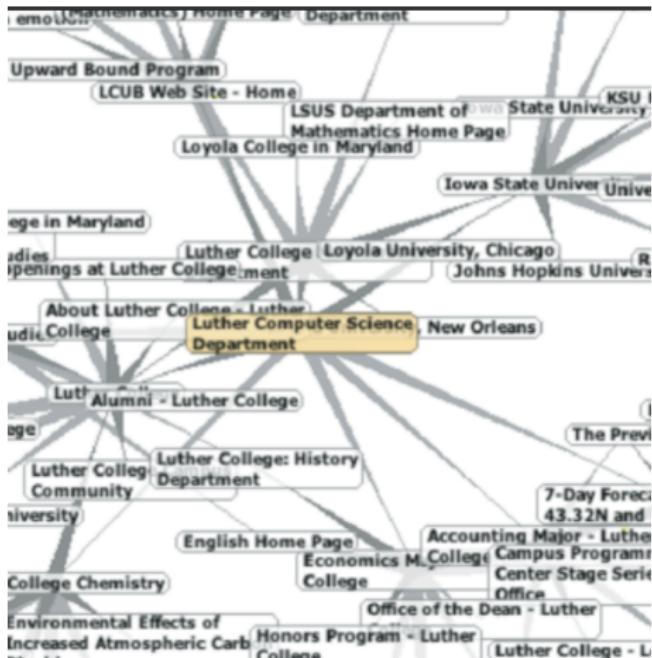
- 我们关注一下互联网相关的非常巨大图：由主机通过网线（或无线）连接而形成的图；以及由网页通过超链接连接而形成的图。
  - 先看网页形成的图：以网页（URI 作为 id）顶点，网页内包含的超链接（Hyperlink：指向另一个网页的可点击链接）作为边，可以转换为一个有向图。



# 强连通分支-关联结点的聚集

- 上图是 Luther College 的 Computer Science 网站链接指向情况，有三个有趣的现象：

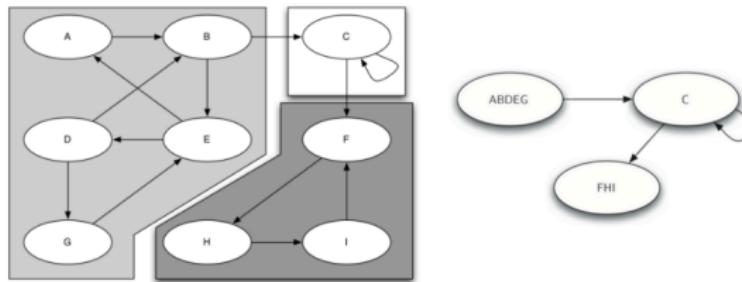
- 图中包含了许多 Luther College 其它系的网站
- 包含了一些 Iowa 其它大学学院的网站
- 还包含了一些人文学院的网站



- 我们可以猜想，Web 的底层结构可能存在某些同类网站的聚集
- 在图中发现高度聚集结点群的算法，即寻找“强连通分支 Strongly Connected Components”算法。
- 强连通分支 SCC，定义为图 G 的一个子集 C
- C 中的任意两个顶点 v,w 之间都强连通，即有路径来回
  - 即  $v \rightarrow w$  且  $w \rightarrow v$
- 而且 C 是具有这样性质的极大 (maximal) 子集
  - 增一分则太肥：增加任意顶点 u 到 C，都会破坏 C 的强联通性
- DAG 中的所有 SCC 都是平凡的
- 顶点与顶点之间的强连通关系 R 是一种等价关系：交换律/传递律
  - $R(u, v) \Rightarrow R(v, u)$
  - $R(u, v), R(v, w) \Rightarrow R(u, w)$

# 强连通分支应用的例子：顶点聚类

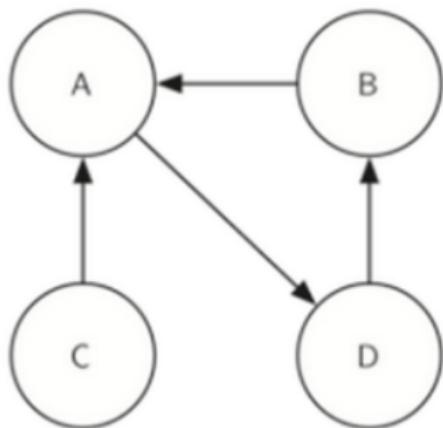
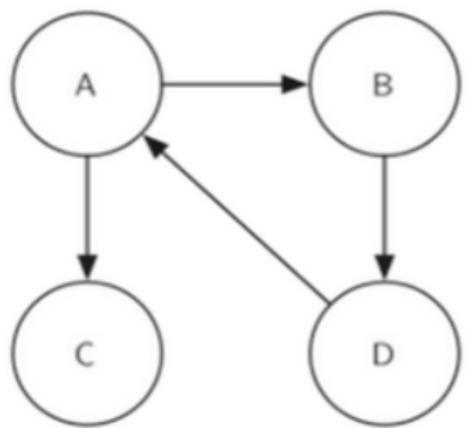
- 下图是具有 3 个强连通分支的 9 顶点有向图
- 通过强连通分支可以对图的顶点进行聚类，并把图化简成 DAG。
- 在社交网络中，聚类形成的圈子反应某一方面的共性，比如：
  - 都对某款游戏感兴趣，计算出圈子后集中推销广告（精准营销）



- 更多应用：布尔方程可满足性问题 (SAT)

# 强连通分支问题：转置 Transposition 概念

- 在使用深度优先搜索 DFS 算法来发现强连通分支之前，先熟悉一个概念：Transposition 转置
  - 一个有向图  $G$  的转置  $G^T$ ，定义为将图  $G$  的所有边的顶点交换次序，如将  $(v,w)$  转换为  $(w,v)$ ，如图所示的图  $G$  和转置  $G^T$ ：
  - 可以观察到图和转置图在强连通分支的数量和划分上，是等价的。

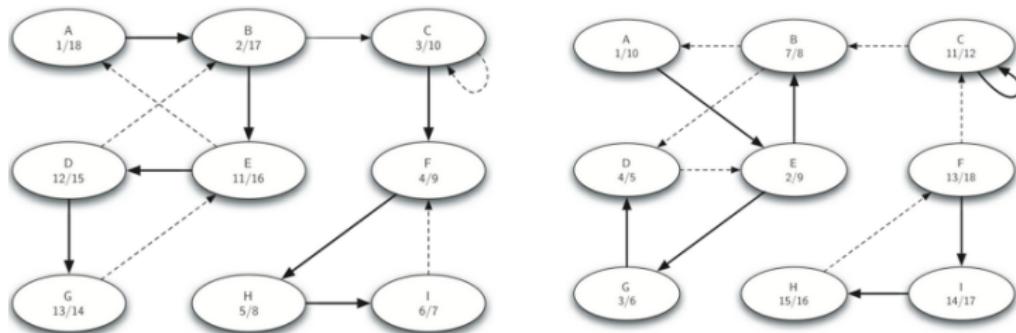


# 强连通分支问题：Kosaraju<sup>1</sup>算法

一个在线性时间内寻找一个有向图中的强连通分量的算法。

course/DG2SC.py

- 首先，对图  $G$  调用 `dfs` 算法，为每个顶点计算“结束时间”；
- 然后，将图  $G$  进行转置，得到  $G^T$ ；
- 再对  $G^T$  调用 `dfs` 算法，但在 `dfs` 函数产生树根的循环中<sup>[56]</sup>，要以顶点的“结束时间”倒序来搜索
- 最后，深度优先森林中的每一棵树就是一个强连通分支



