## 2.8.1 复杂网络（图论）基础与NetworkX

复杂网络（图，graph）的研究已经成为涉及物理学、数学、生物学、社会科学、信息学以及其它理论和应用科学的多学科研究的重要领域。这一领域的重要性在于存在一种统一的语言来描述在现代社会中具有重大相关性的不同的现实世界系统，从互联网或电网到代谢或蛋白质相互作用的网络[1]。复杂网络也成为城市空间数据分析中解决城市问题的重要工具，例如城市街道（交通）系统，斑廊基景观生态，业态分布网络等。图的研究自从18世纪上半叶诞生以来，到19世纪下半叶已经发展成为数学一个系统的分支，至20世纪开始迅速发展，开发了大量算法、属性识别并定义数学模型以更好的理解复杂网络。为了避免对大量繁复算法的直接计算，[NetworkX](https://networkx.org/documentation/stable/index.html)①库定义了大量函数方法再现了许多图的算法，可以直接调用用于实际相关的研究问题中。对于图的解释也以NetworkX库为工具来说明，不仅可以图示复杂网络，也可以再现复杂算法。

对于复杂网络的解释会涉及到很多名词定义（术语）、数学符号表达，这里以Reihhard Diestel的《图论》[2]，结合NetworkX库作为复杂网络解释的参考标准，并适当的引入其它论文专著的表述。

### 2.8.1.1 图[2]

用表示包括零在内的自然数的集合。模整数集  
Z/*n*Z Z*n :* * * F2​={0,1}*x*  
⌊*x*⌋*x*  
⌈*x*⌉*xloglnA*iAi≠ Ø（空集）** AA{ A’ *{1}, A’* {2} , , A’ *{ } }A’* {i}A\_{j}A’A[A] kAkk个元素的子集称为**k-子集（k-subset）**。

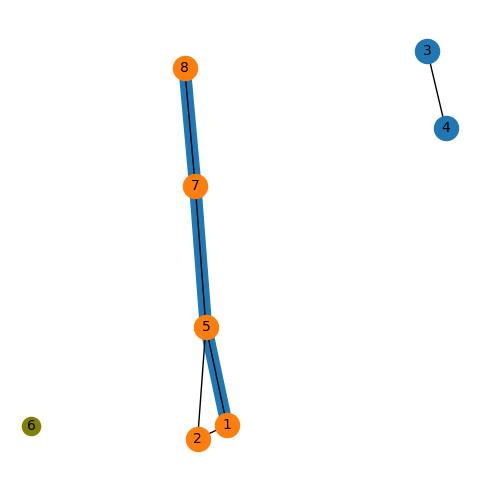
**图（graph）**是一个二元组使得 EVV E=Ø VGE的元素称为**边（edge）**(或**线（line）**)。通常，描绘一个图的方法是把顶点画成一个小圆圈，如果相应的顶点之间有一条边，就用一条线连接这两个小圆圈。如何绘制这些小圆圈和连线时无关紧要的，重要的是要正确体现哪些顶点对之间有边，哪些顶点对之间没有边。

为了方便绘制复杂网络，定义G\_drawing()函数，可以配置是否显示顶点属性或边属性，及配置图的样式，可以调整顶点的大小和颜色，字体的大小和颜色，及顶点位置、边宽度、图大小等样式。

def G\_drawing(G,edge\_labels=None,node\_labels=None,routes=[],nodes=[],\*\*kwargs):  
 '''  
 绘制复杂网络  
  
 Parameters  
 ----------  
 G : networkx.classes.graph.Graph  
 复杂网络（图）.  
 edge\_labels : string, optional  
 显示边属性. The default is None.  
 node\_labels : string, optional  
 显示节点属性. The default is None.  
 routes : list(G vertex), optional  
 构成图路径的顶点. The default is None.   
 nodes : list(G vertex), optional  
 顶点的嵌套列表，用于不同顶点集的不同显示（颜色和大小等）. The default is None.   
 \*\*kwargs : kwargs  
 图表样式参数，包括options和sytle，默认值为：  
 options={  
 "font\_size": 20,  
 "font\_color":"black",  
 "node\_size": 150,  
 "node\_color": "olive",  
 "edgecolors": "olive",  
 "linewidths": 7,  
 "width": 1,  
 "with\_labels":True,   
 }  
 style={  
 "figsize":(3,3),   
 "tight\_layout":True,  
 "pos\_func":nx.spring\_layout,  
 "edge\_label\_font\_size":10,  
 "pos":None  
 }.  
  
 Returns  
 -------  
 None.  
  
 '''   
 import matplotlib.pyplot as plt  
 import networkx as nx  
 import matplotlib.colors as mcolors  
 import random  
 from pylab import mpl  
 plt.rc('axes', unicode\_minus=False) # 解决图表负号不正确显示问题  
 mpl.rcParams['font.sans-serif']=['SimHei'] # 解决中文字符乱码问题  
  
 def generate\_color():  
 color = '#{:02x}{:02x}{:02x}'.format(\*map(lambda x: random.randint(0, 255), range(3)))  
 return color  
   
 options={  
 "font\_size": 20,  
 "font\_color":"black",  
 "node\_size": 150,  
 "node\_color": "olive",  
 "edgecolors": "olive",  
 "linewidths": 7,  
 "width": 1,  
 "with\_labels":True,   
 "cmap":None,  
 }  
 options.update((k, kwargs[k]) for k in set(kwargs).intersection(options))  
   
 style={  
 "figsize":(3,3),   
 "tight\_layout":True,  
 "pos\_func":nx.spring\_layout,  
 "edge\_label\_font\_size":10,  
 "pos":None,  
 "edge\_colors":list(mcolors.TABLEAU\_COLORS.values()),  
 "edge\_widths":[3]\*len(routes),  
 "title":None,  
 "nodes\_size":[200]\*len(nodes),  
 "nodes\_color":[generate\_color() for i in range(len(nodes))]#list(mcolors.TABLEAU\_COLORS.values()),  
 }  
   
 style.update((k, kwargs[k]) for k in set(kwargs).intersection(style))   
 fig,ax=plt.subplots(figsize=style['figsize'],tight\_layout=style["tight\_layout"])   
   
 if style['pos']:  
 pos=style['pos']  
 else:  
 pos=list(map(style["pos\_func"],[G]))[0]   
   
 if routes:  
 route\_edges=[[(r[n],r[n+1]) for n in range(len(r)-1)] for r in routes]  
 [nx.draw\_networkx\_edges(G,pos=pos,edgelist=edgelist,edge\_color=style['edge\_colors'][idx],width=style['edge\_widths'][idx],) for idx,edgelist in enumerate(route\_edges)]   
  
   
 if node\_labels:  
 options["with\_labels"]=False  
 nx.draw(G, pos=pos,ax=ax,\*\*options)  
 node\_labels=nx.get\_node\_attributes(G,node\_labels)  
 nx.draw\_networkx\_labels(G, pos, labels=node\_labels,ax=ax)  
 else:  
 nx.draw(G, pos=pos,ax=ax,\*\*options)   
   
 if edge\_labels:  
 edge\_labels=nx.get\_edge\_attributes(G,edge\_labels)  
 nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels,ax=ax,font\_size=style["edge\_label\_font\_size"])   
   
 if nodes:  
 [nx.draw\_networkx\_nodes(G,pos=pos,nodelist=sub\_nodes,node\_size=style['nodes\_size'][idx],node\_color=style['nodes\_color'][idx]) for idx,sub\_nodes in enumerate(nodes)]   
   
 plt.title(style['title'])  
 plt.show()

用NetworkX库的nx.Graph()方法定义空的图G，这里通过add\_edges\_from增加边（顶点对）的方式和add\_node增加点的方式定义图G。其顶点集为，边集为的图。

import networkx as nx  
G=nx.Graph()  
G.add\_edges\_from([(1,2),(1,5),(2,5),(3,4),(5,7),(7,8)])  
G.add\_node(6)  
G\_drawing(G,node\_size=50,font\_size=10,nodes=[[3,4],[1,2,5,7,8]],nodes\_size=[300,300],figsize=(5,5),routes=[[1,5,7,8]],edge\_widths=[9])



具有顶点集的图亦称为**V上的图**（**a graph on V**）。图G的顶点集记为，边集记为。这些约定俗成与这两个集合的记法是独立的：图的顶点集仍记为，而不是。通常并不把一个图和它的顶点集或边集严格的区分开来，例如称一个顶点（而不是），一条边等。

一个图的顶点个数称为图的**阶（order）**，记为$GG $。根据图的阶，把图分为**有限的（finite）**、**无限的（infinite）**或**可数的（countable）**等。一般假定图均为有限的。

如下代码，NetworkX库提供了获取图顶点G.nodes、顶点数G.number\_of\_nodes()和边G.edges、边数G.number\_of\_edges()的方法。

print(G.nodes)  
print(G.number\_of\_nodes())  
print(G.edges)  
print(G.number\_of\_edges())

[1, 2, 5, 3, 4, 7, 6]  
7  
[(1, 2), (1, 5), (2, 5), (5, 7), (3, 4)]  
5

对于**空图（empty graph）**，记作Ø。阶为0或1的图称为**平凡的（trivial）**。有时平凡图会很有用，例如使用归纳法时；而在其它一些情景中，平凡图则无意义，因此避免非平凡性条件的假设，省略对平凡图和空图的讨论。

给定顶点和边，如果，则称与**关联（incident）**，从而是**在（at）**的边。关联同一条边的两个顶点称为这条边的**端点（endvertex）**或**顶端（end）**，而这条边**连接（joint）**它的两个端点。边通常记为（或）。如果且，则称边为**一条X-Y边（X-Y edge）**；集合中所有边的集合，记为；而和会记为和。中所有和顶点关联的边记为。

如果是的一条边，则称两个顶点和是**相邻的（adjacent）**或**邻点（neighbor）**；如果两条边有一个公共端点，则称和是**相邻的**。若的所有顶点都是两列相邻的，则称是**完全的（complete）**。个顶点的完全图记为 称为**三角形（triangle）**。

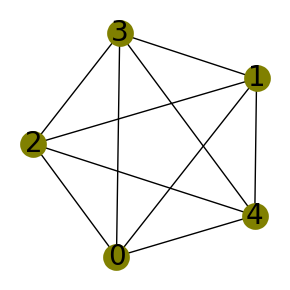
用G.neighbors或nx.all\_neighbors获取给定顶点的邻点；用G.edges(node)可以提取给定公共端点的所有边。

print(list(G.neighbors(5)))  
print(list(nx.all\_neighbors(G,5)))  
print(G.edges(5))

[1, 2, 7]  
[1, 2, 7]  
[(5, 1), (5, 2), (5, 7)]

用nx.complete\_graph方法可以生成给定阶的完全图。

G=nx.complete\_graph(5)  
G\_drawing(G)

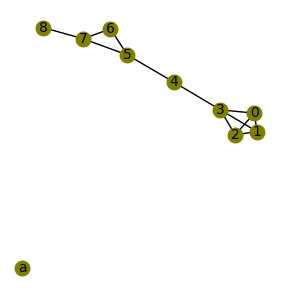


互不相邻的顶点或边称为**独立顶点**或**独立边（independent vertex/edge）**，更正式地，若一个顶点集或边集中没有两个元素是相邻的，则该集合称为**独立集（independent set）**；独立的顶点集也称为**稳定集（stable set）**。

maximal\_independent\_set(G, nodes=None, seed=None)：返回图的随机最大独立集。可以通过nodes参数指定必须包含的顶点列表。最大独立集是一个独立集，因此不可能添加一个新的节点，仍然可以得到一个独立集。

maximum\_independent\_set(G)[3]：返回一个近似的最大独立集。

G=nx.lollipop\_graph(4, 3)  
G.add\_edges\_from([(6,7),(5,7),(7,8)])  
G.add\_node("a")  
G\_drawing(G,node\_size=20,font\_size=10)  
print(nx.maximal\_independent\_set(G))  
print(nx.maximal\_independent\_set(G, [1]))  
  
from networkx.algorithms.approximation.clique import maximum\_independent\_set  
print(maximum\_independent\_set(G))



[1, 5, 'a', 8]  
[1, 8, 6, 'a', 4]  
{0, 4, 6, 8, 'a'}

设和G’=( V’, E’ ) GG’: V⟶V′  
{x,y}∈E{φ(x),φ(y)}∈E′φφx’ Gφ  
x,y∈V，有xy∈E⇔φ(x)φ(y)∈E′GG’G≃G’ G

G=G’G≃ G’。如果强调只对给定图的同构类感兴趣，会非正式的称它为抽象图（abstract graph）。

在同构意义下封闭的图族叫做**图性质（graph property）**。例如“包含三角形”就是一个图性质：如果包含三个两两相邻的顶点，则每个同构于的图亦有此性质。对于图上的一个映射，如果对于每个同构图它均取相同的值，则这样的映射称为一个图不变量（graph invvariant）。一个图的顶点数和边数就是两个简单的图不变量；图中两两相邻的最大顶点数也是**图不变量**。

