

# 14

## Fundamentals of Graph Theory

# 图论入门

由一组节点和连接节点的边组成结构



从某种意义上说，数学是逻辑思想的诗歌。

*Mathematics is, in its way, the poetry of logical ideas.*

—— 阿尔伯特·爱因斯坦 (Albert Einstein) | 理论物理学家 | 1879 ~ 1955



- networkx.DiGraph() 创建有向图的类，用于表示节点和有向边的关系以进行图论分析
- networkx.draw\_networkx() 用于绘制图的节点和边，可根据指定的布局将图可视化呈现在平面上
- networkx.draw\_networkx\_edge\_labels() 用于在图可视化中绘制边的标签，显示边上的信息或权重
- networkx.get\_edge\_attributes() 用于获取图中边的特定属性的字典，其中键是边的标识，值是对应的属性值
- networkx.Graph() 创建无向图的类，用于表示节点和边的关系以进行图论分析
- networkx.MultiGraph() 创建允许多重边的无向图的类，可以表示同一对节点之间的多个关系
- networkx.random\_layout() 用于生成图的随机布局，将节点随机放置在平面上，用于可视化分析
- networkx.spring\_layout() 使用弹簧模型算法将图的节点布局在平面上，模拟节点间的弹簧力和斥力关系，用于可视化分析
- networkx.to\_numpy\_matrix() 用于将图表示转换为 NumPy 矩阵，方便在数值计算和线性代数操作中使用

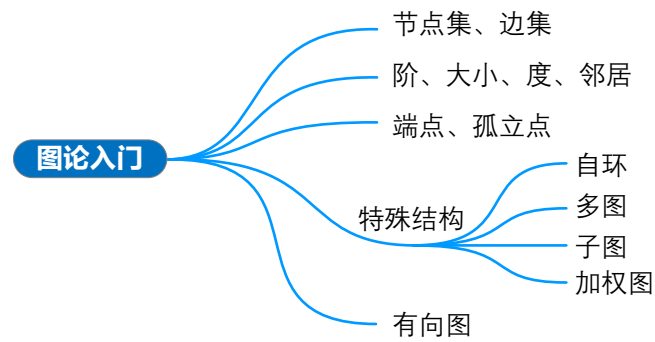
本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)



# 14.1 什么是图？

## 图

**图论** (Graph Theory) 是数学的一个分支，研究的是图的性质和图之间的关系。图由节点 (节点) 和边组成，节点表示对象，边表示对象之间的关系。

历史上，图论起源于 18 世纪，数学家**欧拉** (Leonhard Euler) 最先提出解决了七桥问题的数学方法，开创了图论的先河。

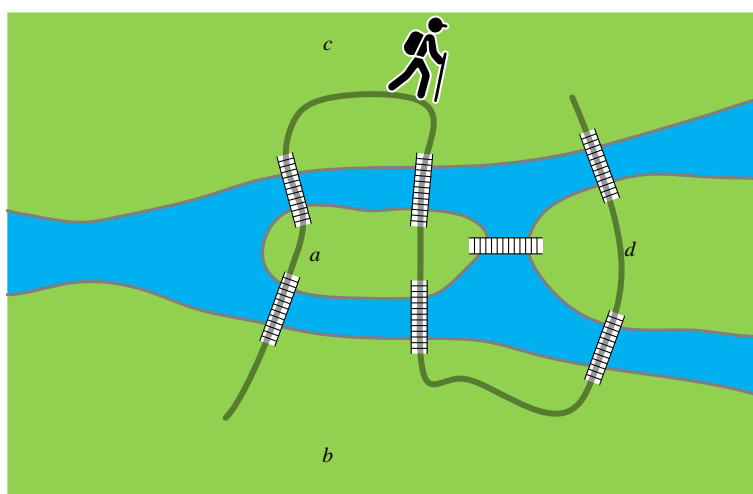


图 1. 七桥问题，走遍图中七座桥，每座桥只经过一次

**柯尼斯堡七桥问题** (Seven Bridges of Königsberg)，简称七桥问题，的背景是**基尔岛** (Königsberg) 的**普雷格尔河** (Pregel River) 上有两座岛 (a、d)，有 7 座桥将两座岛和两岸 (b、c) 相连。问题是能否走遍这七座桥，每座桥只经过一次。

欧拉解决这个问题的方法是抽象化。他将问题中的地理元素简化成**节点** (nodes) 和**边** (edge) 的**图** (graph)。节点也称**顶点** (vertex 复数 vertices)。

**⚠ 注意**，本书一般采用节点这一表达，目的是和 NetworkX 统一。

每座桥成为图中的一条边，每个岸上的土地成为一个节点。这样，问题就转变成了在这个图上找一条路径，经过每条边一次且仅一次。这就是所谓的“一笔画问题”，即 Eulerian path。

欧拉将问题抽象化，引入了图论的概念，奠定了图论这一数学分支的基础。他的方法和思想对后来图论和网络理论的发展产生了深远的影响。

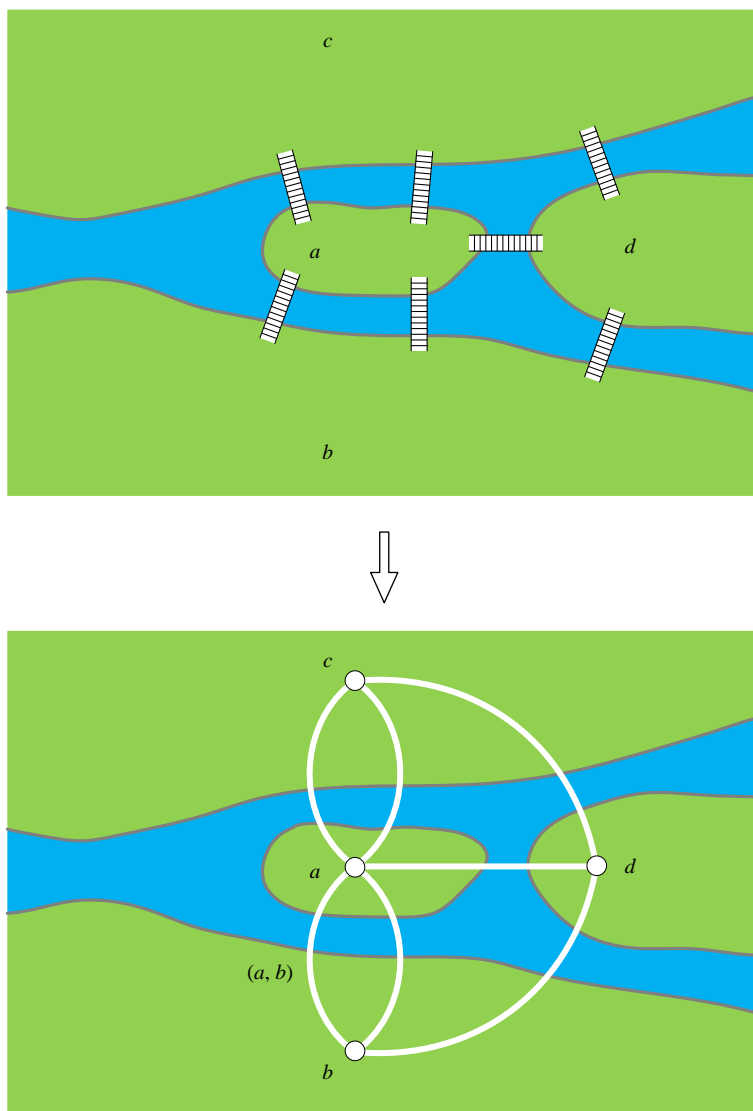


图 2. 七桥问题

## 机器学习中的图

在机器学习中，图论常用于表示和分析数据之间的关系。图模型可以用来建模复杂的关联关系，尤其在结构化数据和网络数据方面。例如，社交网络中的用户关系可以被建模成一个图，其中节点表示用户，边表示用户之间的连接。这种图结构有助于分析信息传播、社交网络分析等问题。

在深度学习中，**图神经网络** (Graph Neural Networks, GNNs) 是图论的一种扩展，专门用于处理图结构数据。GNNs 能够在图上进行节点和边的信息传递，使得模型能够理解节点之间的复杂关系。

至于**大语言模型** (Large Language Model, LLM)，图论在自然语言处理中的应用主要体现在语言结构的建模上。语言结构可以被视为一个图，其中单词或子词是节点，语法和语义关系则是边。通过图模型，大语言模型可以更好地理解语言的层次结构和关联关系，从而提高对文本理解和生成的能力。