<sup>1</sup>科萨拉朱, 1978

# 强连通分支问题：Kosaraju 算法

- 输出的强连通分支如图

- 算法的证明

- 参考：[edward-mj.com/archives/455](http://edward-mj.com/archives/455)
- 该算法求出的都是强连通分量
- 所有的极大强连通分量都会被该算法求到

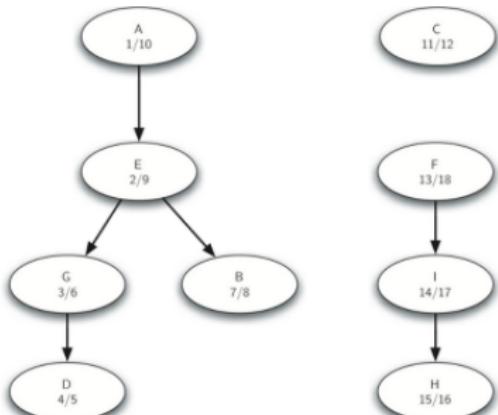
- 另一个常用的强连通分支算法

- Tarjan 算法

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

- 课后练习 1：写出本算法（Kosaraju 算法）的代码

- 课后练习 2：根据参考资料写出 Tarjan 算法的代码

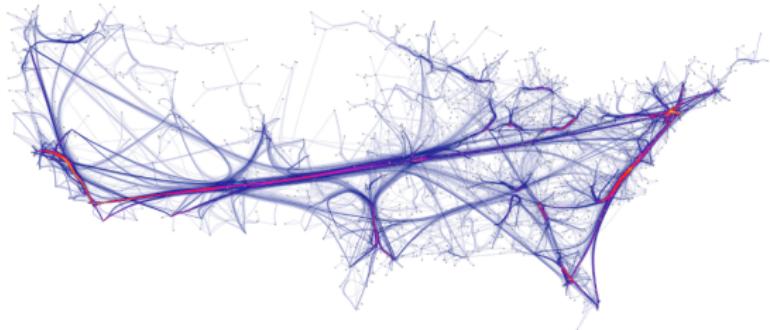


# 课后思考：强连通性在社会网络研究中的应用

- 社会网络：每个人是一个顶点，两个人之间的认识关系作为有向边。
  - 我认识某明星，但该明星不认识我
- 六度分隔理论：世界上任何互不相识的两人，只需要很少的中间人就能够建立起联系。
- 理论在一个大的范围“世界”上成立，但在一些特殊情况下，在较小的“子集”里面反而不成立
  - 比如我们的班级：作为公共选修课，来自不同的院系/年级，建立联系反而困难起来
  - 我们来研究这张“子图”性质，并通过施加最小的影响来改变某些性质。
- 写出自己和**脑海中的**班里另外三个同学的名字
- 把学过的图算法组合起来，尝试解决一些实际问题

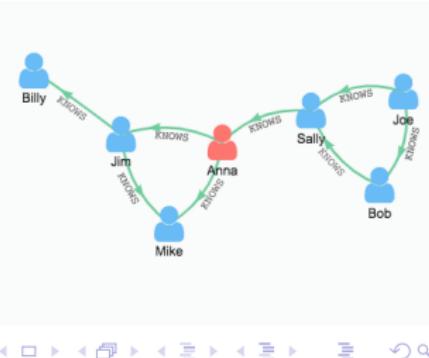
# 目录

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

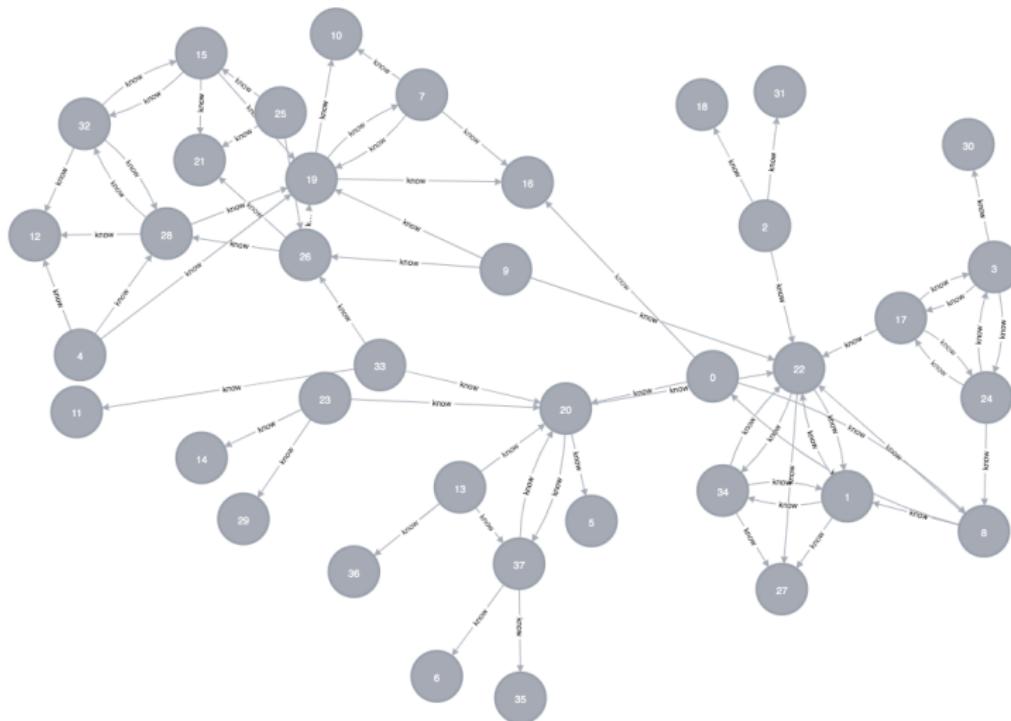


# 有向图的增强

- 问题：给定一有向图  $G$ ，最少添加几条边，使之成为一个强连通图。
  - 这个问题的最早解是由 Eswaran 和 Tarjan 在 1976 年给出的。
  - 1995 年，Frank 将问题推广到  $k$ -强连通，并给出了解。
    - $V$  中的任意  $k$  个顶点被“战争”摧毁，剩下的部分仍然保持强连通
  - 2005 年还有人发表论文，指出 Eswaran 和 Tarjan 的解有点错误，给予了纠正。
  - 论文比较复杂，我们尝试一种综合运用前面的基本图算法，逐步解决实际问题的办法。
- 强连通问题在交通、通讯、控制方面都有着重大的意义
  - 假设“认识”有向边意味着“可以直接传话”，如果要让每个同学都能（直接或间接地）传话给另一个同学（即两个顶点之间存在有向路径），最少需要添加几条边？