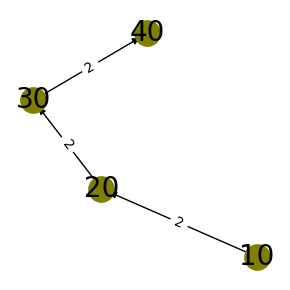
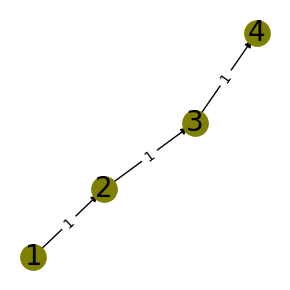
is\_isomorphic(G1, G2, node\_match=None, edge\_match=None)[4]：如果图G1和G2同构，则返回true，否则返回false。如果需要考虑顶点和边属性则可以配置node\_match和edge\_match参数。

numerical\_edge\_match(attr, default, rtol=1e-05, atol=1e-08)：返回数值边缘属性的比较函数。属性值必须为数值或可排序的对象，如果G1和G2排序后的值列表相同或者在一个允许的范围内则返回True。

numerical\_node\_match(attr, default, rtol=1e-05, atol=1e-08)：返回数值顶点属性的比较函数。

could\_be\_isomorphic(G1, G2)，fast\_could\_be\_isomorphic(G1, G2)和faster\_could\_be\_isomorphic(G1, G2)：如果图绝对不是同构的，则返回false；不能保证True为同构。

import networkx.algorithms.isomorphism as iso  
  
G1=nx.DiGraph()  
G2=nx.DiGraph()  
nx.add\_path(G1, [1, 2, 3, 4], weight=1)  
nx.add\_path(G2, [10, 20, 30, 40], weight=2)  
G\_drawing(G1,edge\_labels='weight')  
G\_drawing(G2,edge\_labels='weight')  
em=iso.numerical\_edge\_match("weight", 1)  
print(em)  
print(nx.is\_isomorphic(G1, G2)) # no weights considered  
print(nx.is\_isomorphic(G1, G2, edge\_match=em)) # match weights



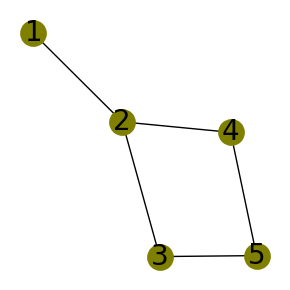
<function numerical\_node\_match.<locals>.match at 0x00000251FA0B9430>  
True  
False

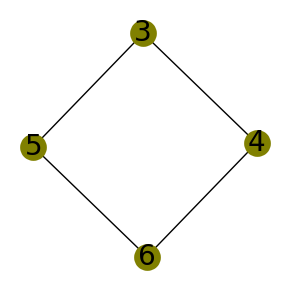
记及，若G∩ G’=Ø GG’V’ V E’ E G’GGG’G’ G GG’ G’ G G’G G’ G$的**真子图（proper subgraph）**。

若$G’ G G’ Ex,y V’ xyG’ GV’ GG’ G’:=G[ V’ ] U VG[U]UGUHGG[V[H]]G[H]V’GV’=V G’ G$的一个**支撑子图（spanning subgraph）**。

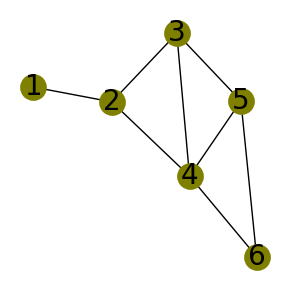
如下代码定义图和，调用compose方法实现图的并；R.remove\_nodes\_from(n for n in G if n in H)方法实现图的差（对于提供的difference(G, H)方法会提示错误Node sets of graphs not equal）；用intersection实现图的交。顶点2,3,4在G∪ G’ G中则不导出三角形。

G=nx.Graph()  
G.add\_edges\_from([(1,2),(2,3),(2,4),(3,5),(4,5)])  
G\_drawing(G)  
H=nx.Graph()  
H.add\_edges\_from([(3,4),(3,5),(4,6),(5,6)])  
G\_drawing(H)

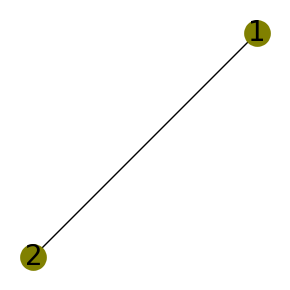




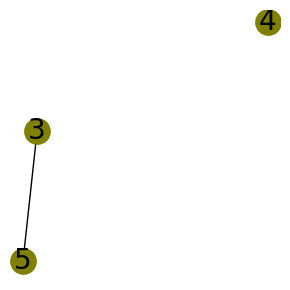
U=nx.compose(G, H)  
G\_drawing(U)



R=G.copy()  
R.remove\_nodes\_from(n for n in G if n in H)  
G\_drawing(R)



I=nx.intersection(G, H)  
G\_drawing(I)



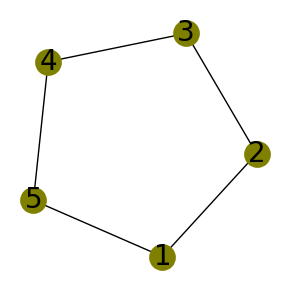
设是的任意一个顶点集合，把简记为，即是从中删除中所有顶点及相关联的边得到的图。如果是个单点集，把简记为；而把简单记作G- G’ [V]2 FG-F:=(V,E \F)G+F:=(V,E ∪F)G-{e}G+{e}G-eG+eG(V,F)F⊋ EG关于此性质是边极大的（edge-maxmal）。

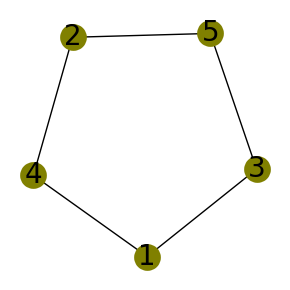
更一般的，当称一个图对于某性质是**极大的（maximal）**或**极小的（minimal）**，但没有强调具体的序关系时，均指子图关系。 当提到顶点集或边集的极大性或极小性时，均指集合的包含关系。

如果和是不交的，那么表示在G∪ G’ GG’ GV[V]2\E GGGL(G)EEx,y ∈EG中是相邻的。

下述代码定义了一个图，使用complement方法计算补图，从结果可知补图的边不在图中存在。

G=nx.Graph([(1, 2), (2, 3), (3, 4), (4, 5),(1,5)])  
G\_drawing(G)  
G\_complement=nx.complement(G)  
G\_drawing(G\_complement)  
G\_complement.edges() # This shows the edges of the complemented graph

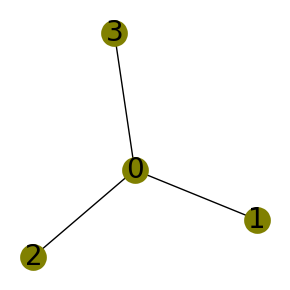


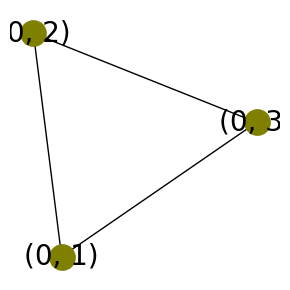


EdgeView([(1, 3), (1, 4), (2, 4), (2, 5), (3, 5)])

下述代码定义了一个图𝐺 ，使用line\_graph方法计算线图，将中的每条边抽象成一个顶点，如若原图中两条边相邻，那么就给线图中对应顶点之间连接一条边。因为线图将原图的边化作了顶点，所以也可以将其视作原图的一种对偶。线图的顶点保留了原图边的信息，例如顶点表示原图中边.

G=nx.star\_graph(3)  
G\_drawing(G)  
L=nx.line\_graph(G)  
G\_drawing(L)





### 2.8.1.2 顶点度

设是一个非空图，中的顶点的邻点集记为  
NG​(v)N(v)U ⊆VUV UUN(U)。

顶点的**度（degree）**（或**价（valenccy）**）v| E(v) | vGδ(G):=min{d(v)∣v∈V} △(G):=max{d(v)∣v∈V}GkG是k-正则的(k-regular)，或简称正则的（regular）。3-正则图亦称立方图（cubic graph）。

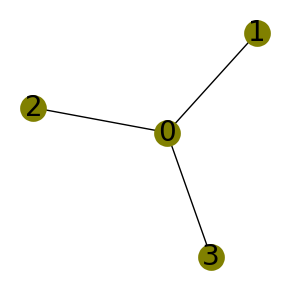
图的平均度（average degree）定义为：  。

顶点度是连接每个顶点的边数，它是一个局部参数，而平均度则是一个整体性的度量，有时，可以方便的把这个比率记为。

当然，和这两个量是密切相关的，如果对中所有顶点度求和，那么每条边恰被计算两次，即每个端点计算一次，所以。

G.degree方法可以计算返回所有顶点的度。

from statistics import mean  
G=nx.star\_graph(3)  
G\_drawing(G)  
print( G.degree)  
max\_degree=max(deg for n, deg in G.degree)  
print(max\_degree)  
average\_degree=sum(deg for n, deg in G.degree)/G.number\_of\_nodes()  
print(average\_degree)



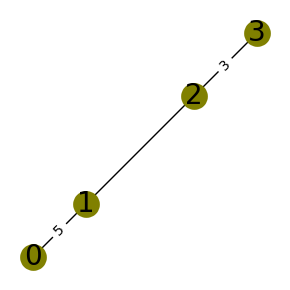
[(0, 3), (1, 1), (2, 1), (3, 1)]  
3  
1.5

average\_neighbor\_degree(G, source=‘out’, target=‘out’, nodes=None, weight=None)[5]：返回每一个顶点邻点的平均度。在无向图中，顶点的邻域包括通过边连接到的顶点。对于有向图，根据source参数确定：

1. 如果source参数为in，则由节点的前驱顶点组成；
2. 如果source参数为out，则由节点的后继顶点组成；
3. 如果source参数为in+out，则由节点的前驱和后继顶点共同组成。

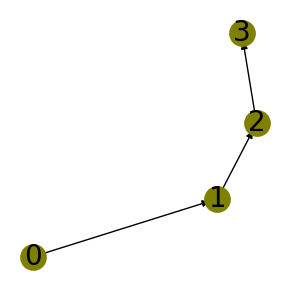
顶点的平均邻点度为：，式中，为顶点的邻点集；是属于邻点集顶点的度。对于有向图，定义一个类似的度量，公式为： ，式中，是顶点的加权度；是连接和的边权重；为顶点的邻点集。

G=nx.path\_graph(4)  
G.edges[0, 1]["weight"]=5  
G.edges[2, 3]["weight"]=3  
G\_drawing(G,edge\_labels='weight')  
print(nx.average\_neighbor\_degree(G))  
print(nx.average\_neighbor\_degree(G, weight="weight"))



{0: 2.0, 1: 1.5, 2: 1.5, 3: 2.0}  
{0: 2.0, 1: 1.1666666666666667, 2: 1.25, 3: 2.0}

G=nx.DiGraph()  
nx.add\_path(G, [0, 1, 2, 3])  
G\_drawing(G)  
nx.average\_neighbor\_degree(G, source="in", target="in")



{0: 0.0, 1: 1.0, 2: 1.0, 3: 0.0}

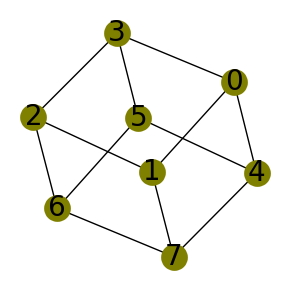
nx.average\_neighbor\_degree(G, source="out", target="out")

{0: 1.0, 1: 1.0, 2: 0.0, 3: 0.0}

is\_regular(G)：确定图G是否为正则图；

is\_k\_regular(G, k)：确定图G是否为k-正则的。

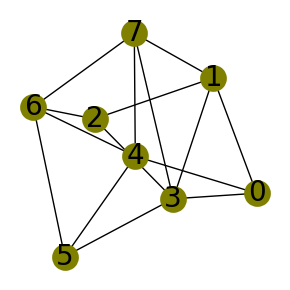
G=nx.cubical\_graph()  
G\_drawing(G)  
print(nx.is\_regular(G))  
print(nx.is\_k\_regular(G, 3))

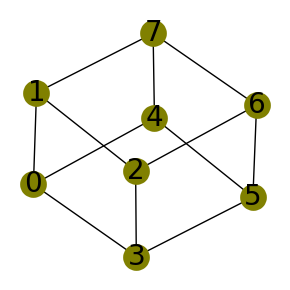


True  
True

k\_factor(G, k, matching\_weight=‘weight’)[6]：计算图G的k-factor（k-因子）。图的k-因子是生成k-正则子图。G的生成k-正则子图是包含G的每个顶点和G的边的子集的子图，使得每个顶点都有度k。

G=nx.cubical\_graph()  
G.add\_edges\_from([(3,7),(3,1),(4,6)])  
G\_drawing(G)  
G1=nx.k\_factor(G, 3)  
G\_drawing(G1)





### 2.8.1.3 路和圈

**路（path）**是一个非空图，其顶点集和边集分别为，，这里的xix0xk x1, , xk-1 kPk k。

经常用顶点的自然顺序排列表示路，记为P=x0​x1​…xk​ Px0xkx0 xk之间（between）的路）。

对，记的各种子路如下：

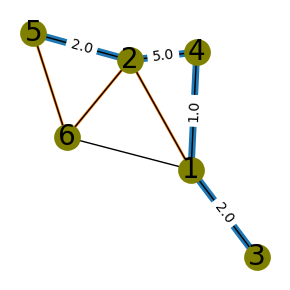
Pxi​:=x0​…xi​  
xi​P:=xi​…xk​  
xi​Pxj​:=xi​…xj​  
以及：

P˙:=x1​…xk−1​  
Pxi​˙​:=x0​…xi−1​  
xi​˙​P:=xi+1​…xk​  
xi​˙​Pxj​˙​:=xi+1​…xj−1​

可以用类似直观的方法表示路的串联，例如，如果三条路的并还是一条路，则可以简记为。

下述代码绘制了两条路，和。其中，is\_path(G, path)：判读指定路径是否存在。path\_weight(G, path, weight)：返回指定路径给定边属性（权重）的总成本。

G=nx.Graph()  
G.add\_edges\_from([(1,2),(1,3),(2,4),(1,4),(2,5),(2,6),(5,6),(1,6)])  
nx.set\_edge\_attributes(G,{(3,1):{'weight':2.0},(1,4):{'weight':1.0},(4,2):{'weight':5.0},(2,5):{'weight':2.0}})  
route=[3,1,4,2,5]  
G\_drawing(G,routes=[route,[1,2,6,5]],edge\_widths=[5,2],edge\_labels='weight')  
  
print(nx.is\_path(G,route))  
print(nx.path\_weight(G,route,weight='weight'))