## 无向图

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

**无向图** (undirected graph) 是一种图，它的边没有方向。节点之间的连接是双向的，没有箭头指示方向。无向图常用于描述简单的关系，如社交网络中的朋友关系。

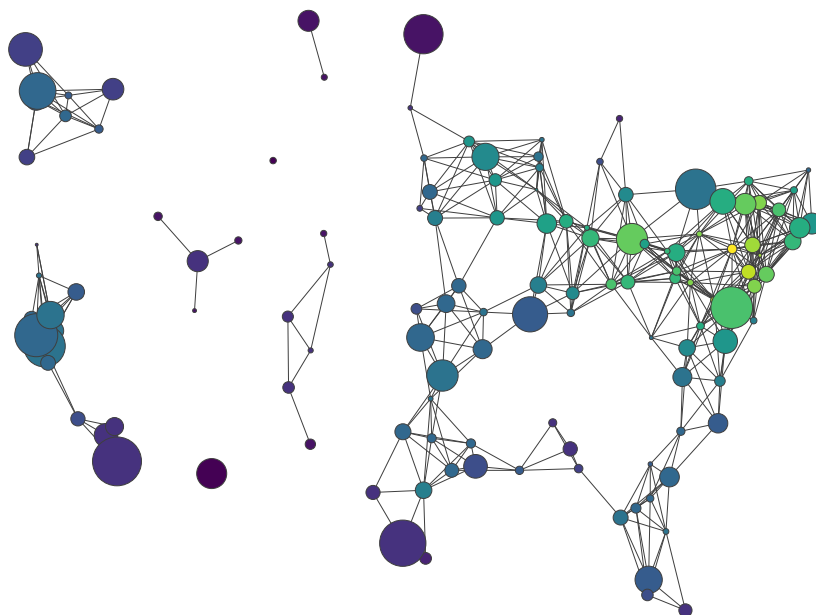


图 3. 128 个美国城市人口和距离组成的无向图

简单来说，无向图由节点集合和边集合构成，其中节点集合表示图中的元素，边集合定义了连接这些节点的关系。如图 4 所示，无向图就好比按特定方式布置的人行步道，任意两个节点并不限制通行方向。

在无向图中，边无权重意味着连接节点的边没有相关的数值信息。在无权重无向图中，通常使用 0 和 1 表示边的存在或不存在。

具体而言，如果节点之间有边相连，则用 1 表示，否则用 0 表示。而有权重无向图的边有关联的数值，这些数值可以是距离、相似度、相关性系数等等。

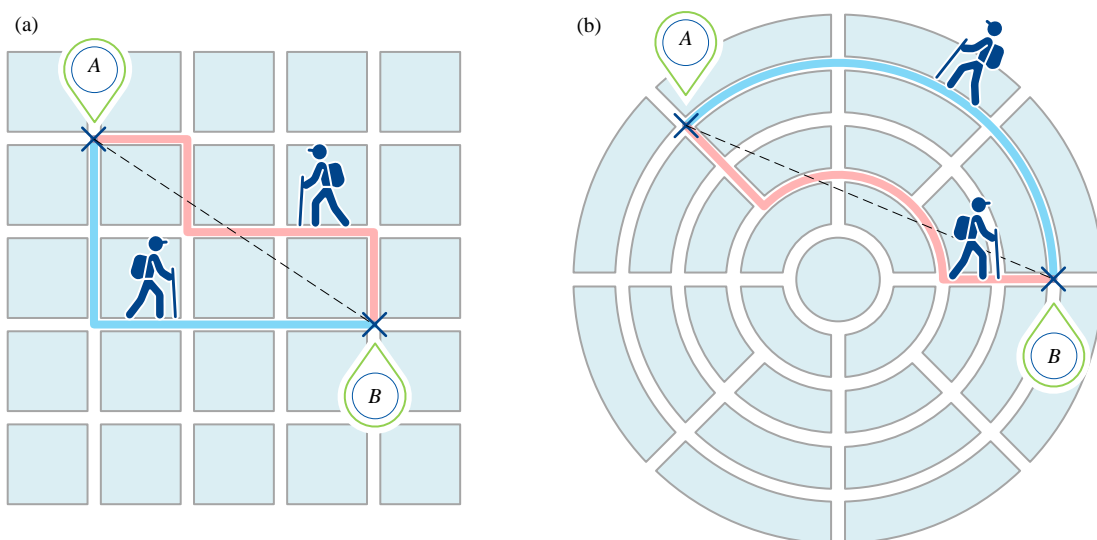


图 4. 不同方法布置的步道

## 有向图

**有向图** (directed graph) 则是边有方向的图，每条边从一个节点指向另一个节点。有向图常用于描述有向关系，例如网页之间的链接、任务执行的顺序等。

顾名思义，边有方向的图就是**有向图** (directed graph, digraph)。在图 4 的步道中任意两个节点规定通行方向，我们便得到了有向图。

生活中，有向图无处不在。图 5 所示的陆地物质能量流动链条就可以抽象成有向图。

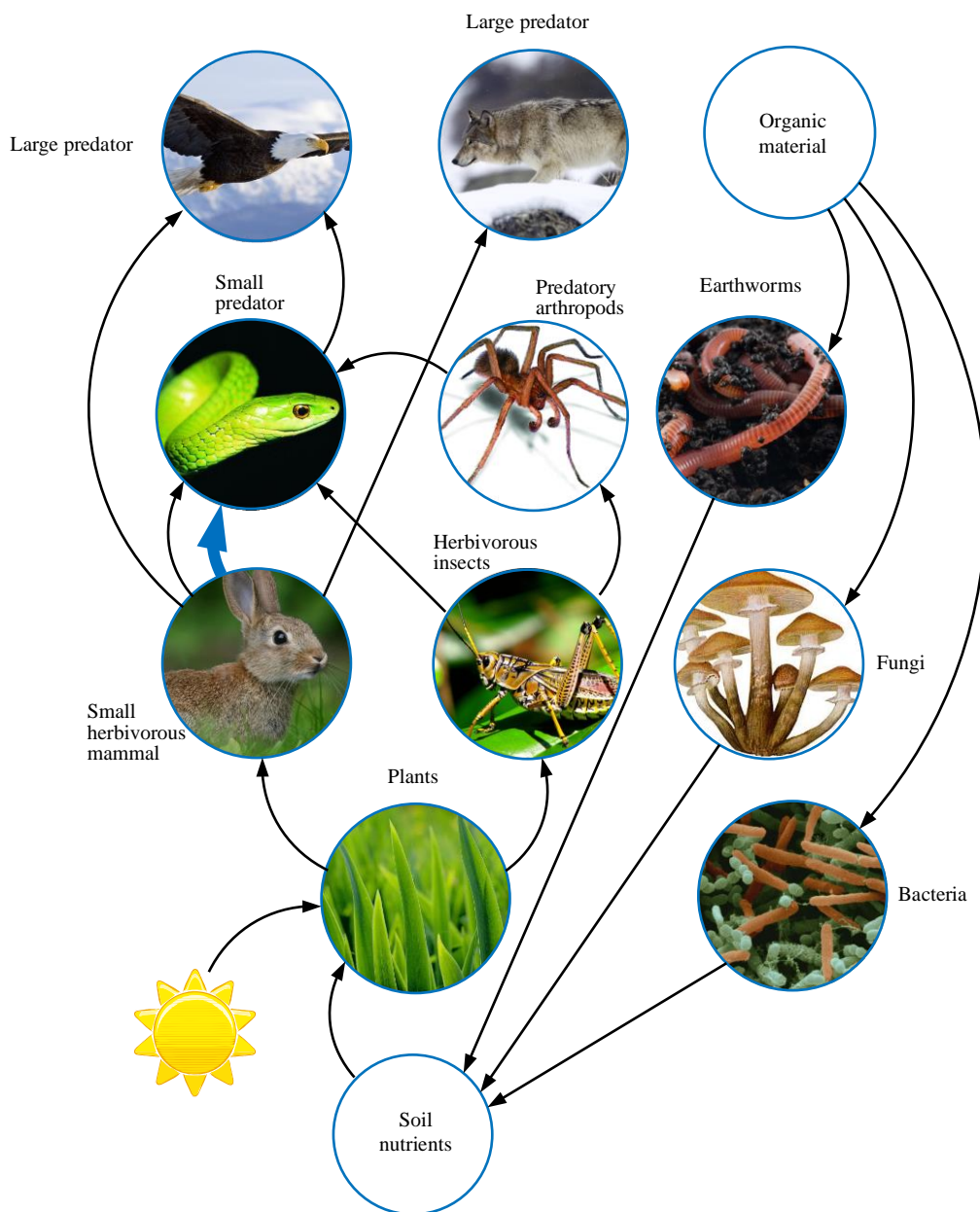


图 5. 食物链中物质能量流动链条具有方向性

## NetworkX

NetworkX 是一个用 Python 编写的图论和复杂网络分析的开源软件包。它提供了创建、操作和研究复杂网络结构的工具。以下是一些 NetworkX 的主要特点和用途。

NetworkX 允许用户轻松创建各种类型的图，包括无向图、有向图、加权图等。它提供了丰富的图操作和算法，使用户能够对图进行修改、查询和分析。

NetworkX 支持图的可视化，可以使用各种布局算法将图形绘制成可视化图形。这有助于直观地理解和展示复杂网络的结构。

NetworkX 包含许多图算法，涵盖了图的各个方面，如最短路径、连通性、中心性度量等。用户可以利用这些算法来分析和研究图的特性。

除了基本的图操作和算法外，NetworkX 还提供了用于复杂网络分析的工具。这包括社区检测、小世界网络、度分布等分析方法。

NetworkX 支持从多种数据源导入图数据，也可以将图数据导出为不同的格式，如 GML、GraphML、JSON 等。

由于 NetworkX 是用 Python 编写的，因此具有很高的灵活性和可扩展性。用户可以方便地自定义算法和功能，以满足特定的需求。

本书下面会结合 NetworkX 介绍图论基础内容，并用 NetworkX 构造并求解一些和图论相关的常见数学问题。

## 14.2 无向图：边没有方向

### 节点集、边集

将图 6 中的无向图记做  $G$ 。一个图有两个重要集合：(1) 节点集  $V(G)$ ；(2) 边集  $E(G)$ 。因此， $G$  也常常被写成  $G = (V, E)$ 。

