

第 10 章 图论模型



图论是运筹学的一个经典和重要分支,专门研究图与网络模型的特点、性质以及求解方法.许多优化问题,可以利用图与网络的固有特性所形成的特定方法来解决,比用数学规划等其他模型求解往往要简单且有效得多.

图论起源于 1736 年欧拉对哥尼斯堡七桥问题的抽象和论证.1936 年,匈牙利数学家柯尼西出版的第一部图论专著《有限图与无限图理论》,树立了图论发展的第一座里程碑.近几十年来,计算机科学和技术的飞速发展,大大地促进了图论的研究和应用,其理论和方法已经渗透到物理学、化学、计算机科学、通信科学、建筑学、生物遗传学、心理学、经济学、社会学等各个学科中.

10.1 图的基础理论及 networkx 简介

10.1.1 图的基本概念

所谓图,概括地讲就是由一些点和这些点之间的连线组成的.定义为 $G = (V, E)$, 其中 V 是顶点的非空有限集合,称为顶点集. E 是边的集合,称为边集.边一般用 (v_i, v_j) 表示,其中 v_i, v_j 属于顶点集 V .

以下用 $|V|$ 表示图 $G = (V, E)$ 中顶点的个数, $|E|$ 表示边的条数.

图 10.1 是三个图的示例,其中图 10.1 (a) 中的图有 3 个顶点、2 条边,将其表示为 $G = (V, E), V = \{v_1, v_2, v_3\}, E = \{(v_1, v_2), (v_1, v_3)\}$.

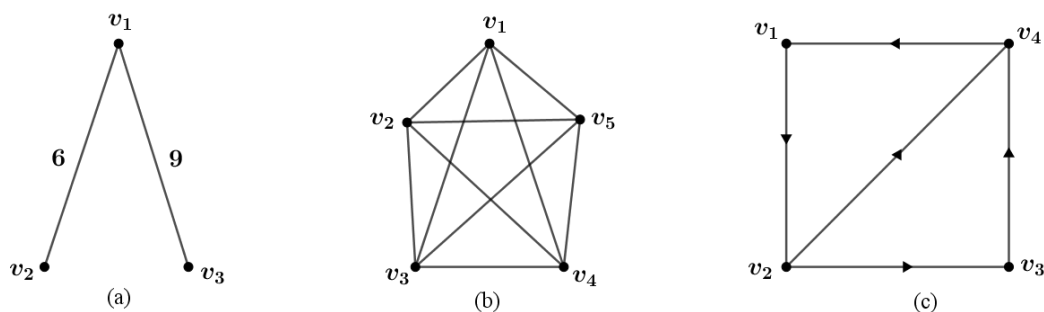


图 10.1: 图的示意图

1. 无向图和有向图

如果图的边是没有方向的,则称此图为无向图(简称为图),无向图的边称为无向边(简称边).图 10.1 (a) 和 (b) 中的图都是无向图.连接两顶点 v_i 和 v_j 的无向边记为 (v_i, v_j) 或 (v_j, v_i) .

如果图的边是有方向 (带箭头) 的, 则称此图为有向图, 有向图的边称为弧 (或有向边), 如图 10.1 (c) 中的图是一个有向图. 连接两顶点 v_i 和 v_j 的弧记为 $\langle v_i, v_j \rangle$, 其中 v_i 称为起点, v_j 称为终点. 显然此时弧 $\langle v_i, v_j \rangle$ 与弧 $\langle v_j, v_i \rangle$ 是不同的两条有向边. 有向图的弧的起点称为弧头, 弧的终点称为弧尾. 有向图一般记为 $D = (V, A)$, 其中 V 为顶点集, A 为弧集.

例如, 图 10.1 (c) 可以表示为 $D = (V, A)$, 顶点集 $V = \{v_1, v_2, v_3, v_4\}$, 弧集为 $A = \{\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle, \langle v_2, v_4 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_1 \rangle\}$.

对于图除非指明是有向图, 一般地, 所谓的图都是指无向图. 有向图也可以用 G 表示.

例 10.1: 设 $V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5\}$, 其中

$$e_1 = (v_1, v_2), \quad e_2 = (v_2, v_3), \quad e_3 = (v_2, v_3), \quad e_4 = (v_3, v_4), \quad e_5 = (v_4, v_4).$$

则 $G = (V, E)$ 是一个图, 其图形如图 10.2 所示.