True  
10.0

给定顶点集和及路*P*=*x*0​*x*1​…*xk*   
*V*(*P*)∩*A*={*x*0​}P{a}-Ba-B

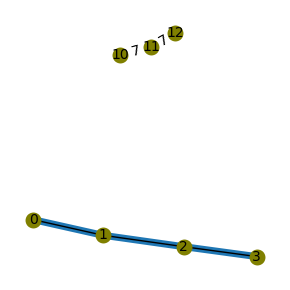
a-bab是其唯一的公共顶点。

给定图，如果路是非平凡的且只与在端点接触，则称是一条**H-路（H-path）**。特别的，任何长度为1的H-路的边不可能是的边。

add\_path(G\_to\_add\_to, nodes\_for\_path, \*\*attr)：将路径添加到图G中。

has\_path(G, source, target)：判断图G中是否有从源source到汇target顶点的路径。

G=nx.Graph()  
route\_1=[0, 1, 2, 3]  
route\_2=[10, 11, 12]  
nx.add\_path(G, route\_1)  
nx.add\_path(G,route\_2, weight=7)  
G\_drawing(G,routes=[route\_1,route\_2],edge\_widths=[5,2],edge\_labels='weight',node\_size=20,font\_size=10)  
print(nx.has\_path(G,0,3))



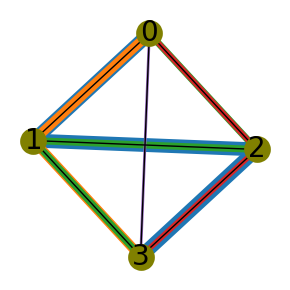
True

all\_simple\_paths(G, source, target, cutoff=None)：生成图G中给定源汇顶点的所有路。使用map(nx.utils.pairwise, paths)方法将路的顶点列表转换为对应的边列表。

shortest\_simple\_paths(G, source, target, weight=None)[7]：生成图G中给定源汇顶点的所有路。

G=nx.complete\_graph(4)  
paths=list(nx.all\_simple\_paths(G, source=0, target=3))  
print(paths)  
print(print(list(nx.shortest\_simple\_paths(G, 0, 3))))  
print([list(path) for path in map(nx.utils.pairwise, paths)])  
G\_drawing(G,routes=paths,edge\_widths=[10,7,5,4,2])

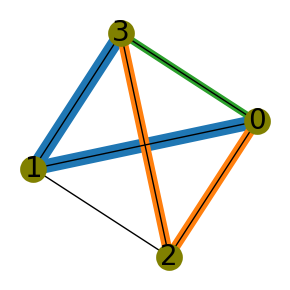
[[0, 1, 2, 3], [0, 1, 3], [0, 2, 1, 3], [0, 2, 3], [0, 3]]  
[[0, 3], [0, 1, 3], [0, 2, 3], [0, 1, 2, 3], [0, 2, 1, 3]]  
None  
[[(0, 1), (1, 2), (2, 3)], [(0, 1), (1, 3)], [(0, 2), (2, 1), (1, 3)], [(0, 2), (2, 3)], [(0, 3)]]



配置cutoff参数，仅返回小于等于给定路长度的路。

paths=list(nx.all\_simple\_paths(G, source=0, target=3, cutoff=2))  
print(paths)  
G\_drawing(G,routes=paths,edge\_widths=[10,7,5])

[[0, 1, 3], [0, 2, 3], [0, 3]]



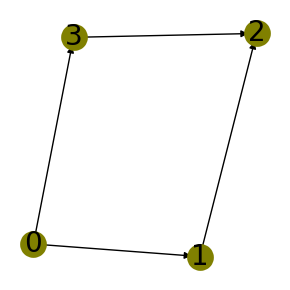
配置target参数为一个列表时，返回以任意多个节点结尾的所有路径。

paths=list(nx.all\_simple\_paths(G, source=0, target=[3,2]))  
print(paths)

[[0, 1, 2], [0, 1, 2, 3], [0, 1, 3], [0, 1, 3, 2], [0, 2], [0, 2, 1, 3], [0, 2, 3], [0, 3], [0, 3, 1, 2], [0, 3, 2]]

使用函数编程方法（functional programming approach）返回有向非循环图中（ directed acyclic graph）迭代从根节点（root nodes）到叶节点（leaf nodes）的每个路径。

from itertools import chain  
from itertools import product  
from itertools import starmap  
from functools import partial  
  
chaini=chain.from\_iterable  
G=nx.DiGraph([(0, 1), (1, 2), (0, 3), (3, 2)])  
G\_drawing(G)  
roots=(v for v, d in G.in\_degree() if d == 0)  
leaves=(v for v, d in G.out\_degree() if d == 0)  
all\_paths=partial(nx.all\_simple\_paths, G)  
list(chaini(starmap(all\_paths, product(roots, leaves))))



[[0, 1, 2], [0, 3, 2]]

使用迭代方法（iterative approach）返回有向非循环图中（ directed acyclic graph）迭代从根节点（root nodes）到叶节点（leaf nodes）的每个路径。

roots=(v for v, d in G.in\_degree() if d == 0)  
leaves=(v for v, d in G.out\_degree() if d == 0)  
print([list(nx.all\_simple\_paths(G, root, leaf)) for leaf in leaves for root in roots])

[[[0, 1, 2], [0, 3, 2]]]

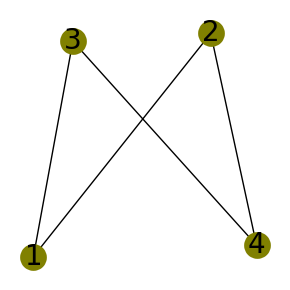
在有向非循环图中，迭代从根节点到叶节点的每个路径，将所有叶传递到一起，以避免不必要的计算。

G=nx.DiGraph([(0, 1), (2, 1), (1, 3), (1, 4)])  
G\_drawing(G)  
roots=(v for v, d in G.in\_degree() if d == 0)  
leaves=[v for v, d in G.out\_degree() if d == 0]  
print([list(nx.all\_simple\_paths(G, root, leaves)) for root in roots])

[[[0, 1, 3], [0, 1, 4]], [[2, 1, 3], [2, 1, 4]]]

all\_simple\_edge\_paths(G, source, target, cutoff=None)[8]：返回图G中所有源汇顶点路的边列表。

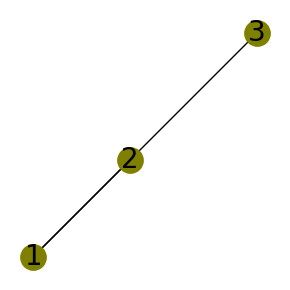
G=nx.Graph([(1, 2), (2, 4), (1, 3), (3, 4)])  
G\_drawing(G)  
sorted(nx.all\_simple\_edge\_paths(G, 1, 4))



[[(1, 2), (2, 4)], [(1, 3), (3, 4)]]

对于多重图（MultiGraph），返回路的边时，也包含其关联的键。

MG=nx.MultiGraph()  
MG.add\_edge(1, 2, key="k0")  
MG.add\_edge(1, 2, key="k1")  
MG.add\_edge(2, 3, key="k0")  
G\_drawing(MG)  
sorted(nx.all\_simple\_edge\_paths(MG, 1, 3))



[[(1, 2, 'k0'), (2, 3, 'k0')], [(1, 2, 'k1'), (2, 3, 'k0')]]

is\_simple\_path(G, nodes)：如果顶点来自于简单路（simple path）则返回为True。一个简单路是一个非空顶点列表，其中没有顶点在序列中出现一次以上，且序列中的每对邻点在图中相邻。

G=nx.cycle\_graph(4)  
G\_drawing(G)  
print(nx.is\_simple\_path(G, [2, 3, 0]))  
print(nx.is\_simple\_path(G, [0, 2]))

True  
False

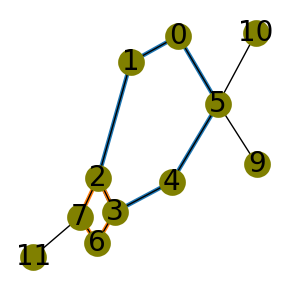
若P=x0​…xk k 

Cx0​x1​…xk−1​x0​ k。

图G中最短圈的长度叫做**围长（girth）**，记作，而中最长圈的长度称为**周长（circumference）**。（若中不含圈，则围长设为G的**导出圈（induced cycle）**是不含弦的圈（即G的导出子图是个圈）。

add\_cycle(G\_to\_add\_to, nodes\_for\_cycle, \*\*attr)：将圈添加到图G中。

G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc  
cycle\_1= [0, 1, 2, 3,4,5]  
cycle\_2=[2,3,6,7]  
G.add\_edges\_from([(5,9),(5,10),(7,11)])  
nx.add\_cycle(G,cycle\_1)  
nx.add\_cycle(G, cycle\_2, weight=7)  
G\_drawing(G,routes=[cycle\_1+[cycle\_1[0]],cycle\_2+[cycle\_2[0]]])



find\_cycle(G, source=None, orientation=None)：通过深度优先（depth-first traversal）遍历找到圈，返回圈的有向边列表。

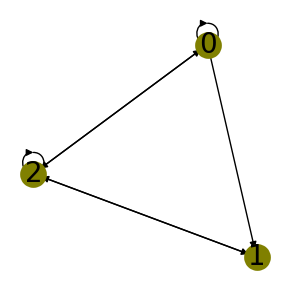
cycle\_basis(G, root=None)[9]：返回基础圈（basis for cycles）的一个列表，且为圈的最小集合，即图G中任何一个圈都为基础圈对象的和。

print(nx.find\_cycle(G))  
print(nx.cycle\_basis(G))  
print([sorted(c) for c in nx.minimum\_cycle\_basis(G)])

[(5, 4), (4, 3), (3, 2), (2, 1), (1, 0), (0, 5)]  
[[2, 7, 6, 3], [1, 2, 3, 4, 5, 0]]  
[[0, 1, 2, 3, 4, 5], [2, 3, 6, 7]]

simple\_cycles(G)[10][11][12]：返回有向图的简单圈（simple cyccle）。

edges=[(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]  
G=nx.DiGraph(edges)  
G\_drawing(G)  
print(sorted(nx.simple\_cycles(G)))



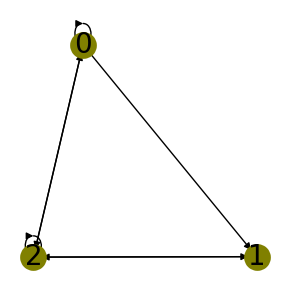
[[0], [0, 1, 2], [0, 2], [1, 2], [2]]

recursive\_simple\_cycles(G)[10]：返回有向图的简单圈（simple cyccle）。

print(nx.recursive\_simple\_cycles(G))

[[0, 1, 2], [0, 2], [1, 2]]

H=G.copy()  
H.remove\_edges\_from(nx.selfloop\_edges(G))  
c  
sorted(nx.simple\_cycles(H))



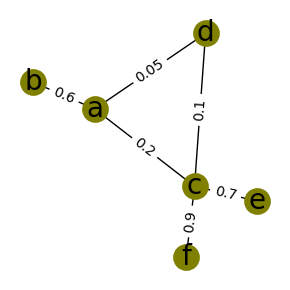
[[0, 1, 2], [0, 2], [1, 2]]

图G中的两个顶点之间的**距离（distance）**定义为中最短路的长度；如果这样的路不存在，则令GGdiam G。当然直径和周长是密切相关的。

diameter(G, e=None, usebounds=False)：返回图G的直径。直径是最大偏心距（maximum eccentricity）。

shortest\_path(G, source=None, target=None, weight=None, method=‘dijkstra’)：给定源汇顶点计算最短路径。如果配置参数weight，则按权重值计算最短路径。

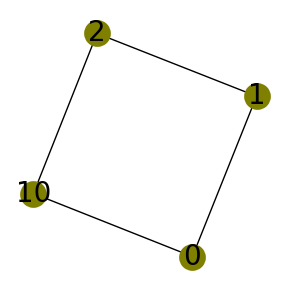
G = nx.Graph()  
G.add\_edge("a", "b", weight=0.6)  
G.add\_edge("a", "c", weight=0.2)  
G.add\_edge("c", "d", weight=0.1)  
G.add\_edge("c", "e", weight=0.7)  
G.add\_edge("c", "f", weight=0.9)  
G.add\_edge("a", "d", weight=0.05)  
G\_drawing(G,edge\_labels='weight')  
print(nx.diameter(G))  
print(nx.shortest\_path(G, source="b", target="e"))  
print(nx.shortest\_path(G, source="b", target="e",weight='weight'))



3  
['b', 'a', 'c', 'e']  
['b', 'a', 'd', 'c', 'e']

all\_shortest\_paths(G, source, target, weight=None, method=‘dijkstra’)：给定源汇顶点返回所有最短路径。

G=nx.Graph()  
nx.add\_path(G, [0, 1, 2])  
nx.add\_path(G, [0, 10, 2])  
G\_drawing(G)  
print([p for p in nx.all\_shortest\_paths(G, source=0, target=2)])

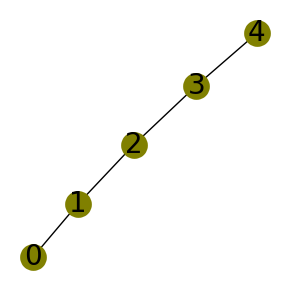


[[0, 1, 2], [0, 10, 2]]

shortest\_path\_length(G, source=None, target=None, weight=None, method=‘dijkstra’)：计算源汇之间最短路长度。如果指定源汇则返回最短路长度；如果只指定源则返回源到各个顶点的最短路长度；如果只指定汇则返回各个顶点到汇的最短路长度；如果不指定源汇则返回所有顶点对的最短路长度。

average\_shortest\_path\_length(G, weight=None, method=None：返回平均最短路长度，其公式为：，式中，是的顶点集，是从顶点到的最短路长度，是图顶点数（阶）。