以图 6 的图  $G$  为例， $G$  的节点集  $V(G)$  为：

$$V(G) = \{a, b, c, d\} \quad (1)$$

$G$  的边集  $E(G)$  为：

$$E(G) = \{ab, bc, bd, cd, ca\} = \{(a, b), (b, c), (b, d), (c, d), (c, a)\} \quad (2)$$

上式的第二种集合记法是为了配合 NetworkX 语法。

由于图 6 中图  $G$  是无向图，因此节点  $a$  到节点  $b$  的边  $ab$ ，和节点  $b$  到节点  $a$  边  $ba$ ，没有区别。但是，下一章介绍有向图时，我们就需要注意连接节点的先后顺序了。

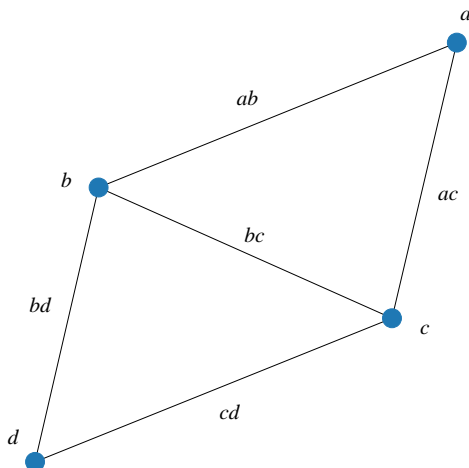


图 6.4 个节点，5 条边的无向图

图 6 是用 NetworkX 绘制，下面聊聊代码 1。

```

import matplotlib.pyplot as plt
import networkx as nx

a undirected_G = nx.Graph()
# 创建无向图的实例

b undirected_G.add_node('a')
# 添加单一顶点

c undirected_G.add_nodes_from(['b', 'c', 'd'])
# 添加多个顶点

d undirected_G.add_edge('a', 'b')
# 添加一条边

e undirected_G.add_edges_from([('b', 'c'),
                                ('b', 'd'),
                                ('c', 'd'),
                                ('c', 'a')])
# 增加一组边

f random_pos = nx.random_layout(undirected_G, seed=188)
# 设定随机种子，保证每次绘图结果一致

g pos = nx.spring_layout(undirected_G, pos=random_pos)
# 使用弹簧布局算法来排列图中的节点
# 使得节点之间的连接看起来更均匀自然

h plt.figure(figsize = (6,6))
nx.draw_networkx(undirected_G, pos = pos,
                  node_size = 180)
plt.savefig('G_4顶点_5边.svg')

```

代码 1. 用 NetworkX 绘制无向图 | Bk6\_Ch14\_01.ipynb

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)



❶ 用 `networkx.Graph()` 创建一个空的无向图对象实例。在这个实例中，我们可以添加节点和边，进行图的各种操作和分析。

注意，一个没有边的图叫做**空图** (empty graph)。也就是说，一个非空图至少要有一条边。

❷ 用 `add_node()` 方法增加单一节点 `a`。

注意，只有一个节点的图叫做**平凡图** (trivial graph)。

❸ 用 `add_nodes_from()` 方法增加另外三个节点，这三个节点以列表形式保存 `['b', 'c', 'd']`。

❹ 用 `add_edge()` 方法向图中添加一条连接节点 `'a'` 和 `'b'` 的无向边。

❺ 用 `add_edges_from()` 方法向图中添加一组无向边，连接 `'b'` 与 `'c'`，`'b'` 与 `'d'`，`'c'` 与 `'d'`，`'c'` 与 `'a'`。

❻ 用 `networkx.random_layout()` 设定随机种子值，以确保每次可视化采用相同的布局。

❼ 用 `networkx.spring_layout()` 弹簧布局算法来排列图中的节点。

❽ 用 `networkx.draw_networkx()` 绘制无向图，传入图 `undirected_G`、节点的位置信息 `pos`、节点的大小 `node_size`。前文在设定随机数种子时，`pos` 已经固定下来；如果没有传入 `pos`，每次运行可视化得到的图外观会随机变化（但是图的基本性质不变）。

图8所示为供大家在 NetworkX 练习的几个无向图，请大家注意对每个图用不同的命名。

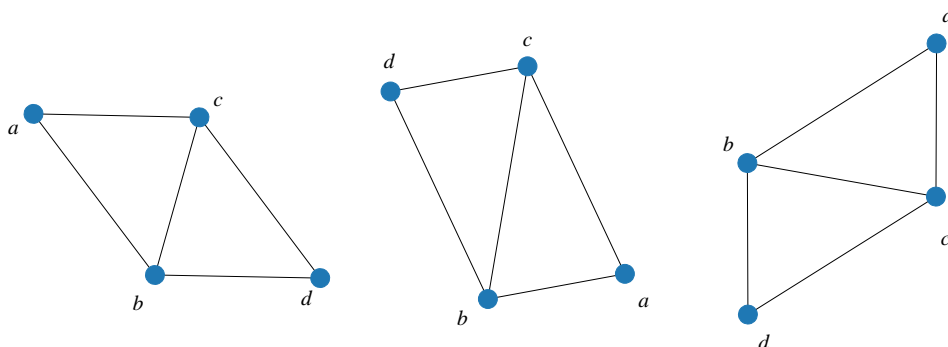


图 7. 图随机布局

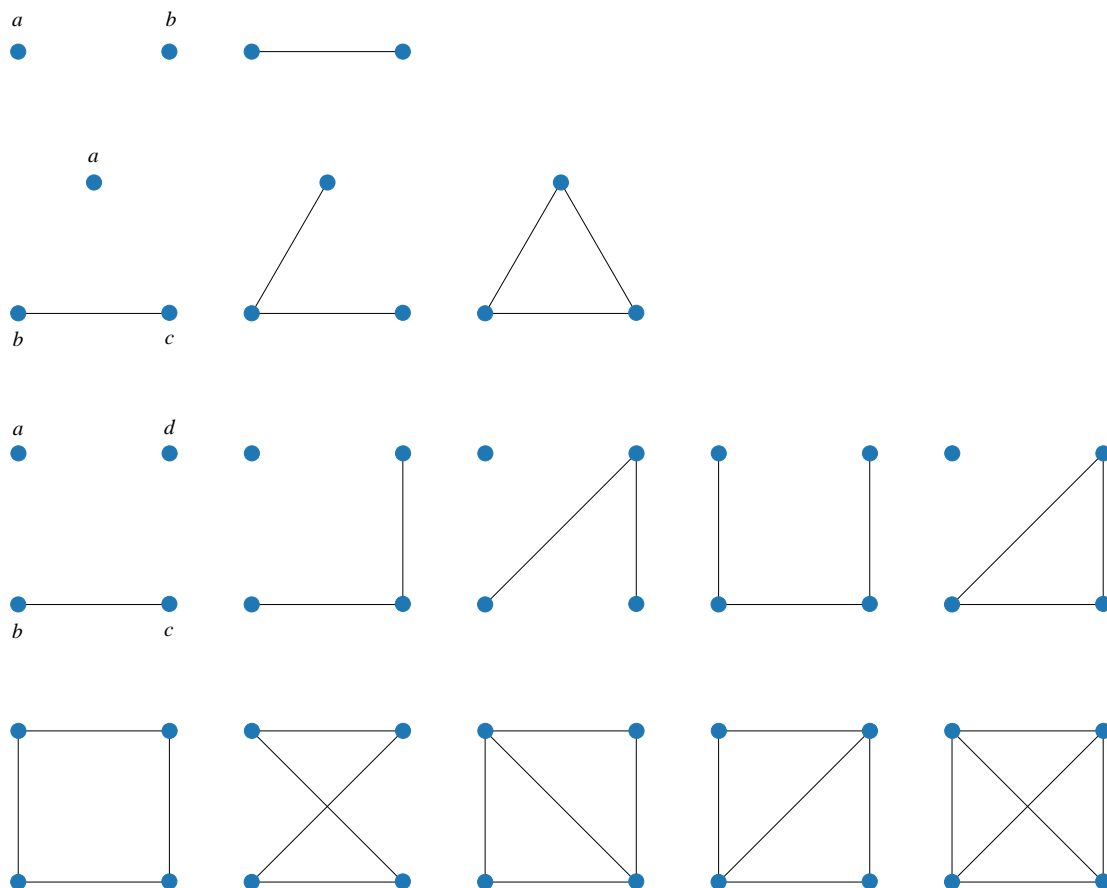


图 8. 供大家练习的几个无向图

### 阶、大小、度、邻居

图  $G$  的节点数量叫做**阶** (order)，常用  $n$  表示。图 6 所示的图  $G$  的阶为 4 ( $n = 4$ )，也就是说  $G$  为 4 阶图。