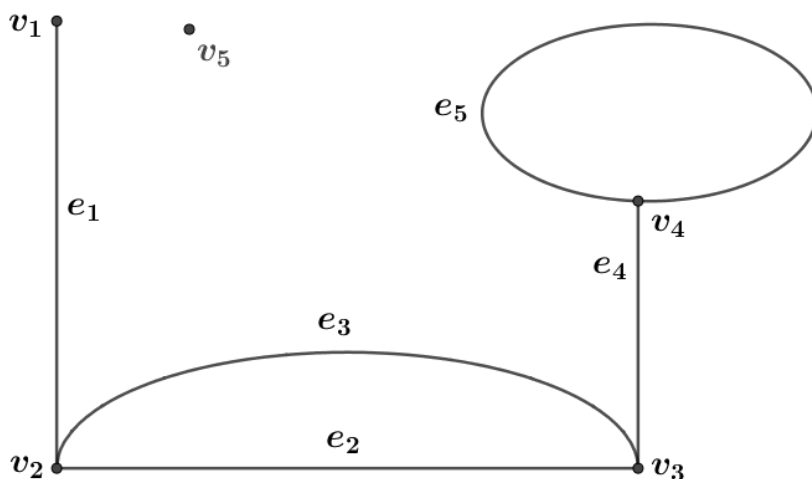


图 10.2: 非简单图示例

2. 简单图和完全图

定义 10.1: 相邻、重边、环、孤立点

设 $e = (u, v)$ 是图 G 的一条边, 则称 u, v 是 e 的端点, 并称 u 与 v 相邻, 边 e 与顶点 u (或 v) 相关联. 若两条边 e_i 与 e_j 有共同的端点, 则称边 e_i 与 e_j 相邻; 称有相同端点的两条边为重边; 称两端点均相同的边为环; 称不与任何边相关联的顶点为孤立点.

图 10.2 中, 边 e_2 与 e_3 为重边, e_5 为环, 顶点 v_5 为孤立点.

定义 10.2: 简单图

无环且无重边的图称为简单图.




图 10.2 不是简单图, 因为图中既含重边 (e_2 与 e_3) 又含环 (e_5).

定义 10.3: 完全图

任意两点均相邻的简单图称为完全图. 含 n 个顶点的完全图记为 K_n . 

3. 赋权图

定义 10.4: 赋权图


如果图 G 的每条边 e 都附有一个实数 $w(e)$, 则称图 G 为赋权图, 实数 $w(e)$ 称为边 e 的权. 

赋权图也称为网络, 图 10.1 (a) 中的图就是一个赋权图. 赋权图中的权可以是距离、费用、时间、效益、成本等.

如果有向图 D 的每条弧都被赋予了权, 则称 D 为有向赋权图.

4. 顶点的度

定义 10.5: 顶点的度

- (1) 在无向图中, 与顶点 v 关联的边的数目 (环算两次) 称为 v 的度, 记为 $d(v)$.
- (2) 在有向图中, 从顶点 v 引出的弧的数目称为 v 的出度, 记为 $d^+(v)$, 从顶点 v 引入的弧的数目称为 v 的入度, 记为 $d^-(v)$, $d(v) = d^+(v) + d^-(v)$ 称为 v 的度. 

度为奇数的顶点称为奇顶点, 度为偶数的顶点称为偶顶点.

定理 10.1

给定图 $G = (V, E)$, 所有顶点的度数之和是边数的 2 倍, 即


$$\sum_{v \in V} d(v) = 2|E|.$$



推论 任何图中奇顶点的总数必为偶数.

5. 子图

定义 10.6: 子图、生成子图

设 $G_1 = (V_1, E_1)$ 与 $G_2 = (V_2, E_2)$ 是两个图, 并且满足 $V_1 \subset V_2$, $E_1 \subset E_2$, 则称 G_1 是 G_2 的子图. 如果 G_1 是 G_2 的子图, 且 $V_1 = V_2$, 则称 G_1 是 G_2 的生成子图. 