G=nx.path\_graph(5)  
G\_drawing(G)  
print(nx.shortest\_path\_length(G, source=0, target=4))  
print(nx.shortest\_path\_length(G, source=0)) # target not specified  
print(nx.shortest\_path\_length(G, target=4)) # source not specified  
print(dict(nx.shortest\_path\_length(G))) # source,target not specified  
print(nx.average\_shortest\_path\_length(G))



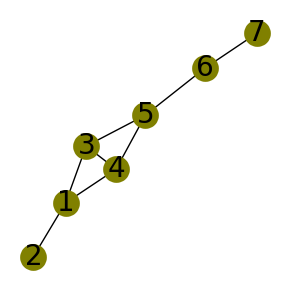
4  
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}  
{4: 0, 3: 1, 2: 2, 1: 3, 0: 4}  
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 4}, 1: {1: 0, 0: 1, 2: 1, 3: 2, 4: 3}, 2: {2: 0, 1: 1, 3: 1, 0: 2, 4: 2}, 3: {3: 0, 2: 1, 4: 1, 1: 2, 0: 3}, 4: {4: 0, 3: 1, 2: 2, 1: 3, 0: 4}}  
2.0

图的**中心点（central vertex）**是指能使得它到任何其它顶点的距离尽可能小的顶点（中心点可以不只一个），这个最短距离称作**半径（radius）**并记为。严格的说，为各顶点到其它各顶点的最大距离的最小值。

center(G, e=None, usebounds=False)：返回图G的中心点。

radius(G, e=None, usebounds=False)：返回图G的半径。

G=nx.Graph([(1, 2), (1, 3), (1, 4), (3, 4), (3, 5), (4, 5),(5,6),(6,7)])  
G\_drawing(G)  
print(list(nx.center(G)))  
print(nx.radius(G))



[3, 4, 5]  
3

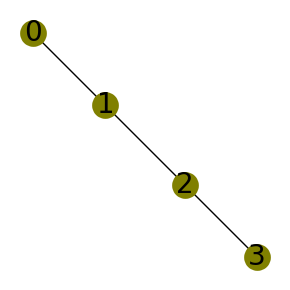
图中长度为的**途径（walk）**是一个非空的顶点和边的交错序列 i<kei​={vi​,vi+1​} G中的一条路。

### 2.8.1.4 连通性

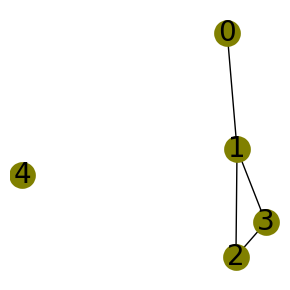
如果非空图中的任意两个顶点之间均有一条路相邻，称是**连通的（connected）**（若一个图中有个顶点，并且边数小于，则此图一定是非连通的）。若且是联通的，则称（在中）本身是连通的。

is\_connected(G)：如图是连通的，则返回True，否则返回False。

G=nx.path\_graph(4)  
G\_drawing(G)  
print(nx.is\_connected(G))  
G.add\_edges\_from([(1,3)])  
G.add\_node(4)  
G\_drawing(G)  
print(nx.is\_connected(G))



True

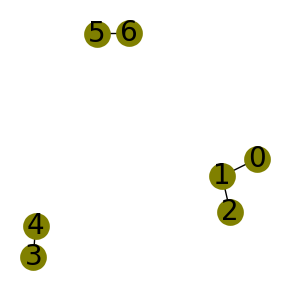


False

设是一个图，则它的极大连通子图称为**分支（component）**。显然，分支都是导出子图且它们的顶点集划分。因为连通图是非空的，所以空图没有分支。

number\_connected\_components(G)：返回连通分支（connected components）的数量。

G=nx.Graph([(0, 1), (1, 2), (5, 6), (3, 4)])  
c  
print(nx.number\_connected\_components(G))

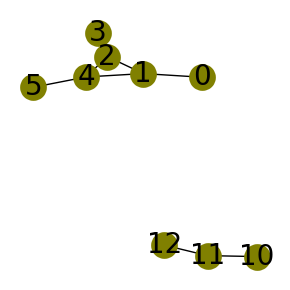


3

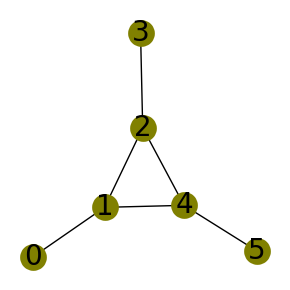
connected\_components(G)：返回所有连通分支。

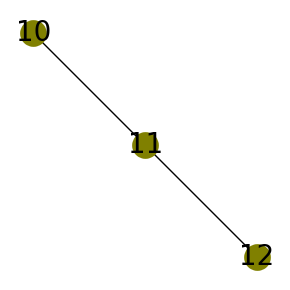
subgraph(G, nbunch)：返回顶点列表中的导出子图。

G=nx.path\_graph(4)  
nx.add\_path(G, [10, 11, 12])  
G.add\_edges\_from([(2,4),(1,4),(4,5)])  
G\_drawing(G)  
print(list(nx.connected\_components(G)))  
print([len(c) for c in sorted(nx.connected\_components(G), key=len, reverse=True)])  
largest\_cc=max(nx.connected\_components(G), key=len) # 返回最大阶的连通分支  
print(largest\_cc)  
S=[G.subgraph(c).copy() for c in nx.connected\_components(G)] # 创建每个连通分支的导出子图  
[G\_drawing(s) for s in S];



[{0, 1, 2, 3, 4, 5}, {10, 11, 12}]  
[6, 3]  
{0, 1, 2, 3, 4, 5}





node\_connected\_component(G, n)：返回包含顶点图的连通分支。

print(nx.node\_connected\_component(G, 2)) # nodes of component that contains node 0  
print(nx.node\_connected\_component(G, 11))

{0, 1, 2, 3, 4, 5}  
{10, 11, 12}

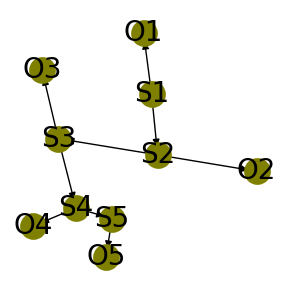
给定和，如果的每条路均包含中的一个顶点或一条边，则称在中**X分离（separate）**集合和。注意到，这蕴含着。若两个顶点且分离顶点，，则称**X分离**顶点；如果分离中的两个顶点，就称**X分离G**。顶点所形成的分离集合亦称为**分隔（separator）**。边的分离集合没有通用的名称，但某些这样的集合会有专用名称，例如**割（cut）**和**键（bond）**。如果一个顶点分离同一个分支中的两个顶点，则称它为**割点（cutvertex）**，而**桥（bridge）**则为分离其两个端点的边，所以图中的桥恰为那些不在任何圈中的边。

如果且没有和之间的边，则无序对称为的**分离（separation）**。显然，第二个条件等于说，分离和。若和均非空，则称这个分离式**真的（proper）**，而| A ∩B | {A,B}A,B$叫做分离的**侧面（side）**。

割点：如果去掉一个点及与它连接的边，该点原来所在的图被分成两部分（不连通），则称该点为割点；割边：如果去掉一条边，该边原来所在的图被分成两部分（不连通），则称该点为割边（桥）。

d\_separated(G, x, y, z)[13][14][15][16]：返回是否顶点集分离顶点集和。

g = nx.DiGraph()  
g.add\_edges\_from(  
 [  
 ("S1", "S2"),  
 ("S2", "S3"),  
 ("S3", "S4"),  
 ("S4", "S5"),  
 ("S1", "O1"),  
 ("S2", "O2"),  
 ("S3", "O3"),  
 ("S4", "O4"),  
 ("S5", "O5"),  
 ]  
)  
G\_drawing(g)  
print(nx.d\_separated(g, {"S1", "S2", "O1", "O2"}, {"S4", "S5", "O4", "O5"}, {"S3","S4"}))  
print(nx.d\_separated(g, {"S1", "S2", "O1", "O2"}, {"S3", "S4", "O4", "O3"}, {"S2"}))  
print(nx.d\_separated(g, {"S1", "S2", "O1", "O2"}, {"S4", "S5", "O4", "O5"}, {"S1"}))



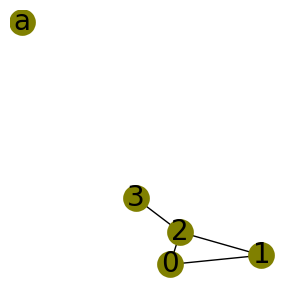
True  
True  
False

若，且对任意满足|X| <k X VG-XGGkGkG(G)(G)=0GK^{1} n 。

一个网络的健壮程度是指它不容易因为顶点的移除使得源汇顶点连接断开，因此顶点的连通度指，要断开源汇顶点的连通所要移除的最小顶点数。可以证明，连通度也等于顶点之间独立于节顶点的路径数，因此可以通过计算独立于顶点的路径数来量化网络的健壮性[17]。

all\_pairs\_node\_connectivity(G, nbunch=None, cutoff=None)：计算所有顶点对之间的连通度（node connectivity）。

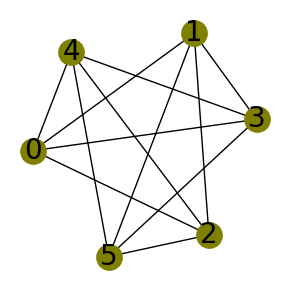
import pprint  
  
G=nx.cycle\_graph(3)  
G.add\_node("a")  
G.add\_edge(2, 3)  
G\_drawing(G)   
pprint.pprint(nx.all\_pairs\_node\_connectivity(G))



{0: {1: 2, 2: 2, 3: 1, 'a': 0},  
 1: {0: 2, 2: 2, 3: 1, 'a': 0},  
 2: {0: 2, 1: 2, 3: 1, 'a': 0},  
 3: {0: 1, 1: 1, 2: 1, 'a': 0},  
 'a': {0: 0, 1: 0, 2: 0, 3: 0}}

local\_node\_connectivity(G, source, target, cutoff=None)： 计算源汇顶点之间的连通度。

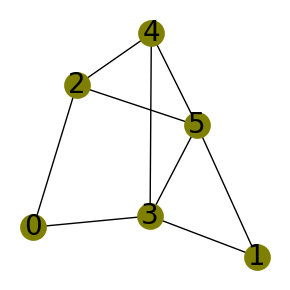
from networkx.algorithms import approximation as approx  
G=nx.octahedral\_graph()  
G\_drawing(G)   
approx.local\_node\_connectivity(G, 1,3)



4

node\_connectivity(G, s=None, t=None)：计算有向图或无向图近似的顶点连通度。为使得图为不连通所要删除的最小顶点数。

from networkx.algorithms import approximation as approx  
G=nx.octahedral\_graph()  
G.remove\_edges\_from([(0,4),(1,2),(0,1)])  
G\_drawing(G)  
print(approx.node\_connectivity(G))



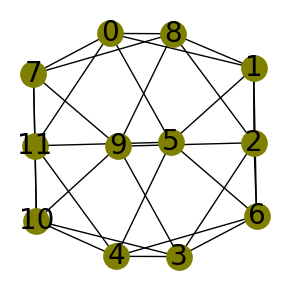
2

若且对任意少于  
ℓF ⊆EG-FGGG(G)Gλ(G)=0。

edge\_connectivity(G, s=None, t=None, flow\_func=None, cutoff=None)[18]：返回有向图或无向图G的边连通度。如果给定一对顶点，则可以返回本地边连通度（local edge connectivity）。

local\_edge\_connectivity(G, s, t, flow\_func=None, auxiliary=None, residual=None, cutoff=None)[18]： 给定一对顶点，返回本地边连通度（local edge connectivity）。

# Platonic icosahedral graph is 5-edge-connected  
G=nx.icosahedral\_graph()  
G\_drawing(G)  
print(nx.edge\_connectivity(G))  
print(nx.edge\_connectivity(G, 1, 7))  
  
from networkx.algorithms.connectivity.connectivity import local\_edge\_connectivity  
print(local\_edge\_connectivity(G, 1, 7))



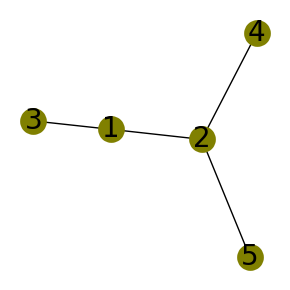
5  
5  
5

### 2.8.1.5 树和森林

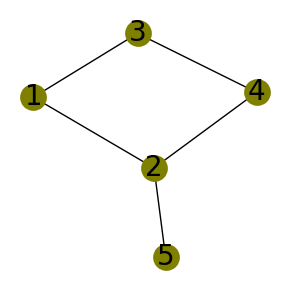
一个**无圈（acyclic）**图，即不含任何圈的图，亦称为**森林（forest）**，而连通的森林则称为**树（tree）**。（所以森林里是分支为树的图）。树中度为1的顶点为**叶子（leaf）**，而其它顶点称为内部顶点。每个非平凡的树都有叶子，例如，最长路的端点。这一简单的结论有时会很有用，尤其是对树使用归纳法时：去掉树的一个叶子，剩下的图还是一个树。对于树，当是图的一颗支撑树时，中的边是在中的**弦（chord）**。

is\_tree(G)：返回图是否为树。

G=nx.Graph()  
G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5)])  
G\_drawing(G)  
print(nx.is\_tree(G)) # n-1 edges  
  
G.add\_edge(3, 4)  
G\_drawing(G)  
print(nx.is\_tree(G)) # n edges



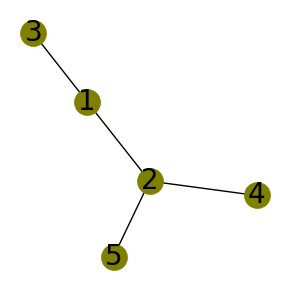
True



False

is\_forest(G)：返回图是否为森林。

G=nx.Graph()  
G.add\_edges\_from([(1, 2), (1, 3), (2, 4), (2, 5)])  
G\_drawing(G)  
print(nx.is\_forest(G))  
  