图  $G$  的边的数量叫做图的**大小** (size)，常用  $m$  表示。图 6 所示的图  $G$  的大小为 5 ( $m = 5$ )。

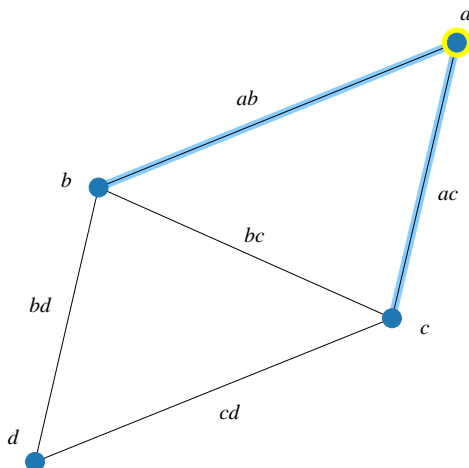
对于无向图，一个节点的**度** (degree) 是与它相连的边的数量。

比如，如图 9 所示，图  $G$  中节点  $a$  的度为 2，记做  $\deg_G(a) = 2$ 。图  $G$  中节点  $b$  的度为 3，记做  $\deg_G(b) = 3$ 。

无向图中，给定一个节点的**邻居** (neighbors) 指的是与该节点直接相连的其他节点。简单来说，如果两个节点之间存在一条边，那么它们就互为邻居。

如图 9 所示，节点  $a$  有两个邻居—— $b$ 、 $c$ 。

如果一个图有  $n$  个节点，那么其中任意节点最多有  $n - 1$  个邻居，它的度最大值也是  $n - 1$ 。注意，这是在不考虑自环的情况下！本章马上介绍自环有关内容。

图 9. 节点  $a$  的度为 2，有 2 个邻居

在代码 1 基础上，代码 2 计算阶、大小、度、邻居等值。下面聊聊这段代码。

```

a undirected_G.order()
# 图的阶

b undirected_G.number_of_nodes()
# 图的节点数

c undirected_G.nodes
# 列出图的节点

d undirected_G.size()
# 图的大小

e undirected_G.edges
# 列出图的边

f undirected_G.number_of_edges()
# 图的边数


g undirected_G.has_edge('a', 'b')
# 判断是否存在ab边
# 结果为 True

h undirected_G.has_edge('a', 'd')
# 判断是否存在ad边
# 结果为 False

i undirected_G.degree()
# 图的度

j list(undirected_G.neighbors('a'))
# 邻居

```

代码 2. 用 NetworkX 计算无向图的阶、大小、度、邻居 |  Bk6\_Ch14\_01.ipynb

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger：<https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

- a 用 `order()` 方法计算无向图的阶，即图中节点总数。
- b 用 `number_of_nodes()` 方法计算无向图中节点数量，结果与阶相同。
- c 用 `nodes` 列出无向图所有节点。
- d 用 `size()` 方法计算了图的大小，即无向图中边数总和。
- e 用 `edges` 列出无向图所有的边。
- f 计算了无向图的边数。
- g 用 `has_edge()` 判断无向图中是否存在连接节点 `a` 和 `b` 的边，结果为 `True` 表示存在。
- h 用 `has_edge()` 判断无向图中是否存在连接节点 `a` 和 `d` 的边，结果为 `False` 表示不存在。
- i 用 `degree()` 方法计算无向图的度。结果列出所有节点的各自度。  
用 `dict(undirected_G.degree())` 可以将结果转化为字典 `dict`。  
也可以用 `undirected_G.degree('a')` 计算某个特定节点，比如 `a`，的度。
- j 用 `neighbors()` 方法查找特定节点的邻居，结果是可迭代键值对；用 `list()` 将结果转化为列表。

请大家也计算图 8 中每幅图的阶、大小、度、邻居等值。

### 端点、孤立点

度数为 1 的节点叫**端点** (end vertex, end node)，如图 10 (a) 所示。

我们可以用 `remove_edge()` 方法删除 `ac` 这条边，比如 `undirected_G.remove_edge('c','a')`。

度数为 0 的节点叫**孤立点** (isolated node, isolated vertex)，如图 10 (b) 所示。

请大家指出图 8 中每幅图可能存在的端点和孤立点。

我们可以用 `undirected_G.remove_edges_from([('b','a'),('a','c')])` 方法删除两条边。

类似地，我们可以用 `undirected_G.remove_node('a')` 从 `undirected_G` 图上删除一个节点；或者用 `undirected_G.remove_nodes_from(['b','a'])` 删除若干节点。

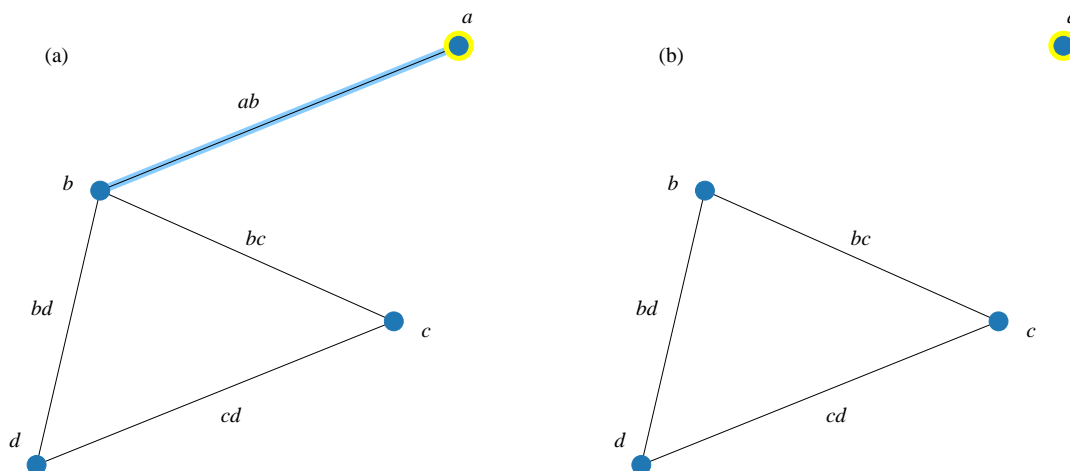


图 10. 端点和孤立点

## 自环

在图论中，一个节点到自身的边被称为**自环** (self-loop)，自环边。简单来说，如图 11 所示，自环就是图中节点  $a$  与它自己之间存在一条边。

这时候，图 11 的大小变为 6；因为在原来 5 条边的基础上，又增加一条边  $aa$ 。

特别请大家注意，这时候节点  $a$  的度从 2 变成了 4。当某个节点增加自环时，它的度将增加 2，因为自环会导致节点与自己连接两次，每次连接都增加了节点的度。自环的存在使得节点的度增加了 2，而不是 1。

从图 11 图上来看，节点伸出了 4 根“触须”。

多了自环，节点  $a$  的邻居变成了 3 个—— $a$ 、 $b$ 、 $c$ 。

也就是说，考虑自环的情况下，如果一个图有  $n$  个节点，那么其中任意节点最多有  $n$  个邻居，它的度最大值也是  $n + 1$ 。

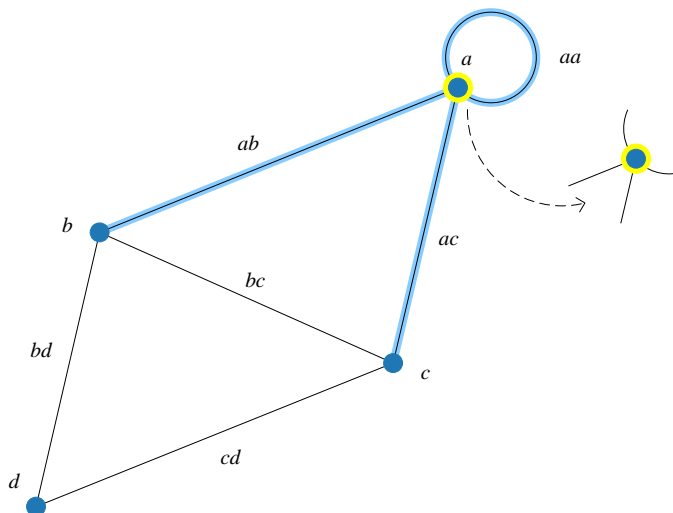


图 11. 节点  $a$  增加自环

在前文代码基础上，代码 3 添加节点  $a$  的自环，并计算大小、度、邻居具体值。请大家自行分析这段代码，并运行结果。

```

a undirected_G.add_edge('a', 'a')
  # 添加一条自环

b # 可视化
  plt.figure(figsize = (6,6))
  nx.draw_networkx(undirected_G, pos = pos,
c                      node_size = 180)
  plt.savefig('G_4顶点_5边_a自环.svg')


d undirected_G.size()
  # 图的大小

e undirected_G.edges
  # 列出图的边

f undirected_G.degree('a')
  # 节点a的度

g list(undirected_G.neighbors('a'))
  # 邻居