6. 道路与回路



定义 10.7: 道路、回路、圈

设 $W = v_0 e_1 v_1 e_2 \cdots e_k v_k$, 其中 $e_i \in E$ ($i = 1, 2, \cdots, k$), $v_j \in V$ ($j = 0, 1, \cdots, k$), e_i 与 v_{i-1} 和 v_i 关联, 称 W 是图 G 的一条道路, 简称路, k 为路长, v_0 为起点, v_k 为终点; 各边相异的道路称为迹 (trail); 各顶点相异的道路称为轨道 (path), 记为 $P(v_0, v_k)$; 起点和终点重合的道路称为回路; 起点和终点重合的轨道称为圈, 即对轨道 $P(v_0, v_k)$, 当 $v_0 = v_k$ 时成为一个圈. 称以两顶点 u, v 分别为起点和终点的最短轨道之长为顶点 u, v 的距离.

**7. 连通图与非连通图****定义 10.8: 连通图、强连通图**

在无向图 G 中, 如果从顶点 u 到顶点 v 存在道路, 则称顶点 u 和 v 是连通的. 如果图 G 中的任意两个顶点 u 和 v 都是连通的, 则称图 G 是连通图, 否则称为非连通图. 非连通图中的连通子图, 称为连通分支.

在有向图 G 中, 如果对于任意两个顶点 u 和 v , 从 u 到 v 和从 v 到 u 都存在道路, 则称图 G 是强连通图.

**10.1.2 图的表示及 networkx 简介**

本节均假设图 $G = (V, E)$ 为简单图, 其中 $V = \{v_1, v_2, \cdots, v_n\}$, $E = \{e_1, e_2, \cdots, e_m\}$.

1. 关联矩阵

对于无向图 G , 其关联矩阵 $M = (m_{ij})_{n \times m}$, 其中

$$m_{ij} = \begin{cases} 1, & v_i \text{ 与 } e_j \text{ 相关联,} \\ 0, & v_i \text{ 与 } e_j \text{ 不关联.} \end{cases}$$

对于有向图 G , 其关联矩阵 $M = (m_{ij})_{n \times m}$, 其中

$$m_{ij} = \begin{cases} 1, & v_i \text{ 是 } e_j \text{ 的起点,} \\ -1, & v_i \text{ 是 } e_j \text{ 的终点,} \\ 0, & v_i \text{ 与 } e_j \text{ 不关联.} \end{cases}$$

2. 邻接矩阵

对于无向非赋权图 G , 其邻接矩阵 $W = (w_{ij})_{n \times n}$, 其中

$$w_{ij} = \begin{cases} 1, & v_i \text{ 与 } v_j \text{ 相邻,} \\ 0, & v_i \text{ 与 } v_j \text{ 不相邻.} \end{cases}$$



对于有向非赋权图 D , 其邻接矩阵 $\mathbf{W} = (w_{ij})_{n \times n}$, 其中

$$w_{ij} = \begin{cases} 1, & \langle v_i, v_j \rangle \in A, \\ 0, & \langle v_i, v_j \rangle \notin A. \end{cases}$$

对于无向赋权图 G , 其邻接矩阵 $\mathbf{W} = (w_{ij})_{n \times n}$, 其中

$$w_{ij} = \begin{cases} \text{顶点 } v_i \text{ 与 } v_j \text{ 之间边的权,} & (v_i, v_j) \in E, \\ 0 \text{ (或 } \infty), & v_i \text{ 与 } v_j \text{ 之间无边.} \end{cases}$$

注: 当两个顶点之间不存在边时, 根据实际问题的含义或算法需要, 对应的权可以取为 0 或 ∞ .

有向赋权图的邻接矩阵可类似定义.

3. networkx 简介

networkx 是一个用 Python 语言开发的图论与复杂网络建模工具, 内置了常用的图与复杂网络分析算法, 可以方便地进行复杂网络数据分析、仿真建模等工作.

networkx 支持创建简单无向图、有向图和多重图; 内置许多标准的图论算法, 顶点可为任意数据; 支持任意的边值维度.

networkx 的一些常用函数举例如下:

- (1) Graph(): 创建无向图;
- (2) Graph(A): 由邻接矩阵 A 创建无向图;
- (3) DiGraph(): 创建有向图;
- (4) DiGraph(A): 由邻接矩阵 A 创建有向图;
- (5) MultiGraph(): 创建多重无向图;
- (6) MultiDiGraph(): 创建多重有向图;
- (7) add_edge(): 添加一条边;
- (8) add_edges_from(List): 从列表中添加多条边;
- (9) add_node(): 添加一个顶点;
- (10) add_nodes_from(List): 添加顶点集合;
- (11) dijkstra_path(G, source, target, weight='weight'): 求最短路径;
- (12) dijkstra_path_length(G, source, target, weight='weight'): 求最短距离.