## 有向图的增强：班级社会网络图



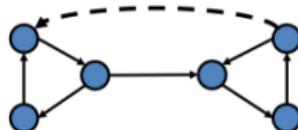
# 有向图的增强：复习一些基本概念

- 有向图  $G=G(V,E)$ ,  $V$  是顶点集合,  $E=\{(u,v)|u,v \in V\}$  是边的集合
- $L=(u_1, u_2, \dots, u_{n-1}, u_n), \forall_{i=1}^{n-1}(u_i, u_{i+1}) \in E$  称为从  $u_1$  到  $u_n$  的有向路径, 也可以用  $u_1 \rightarrow u_n$  表示
- $\forall u, v \in V, u \rightarrow v$ , 称  $G(V,E)$  是强连通的
- 强连通分量 SCC, 最大的  $V' \subset V, \forall u, v \in V', u \rightarrow v$ , 称  $V'$  是  $V$  的 SCC
  - Kosaraju 算法: dfs,  $G_T$ , dfs
- 弱连通性: 参考强连通的定义, 不考虑边的方向性。
  - 路径  $L$  的定义中,  $(u_i, u_{i+1}) \in E$  或者  $(u_{i+1}, u_i) \in E$  都可以
- 顶点的入度:  $G(V,E)$  中  $(u_i, v) \in E$ ,  $u_i$  的个数为  $v$  的入度, 记  $in(v)$
- 顶点的出度:  $G(V,E)$  中  $(v, u_i) \in E$ ,  $u_i$  的个数为  $v$  的出度, 记  $out(v)$

# 有向图的增强：问题的定义（抽象）与初步分析

- 问题：给定一有向图  $G$ ，如何添加最少的边，将它增广为一个强连通图。
- 为了避免一开始就讨论细枝末节，先假设  $G$  是弱连通的
- 强连通图中，不能存在出度或入度为 0 的顶点
  - $v$  的入度为 0，则  $u \rightarrow v$  不存在； $v$  的出度为 0，则  $v \rightarrow u$  不存在
- 后面的讨论中，用  $p$  表示 0 入度顶点个数， $q$  表示 0 出度顶点个数记： $P=\{v|in(v)=0\}$ ,  $Q=\{v|out(v)=0\}$ ,  $p=|P|$ ,  $q=|Q|$
- 至少需要  $\max\{p,q\}$  条边才能将  $G$  增广为强连通图，必要条件。
  - 新增一条边，最多消除一个 0 入度顶点和一个 0 出度顶点。
- $\max\{p,q\}$  是充分条件吗？

# 有向图的增强： $\max\{p,q\}$ 充分性的一个反例

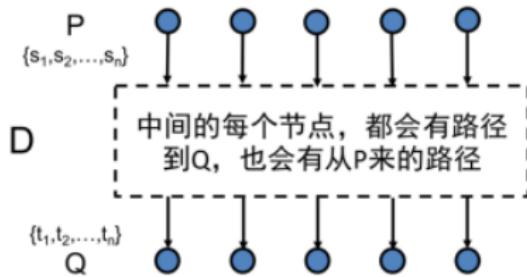


- 上图每一个顶点都在一个环中，于是  $p=q=0$ ，显然最少还需要 1 条边（虚线）才能让它强连通，于是  $\max\{p,q\}$  不是充分条件。
- 由非平凡 SCC（即不是单个顶点）构成的有向图都会有这个问题
- 将一个图 ( $G$ ) 的非平凡 SCC“收缩”成一个顶点，所导致的结果图应该和  $G$  在增广强连通上具有某种等价性
- “收缩” $G$  的强连通分量 SCC 为一个顶点，得到一个 DAG（有向无圈图）
  - 如何证明：增广  $G$  为强连通与增广 DAG 为强连通是等价的？（课后思考）

# 有向图的增强：变为 DAG 之后，重新考虑增广问题

- 若  $G$  是有向无环图 (DAG)，为了将它增广为强连通图，用  $\max\{p,q\}$  条边是充分的吗
- 添加的边用来消除 0 出度顶点和 0 入度顶点，最佳办法就是要跨在它们之间
- 如果  $p \neq q$ ，假设  $p > q$  ( $q > p$  做对等处理)，只要考虑多出来的那些 0 入度顶点怎么办？
- 用  $p-q$  条边在入度为 0 的顶点之间做连接，只要不形成环，我们就得到一个入度为 0 与出度为 0 个数相等（记为  $n$ ）的图，记为  $D$ 。
  - 比方说加  $p-q$  条边，从  $P[0]$  指向  $P[i]|_{i=1}^{p-q}$ ，就消除了  $P[i]|_{i=1}^{p-q}$  共  $p-q$  个 0 入度顶点，0 入度顶点和 0 出度顶点都只剩  $q$  个，并且过程中没有形成环：新增边的起始点  $P[0]$  入度仍然为零。
- 实现了对问题的又一次简化，即只要考虑  $p=q=n$  的情形
- 若能用  $n$  条边解决  $D$  的问题，也就解决了 DAG 的问题。

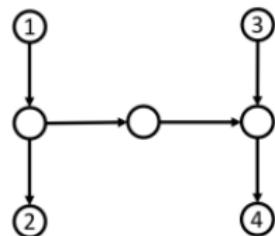
# 有向图的增强：G->DAG->D，一个“规范化”表示的无圈有向图



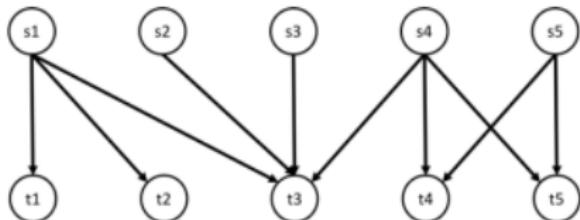
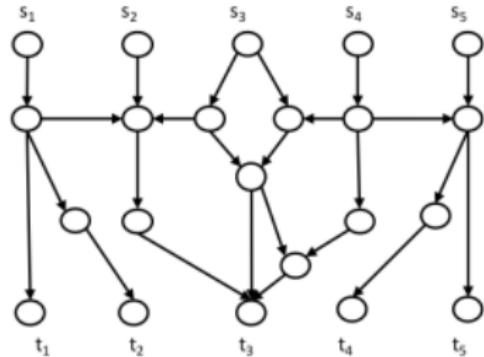
- “中间的每个顶点，都会有路径到 **Q**，也会有从 **P** 来的路径”，证明？(hint:**D** 是无环弱连通)
- 在 **D** 的基础上添加  $n$  条边，让它成为强连通
- 是不是从 **Q** 到 **P**，任意一个无冗余、无遗漏的 1-1 映射就可以了呢？