```

代码 3. 节点 *a* 增加自环后计算无向图的大小、度、邻居 |  Bk6\_Ch14\_01.ipynb

## 14.3 多图

观察图 2 下图，我们会发现一个有趣的现象——连接两个节点的边可能不止一条！我们管这种图叫**多图** (multigraph)。

在图论中，一个多图是一种图的扩展形式，允许在同一对节点之间存在多条边。**普通图** (simple graph) 中，任意两个节点之间只能有一条边。注意，严格来说，普通图也不能有自环。

多图的定义允许图中存在**平行边** (parallel edge)，也叫**重边**，即连接相同两个节点的多条边。

如图 12 所示，在多图中，两个节点之间可以有多条边，每条边可能具有不同的权重或其他属性。对于本书前文介绍的七桥问题，显然平行边代表不同位置的桥。

多图的概念对于某些应用很有用，例如网络建模、流量分析等。在多图中，我们可以更灵活地表示节点之间复杂的关系，以及在同一对节点之间可能存在不同类型或性质的连接。

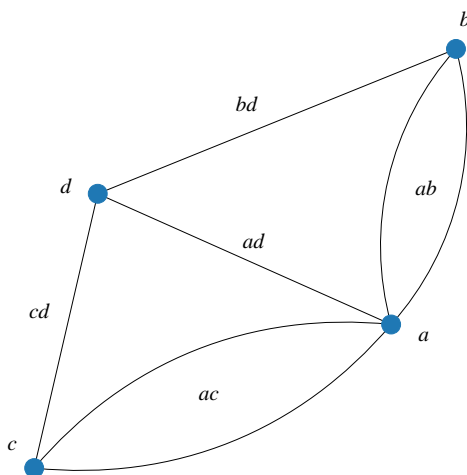


图 12. 多图

很明显节点  $a$ 、 $b$  之间的边数为 2，节点  $a$ 、 $c$  之间的边数也是 2。

代码 4 定义并可视化多图。请大家自行分析这段代码。

值得注意的是目前 `networkx.draw_networkx()` 函数还不能很好呈现多图中的平行边。图 12 中带有弧度的平行边是 NetworkX 出图后再处理的结果。在 StackOverflow 中可以找到几种解决方案，但都不是特别理想；希望 NetworkX 推出新版本时，能够解决这一问题。

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx

a Multi_G = nx.MultiGraph()
# 多图对象实例

b Multi_G.add_nodes_from(['a', 'b', 'c', 'd'])
# 添加多个顶点

c Multi_G.add_edges_from([('a', 'b'), # 平行边
                          ('a', 'b'),
                          ('a', 'c'), # 平行边
                          ('a', 'c'),
                          ('a', 'd'),
                          ('b', 'd'),
                          ('c', 'd')])

# 添加多条边

# 可视化
plt.figure(figsize = (6,6))
d nx.draw_networkx(Multi_G, with_labels=True)

e adjacency_matrix = nx.to_numpy_matrix(Multi_G)
# 获得邻接矩阵
```

代码 4. 定义并可视化多图 | Bk6\_Ch14\_02.ipynb

## 14.4 子图

一个图的**子图** (subgraph) 是指原始图的一部分，它由图中的节点和边的子集组成。子图可以包含图中的部分节点和部分边，但这些节点和边的组合必须遵循原始图中存在的连接关系。

子图可以是原始图的任意子集。给定一个图  $G = (V, E)$ ，其中  $V$  是节点集合， $E$  是边集合。如果  $H = (V', E')$  是  $G$  的子图，则：

$V'$  是  $V$  的子集；

$E'$  是  $E$  的子集。

简单来说，子图是通过选择原始图中的一些节点和边而形成的一个图，保持了这些选定的节点之间的连接关系。这个过程并不创造新的节点，也不产生新的边。

图 6 中的图节点集合为  $V(G) = \{a, b, c, d\}$ ，如果选取  $V$  的子集  $\{a, b, c\}$  作为一个  $G$  子图的节点，我们便得到图 13 右侧子图。

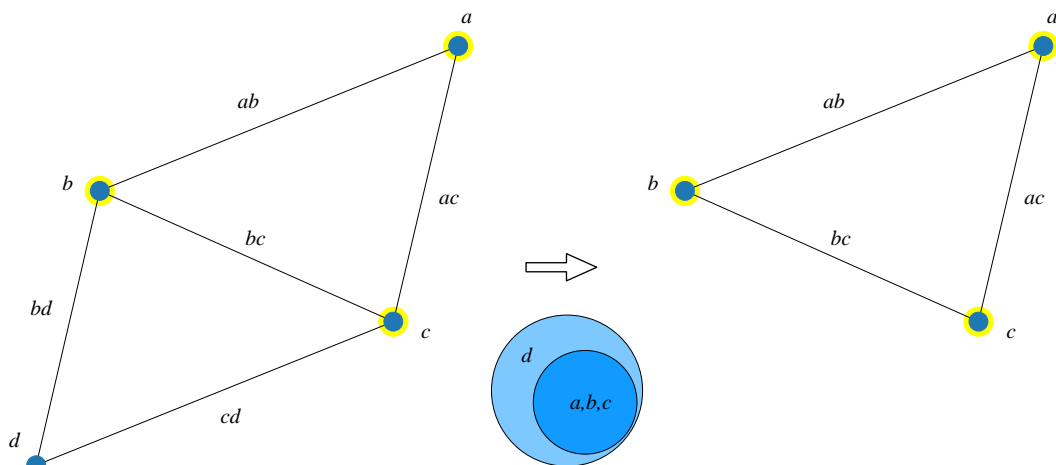


图 13. 基于节点集合子集的子图

此外，我们还可以利用图的边集合子集构造子图。图 6 中的图节点集合为  $E(G) = \{ab, ac, bc, bd, cd\}$ ，如果选取  $E$  的子集  $\{ab, bc, cd\}$  作为一个  $G$  子图的节点，我们便得到图 14 右侧子图。



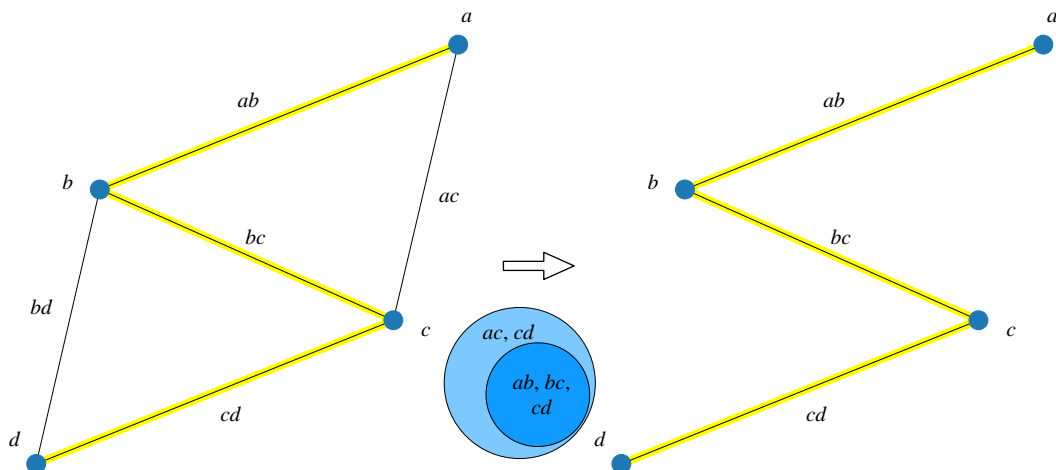


图 14. 基于边集合子集的子图

代码 5 展示如何用 NetworkX 创建子图。请大家注意以下几句。

- d** 用 `subgraph()` 方法基于节点集合子集创建子图。
- e** 计算原始图节点集合和子图节点集合之差。
- f** 用 `edge_subgraph()` 方法基于边集合子集创建子图。
- g** 计算原始图边集合和子图边集合之差。

```

import matplotlib.pyplot as plt
import networkx as nx

a G = nx.Graph()
# 创建无向图的实例

b G.add_nodes_from(['a', 'b', 'c', 'd'])
# 添加多个顶点

c G.add_edges_from([( 'a', 'b' ),
                     ( 'b', 'c' ),
                     ( 'b', 'd' ),
                     ( 'c', 'd' ),
                     ( 'c', 'a' )])
# 增加一组边

d Sub_G_nodes = G.subgraph(['a', 'b', 'c'])
# 基于节点子集的子图

e set(G.nodes) - set(Sub_G_nodes.nodes)
# 计算节点集合之差
# 结果为 {'d'}

f Sub_G_edges = G.edge_subgraph([( 'a', 'b' ),
                                   ( 'b', 'c' ),
                                   ( 'c', 'd' )])
# 基于边子集的子图