例 10.2: 图 10.3 所示的无向图, 其邻接矩阵为

$$\mathbf{A} = \begin{bmatrix} 0 & 9 & 2 & 4 & 7 \\ 9 & 0 & 3 & 4 & 0 \\ 2 & 3 & 0 & 8 & 4 \\ 4 & 4 & 8 & 0 & 6 \\ 7 & 0 & 4 & 6 & 0 \end{bmatrix}.$$



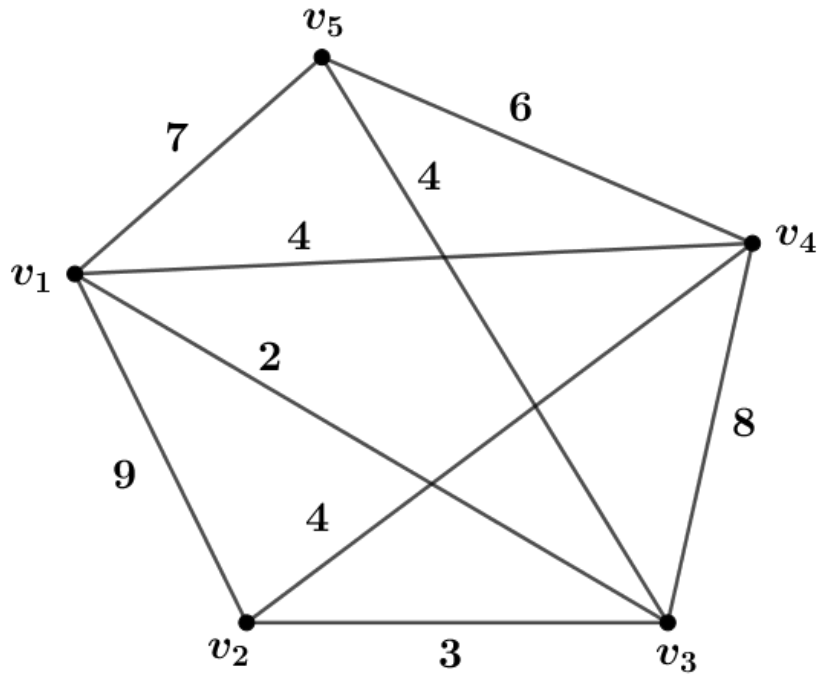


图 10.3: 赋权无向图

Python 代码

```
import numpy as np
import networkx as nx
import pylab as plt

# 构造邻接矩阵
a = np.zeros((5, 5))
a[0, 1:5] = [9, 2, 4, 7] # 上三角第一行的元素
a[1, 2:4] = [3, 4] # 上三角第二行的元素
a[2, [3, 4]] = [8, 4] # 上三角第三行元素
a[3, 4] = 6 # 上三角第四行元素
print(a)
np.savetxt("Pdata10_2.txt", a) # 保存邻接矩阵供以后使用

# 构造图的边
i, j = np.nonzero(a) # 提取顶点的编号
w = a[i, j] # 提出a中的非零元素
edges = list(zip(i, j, w))

# 画图
G = nx.Graph()
G.add_weighted_edges_from(edges)
key = range(5)
```



```

s = [str(i + 1) for i in range(5)]
s = dict(zip(key, s)) # 构造用于顶点标注的字符字典
plt.rc('font', size=18)

plt.subplot(121)
nx.draw(G, font_weight='bold', labels=s) # 图1 没有权重的图

plt.subplot(122)
pos = nx.shell_layout(G) # 布局设置
nx.draw_networkx(G, pos, node_size=260, labels=s) # 图2 有权重的图
w = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, font_size=12, edge_labels=w) # 标注
                                                                    权重
plt.savefig("figure10_2.png", dpi=500)
plt.show()

```

所画的图形如图 10.4 所示.