# 有向图的增强： $Q \rightarrow P$ 不能随意加边

- 如果添加的两条边是  $2 \rightarrow 1$  和  $4 \rightarrow 3$ ，虽然不再有入度为 0 或出度为 0 的顶点了，但结果图显然还不是强连通的
- 若添加的边是  $2 \rightarrow 3$  和  $4 \rightarrow 1$ ，就解决问题了
- 如果添加一些从  $Q$  到  $P$  的边，加上虚线框中的某些从  $P$  到  $Q$  的路径，能保证  $P$  和  $Q$  中的顶点两两之间都有双向路径，则  $D$  中所有顶点两两之间也就都存在双向路径，即成为强连通了



# 有向图的增强：D 变化成二部图 $B_n$

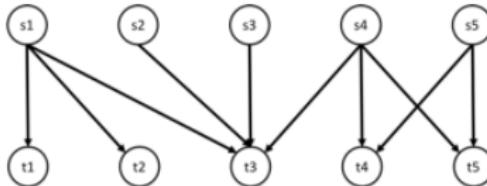


- 二部图定义： $B=B(P,Q,E)$ ,  $\forall(u,v) \in E$  有  $u \in P, v \in Q$
- 完全二部图： $\forall u \in P, v \in Q$  有  $(u,v) \in E$
- $P=\{s_1, s_2, \dots, s_n\}$  和  $Q=\{t_1, t_2, \dots, t_n\}$ ,  $B_n$  中存在边  $(s_i, t_j)$ , 当且仅当在  $D$  中有一条从  $s_i$  到  $t_j$  的路径
- 如何添加  $n$  条边, 让  $B_n$  成为强连通

# 有向图的增强： $B_n$ 的增强

- 是否可以添加 1 条边，将  $B_n$  变成  $B_{n-1}$
- 然后  $B_n \rightarrow B_{n-1} \dots \rightarrow B_k$ ，直到  $B_k$  是一个“完全二部图”
- 对于  $B_k$  的增强就很简单了， $\forall i$  添加  $k$  条边  $(t_i, s_i)$  后  
 $(s_1, t_2, s_2, t_3, \dots, t_k, s_k, t_1, s_1)$  形成一个圈，将所有的顶点串起来（强连通）
- 关键是“如何添加 1 条边，将  $B_n$  变成  $B_{n-1}$ ? ”

# 有向图的增强： $B_n$ 变成 $B_{n-1}$



- 记  $O(s_i) \subset Q$  和  $I(t_j) \subset P$  分别为顶点  $s_i$  和  $t_j$  的邻居集合
- $B_n$  不是完全二部图，所以  $\exists (s_i, t_j) \notin B_n$ ，我们添加一条边  $(t_j, s_i)$
- $B_n$  中入度为 0 和出度为 0 的顶点都少了一个，但它不再是二部图了。
- $B_{n-1}$  除了没有  $s_i$  和  $t_j$ ，应该体现通过新增边  $(t_j, s_i)$  形成的， $P$  和  $Q$  中剩余顶点之间的新路径，就是：

$$B_{n-1} = B_n - V\{s_i, t_j\} + E\{I(t_j) \rightarrow O(s_i)\}$$

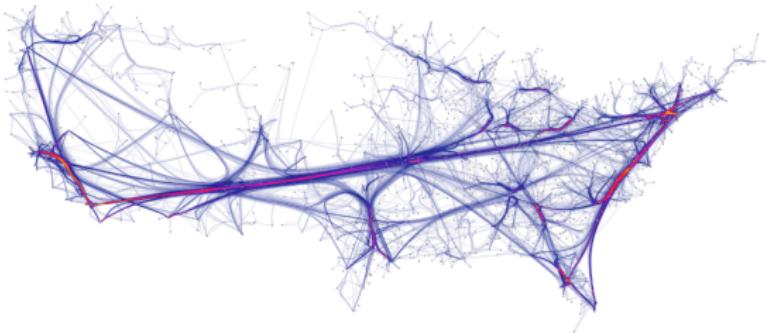
$I(t_j)$  和  $O(s_i)$  中的顶点两两全连接

# 有向图的增强：问题总结

- 去繁化简的思路:  $G \rightarrow DAG \rightarrow D \rightarrow B_n \rightarrow B_{n-1} \dots \rightarrow B_k$
- 完全二部图  $B_k$  总会达到的 (极端情况就是两部各只有一个顶点)
- 若  $DAG$  中有  $\max\{p, q\} = p \geq q = n$ , 这个过程用到的边数为  $(p-q)+(q-k)+k=p=\max\{p,q\}$
- 每一个环节的落实都会涉及到某些具体算法
  - 从  $D$  到  $B_n$  这个环节会用到广度优先搜索,
  - 从  $G$  到  $DAG$  的环节需要检测强连通分量等等。
- 作业：在我们班级的关系图上实现增广强连通图。

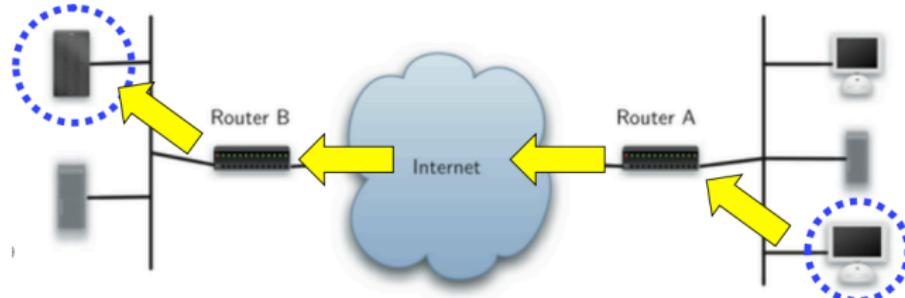
# 目录

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



# 最短路径问题：介绍

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



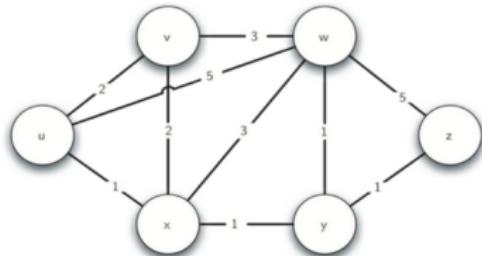
# 最短路径问题：介绍

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

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

# 最短路径问题：介绍

- 所以我们可以将互联网路由器体系表示为一个带权边的图
  - 路由器作为顶点，路由器之间的网络连接作为边
  - 权重可以包括网络连接的速度、网络负载程度、分时段优先级等影响因素
    - 空间距离往往是最不重要的一个因素
  - 作为一个抽象，我们把所有影响因素合成为单一的权重
- 解决信息在路由器网络中选择传播速度最快路径的问题，就转变为在带权图上最短路径的问题。
- 这个问题与广度优先搜索 BFS 算法解决的词梯问题相似，只是在边上增加了权重（如果所有权重相等，则还原到词梯问题）