g set(G.edges) - set(Sub_G_edges.edges)
# 计算边集合之差
# 结果为 {( 'a', 'c' ), ( 'b', 'd' )}

```



代码 5. 创建子图 | Bk6\_Ch14\_03.ipynb

## 14.5 加权图：边自带权重

**加权无向图** (weighted undirected graph) 是一种图论中的数据结构，基于无向图，但在每条边上附加了一个权重或值。这个权重表示了连接两个节点之间的某种度量，例如距离、成本、时间等。

每条边上的权重可以是实数或整数，用来表示相应边的重要性或其他度量。

在加权无向图中，通常通过在图的边上添加权值来模拟现实世界中的关系或约束。这种图结构在许多应用中都很实用，如网络规划、交通规划、社交网络分析等。在算法和问题解决中，加权无向图的引入使得我们能够更准确地建模和分析实际情况中的关系。

举个例子，图 15 可视化 1886 年至 1985 年的所有 685 场世界国际象棋锦标赛比赛参赛者、赛事、成绩。边宽度代表对弈的数量，点的大小代表获胜棋局数量。

图 15 这个例子来自 NetworkX 官方示例，大家可以自己学习。

[https://networkx.org/documentation/stable/auto\\_examples/drawing/plot\\_chess\\_masters.html](https://networkx.org/documentation/stable/auto_examples/drawing/plot_chess_masters.html)

此外，图 15 也告诉我们用 NetworkX 绘制图时，节点、边可以调整设计来展示更多有价值的信息。

本 PDF 文件为作者草稿，发布目的为方便读者在移动终端学习，终稿内容以清华大学出版社纸质出版物为准。

版权归清华大学出版社所有，请勿商用，引用请注明出处。

代码及 PDF 文件下载：<https://github.com/Visualize-ML>

本书配套微课视频均发布在 B 站——生姜 DrGinger: <https://space.bilibili.com/513194466>

欢迎大家批评指教，本书专属邮箱：[jiang.visualize.ml@gmail.com](mailto:jiang.visualize.ml@gmail.com)

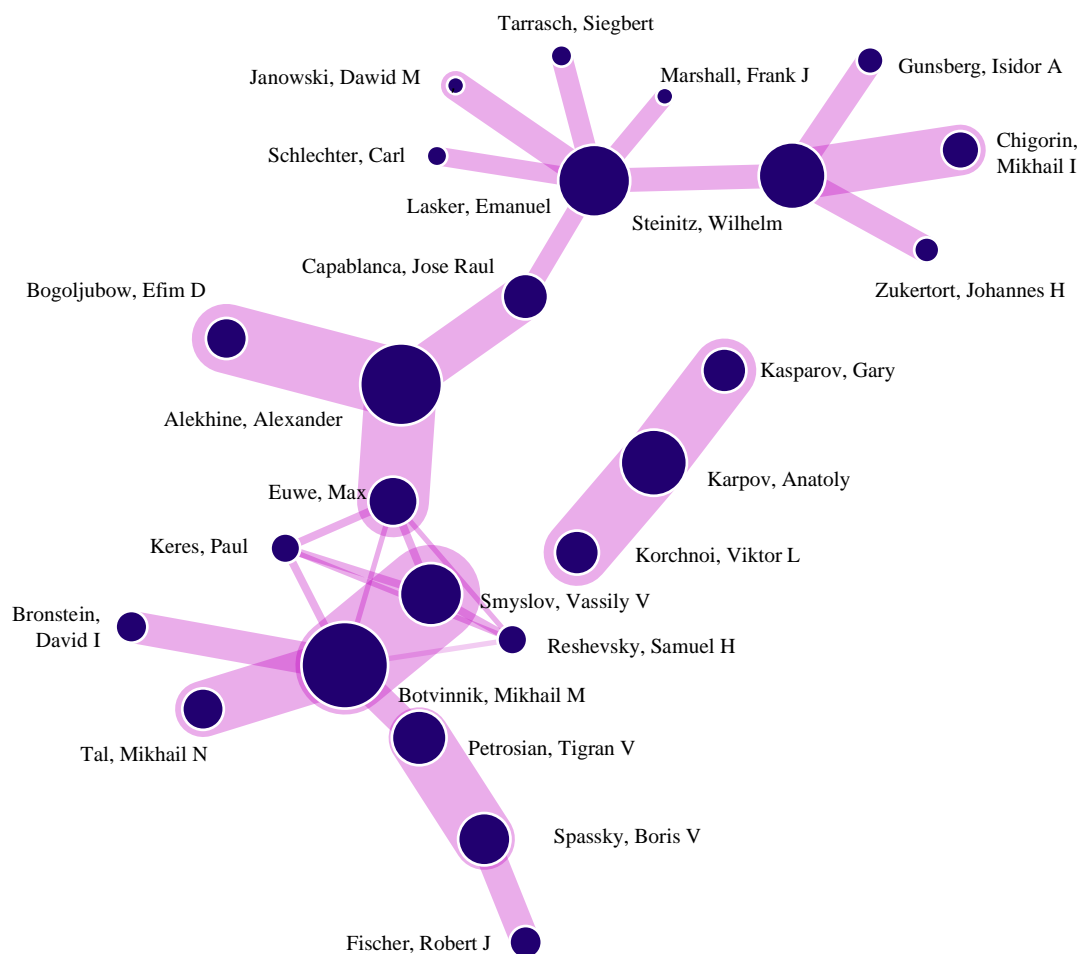


图 15. 可视化 1866 年至 1985 年的所有 685 场世界国际象棋锦标赛比赛参赛者、赛事、成绩；图片来自《可视之美》

大家应该对图 16 这幅图很熟悉了，上一章介绍过这个图。图 16 不同的是，图的每条边都有自己权重值。

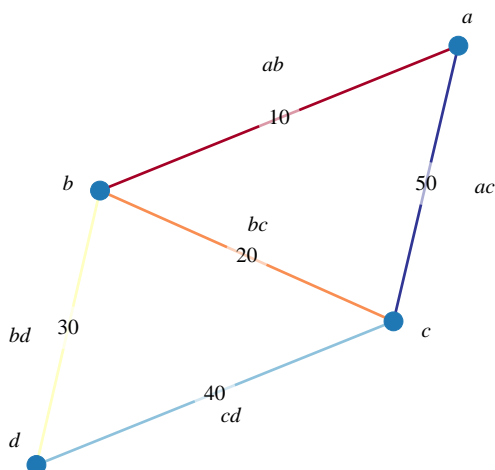


图 16. 加权无向图

下面聊聊代码 6 如何用 NetworkX 绘制加权无向图。

```

import matplotlib.pyplot as plt
import networkx as nx

a weighted_G = nx.Graph()
# 创建无向图的实例

b weighted_G.add_nodes_from(['a', 'b', 'c', 'd'])
# 添加多个顶点

c weighted_G.add_edges_from([('a', 'b', {'weight': 10}),
                              ('b', 'c', {'weight': 20}),
                              ('b', 'd', {'weight': 30}),
                              ('c', 'd', {'weight': 40}),
                              ('c', 'a', {'weight': 50})])

# 增加一组边，并赋予权重

d weighted_G['a']
# 取出节点a的邻居

e weighted_G['a']['b']
# 取出ab边的权重，结果为字典

f weighted_G['a']['b']['weight']
# 取出ab边的权重，结果为数值

g edge_weights = [weighted_G[i][j]['weight'] for i, j in weighted_G.edges]
# 所有边的权重

h edge_labels = nx.get_edge_attributes(weighted_G, "weight")
# 所有边的标签

plt.figure(figsize = (6,6))
i pos = nx.spring_layout(weighted_G)
j nx.draw_networkx(weighted_G,
                    pos = pos,
                    with_labels = True,
                    node_size = 180,
                    edge_color=edge_weights,
                    edge_cmap = plt.cm.RdYlBu,
                    edge_vmin = 10, edge_vmax = 50)

k nx.draw_networkx_edge_labels(weighted_G,
                               pos = pos,
                               edge_labels=edge_labels,
                               font_color='k')