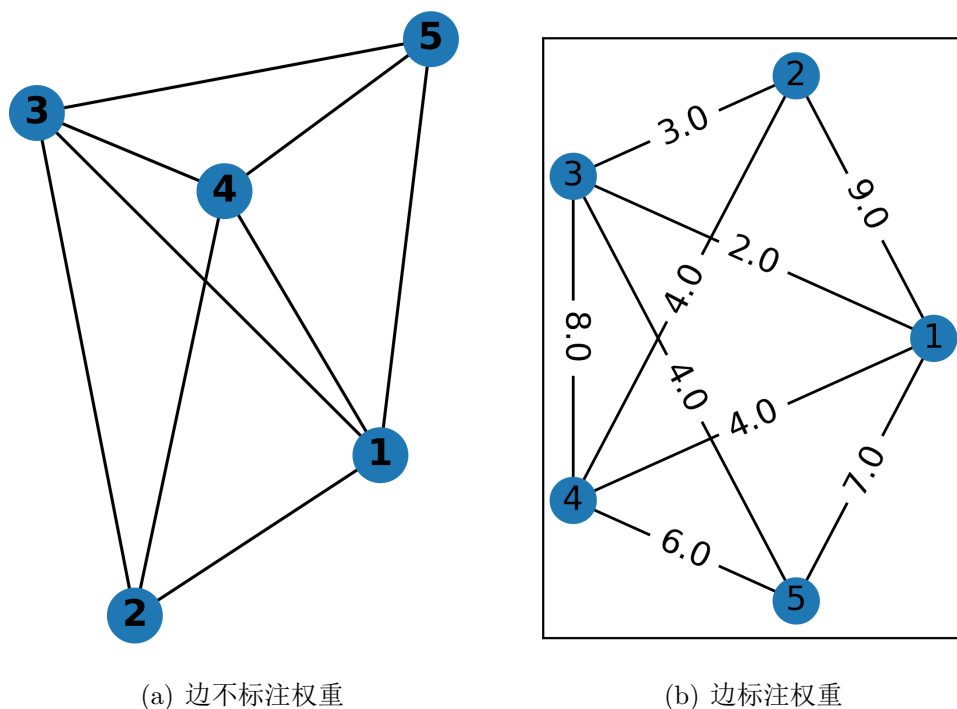


图 10.4: Python 所画的无向图

注: (1) 图形的布局有五种设置:

circular_layout: 顶点在一个圆环上均匀分布;

random_layout: 顶点随机分布;

shell_layout: 顶点在同心圆上分布;

spring_layout: 用 Fruchterman-Reingold 算法排列顶点;

spectral_layout: 根据图的 Laplace 特征向量排列顶点.



(2) 上面使用较复杂的构造图方法, 直接使用邻接矩阵 a 构造图的命令为 `Graph(a)`, 这里是为了让读者熟悉各种数据结构的使用方法. 直接输入列表构造赋权图的 Python 程序如下:

```
import networkx as nx
import pylab as plt
import numpy as np

List = [(1, 2, 9), (1, 3, 2), (1, 4, 4), (1, 5, 7), (2, 3, 3), (2, 4, 4),
        (3, 4, 8), (3, 5, 4), (4, 5, 6)]

G = nx.Graph() # 创建无向图
G.add_nodes_from(range(1, 6)) # 添加顶点集合
G.add_weighted_edges_from(List) # 通过列表增加权重
pos = nx.shell_layout(G) # 图形样式, 顶点在同心圆上分布
w = nx.get_edge_attributes(G, "weight") # 获取 graph 中的边权重
nx.draw(G, pos, with_labels=True, font_weight="bold", font_size=12)
nx.draw_networkx_edge_labels(G, pos, edge_labels=w) # 把边权重画出来
plt.show()
```