# 最短路径问题：Dijkstra<sup>2</sup>算法

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

<sup>2</sup>/deikstra/



# 最短路径问题：Dijkstra 算法代码

- course/dijkstra.py

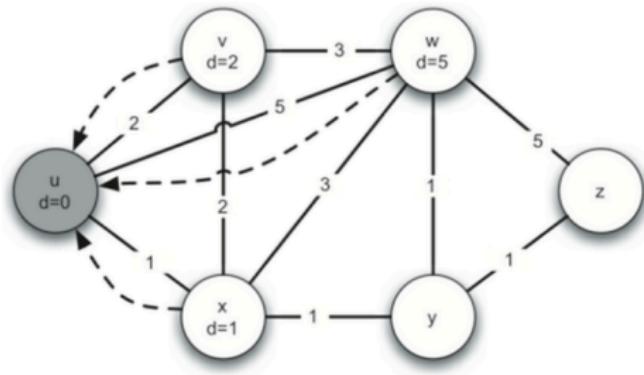
```
1 from pythonds.graphs import PriorityQueue, Graph, Vertex
2 def dijkstra(aGraph, start):
3     pq = PriorityQueue()
4     start.setDistance(0) #对所有顶点建堆，形成优先队列
5     pq.buildHeap([(v.getDistance(), v) for v in aGraph])
6     while not pq.isEmpty():
7         currentVert = pq.delMin() #优先队列出队
8         for nextVert in currentVert.getConnections():
9             newDist = currentVert.getDistance() \
10                + currentVert.getWeight(nextVert)
11                #修改出队顶点所邻顶点dist和前驱节点,
12                #并重排队列
13                if newDist < nextVert.getDistance():
14                    nextVert.setDistance(newDist)
15                    nextVert.setPred(currentVert)
16                    pq.decreaseKey(nextVert, newDist)
```

dijkstra.py

- newDistance 需要重算
- BFS 对比

```
24 def bfs(g, start):
25     start.setDistance(0)
26     start.setPred(None)
27     start.setColor('white')
28     vertQueue = Queue()
29     vertQueue.enqueue(start)
30     while (vertQueue.size() > 0):
31         currentVert = vertQueue.dequeue()
32         for nbr in currentVert.getConnections():
33             if (nbr.getColor() == 'white'):
34                 nbr.setColor('gray')
35                 nbr.setDistance(currentVert.getDistance() + 1)
36                 nbr.setPred(currentVert)
37                 vertQueue.enqueue(nbr)
38         currentVert.setColor('black')
39
40 def traverse(v):
    wordLadder.py [+]
```