plt.savefig('加权无向图.svg')

```

代码 6. 绘制加权无向图 | Bk6\_Ch14\_04.ipynb

- a 用 `networkx.Graph()` 创建无向图实例。
- b 用 `add_nodes_from()` 方法添加四个节点，节点的标签分别为 'a', 'b', 'c', 'd'。
- c 用 `add_edges_from()` 方法向图中添加多条边，每条边用一个包含起点、终点和权重的元组表示。这里的权重是使用字典形式的边属性进行设置。
- d 取出节点 'a' 的邻居，返回一个邻居节点的字典。

- e 取出节点 'a' 到 'b' 的边的属性，返回一个字典，包含边的所有属性。
- f 取出节点 'a' 到 'b' 的边的权重值。
- g 创建一个列表，包含图中所有边的权重。通过遍历图中所有的边，将每条边的权重添加到列表中。
- h 用 `networkx.get_edge_attributes()` 获取图中所有边的权重作为字典。
- i 用 `networkx.spring_layout()` 计算节点的布局位置，返回一个包含节点位置信息的字典。
- j 用 `networkx.draw_networkx()` 绘制图，其中包括节点、边和边的权重。`edge_color` 参数用于指定边的颜色，根据权重值映射到 `plt.cm.RdYlBu`。`edge_vmin` 和 `edge_vmax` 指定边颜色映射的范围。
- k 用 `networkx.draw_networkx_edge_labels()` 在图上添加边的标签，这里是边的权重值。

## 14.6 有向图

将图 6 中的有向图记做  $G_D$ 。有向图两个重要集合：(1) 节点集  $V(G_D)$ ；(2) 有向边集  $A(G_D)$ 。因此， $G_D$  也常常被写成  $G_D = (V, A)$ 。

注意，无向图中边集记做  $E(G)$ ， $E$  代表 edge。有向图中有向边集记做  $A(G_D)$ ， $A$  代表弧 (arc，复数 arcs)，也叫 arrows，本书叫有向边 (directed edge)。有向边是节点的有序对。下标  $D$  代表 directed。

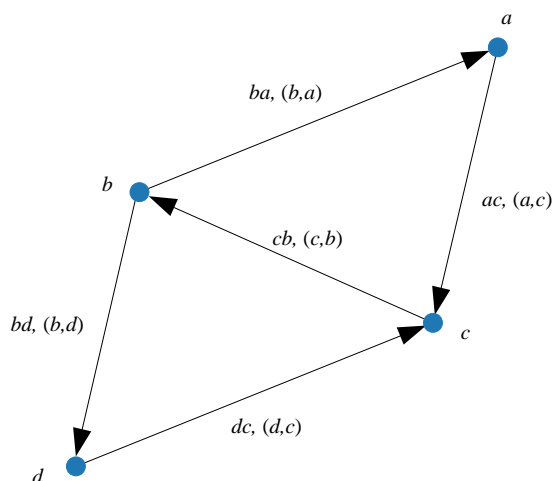


图 17.4 个节点，5 条有向边的有向图

同样为了和有向图对照学习，图 6 采用和本书前文无向图一样的结构，不同的是每条赋予了方向。

以图 6 的有向图  $G_D$  为例， $G_D$  的节点集  $V(G_D)$  为：

$$V(G_D) = \{a, b, c, d\} \quad (3)$$

有向图  $G_D$  节点集和无向图并无差别。

$G_D$  的有向边集  $A(G_D)$  为：

$$A(G_D) = \{ba, cb, bd, dc, ac\} = \{(b, a), (c, b), (b, d), (d, c), (a, c)\} \quad (4)$$

由于图 6 中图  $G_D$  是有向图，因此节点  $b$  到节点  $a$  的边  $ba$ ，不同于节点  $a$  到节点  $b$  边  $ab$ 。

有向边  $ab$  中  $a$  叫头 (head)， $b$  叫尾 (tail)。

图 6 是用 NetworkX 绘制，下面聊聊代码 1。

**a** 用 `networkx.DiGraph()` 创建一个空的有向图对象实例。在这个实例中，我们可以添加节点和边，进行图的各种操作和分析。

**b** 用 `add_nodes_from()` 方法增加 4 个节点。当然，我们也可以用 `add_node()` 方法增加单一节点，这和无向图一致。

**c** 用 `add_edges_from()` 方法向图中添加一组有向边。注意，`('a', 'b')` 不同于 `('b', 'a')`。

**d** 用 `networkx.draw_networkx()` 绘制有向图，传入图 `directed_G`、节点的位置信息 `pos`、箭头大小 `arrowsize`、节点的大小 `node_size`。

图 8 提供了几个供大家练习的有向图。

```
import matplotlib.pyplot as plt
import networkx as nx

a directed_G = nx.DiGraph()
# 创建有向图的实例


b directed_G.add_nodes_from(['a', 'b', 'c', 'd'])
# 添加多个顶点

c directed_G.add_edges_from([(b, 'a'),
                             (c, 'b'),
                             (b, 'd'),
                             (d, 'c'),
                             (a, 'c')])
# 增加一组有向边

random_pos = nx.random_layout(directed_G, seed=188)
# 设定随机种子，保证每次绘图结果一致

pos = nx.spring_layout(directed_G, pos=random_pos)
# 使用弹簧布局算法来排列图中的节点
# 使得节点之间的连接看起来更均匀自然

plt.figure(figsize = (6,6))
d nx.draw_networkx(directed_G, pos = pos,
                   arrowsize = 28,
                   node_size = 180)
plt.savefig('G_D_4顶点_5边.svg')
```

代码 7. 用 NetworkX 绘制无向图 |  Bk6\_Ch14\_05.ipynb

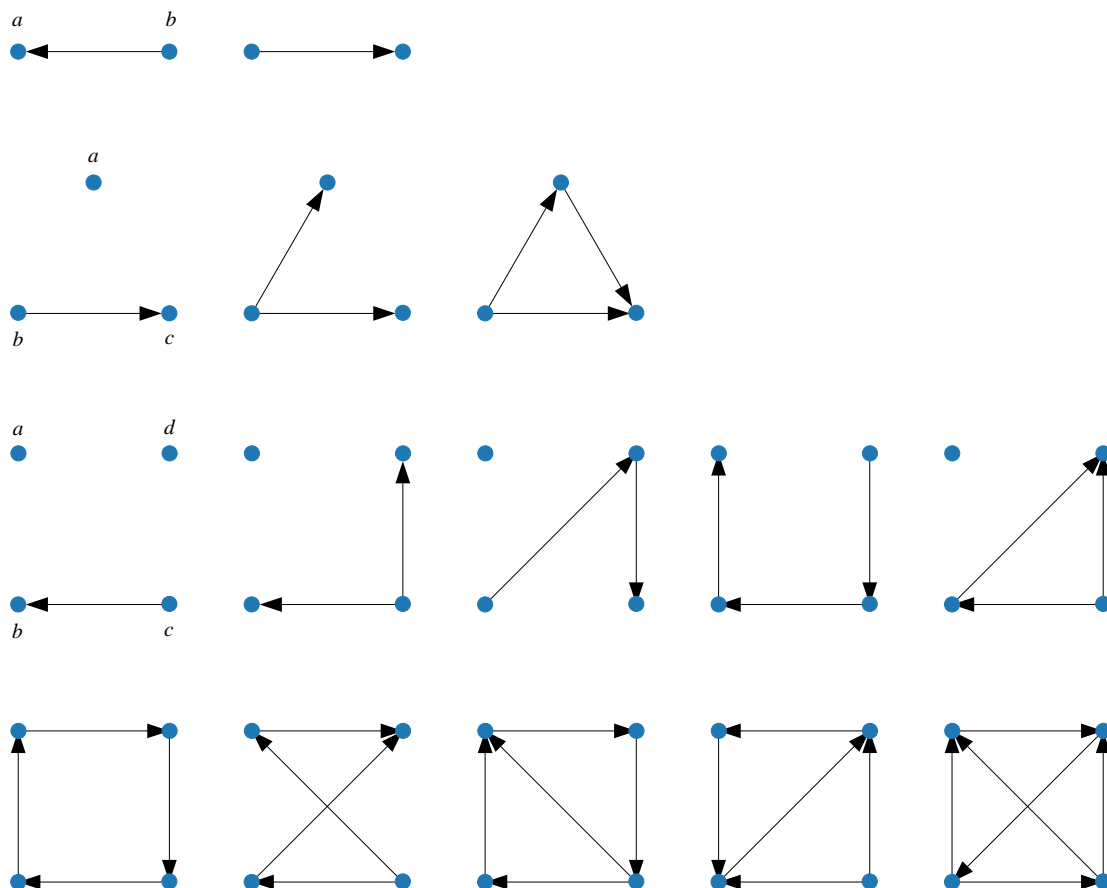


图 18. 供大家练习的几个有向图

### 阶、大小

和无向图一样，有向图  $G_D$  的节点数量叫做**阶** (order)，常用  $n$  表示。图 6 所示的有向图  $G_D$  的阶为 4 ( $n=4$ )，也就是说  $G_D$  为 4 阶图。

和无向图一样，图  $G_D$  的边的数量叫做图的**大小** (size)，常用  $m$  表示。图 6 所示的图  $G_D$  的大小为 5 ( $m=5$ )。

接着前文代码，代码 8 计算有向图的阶、大小等度量。请大家格外注意 **a** 和 **b**。对于有向图实例，用 `has_edge()` 判断边是否存在时，需要注意方向。