G.add\_edge(4, 1)  
G\_drawing(G)  
print(nx.is\_forest(G))



True

False

有时，把树中的一个顶点做特别处理会方便问题的解决，这个顶点称为树的**根（root）**，而具有固定根的树叫做**有根树（rooted tree）**。把*x*∈*rTy* x ≤yTrx<yxTyy:={x | x ≤y}x:={y | y ≥x}yxT中是**上闭的（up-closed）**或是一个**上集合（up-set）**；类似的，可以定义**下闭的（down-closed）**或**下集合（down-set）**。

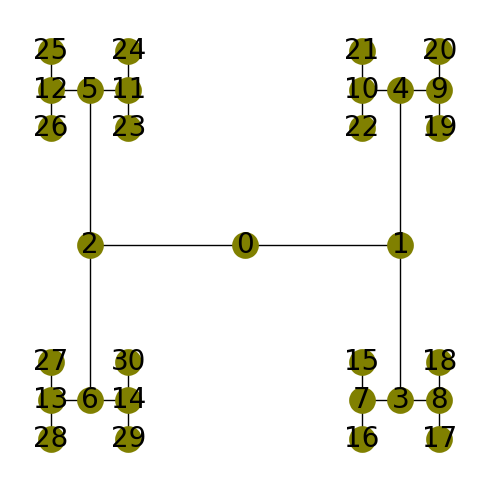
注意到，的根是这个偏序关系中最小的元素，叶子是极大元素，而的任意边的两个端点是可比的，任意顶点的下闭包是一条**链（chain）**，即两两可比较的元素的集合。到根的距离为的顶点具有**高度（height）**，并组成的第**层（level）**。

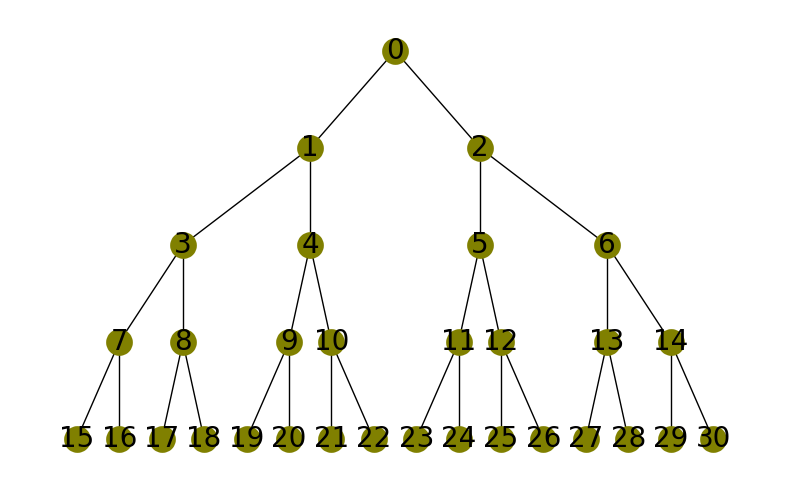
对于包含于中的有根树，如果中的任意T-路的两个端点在的树序中是可比的，称有根树在中是**正规的（normal）**。若支撑，这等于是要求：只要两个顶点在中是相邻的，它们在中一定是可比的。正规支撑树也叫做**深度优先搜索树（depth-first search tree，DFS tree）**，这是因为它和图的计算机搜索相关。

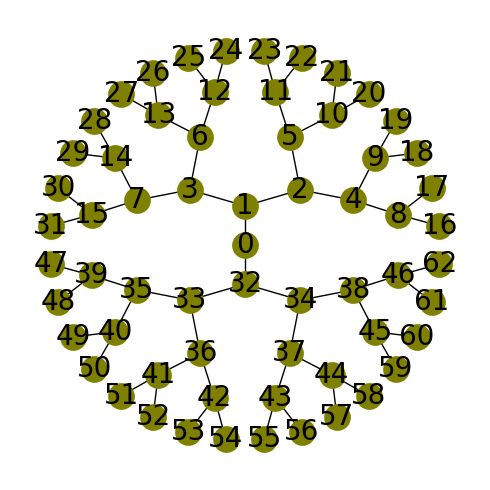
join(rooted\_trees, label\_attribute=None)：给定多个有根树，返回一个新的有根树。新的有根树的根连接到给定多个有根树的根。

为了更清晰的表达树和森林，安装pydot（conda install pydot）库，通过graphviz\_layout方法传递参数prog，包括”twopi”, “dot”, “circo”等获得顶点位置值，绘制图。

from networkx.drawing.nx\_pydot import graphviz\_layout  
import pydot  
  
h=4  
left=nx.balanced\_tree(2, h)  
right=nx.balanced\_tree(2, h)  
  
pos=graphviz\_layout(left, prog="circo")   
G\_drawing(left,pos=pos,figsize=(5,5))  
  
pos=graphviz\_layout(right, prog="dot")  
G\_drawing(right,pos=pos,figsize=(8,5))  
  
joined\_tree=nx.join([(left, 0), (right, 0)])  
pos=graphviz\_layout(joined\_tree, prog="twopi")  
G\_drawing(joined\_tree,pos=pos,figsize=(5,5))







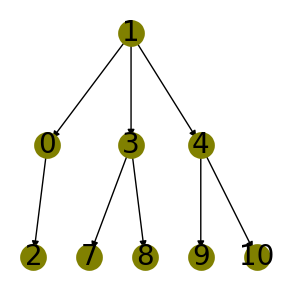
dfs\_edges(G, source=None, depth\_limit=None)：在深度优先搜索（ depth-first-search (DFS)）下沿给定源顶点，迭代返回边。可以配置参数depth\_limit限制搜索的深度。

print(list(nx.dfs\_edges(right, source=0)))  
print('-'\*50)  
print(list(nx.dfs\_edges(right, source=0, depth\_limit=2)))

[(0, 1), (1, 3), (3, 7), (7, 15), (7, 16), (3, 8), (8, 17), (8, 18), (1, 4), (4, 9), (9, 19), (9, 20), (4, 10), (10, 21), (10, 22), (0, 2), (2, 5), (5, 11), (11, 23), (11, 24), (5, 12), (12, 25), (12, 26), (2, 6), (6, 13), (13, 27), (13, 28), (6, 14), (14, 29), (14, 30)]  
--------------------------------------------------  
[(0, 1), (1, 3), (1, 4), (0, 2), (2, 5), (2, 6)]

dfs\_tree(G, source=None, depth\_limit=None)：在深度优先搜索（ depth-first-search (DFS)）下给定源顶点，返回指定深度的定向树（oriented tree）。

T=nx.dfs\_tree(right, source=1, depth\_limit=2)  
pos=graphviz\_layout(T, prog="dot")  
G\_drawing(T,pos=pos)



dfs\_predecessors(G, source=None, depth\_limit=None)：在深度优先搜索（ depth-first-search (DFS)）下给定源顶点，返回指定深度，顶点为键，值为该顶点的祖先（predecessors ）。

print(nx.dfs\_predecessors(right, source=0,depth\_limit=2))

{1: 0, 3: 1, 4: 1, 2: 0, 5: 2, 6: 2}

dfs\_successors(G, source=None, depth\_limit=None)：在深度优先搜索（ depth-first-search (DFS)）下给定源顶点，返回指定深度，顶点为键，值为该顶点的后代（successors）。

print(nx.dfs\_successors(right, source=0, depth\_limit=2))

{0: [1, 2], 1: [3, 4], 2: [5, 6]}

### 2.8.1.6 二部图

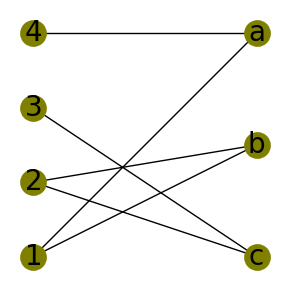
设是一个整数，对于图，如果可以划分为个类使得任意一条边的端点都属于不同的类中（即同一类中的顶点不相邻），则称为**r-部图（r-partite graph）**。通常把2-部图称为**二部图（bipartite graph）**。

若-部图中，不同类中任意两个顶点均相邻，则称它为**完全r-部图（complete r-partite graph）**。所有的-部图被统称为**完全多部图（complete multipartite graph）**。把完全部图 *xx*1​,…,*nr*​​n1 = = nr =:s  srK1,n K1,n n的顶点称为星的**中心（center）**。任意二部图不能包含长度为奇数的圈（**奇圈（odd cycle）**）。

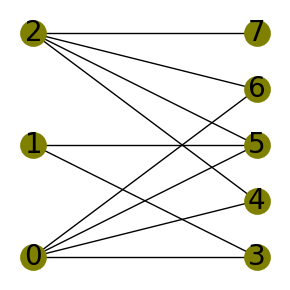
下述使用了两种方法定义二部图，第1种通过参数bipartite指定两个顶点集，并自定义边实现；第2种使用networkx.bipartite模块提供的生成方式构建。为了清晰显示二部图的关系，用networkx.bipartite.sets方法提取顶点集，并配合使用networkx.bipartite\_layout定义顶点位置，传入到自定义G\_drawing()函数中绘制二部图。

B=nx.Graph()  
# Add nodes with the node attribute "bipartite"  
B.add\_nodes\_from([1, 2, 3, 4], bipartite=0)  
B.add\_nodes\_from(["a", "b", "c"], bipartite=1)  
# Add edges only between nodes of opposite node sets  
B.add\_edges\_from([(1, "a"), (1, "b"), (2, "b"), (2, "c"), (3, "c"), (4, "a")])  
print(nx.bipartite.sets(B))  
pos=nx.bipartite\_layout(B, nx.bipartite.sets(B)[0])  
print(pos)  
G\_drawing(B,pos=pos)

({1, 2, 3, 4}, {'c', 'b', 'a'})  
{1: array([-0.75 , -0.65625]), 2: array([-0.75 , -0.21875]), 3: array([-0.75 , 0.21875]), 4: array([-0.75 , 0.65625]), 'c': array([ 1. , -0.65625]), 'b': array([1.00000000e+00, 2.08166817e-17]), 'a': array([1. , 0.65625])}



G=nx.bipartite.gnmk\_random\_graph(3, 5, 10, seed=123)  
top=nx.bipartite.sets(G)[0]  
pos=nx.bipartite\_layout(G, top)  
G\_drawing(G,pos=pos)



除了用networkx.bipartite.sets方法获取顶点集，也可以直接使用顶点属性的方法提取。

top\_nodes={n for n, d in B.nodes(data=True) if d["bipartite"] == 0}  
bottom\_nodes=set(B)-top\_nodes  
print(top\_nodes,bottom\_nodes)

{1, 2, 3, 4} {'c', 'b', 'a'}

* Basic functions（基本函数）

is\_bipartite(G)：判断是否为二部图。

is\_bipartite\_node\_set(G, nodes)：判断顶点集是否为二分部顶点集。

sets(G[, top\_nodes])：返回顶点集。

color(G)：返回二部图顶点集染色（ two-coloring）。

density(B, nodes)：返回二部图的密度。

degrees(B, nodes[, weight])：返回二部图顶点集顶点的度值。

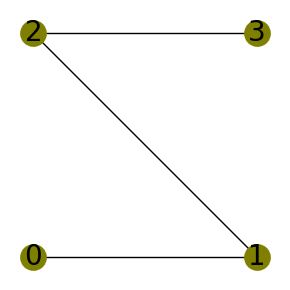
from networkx.algorithms import bipartite  
print(bipartite.is\_bipartite(B))   
print(bipartite.is\_bipartite\_node\_set(B, top\_nodes))  
print(bipartite.sets(B))  
print(bipartite.color(B))  
print(bipartite.density(B,top\_nodes))  
print(bipartite.density(B, bottom\_nodes))  
print(bipartite.degrees(B, top\_nodes))

True  
True  
({1, 2, 3, 4}, {'c', 'b', 'a'})  
{1: 1, 'a': 0, 'b': 0, 2: 1, 'c': 0, 3: 1, 4: 1}  
0.5  
0.5  
(DegreeView({'c': 2, 'b': 2, 'a': 2}), DegreeView({1: 2, 2: 2, 3: 1, 4: 1}))

* Edgelist （边集）

generate\_edgelist(G[, delimiter, data])： 以列表形式返回二部图的边，可以指定返回边的属性值。

from networkx.algorithms import bipartite  
G=nx.path\_graph(4)  
G.add\_nodes\_from([0, 2], bipartite=0)  
G.add\_nodes\_from([1, 3], bipartite=1)  
G[1][2]["weight"]=3  
G[2][3]["capacity"]=12  
pos=nx.bipartite\_layout(G, nx.bipartite.sets(G)[0])  
G\_drawing(G,pos=pos)  
print(list(bipartite.generate\_edgelist(G, data=False)))  
print(list(bipartite.generate\_edgelist(G, data=True)))  
print(list(bipartite.generate\_edgelist(G, data=['weight'])))



['0 1', '2 1', '2 3']  
['0 1 {}', "2 1 {'weight': 3}", "2 3 {'capacity': 12}"]  
['0 1', '2 1 3', '2 3']

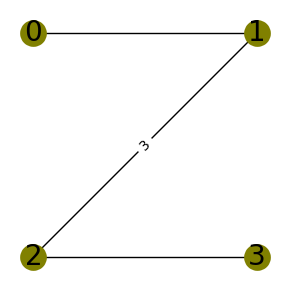
write\_edgelist(G, path[, comments, …])：将二部图边集合写入至文件。

read\_edgelist(path[, comments, delimiter, …])：配合write\_edgelist方法读取边集合文件为二部图。

nx.write\_edgelist(G, "./data/test.edgelist")

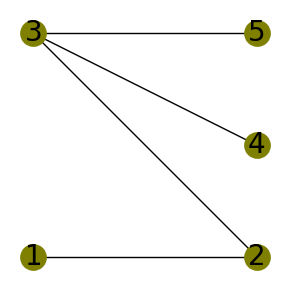
fh=open("./data/test.edgelist", "r")  
print(fh.readlines())  
G=bipartite.read\_edgelist("./data/test.edgelist")  
pos=nx.bipartite\_layout(G, nx.bipartite.sets(G)[0])  
G\_drawing(G,pos=pos,edge\_labels='weight')