例 10.3: 图 10.5 所示的有向图的邻接矩阵为

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

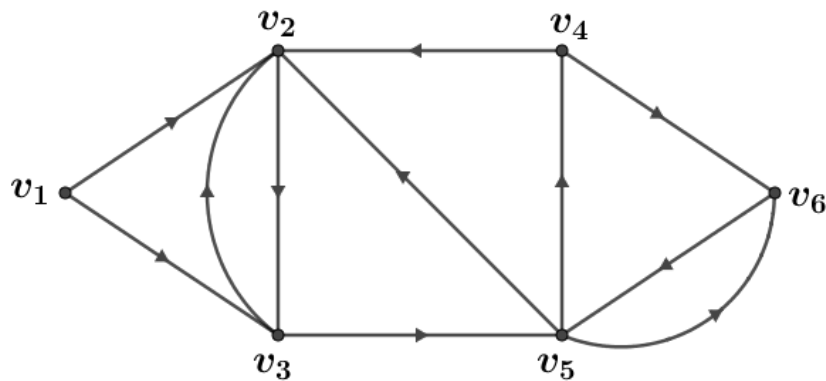


图 10.5



Python 程序

```
import numpy as np
import networkx as nx
import pylab as plt

G = nx.DiGraph() # 创建有向图
List = [(1, 2), (1, 3), (2, 3), (3, 2), (3, 5), (4, 2), (4, 6), (5, 2), (
        5, 4), (5, 6), (6, 5)]

G.add_nodes_from(range(1, 7))
G.add_edges_from(List)
plt.rc("font", size=16)
pos = nx.shell_layout(G)
nx.draw(G, pos, with_labels=True, font_weight='bold', node_color='r')
plt.savefig("figure10_3.png", dpi=500)
plt.show()
```

所画的图形如图 10.6 所示.

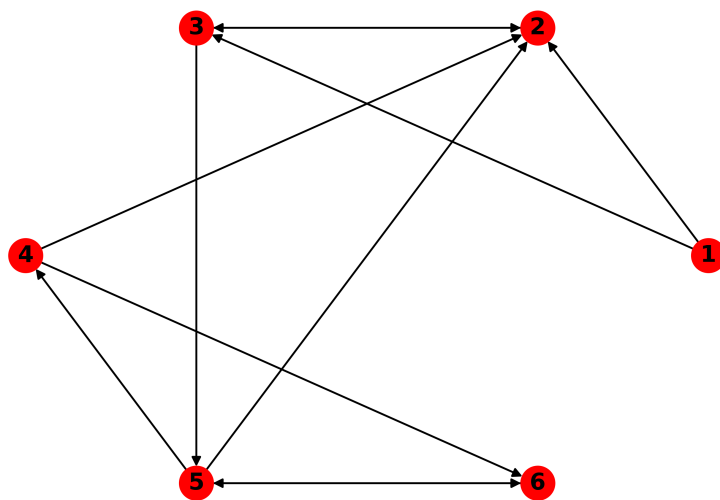


图 10.6: Python 所画的有向图

4. 图的其他表示和图数据的导出

描述图的方法有多种, 还可以使用邻接表 (adjacency list). 它列出了每个顶点的邻居顶点.

为了使描述更清楚, 可以将图表示为列表的字典. 这里, 顶点名称就是字典的键, 值是顶点的邻接表.

例 10.4: 图的相关操作示例.



Python 程序

```
import numpy as np
import networkx as nx
import pylab as plt

a = np.loadtxt("Pdata10_2.txt")
G = nx.Graph(a) # 利用邻接矩阵构造赋权无向图
print("图的顶点集为:", G.nodes(), "\n边集为:", G.edges())
print("邻接表为:", list(G.adjacency())) # 显示图的邻接表
print("列表字典为:", nx.to_dict_of_lists(G))
B = nx.to_numpy_matrix(G) # 从图G中导出邻接矩阵B, 这里B=a
C = nx.to_scipy_sparse_matrix(G) # 从图G中导出稀疏矩阵C
```

运行结果

图的顶点集为: [0, 1, 2, 3, 4]

边集为: [(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]


邻接表为: [(0, 1: 'weight': 9.0, 2: 'weight': 2.0, 3: 'weight': 4.0, 4: 'weight': 7.0), (1, 0: 'weight': 9.0, 2: 'weight': 3.0, 3: 'weight': 4.0), (2, 0: 'weight': 2.0, 1: 'weight': 3.0, 3: 'weight': 8.0, 4: 'weight': 4.0), (3, 0: 'weight': 4.0, 1: 'weight': 4.0, 2: 'weight': 8.0, 4: 'weight': 6.0), (4, 0: 'weight': 7.0, 2: 'weight': 4.0, 3: 'weight': 6.0)]

列表字典为: 0: [1, 2, 3, 4], 1: [0, 2, 3], 2: [0, 1, 3, 4], 3: [0, 1, 2, 4], 4: [0, 2, 3]

10.2 最短路算法及其 Python 实现

最短路径问题是图论中非常经典的问题之一, 旨在寻找图中两顶点之间的最短路径. 作为一个基本工具, 实际应用中的许多优化问题, 如管道铺设、线路安排、厂区布局、设备更新等, 都可被归结为最短路径问题来解决.