```

directed_G.order()
# 图的阶

directed_G.number_of_nodes()
# 图的节点数

directed_G.nodes
# 列出图的节点

directed_G.size()
# 图的大小

directed_G.edges
# 列出图的边

directed_G.number_of_edges()
# 图的边数

a directed_G.has_edge('a', 'b')
# 判断是否存在ab有向边

b directed_G.has_edge('b', 'a')
# 判断是否存在ba有向边

```

代码 8. 用 NetworkX 计算有向图的阶、大小 |  Bk6\_Ch14\_05.ipynb

## 度：入度、出度

和无向图一样，有向图任意一个节点的**度** (degree) 是与它相连的边的数量。

但是，有向图中由于边有方向，我们更关心**入度** (indegree)、**出度** (outdegree) 这两个概念。

在有向图中，节点的**入度**是指指向该节点的边的数量，即从其他节点指向该节点的有向边的数量。而**出度**是指从该节点出发的边的数量，即从该节点指向其他节点的有向边的数量。这两个概念用于描述有向图中节点的连接性质，**入度**和**出度**的总和等于节点的**度数**。

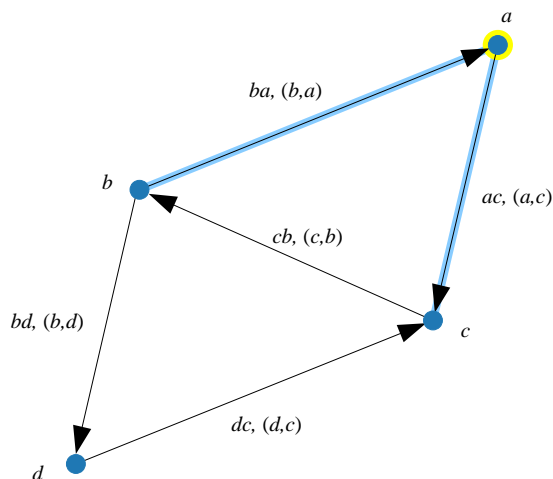
比如，如图 9 所示，图  $G_D$  中节点  $a$  的度为 2，记做  $\deg_{G_D}(a) = 2$ 。

而图  $G_D$  中节点  $a$  的入度为 1，即有 1 条有向边“进入”节点  $a$ ，记做  $\deg_{G_D}^+(a) = 1$ 。图  $G_D$  中节点  $a$  的出度为 1，即有 1 条有向边“离开”节点  $a$ ，记做  $\deg_{G_D}^-(a) = 1$ 。显然，节点  $a$  的入度和出度之和为其度数

$$\deg_{G_D}(a) = \deg_{G_D}^+(a) + \deg_{G_D}^-(a) \quad (5)$$

整个有向图来看，入度之和等于出度之和。

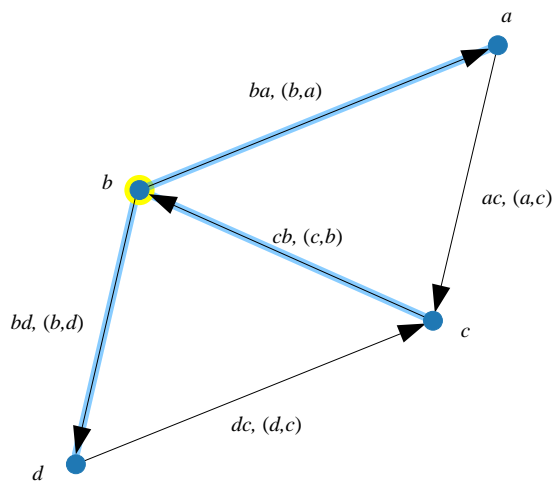


图 19. 节点  $a$  的度为 2，入度为 1，出度为 1

再看个例子，图  $G_D$  中节点  $b$  的度为 3，记做  $\deg_G(a) = 3$ 。

而图  $G_D$  中节点  $b$  的入度为 1，即有 1 条有向边“进入”节点  $b$ ，记做  $\deg_{G_D}^+(b) = 1$ 。图  $G_D$  中节点  $b$  的出度为 2，即有 2 条有向边“离开”节点  $b$ ，记做  $\deg_{G_D}^-(b) = 2$ 。

同样，入度和出度之和为度数，即  $\deg_{G_D}(b) = \deg_{G_D}^+(b) + \deg_{G_D}^-(b)$ 。

图 20. 节点  $b$  的度为 3，入度为 1，出度为 2

```

a directed_G.degree()
# 图的度

dict(directed_G.degree())

b directed_G.in_degree()
# 有向图的入度

c directed_G.out_degree()
# 有向图的出度

d directed_G.degree('a')
# 节点a的度

e directed_G.in_degree('a')
# 节点a的入度

f directed_G.out_degree('a')
# 节点a的出度

```



代码 9. 用 NetworkX 计算有向图的度、入度、出度 | Bk6\_Ch14\_05.ipynb

## 邻居

无向图中，给定特定节点的**邻居** (neighbors) 指的是与该节点直接相连的其他节点。简单来说，如果两个节点之间存在一条边，那么它们就互为邻居。

但是，在有向图中，邻居的定义则多了一层考虑——边的方向。

由于，对于任意节点的度分为入度、出度。据此，我们把邻居也分为——**入度邻居** (incoming neighbor, indegree neighbor)、**出度邻居** (outgoing neighbor, outdegree neighbor)。

入度邻居，可以理解为**上家** (predecessor)。对于节点  $a$  而言，入度邻居是所有指向节点  $a$  的节点，即节点  $b$ 。

出度邻居，可以理解为**下家** (successor)。节点  $a$  的出度邻居是所节点  $a$  指向的节点，即节点  $c$ 。

请大家自行分析节点  $b$  的邻居有哪些？入度邻居、出度邻居分别是谁？

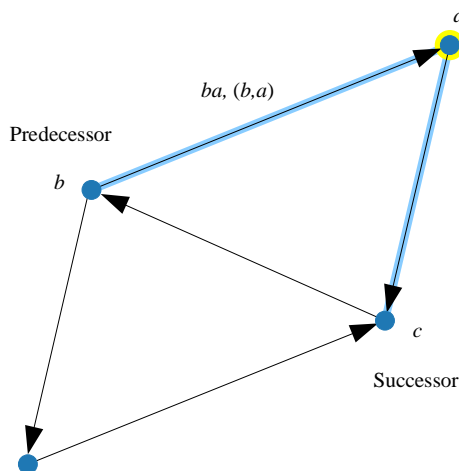


图 21. 节点  $a$  的入度邻居、出度邻居

接着前文代码，代码 10 计算有向图节点 *a* 的邻居、入度邻居、出度邻居。

- a 用 `networkx.all_neighbors()` 获取有向图中节点 *a* 所有的邻居，包括入度、出度。
- b 对有向图 `directed_G` 节点 *a* 用 `neighbors()` 方法只能获取其出度邻居。
- c 对有向图 `directed_G` 节点 *a* 也可以用 `successors()` 方法获取其出度邻居。
- d 对有向图 `directed_G` 节点 *a* 用 `predecessors()` 方法获取其入度邻居。

```
a list(nx.all_neighbors(directed_G, 'a'))
# 节点a所有邻居

b list(directed_G.neighbors('a'))
# 节点a的（出度）邻居

c list(directed_G.successors('a'))
# 节点a的出度邻居

d list(directed_G.predecessors('a'))
# 节点a的入度邻居
```

代码 10. 用 NetworkX 计算有向图的邻居、入度邻居、出度邻居 | Bk6\_Ch14\_05.ipynb

## 有向多图

本书前文介绍过无向图的多图 (multigraph)，即允许在同一对节点之间存在平行边 (parallel edge)，也叫重边。

图 12 所示为用 NetworkX 绘制的有向多图。节点 *a* 和 *b* 之间有两条有向边，节点 *a* 和 *c* 之间也有两条有向边。

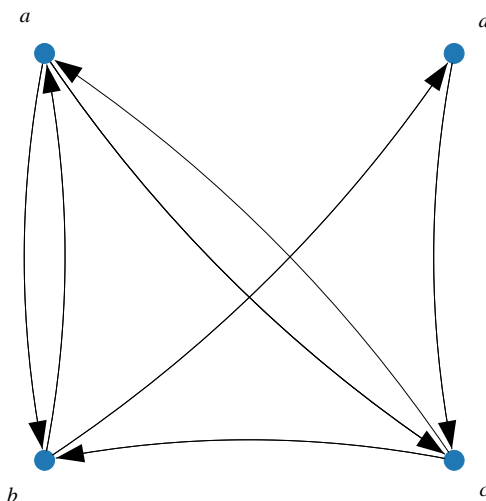


图 22. 有向多图

代码 11 绘制图 12。请大家格外注意 d 和 e 两句。d 人为设定每个节点在平面上的坐标位置。e 在使用 `networkx.draw_networkx()` 绘图时，通过 `pos` 参数输入每个节点坐标，利用 `connectionstyle` 将有向边设为圆弧，并指定弧度。

```

import matplotlib.pyplot as plt
import networkx as nx

a directed_G = nx.MultiDiGraph()
# 创建有向图的实例

b directed_G.add_nodes_from(['a', 'b', 'c', 'd'])
# 添加多个顶点

c directed_G.add_edges_from([('b', 'a'),
                             ('a', 'b'),
                             ('c', 'b'),
                             ('b', 'd'),
                             ('d', 'c'),
                             ('a', 'c'),
                             ('c', 'a')])

# 增加一组有向边

# 人为设定节点位置
d nodePosDict = {'b': [0, 0],
                 'c': [1, 0],
                 'd': [1, 1],
                 'a': [0, 1]}

e plt.figure(figsize = (6,6))
nx.draw_networkx(directed_G,
                 pos = nodePosDict,
                 arrowsize = 28,
                 connectionstyle='arc3, rad = 0.1',
                 node_size = 180)
plt.savefig('G_D_4顶点_7边.svg')

```

代码 11. 用 NetworkX 绘制有向多图 | Bk6\_Ch14\_06.ipynb

图论是数学的一个分支，研究的是图的性质和结构以及与图相关的问题。图由节点和边组成，节点表示对象，边表示对象之间的关系。图论被广泛应用于计算机科学、网络分析、社交网络、电路设计等领域。

在图论中，图可以分为无向图和有向图两种基本类型。无向图中的边没有方向，即连接两个节点的边不区分起点和终点。有向图中的边有方向，即连接两个节点的边有明确的起点和终点。

论在机器学习中的应用有助于处理复杂的关系型数据，提取有用的信息，并为模型提供更深层次的理解。

下一章将专门介绍如何用 NetworkX 完成图的可视化。