['0 1 {}\n', "1 2 {'weight': 3}\n", "2 3 {'capacity': 12}\n"]



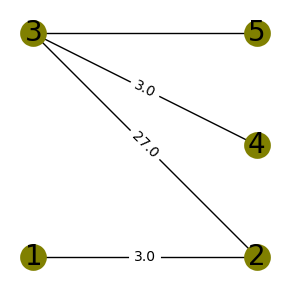
parse\_edgelist(lines[, comments, delimiter, …])：例如以[“1 2”, “2 3”, “3 4”,“3 5”]或[“1 2 {‘weight’:3}”, “2 3 {‘weight’:27}”, “3 4 {‘weight’:3.0}”]，及[“1 2 3”, “2 3 27”, “3 4 3.0”,“3 5”]等方式传入，自动解析构建二部图。

from networkx.algorithms import bipartite  
lines=["1 2", "2 3", "3 4","3 5"]  
G=bipartite.parse\_edgelist(lines, nodetype=int)  
pos=nx.bipartite\_layout(G, nx.bipartite.sets(G)[0])  
G\_drawing(G,pos=pos)



lines=["1 2 {'weight':3}", "2 3 {'weight':27}", "3 4 {'weight':3.0}"]  
G=bipartite.parse\_edgelist(lines, nodetype=int)  
G=nx.complete\_bipartite\_graph(2, 3)

lines=["1 2 3", "2 3 27", "3 4 3.0","3 5"]  
G=bipartite.parse\_edgelist(lines, nodetype=int, data=(("weight", float),))  
pos=nx.bipartite\_layout(G, nx.bipartite.sets(G)[0])  
G\_drawing(G,pos=pos,edge\_labels='weight')

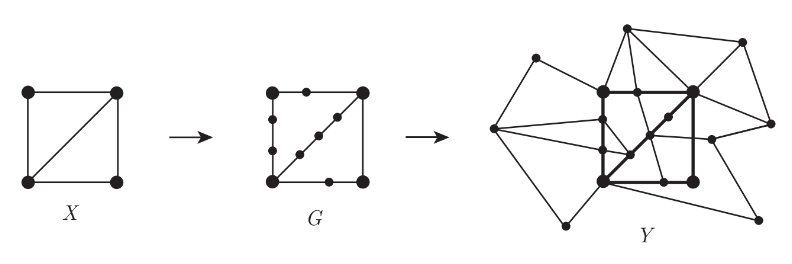


### 2.8.1.7 收缩运算和子式

非正式的，对中的某些或全部边进行“细分”，即在这些边上插入若干新的顶点，这样所得到的图叫做的**细分（subdivision）**。换句话说，把中的某些边替换成具有相同端点的路，使得这些路的内点即不在中，也不在其它任何新的路上。当是的细分时，亦称是一个 XTX

X中的顶点度。

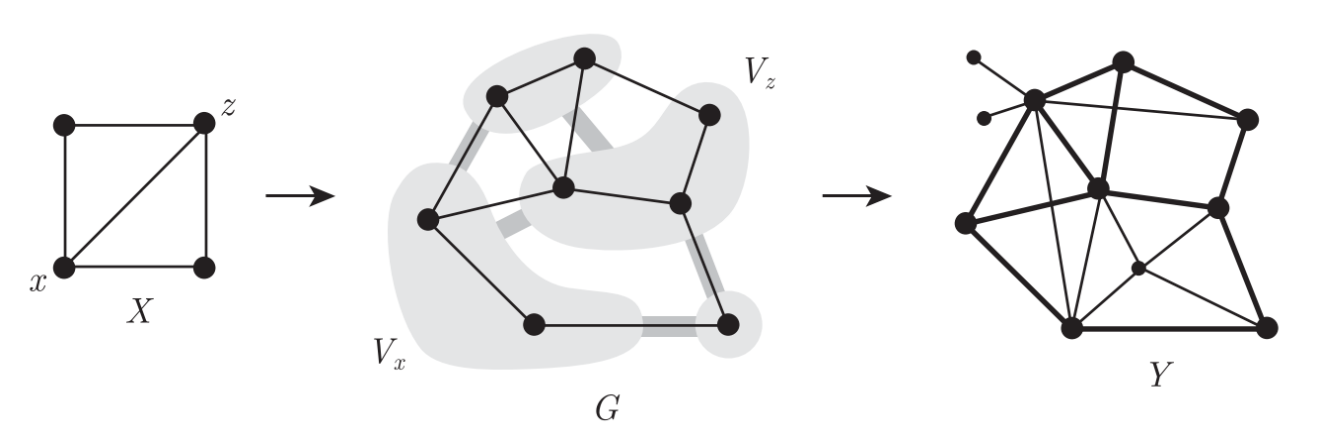
如果包含作为子图，那么是的**拓扑子式（topological minor）**。下图[2]18中图是一个，即的细分，因为，故是的拓扑子式。



类似的，把的每个顶点由不相交的连通子图Gx XxyGx-Gy GVxx,y ∈XXGGIXVxIXXGGxG的**收缩子式（contraction minor）**。

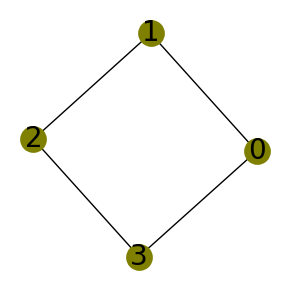
若图包含子图，则称是的**子式（minor）**。而是中的**模型（model）**，记作（下图[2]18）。因此是的子式，当且仅当存在从的子集到上的映射*φ*x ∈X Yx x’ ∈ XY   V(Y)x x’ x ≠x’ XY  YIX YX的收缩（contraction）。

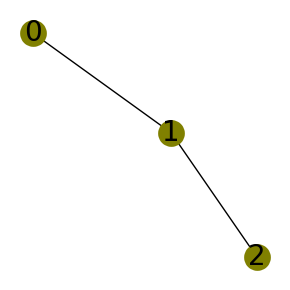
因为分支集可能是单点集，所以图的每个子图也是它的一个子式。在无限图中，分支集也允许式无限的。

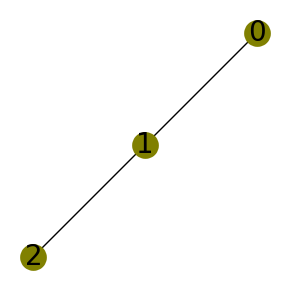


contracted\_nodes(G, u, v, self\_loops=True, copy=True：收缩两个顶点为一个，收缩后的顶点与原两个顶点上的任何边相关联。

G=nx.cycle\_graph(4)  
G\_drawing(G)  
M=nx.contracted\_nodes(G, 1, 3)  
G\_drawing(M)  
P3=nx.path\_graph(3)  
G\_drawing(P3)  
nx.is\_isomorphic(M, P3)





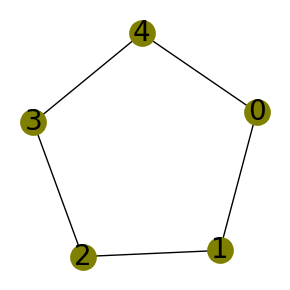


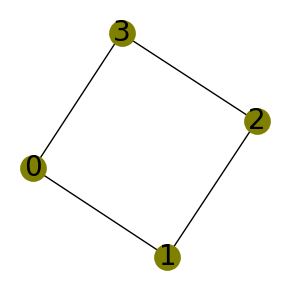
True

contracted\_edge(G, edge, self\_loops=True, copy=True)：返回收缩指定边的结果图。边收缩是将边的两个端点收缩为一个顶点，该顶点与收缩边端点所关联的边相关联。由边收缩产生的图为原始图的子式（minor）。

下述代码给定的参数edge为边，可描述为图是图**收缩边**而得到的结果。

C5=nx.cycle\_graph(5)  
G\_drawing(C5)  
C4=nx.cycle\_graph(4)  
G\_drawing(C4)  
M=nx.contracted\_edge(C5, (0, 1), self\_loops=False)  
G\_drawing(M)  
nx.is\_isomorphic(M, C4)

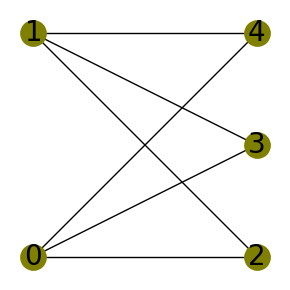


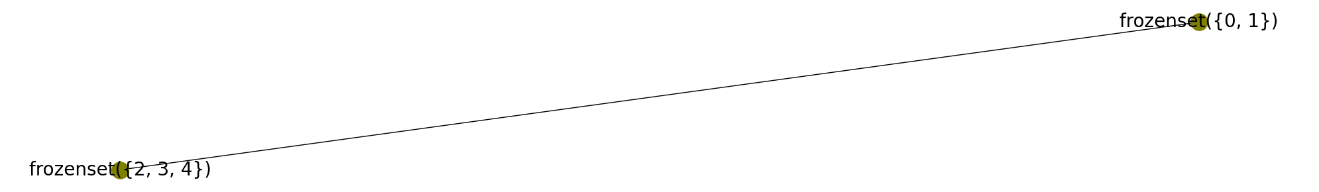


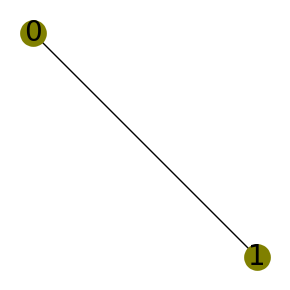
True

quotient\_graph(G, partition, edge\_relation=None, node\_data=None, edge\_data=None, relabel=False, create\_using=None)：指定顶点的等价关系（equivalence relation）返回商图（quotient graph）。参数partition可以为自定义函数或字典和嵌套列表。如果是函数，则必须表征图顶点等价关系。其包含两个参数u和v，如果u和v在自定义同一等价关系类中则返回True。返回的图由等价类构成顶点集；如果是字典列表等，则键可以为任何有意义的块标签（block labels），值必须为形成图形节点的有效分区（valid partition），即每个顶点必须正好位于分区的一个块中。

G=nx.complete\_bipartite\_graph(2, 3)  
pos=nx.bipartite\_layout(G, nx.bipartite.sets(G)[0])  
G\_drawing(G,pos=pos)  
  
same\_neighbors = lambda u, v: ( u not in G[v] and v not in G[u] and G[u] == G[v])  
Q=nx.quotient\_graph(G, same\_neighbors)  
print(list(Q.nodes))  
G\_drawing(Q,figsize=(20,3))  
  
K2=nx.complete\_graph(2)  
G\_drawing(K2)  
print(nx.is\_isomorphic(Q, K2))

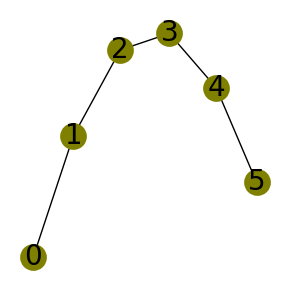


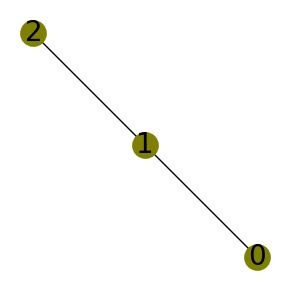




True

G=nx.path\_graph(6)  
G\_drawing(G)  
partition=[{0, 1}, {2, 3}, {4, 5}] # partition = {0: {0, 1}, 2: {2, 3}, 4: {4, 5}}  
M=nx.quotient\_graph(G, partition, relabel=True)  
G\_drawing(M)





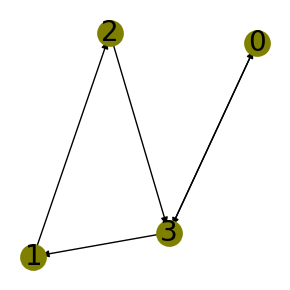
图在中的**嵌入（embdedding）***是一个单射 GH E(G)V(G)GxyH (x) (y)GH GH V(G)HGxyH*φ*(x)* φ*(y)的边。根据研究的对象，可以引进不同的嵌入，例如，可以类似的定义“作为支撑子图”，“作为导出子式”等嵌入。*

### 2.8.1.8 欧拉环游

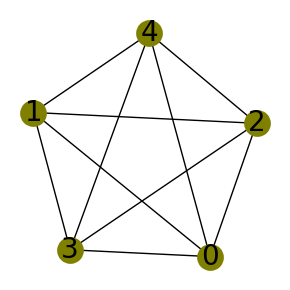
称一个通过图的每条边恰好一次的闭途径为**欧拉环游（Euler tour/circuit）**。如果一个图包含一个欧拉环游，称它是**欧拉的（Eulerain）**。

is\_eulerian(G)：当且仅当图是欧拉的返回True。

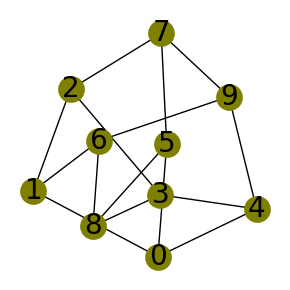
E1=nx.DiGraph({0: [3], 1: [2], 2: [3], 3: [0, 1]})  
G\_drawing(E1)  
print(nx.is\_eulerian(E1))  
E2=nx.complete\_graph(5)  
G\_drawing(E2)  
print(nx.is\_eulerian(E2))  
E3=nx.petersen\_graph()  
G\_drawing(E3)  
print(nx.is\_eulerian(E3))