定义 10.9: 路的长度

设图 G 是赋权图, Γ 为 G 中的一条路. 则称 Γ 的各边权之和为路 Γ 的长度. 

对于 G 的两个顶点 u_0 和 v_0 , 从 u_0 到 v_0 的路一般不止一条, 其中最短的 (长度最小的) 一条称为从 u_0 到 v_0 的最短路; 最短路的长称为从 u_0 到 v_0 的距离, 记为 $d(u_0, v_0)$.

求最短路的算法有 Dijkstra (迪杰斯特拉) 标号算法和 Floyd (弗洛伊德) 算法等方法, 但 Dijkstra 标号算法只适用于边权是非负的情形. 最短路径问题也可以归结为一个 0-1 整数规划模型.



10.2.1 固定起点到其余各点的最短路算法

寻求从一固定起点 u_0 到其余各点的最短路, 最有效的算法之一是 E. W. Dijkstra 于 1959 年提出的 Dijkstra 算法. 这个算法是一种迭代算法, 它的依据有一个重要而明显的性质: 最短路是一条路, 最短路上的任一子段也是最短路.

对于给定的赋权图 $G = (V, E, W)$, 其中 $V = \{v_1, \dots, v_n\}$ 为顶点集合, E 为边的集合, 邻接矩阵 $W = (w_{ij})_{n \times n}$, 这里

$$w_{ij} = \begin{cases} v_i \text{ 与 } v_j \text{ 之间边的权值, } & v_i \text{ 与 } v_j \text{ 之间有边,} \\ \infty, & v_i \text{ 与 } v_j \text{ 之间无边,} \end{cases} \quad (i \neq j),$$

$$w_{ii} = 0, \quad i = 1, 2, \dots, n.$$

u_0 为 V 中的某个固定起点, 求顶点 u_0 到 V 中另一顶点 v_0 的最短距离 $d(u_0, v_0)$, 即为求 u_0 到 v_0 的最短路.

Dijkstra 算法的基本思想是: 按距离固定起点 u_0 从近到远为顺序, 依次求得 u_0 到图 G 某个顶点 v_0 或所有顶点的最短路和距离.

为避免重复并保留每一步的计算信息, 对于任意顶点 $v \in V$, 定义两个标号

$l(v)$: 顶点 v 的标号, 表示从起点 u_0 到 v 的当前路的长度;

$z(v)$: 顶点 v 的父顶点标号, 用以确定最短路的路线.

另外用 S_i 表示具有永久标号的顶点集. Dijkstra 标号算法的计算步骤如下.

(1) 令 $l(u_0) = 0$, 对 $v \neq u_0$, 令 $l(v) = \infty$, $z(v) = u_0$, $S_0 = \{u_0\}$, $i = 0$.

(2) 对每个 $v \in \bar{S}_i$ ($\bar{S}_i = V \setminus S_i$), 令

$$l(v) = \min_{u \in S_i} \{l(v), l(u) + w(uv)\},$$

这里 $w(uv)$ 表示顶点 u 和 v 之间边的权值, 如果此次迭代利用顶点 \tilde{u} 修改了顶点 v 的标号值 $l(v)$, 则 $z(v) = \tilde{u}$, 否则 $z(v)$ 不变. 计算 $\min_{v \in \bar{S}_i} \{l(v)\}$, 把达到这个最小值的一个顶点记为 u_{i+1} , 令 $S_{i+1} = S_i \cup \{u_{i+1}\}$.

(3) 若 $i = |V| - 1$ 或 v_0 进入 S_i , 算法终止; 否则, 用 $i + 1$ 代替 i , 转 (2).

算法结束时, 从 u_0 到各顶点 v 的距离由 v 的最后一次标号 $l(v)$ 给出. 在 v 进入 S_i 之前的标号 $l(v)$ 叫 T 标号, v 进入 S_i 时的标号 $l(v)$ 叫 P 标号. 算法就是不断修改各顶点的 T 标号, 直至获得 P 标号. 若在算法运行过程中, 将每一顶点获得 P 标号所得来的边在图上标明, 则算法结束时, u_0 至各顶点的最短路也在图上标示出来了.