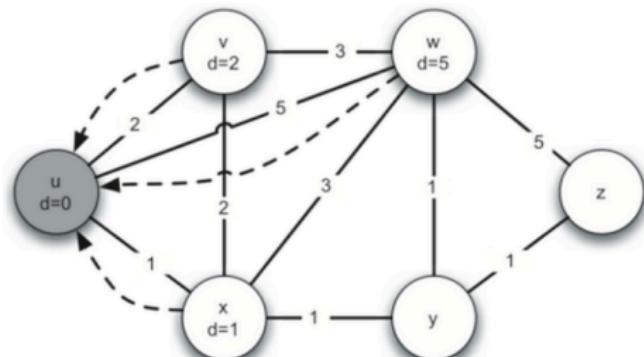
# 最短路径问题：Dijkstra 算法示例



$PQ = x, v, w$

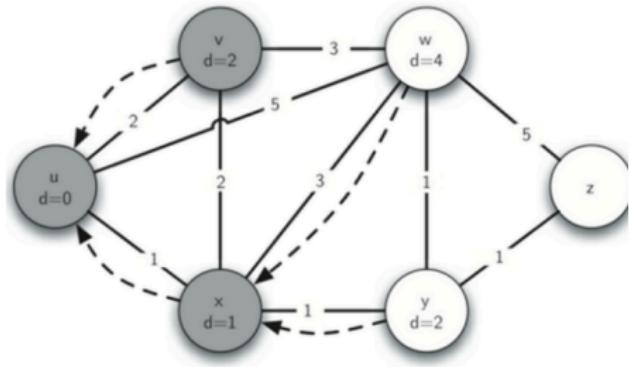
- 起始顶点  $u$  出队，重算  $xvw$

- $x$  出队，重算  $uvw$



$PQ = x, v, w$

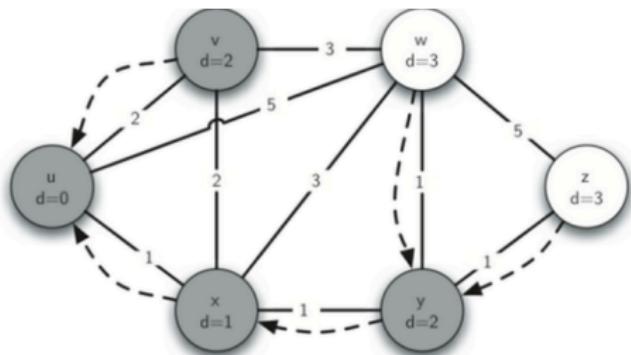
# 最短路径问题：Dijkstra 算法示例



$$PQ = yw$$

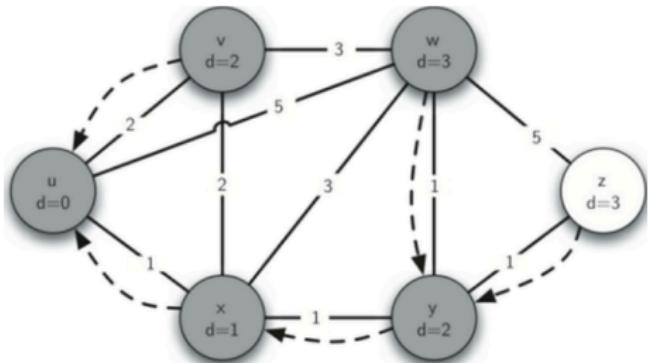
• v 出队，重算uxw

• y 出队，重算xwz



$$PQ = wz$$

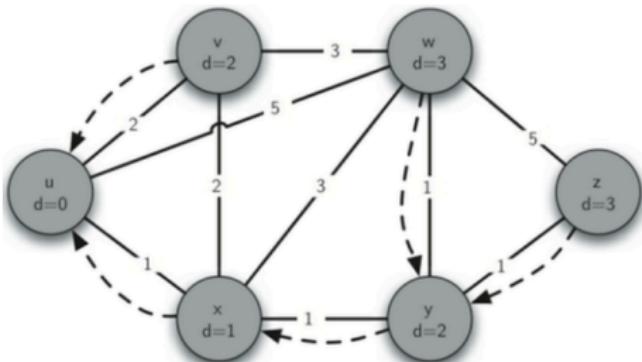
# 最短路径问题：Dijkstra 算法示例



$PQ = z$

- w 出队，重算 $vyz$

- z 出队，重算 $wy$



$PQ = \text{None}$

# 二叉堆与优先队列

- Dijkstra 中用到的 PriorityQueue 与 BinaryHeap 略有不同
- 多了一个 decreaseKey 方法，调整堆中元素 key 值
- Python 库中的 heapq 或 PriorityQueue 中无此“重要”功能
- 难点：堆操作连带改变“所有”数据项的位置
  - Miller 的实现：顺序查找 cry!!

```
73     def decreaseKey(self, val, amt):
74         # this is a little wierd, but we need to find the heap thing to decrease by
75         # looking at its value
76         done = False
77         i = 1
78         myKey = 0
79         while not done and i <= self.currentSize:
80             if self.heapArray[i][1] == val:
81                 done = True
82                 myKey = i
83             else:
84                 i = i + 1
85         if myKey > 0:
86             self.heapArray[myKey] = (amt, self.heapArray[myKey][1])
87             self.percUp(myKey)
```

Figure: <https://github.com/bnmnntp/pythonds.git, graphs/priorityQueue.py>

# 如何调整二叉堆中元素 key 值

- 调整 key 值：删除再重装；删除：标记为“REMOVED”
- 用字典记录任务在二叉堆中的数据项；
- 在优先级之后插入序列号，避免了对 task 比较小的尴尬
  - 副产品：得到了一种使所有排序算法稳定的通用办法

```
6     entry_finder = {}          # mapping of tasks to entries
7     REMOVED = '<removed-task>' # placeholder for a removed task
8     counter = itertools.count() # unique sequence count
9
10    def add_task(task, priority=0):
11        'Add a new task or update the priority of an existing task'
12        if task in entry_finder:
13            remove_task(task)
14        entry = [priority, next(counter), task]
15        entry_finder[task] = entry
16        pq.insert(entry)
17
18    def remove_task(task):
19        'Mark an existing task as REMOVED. Raise KeyError if not found.'
20        entry = entry_finder.pop(task)
21        entry[-1] = REMOVED
22
23    def pop_task():
24        'Remove and return the lowest priority task. Raise KeyError if empty.'
25        while not pq.isEmpty():
26            priority, count, task = pq.delMin()
27            if task is not REMOVED:
28                del entry_finder[task]
29                return task
30        raise KeyError('pop from an empty priority queue')
```

Figure: <https://github.com:dsaClass/dsa2020.git, course/queue.py>

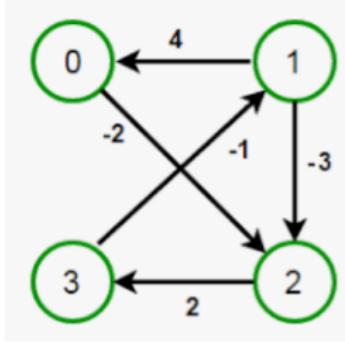
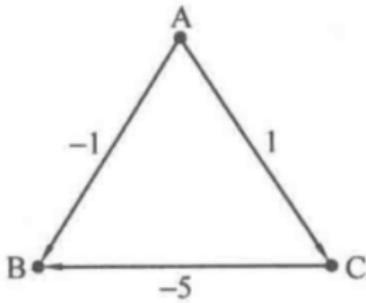
# Dijkstra 算法实际应用：城市导航

250606:Subway openjudge/1504

- 抽象：城市看成一张图；
  - 家、学校和地铁站是图中的顶点；
  - 两个顶点之间的路线是图中的边
- 按需求快速的实现一个图 graph，它的精髓在哪儿？
  - 就用一个字典来表示：存储顶点和顶点的 key 值；这里用顶点的坐标做 key 值；graph 类中的其他功能用不着，只需要主列表就可以了
  - 顶点需要用一个类来实现，记录顶点的坐标、通勤时间和出边表
  - 边的权重是两点之间的通行时间
- 先建图 buildGraph；然后 Dijkstra 找最短路径（最短通勤时间）

# Dijkstra 算法：负权边

- 需要注意的是，Dijkstra 算法只能处理大于 0 的权重，如果图中出现负值边，则算法失效。
  - “没有负权边”保证了前面红色的重算，不会产生任何效果。
  - 以 A 为起始点，到 B 的最短路径应该是  $A \rightarrow C \rightarrow B$ ，而不是  $A \rightarrow B$
  - Bellman-Ford 算法可以处理负权边的情况，但算法效率低一些。
    - 重复  $|V|-1$  次：对每条边进行松弛；复杂度  $O(|V| * |E|)$
    - 但图中如果出现从起始点可达的负权回路，则问题无解。
    - 通过反复走这条回路，从起始点出来的路径长度可以无限小。



- 虽然 Dijkstra 算法完美解决了带权图的最短路径问题，但实际上 Internet 的路由器中采用的是其它算法，Dijkstra 算法广泛应用于导航、物流配送、机器人避障等领域
- 其中最重要的原因是，Dijkstra 算法需要具备整个图的数据，但对于 Internet 的路由器来说，显然无法将整个 Internet 所有路由器及其连接信息保存在本地，这不仅是数据量的问题，Internet 动态变化的特性也使得保存全图缺乏现实性。
- 路由器的选径算法（或“路由算法”）对于互联网极其重要，有兴趣可以进一步参考“距离向量路由算法”。
  - <http://baike.baidu.com/view/1227645.htm>

# 最短路径问题：Dijkstra 算法分析

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

# 目录

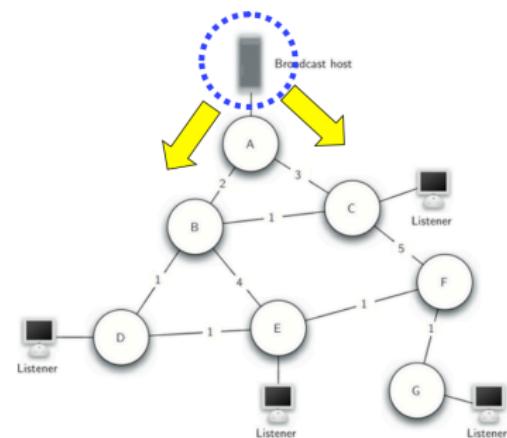
- 本章目标
- 图抽象数据类型及实现
- Word Ladder 词梯问题
- 骑士周游问题
- 拓扑排序和强连通分支
- 最短路径问题
- Prim 最小生成树算法