True



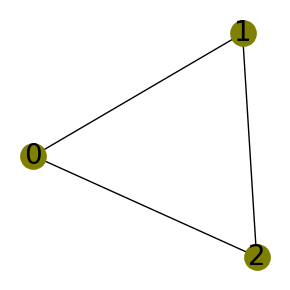
True



False

eulerian\_circuit(G, source=None, keys=False)[19]：返回图中顺序迭代一个欧拉环游所有边。

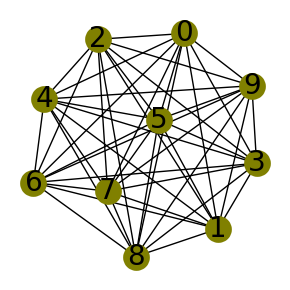
G=nx.complete\_graph(3)  
G\_drawing(G)  
print(list(nx.eulerian\_circuit(G)))  
print(list(nx.eulerian\_circuit(G, source=1)))



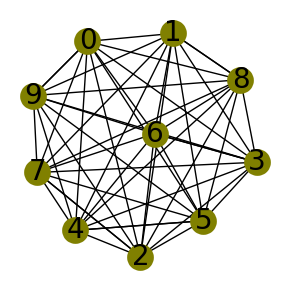
[(0, 2), (2, 1), (1, 0)]  
[(1, 2), (2, 0), (0, 1)]

eulerize(G)[19]：将图G转换为欧拉图（Eulerian graph）。

G=nx.complete\_graph(10)  
G\_drawing(G)  
print(nx.is\_eulerian(G))  
print(G.number\_of\_nodes(),G.number\_of\_edges())  
H=nx.eulerize(G)  
G\_drawing(H)  
print(nx.is\_eulerian(H))  
print(H.number\_of\_nodes(),H.number\_of\_edges())



False  
10 45



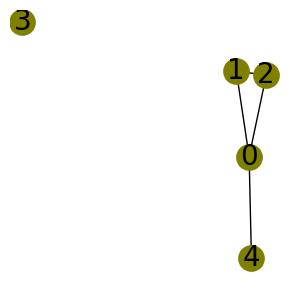
True  
10 50

has\_eulerian\_path(G, source=None)：如果图中有**欧拉路（径）（Eulerian path）**则返回True。欧拉路（径）为通过图的每条边恰好一次的途径，不一定闭合。

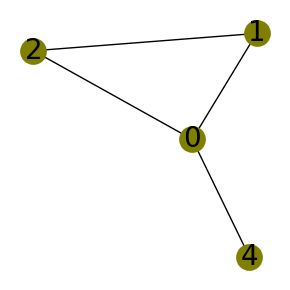
eulerian\_path(G, source=None, keys=False)：返回图中顺序迭代一个欧拉路径所有边。

is\_semieulerian(G)：返回图是否为**半-欧拉（semi-Eulerian）**。半-欧拉是指存在欧拉路径但是没有欧拉环游（Eulerian circuit）。

G=nx.Graph([(0, 1), (1, 2), (0, 2)])  
G.add\_node(3)  
G.add\_edge(0,4)  
G\_drawing(G)  
print(nx.has\_eulerian\_path(G))  
  
G.remove\_nodes\_from(list(nx.isolates(G)))  
G\_drawing(G)  
print(nx.has\_eulerian\_path(G))  
print(nx.has\_eulerian\_path(G,source=0))  
print(nx.has\_eulerian\_path(G,source=2))  
print(nx.is\_semieulerian(G))



False



True  
True  
False  
True

list(nx.eulerian\_path(G))

[(0, 1), (1, 2), (2, 0), (0, 4)]

### 2.8.1.9 若干线性代数知识

设是一个具有个顶点和条边的图，这里和。图的**顶点空间（vertex space）**  
V(G) VF2​   
F2={0,1} V(G) VV V(G) V(G) VU, U’ V U+U’ U ⊆VU=-UØ   
V(G) V(G) dimV(G)=n。

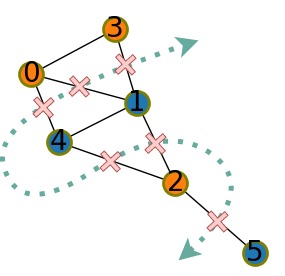
同样地，从到的所有函数构成了的**边空间（edge space）**：的子集对应于它的元素，向量加法采用对称差运算，是零元，且对任意有。同理，是的**标准基（standard basis）**，故。给定边空间中的两个元素和$F’ E*{2}*。

上式等于零当且仅当和有偶数条公共边；特别地，若F≠ Ø ⟨F,F′⟩=0 F⊥:={D∈E(G)∣对所有F∈F满足⟨F,D⟩=0}。。

**圈空间（cycle space）**是由中所有圈（更准确的说，是所有圈的边）支撑的的子空间。的维数有时称为的**圈数（cyclomatic number）**。

如果存在的一个划分，使得，那么称边集是的一个割（cut）；中的边横穿（cross）这个划分；集合和V2 }V1​={v} E(v)G中的极小非空割是一个键。

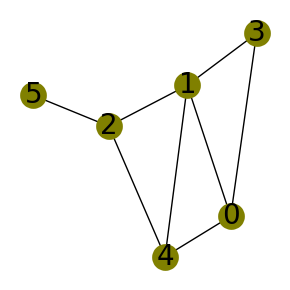
最大割问题（Maximum Cut）是求一种分割方法，将图所有顶点分割成两群，同时使得被切断的边数量最大。当每条边都有权重的时候，则需要保证被切断的边权重和最大。下述计算结果如图示：



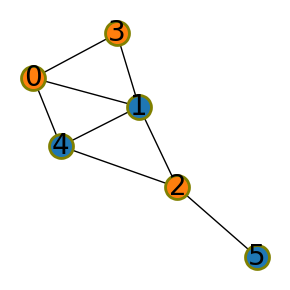
randomized\_partitioning(G[, seed, p, weight])：计算图顶点的随机分割及分割值。返回值cut\_size为最小割值（cut\_size，value of the minimum cut）；partition为定义最小割的顶点划分（partition）。

one\_exchange(G[, initial\_cut, seed, weight])：计算图顶点的分割及分割值。使用贪心交换策略（greedy one exchange strategy）找到局部最大割（locally maximal cut）及其值，添加到当前割并重复此过程，直到无法改进为止。

from networkx.algorithms.approximation.maxcut import randomized\_partitioning  
G=nx.random\_geometric\_graph(6, radius=0.6, seed=3)  
G\_drawing(G)  
print(randomized\_partitioning(G))  
from networkx.algorithms.approximation.maxcut import one\_exchange  
maximal\_cut=one\_exchange(G)  
G\_drawing(G,nodes=maximal\_cut[-1])  
print(maximal\_cut)



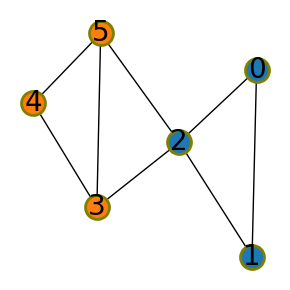
(1, ({5}, {0, 1, 2, 3, 4}))



(6, ({1, 4, 5}, {0, 2, 3}))

cut\_size(G, S, T=None, weight=None)：返回划分两个顶点集割的大小。

G=nx.barbell\_graph(3, 0)  
G.add\_edge(2,5)  
S={0, 1, 2}  
T={3, 4, 5}  
G\_drawing(G,nodes=[S,T])  
nx.cut\_size(G, S, T)

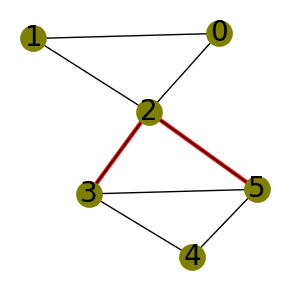


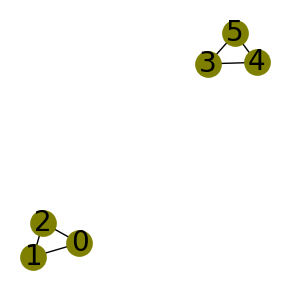
2

minimum\_edge\_cut(G, s=None, t=None, flow\_func=None)[18]：返回断开图最小基数（cardinality）的边集。

cuts=nx.minimum\_edge\_cut(G)  
print(cuts)  
G\_drawing(G,routes=cuts,edge\_colors=['r']\*len(cuts))  
G.remove\_edges\_from(cuts)  
G\_drawing(G)

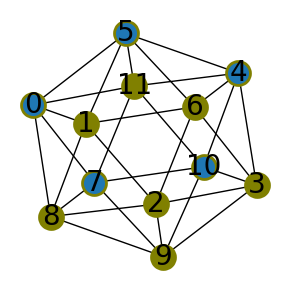
{(2, 3), (2, 5)}



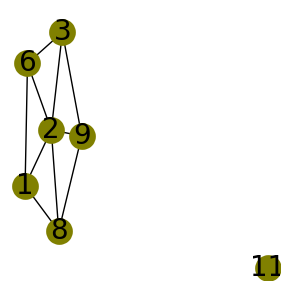


minimum\_edge\_cut(G, s=None, t=None, flow\_func=None)[18]：返回断开图 𝐺 最小基数（cardinality）的顶点集。

G=nx.icosahedral\_graph()  
node\_cut=nx.minimum\_node\_cut(G)  
G\_drawing(G,nodes=[node\_cut])  
print(node\_cut)  
G.remove\_nodes\_from(node\_cut)  
G\_drawing(G)



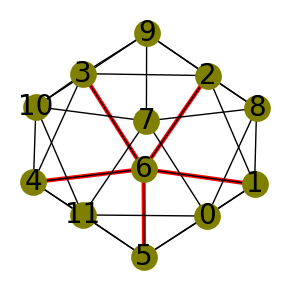
{0, 4, 5, 7, 10}



minimum\_st\_edge\_cut(G, s, t, flow\_func=None, auxiliary=None, residual=None)：返回分割给定源汇顶点的最小基数（cardinality）的边集。

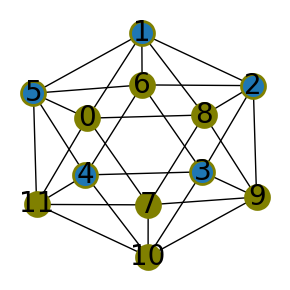
from networkx.algorithms.connectivity import minimum\_st\_edge\_cut  
G=nx.icosahedral\_graph()  
cuts=minimum\_st\_edge\_cut(G, 0, 6)  
print(cuts)  
G\_drawing(G,routes=cuts,edge\_colors=['r']\*len(cuts))

{(4, 6), (2, 6), (5, 6), (3, 6), (1, 6)}

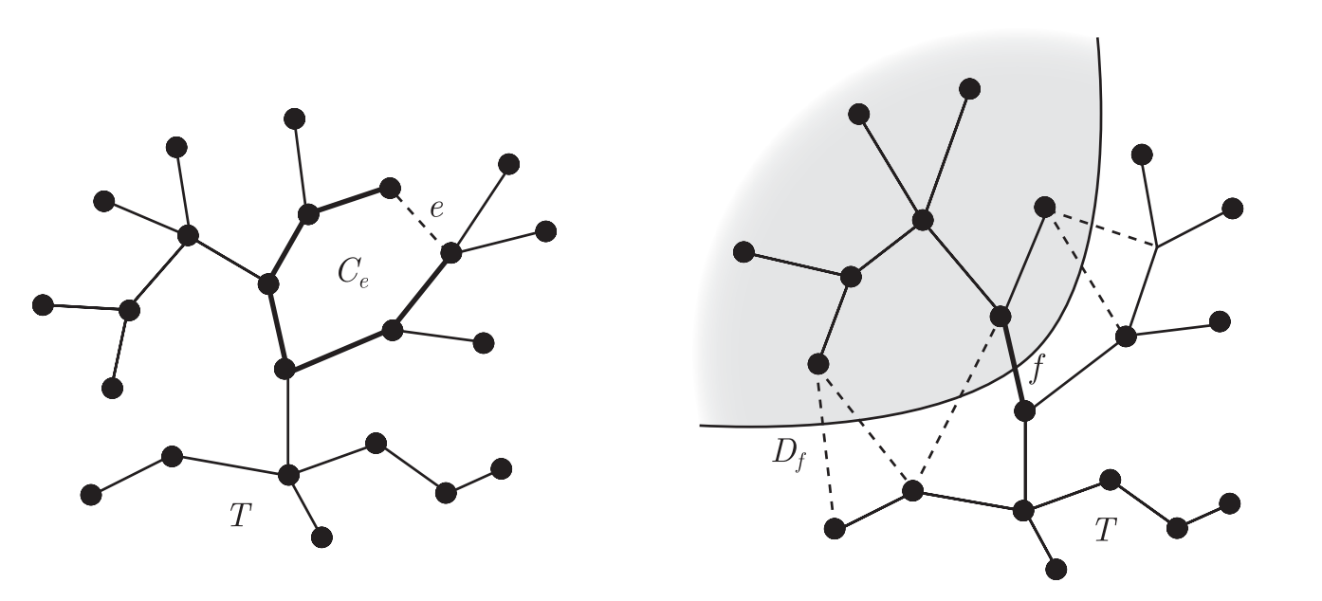


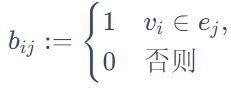
minimum\_st\_node\_cut(G, s, t, flow\_func=None, auxiliary=None, residual=None)minimum\_st\_node\_cut(G, s, t, flow\_func=None, auxiliary=None, residual=None)[18]：返回分割给定源汇顶点的最小基数（cardinality）的顶点集。

from networkx.algorithms.connectivity import minimum\_st\_node\_cut  
G=nx.icosahedral\_graph()  
node\_cut=minimum\_st\_node\_cut(G, 0, 6)  
G\_drawing(G,nodes=[node\_cut])



考虑一个连通图以及它的支撑树。（下图[2]24）对于任意弦，中存在唯一圈Ce CeGTf ∈TT-fDf ∈E GTfe ∈/Tf ∈/T f ∈Ce e∈ Df。意味着存在某种深刻的对偶关系。



设图的顶点集是，而边集是，则它在上的**关联矩阵（incidence matrix）** **

*Bt* BBBt B:E(G)⟶V(G)B:V(G)⟶E(G)BF ⊂EFBt U⊆ VU中的边集合上。

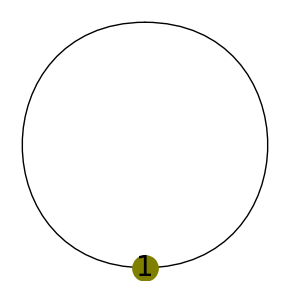
图的**邻接矩阵（adjacency matrix）****

U ⊆VU中有奇数个邻点的顶点集上。

incidence\_matrix(G, nodelist=None, edgelist=None, oriented=False, weight=None)：返回图的关联矩阵。

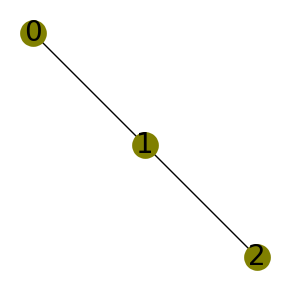
adjacency\_matrix(G, nodelist=None, dtype=None, weight=‘weight’)：返回图的邻接矩阵。

G=nx.Graph([(1, 1)])  
G\_drawing(G)  
A=nx.adjacency\_matrix(G)  
print(A.todense())  
I=nx.incidence\_matrix(G)  
print(I.todense())



[[1]]  
[[0.]]  
  
  
C:\Users\richi\AppData\Local\Temp\ipykernel\_14704\1922076179.py:3: FutureWarning: adjacency\_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.  
 A=nx.adjacency\_matrix(G)  
C:\Users\richi\AppData\Local\Temp\ipykernel\_14704\1922076179.py:5: FutureWarning: incidence\_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.  
 I=nx.incidence\_matrix(G)

G=nx.path\_graph(3)  
G\_drawing(G)  
A=nx.adjacency\_matrix(G)  
print(A.todense())  
I=nx.incidence\_matrix(G)  
print(I.todense())



[[0 1 0]  
 [1 0 1]  
 [0 1 0]]  
[[1. 0.]  
 [1. 1.]  
 [0. 1.]]  
  
  
C:\Users\richi\AppData\Local\Temp\ipykernel\_14704\1125983268.py:3: FutureWarning: adjacency\_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.  
 A=nx.adjacency\_matrix(G)  
C:\Users\richi\AppData\Local\Temp\ipykernel\_14704\1125983268.py:5: FutureWarning: incidence\_matrix will return a scipy.sparse array instead of a matrix in Networkx 3.0.  
 I=nx.incidence\_matrix(G)

NetworkX库提供了to\_pandas\_adjacency和to\_numpy\_matrix方法，可以将图的邻接矩阵直接转换为DataFrame和array（NumPy）的数据格式。

nx.to\_pandas\_adjacency(G)

|  | **0** | **1** | **2** |
| --- | --- | --- | --- |
| **0** | 0.0 | 1.0 | 0.0 |
| **1** | 1.0 | 0.0 | 1.0 |
| **2** | 0.0 | 1.0 | 0.0 |

nx.to\_numpy\_matrix(G)

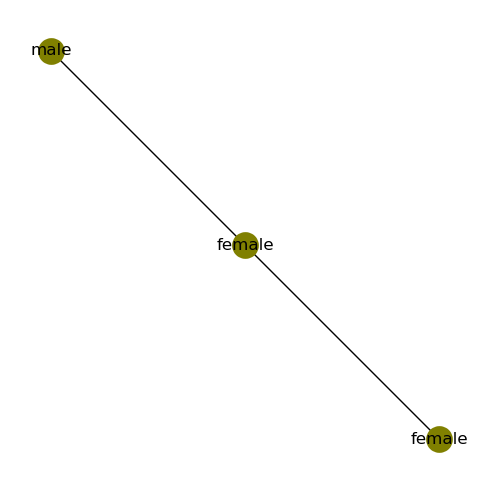
matrix([[0., 1., 0.],  
 [1., 0., 1.],  
 [0., 1., 0.]])

* Mixing（混合）

attribute\_mixing\_matrix(G, attribute[, …])： 返回顶点属性连通的混合矩阵，如果normalized为False，返回的为属性对出现次数，例如出现次数为0，出现次数为1，出现次数为1，出现次数为2；为True则返回的为属性对出现的联合概率（ joint probability）。

numeric\_mixing\_matrix(G, attribute, nodes=None, normalized=True, mapping=None)方法将被移除，直接用attribute\_mixing\_matrix方法，传入属性参数。

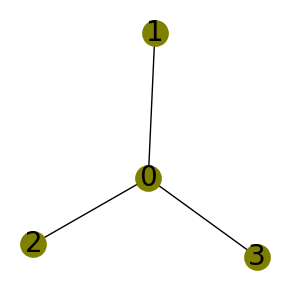
G=nx.path\_graph(3)  
gender={0: 'male', 1: 'female', 2: 'female'}  
nx.set\_node\_attributes(G, gender, 'gender')  
G\_drawing(G,node\_labels='gender',figsize=(5,5))  
mapping={'male': 0, 'female': 1}  
mix\_mat=nx.attribute\_mixing\_matrix(G, 'gender', mapping=mapping,normalized=False)  
# mixing from male nodes to female nodes  
print(mix\_mat)  
print(mix\_mat[mapping['male'], mapping['female']])  
print(nx.attribute\_mixing\_matrix(G, 'gender', mapping=mapping))



[[0. 1.]  
 [1. 2.]]  
1.0  
[[0. 0.25]  
 [0.25 0.5 ]]

degree\_mixing\_matrix(G[, x, y, weight, …])： 返回顶点度连通的混合矩阵，例如下图顶点度只有1和3两种情况，度为1的顶点连通到度为3的顶点次数为3，度为3的顶点连通到度为1的顶点次数为3，无其它情况，因此矩阵其它位置为0。

G=nx.star\_graph(3)  
mix\_mat=nx.degree\_mixing\_matrix(G,normalized=False)  
G\_drawing(G)  
print(mix\_mat)  
mix\_mat[0, 1] # mixing from node degree 1 to node degree 3



[[0. 3.]  
 [3. 0.]]  
  
  
  
  
  
3.0

可以使用mapping参数，显示所有的度，即使度值不存在于图中，例如度为0、1、2、3的混合矩阵。

max\_degree=max(deg for n, deg in G.degree)  
mapping={x: x for x in range(max\_degree + 1)} # identity mapping  
mix\_mat=nx.degree\_mixing\_matrix(G, mapping=mapping,normalized=False)  
print(mix\_mat)  
mix\_mat[3, 1] # mixing from node degree 3 to node degree 1

[[0. 0. 0. 0.]  
 [0. 0. 0. 3.]  
 [0. 0. 0. 0.]  
 [0. 3. 0. 0.]]  
  
  
  
  
  
3.0

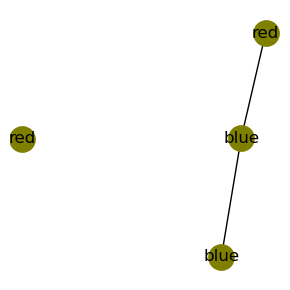
degree\_mixing\_dict(G, x=‘out’, y=‘in’, weight=None, nodes=None, normalized=False)：返回以字典形式存储的顶点度连通次数或者联合概率。

nx.degree\_mixing\_dict(G)

{3: {1: 3}, 1: {3: 3}}

attribute\_mixing\_dict(G, attribute[, nodes, …])：返回字典表示的顶点属性混合矩阵。

G=nx.Graph()  
G.add\_nodes\_from([0, 1], color="red")  
G.add\_nodes\_from([2, 3], color="blue")  
G.add\_edges\_from([(1, 3),(2,3)])  
G\_drawing(G,node\_labels='color')  
d=nx.attribute\_mixing\_dict(G, "color")  
print(d)  
print(d["red"]["blue"])  
print(d["blue"]["red"]) # d symmetric for undirected graphs



{'red': {'blue': 1}, 'blue': {'blue': 2, 'red': 1}}  
1  
1

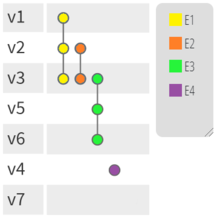
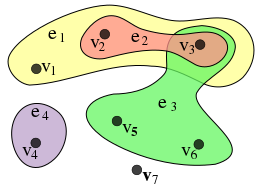
mixing\_dict(xy[, normalized])：给定元组列表的方式计算“顶点”的混合矩阵。

nx.mixing\_dict([(1,2),(3,4),(2,1),(3,4)])

{1: {2: 1}, 2: {1: 1}, 3: {4: 2}, 4: {}}

### 2.8.1.10 图中的其它概念

**超图（hypergraph）**是一对不相交的集合，其中的元素是的（具有任意基数的）非空子集。因此，图是一种特殊的超图。



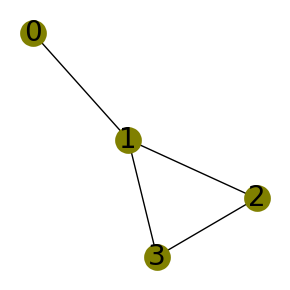
上图（左图一般描绘，右图称为PAOH描绘）②为一个无向超图的例子，其中集合， E={ e1, e2, e3, e4 }={v1,v2,v3},{v2,v3},{v3,v4,v5},{v4}}。这个超图的阶数（order）为7，大小（size）为4。

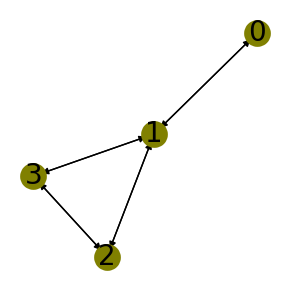
**有向图（directed graph or digraph）**是由一对不相交的集合（分别称作顶点和边）以及两个映射init: *E*⟶*V* ter: *E*⟶*V* initeinit(e)tereter(e)einit(e)ter(e)xyxyinit(e)=ter(e)e叫做**环边（loop）**。

对于有向图和（无向）图，如果，且对每条边有，则称是的一个**定向（orientation）**。直观的看，一个**定向图（oriented graph）**就是把一个无向图的每条边从一个端点到另一个端点给出方向而得到的图，也可以把定向图看作没有重边和环边的有向图。

to\_directed(graph)：返回一个图为有向图。具有相同名称、顶点，且每条边被两条有向边和$ (v, u, data)$替换。类似的方法还有DiGraph.to\_directed(as\_view=False)，MultiGraph.to\_directed(as\_view=False)等。

G=nx.Graph()  
G.add\_edges\_from([(0, 1),(1,2),(2,3),(1,3)])  
G\_drawing(G)  
D=G.to\_directed()  
G\_drawing(D)





**多重图（multigraph）**是由一对不相交的集合（称为顶点和边）以及从到V∪[V]2的一个映射组成的，这里映射给每条边指定一个或两个顶点（叫做端点（end））。所以，多重图可以有重边和环边（或叫做**双边（double edge）**）。

注释（Notes）：

① NetworkX，（[https://networkx.org/documentation/stable/index.html#](https://networkx.org/documentation/stable/index.html)）。

② Hypergraph(Wikipedia)，（<https://en.wikipedia.org/wiki/Hypergraph>）。

参考文献（References）:

[1] Estrada, E. & Rodríguez-Velázquez, J. A. Spectral measures of bipartivity in complex networks. Phys Rev E Stat Nonlin Soft Matter Phys 72, (2005).

[2] [德]Reinhard Diestel著, 于青林译.图论（第五版）[M].北京: 科学出版社, 2020.04.

[3] Boppana, R., & Halldórsson, M. M. (1992). Approximating maximum independent sets by excluding subgraphs. BIT Numerical Mathematics, 32(2), 180–196. Springer.

[4] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, “An Improved Algorithm for Matching Large Graphs”, 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen, pp. 149-159, 2001. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.5342

[5] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, “The architecture of complex weighted networks”. PNAS 101 (11): 3747–3752 (2004).

[6] “An algorithm for computing simple k-factors.”, Meijer, Henk, Yurai Núñez-Rodríguez, and David Rappaport, Information processing letters, 2009.

[7] Jin Y. Yen, “Finding the K Shortest Loopless Paths in a Network”, Management Science, Vol. 17, No. 11, Theory Series (Jul., 1971), pp. 712-716.

[8] R. Sedgewick, “Algorithms in C, Part 5: Graph Algorithms”, Addison Wesley Professional, 3rd ed., 2001.

[9] Paton, K. An algorithm for finding a fundamental set of cycles of a graph. Comm. ACM 12, 9 (Sept 1969), 514-518.

[10] Finding all the elementary circuits of a directed graph. D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975. https://doi.org/10.1137/0204007

[11] Enumerating the cycles of a digraph: a new preprocessing strategy. G. Loizou and P. Thanish, Information Sciences, v. 27, 163-182, 1982.

[12] A search strategy for the elementary cycles of a directed graph. J.L. Szwarcfiter and P.E. Lauer, BIT NUMERICAL MATHEMATICS, v. 16, no. 2, 192-204, 1976.

[13] Pearl, J. (2009). Causality. Cambridge: Cambridge University Press.

[14] Darwiche, A. (2009). Modeling and reasoning with Bayesian networks. Cambridge: Cambridge University Press.

[15] Shachter, R. D. (1998). Bayes-ball: rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In , Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (pp. 480–487). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[16] Koller, D., & Friedman, N. (2009). Probabilistic graphical models: principles and techniques. The MIT Press.

[17] White, Douglas R., and Mark Newman. 2001 A Fast Algorithm for Node-Independent Paths. Santa Fe Institute Working Paper #01-07-035 http://eclectic.ss.uci.edu/~drwhite/working.pdf

[18] Abdol-Hossein Esfahanian. Connectivity Algorithms. http://www.cse.msu.edu/~cse835/Papers/Graph\_connectivity\_revised.pdf

[19] J. Edmonds, E. L. Johnson. Matching, Euler tours and the Chinese postman. Mathematical programming, Volume 5, Issue 1 (1973), 111-114.