# 最小生成树 (Minimum Spanning Trees)

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

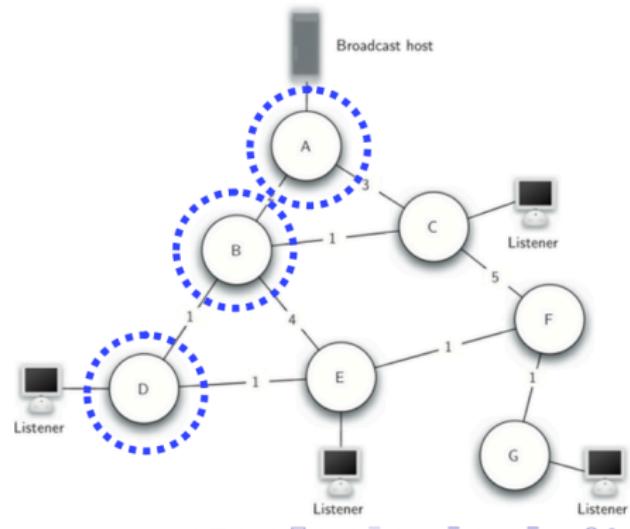
- 网游需要让所有玩家获知其他玩家所在的位置
- 收音机则需要让所有收听用户获取直播的音频数据



# 信息广播问题：单播解法（点对点）

- 信息广播问题最简单的解法是由广播源维护一个收听者的列表，将每条消息向每个收听者发送一次
  - 如图，一个信息源，四个收听者
  - 每条消息会被发送 4 次，每个消息都采用最短路径算法到达收听者
  - 可以把消息想象成快递，边的权重就是快递通过的价格（转运模式）

- $A \rightarrow C: 3$
- $A \rightarrow D: 2+1$
- $A \rightarrow E: 2+1+1$
- $A \rightarrow G: 2+1+1+1$
- 总费用： $3 + (2+1) + (2+1+1) + (2+1+1+1) = 16$

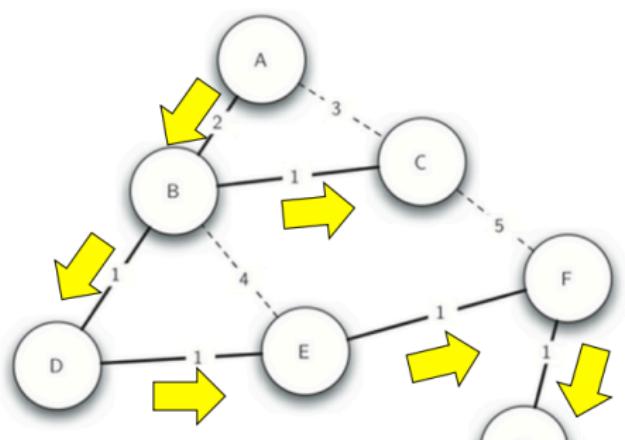


# 信息广播问题：洪水解法

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

# 信息广播问题：最小生成树

- 信息广播问题的最优解法，依赖于路由器关系图上选取具有最小权重的生成树（minimum weight spanning tree）
  - 树的另一个定义：无圈（环）连通图<sup>3</sup>
  - 生成树：如果连通图 G 的一个子图是一棵包含 G 的所有顶点的树，则该子图称为 G 的生成树 (SpanningTree)。它拥有图上所有的 n 个顶点和 n-1 条边。
- 连通无向图  $G(V,E)$  的最小生成树 T，定义为连通子图  $T=T(V,E')$ ，满足  $E' \subset E$ ，且  $E'$  中边的权重之和最小。
- 图为一个最小生成树
  - 这样需要广播的信息就只需要从 A 开始
  - 沿着树的路径层次向下传播
  - 就可以达到每个路由器只需要处理 1 次消息，
  - 消息的传输有成本，消息的复制没有成本

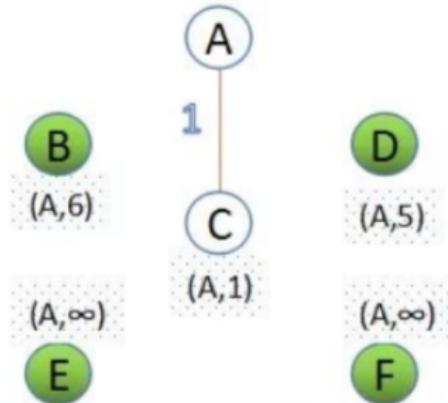
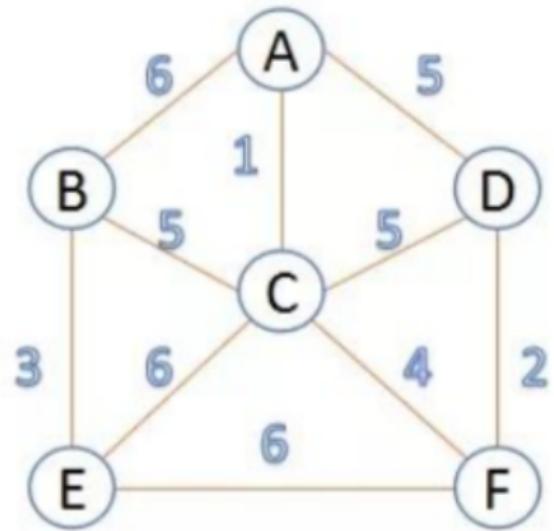


# 最小生成树：Prim 算法

- 解决最小生成树问题的 Prim 算法，属于“贪心算法”，即每步都沿着最小权重的边向前搜索。
- 构造生成树的思路如下：
  - 如果  $T$  还不是生成树，则反复做：
    - 找到一条可以安全添加到树  $T$  的边
    - 将边添加到树  $T$
- “可以安全添加”的边，定义为一端顶点在树中，另一端不在树中的边，以便保持树的无圈特性
- 如果把安全边的定义修改为“两端不在同一个连通分量中的边”，就得到第二种最小生成树 Kruskal<sup>4</sup> 算法，同样保持了树的无圈特性。
- 一句话概括算法核心：找“最小权重安全边”

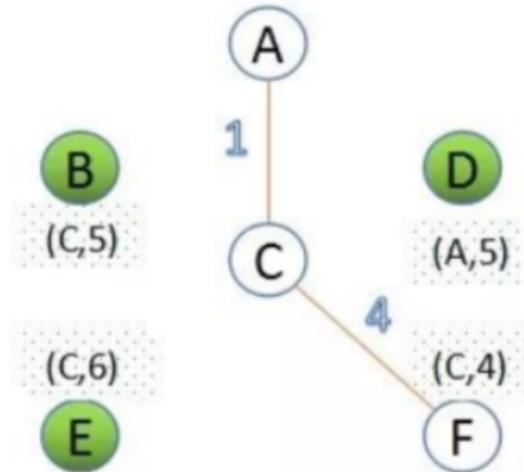
<sup>4</sup>克鲁斯克尔

# 最小生成树：Prim 算法示例

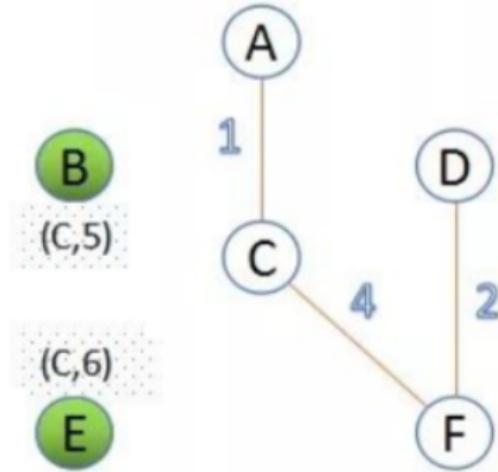


1. 初始  $u=\{A\}, v=\{B,C,D,E,F\}$ ; 顶点B下方(A,6), 表示与集合u中A的代价为6作为最小代价边。选择最小的代价边(A,C), 把C并入到集合u中。

# 最小生成树：Prim 算法示例

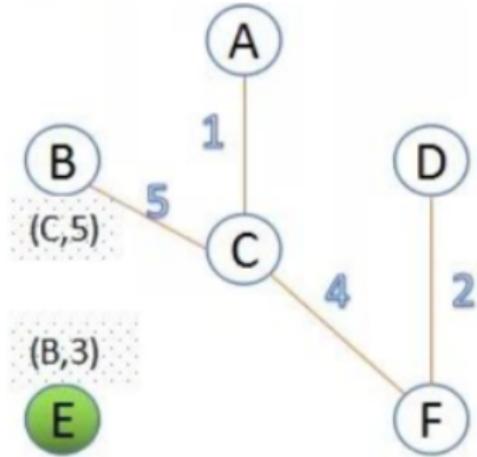


2.  $u=\{A,C\}$ ,  $v=\{B,D,E,F\}$ ; 更新  $v$  中顶点与集合  $u$  的最小的代价边; 例如: 顶点 E 之前为  $(A, \infty)$ , 更新为  $(C, 6)$ ; 选择最小代价边  $(C, F)$ , F 并入  $u$ .

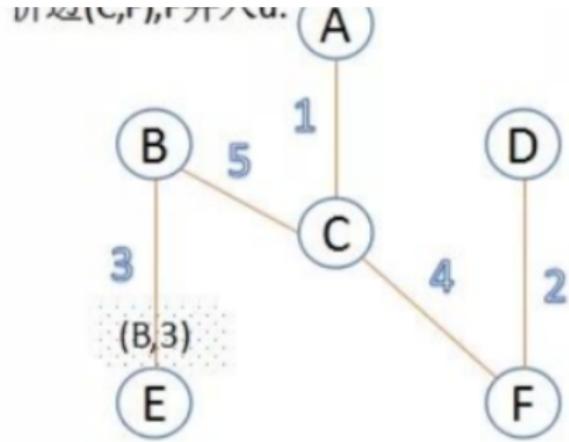


3.  $u=\{A,C,F\}$ ,  $v=\{B,D,E\}$ ; 更新  $v$  中顶点与集合  $u$  的最小的代价边; 选择最小代价边  $(F, D)$ , D 并入  $u$ .

# 最小生成树：Prim 算法示例



4.  $u=\{A,C,F,D\}$ ,  $v=\{B,E\}$ ; 更新  $v$  中顶点与集合  $u$  的最小的代价边; 选择最小代价边(C,B), B并入u.



5.  $u=\{A,C,F,D,B\}$ ,  $v=\{E\}$ ; 更新  $v$  中顶点与集合  $u$  的最小的代价边; 选择最小代价边(B,E), E并入u.

# 最小生成树：Prim 算法数学表述

- $G = G(V, E)$  是一个带权连通图，  
 $E = \{(u, v, weight)\}, u, v \in V, weight > 0$
- 最小生成树  $T = T(V, E')$ , 满足条件:  $T$  是一颗树;  $E' \subset E$ , 且

$$\operatorname{ArgMin} \sum_{e \in E'} e.weight$$

- Prim 算法从起始顶点  $v_1 \in V$  开始产生树的队列 (也称树的生长)

$$T_i = T_i(V_i, E_i), 1 \leq i \leq n = |V|$$

- 生长过程满足  $V_1 = \{v_1\}$ ,  $E_1 = \emptyset$ ,  $V_n = V$ , 中间每一步都满足:  
 $E_{i+1} = E_i + \{e\}$ ,  $e = (u, v, weight) \in E$ ,  $u \in V_i$ ,  $v \in V - V_i$ ,  
称  $e$  为安全边

- 贪心法: 对于上面安全边  $e=(u,v,weight)$ , 找最小 weight 的  $e$

- $(v.distance)_i = \min_{u \in V_i} ((u, v).weight)$

- $\min_i(e.weight) = \min_{v \in V - V_i} (v.distance)_i$

# 最小生成树：Prim 算法代码

```
3 def prim(G,start):
4     pq = PriorityQueue()
5     for v in G:
6         v.setDistance(sys.maxsize)
7         v.setPred(None)
8     start.setDistance(0)
9     pq.buildHeap([(v.getDistance(),v) for v in G])
10    while not pq.isEmpty():
11        currentVert = pq.delMin()
12        for nbr in currentVert.getConnections():
13            """ 原来的distance表示顶点与起始顶点的距离
14            newCost = currentVert.getWeight(nbr) \
15                  + currentVert.getDistance()
16            """
17            #现在的distance表示顶点与当前生成树的最小距离
18            newCost = currentVert.getWeight(nbr)
19            if nbr in pq and newCost<nbr.getDistance():
20                nbr.setPred(currentVert)
21                nbr.setDistance(newCost)
22                pq.decreaseKey(nbr,newCost)
```

- $pg$  中存放的顶点集合为  $V - V_i$
- 优先队列  $pg$  中顶点按  $distance$  排序，取出最小顶点  $currentVert$
- $V_{i+1} = V_i + \{currentVert\}$
- 更新  $pg$  中  $currentVert$  邻居的  $distance$

$$\begin{aligned}(v.distance)_{i+1} &= \min_{u \in V_{i+1}} ((u, v).weight) \\&= \min((v.distance)_i, (currentVert, v).weight)\end{aligned}$$

# 一些经典 NPC 的图问题

- 哈密顿路径问题
- 行商问题（最短路径问题）
  - 给定一系列城市和每对城市之间的距离，求解访问每一座城市一次并回到起始城市的最短回路。它是组合优化中的一个 NP 困难问题，在运筹学和理论计算机科学中非常重要。
- 子图同构 (Graph Isomorphism)
- 分团问题，列举图中所有  $k$  个点的子集合
- 图的顶点覆盖
- 图着色问题

- 本章我们学习了图抽象数据类型，以及若干实现方法
- 本章讨论了一些图的算法和应用
  - 广度优先搜索算法 **BFS**，解决无权图的最短路径问题；
  - 带权图的 **Dijkstra** 算法；
  - 图的深度优先搜索算法；
  - 用于简化图的强连通分支算法；
  - 用于关联任务排序的拓扑排序算法；
  - 用于广播消息的最小生成树算法。