例 10.5: 求图 10.7 所示的图 G 中从 v_3 到所有其余顶点的最短路及最短距离.



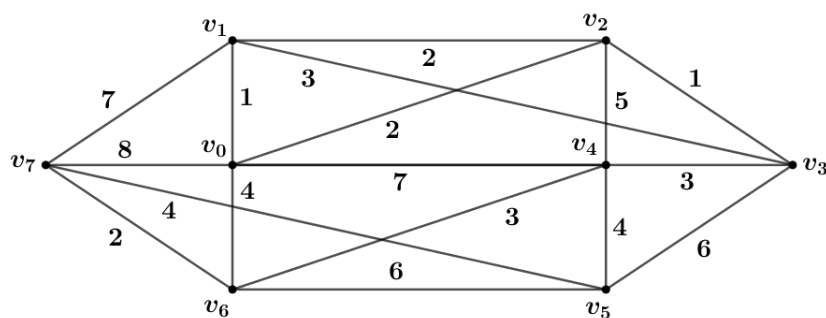


图 10.7

先写出邻接矩阵

$$W = \begin{bmatrix} 0 & 1 & 2 & \infty & 7 & \infty & 4 & 8 \\ 1 & 0 & 2 & 3 & \infty & \infty & \infty & 7 \\ 2 & 2 & 0 & 1 & 5 & \infty & \infty & \infty \\ \infty & 3 & 1 & 0 & 3 & 6 & \infty & \infty \\ 7 & \infty & 5 & 3 & 0 & 4 & 3 & \infty \\ \infty & \infty & \infty & 6 & 4 & 0 & 6 & 4 \\ 4 & \infty & \infty & \infty & 3 & 6 & 0 & 2 \\ 8 & 7 & \infty & \infty & \infty & 4 & 2 & 0 \end{bmatrix}.$$

Python 程序

```
import numpy as np
inf = np.inf

def Dijkstra_all_minpath(matr, start): # matr为邻接矩阵的数组, start表示
                                       # 起点
    n = len(matr) # 该图的节点数
    dis = []
    temp = []
    dis.extend(matr[start]) # 添加数组matr的start行元素
    temp.extend(matr[start]) # 添加矩阵matr的start行元素
    temp[start] = inf # 临时数组会把处理过的节点的值变成inf
    visited = [start] # start已处理
    parent = [start] * n # 用于画路径, 记录此路径中该节点的父节点
    while len(visited) < n:
        i = temp.index(min(temp)) # 找最小权值的节点的坐标
        temp[i] = inf
        for j in range(n):
            if j not in visited:
                if (dis[i] + matr[i][j]) < dis[j]:
                    dis[j] = temp[j] = dis[i] + matr[i][j]
```



```

parent[j] = i # 说明父节点是i
visited.append(i) # 该索引已经处理了
path = [] # 用于画路径
path.append(str(i))
k = i
while (parent[k] != start): # 找该节点的父节点添加到path, 直到父节点是
                           start
    path.append(str(parent[k]))
    k = parent[k]
path.append(str(start))
path.reverse() # path反序产生路径
print(str(i) + ': ', '->'.join(path)) # 打印路径
return dis

a = [[0, 1, 2, inf, 7, inf, 4, 8], [1, 0, 2, 3, inf, inf, inf, 7],
      [2, 2, 0, 1, 5, inf, inf, inf], [inf, 3, 1, 0, 3, 6, inf, inf],
      [7, inf, 5, 3, 0, 4, 3, inf], [inf, inf, inf, 6, 4, 0, 6, 4],
      [4, inf, inf, inf, 3, 6, 0, 2], [8, 7, inf, inf, inf, 4, 2, 0]]
d = Dijkstra_all_minpath(a, 3)
print("v3到所有顶点的最短距离为: ", d)

```

运行结果

```

2: 3->2
0: 3->2->0
1: 3->1
4: 3->4
5: 3->5
6: 3->4->6
7: 3->4->6->7
v3 到所有顶点的最短距离为: [3, 3, 1, 0, 3, 6, 6, 8]

```

例 10.6: (续例 10.5) 求图 10.7 所示的图 G 中从 v_3 到 v_7 的最短路及最短距离. 直接调用 networkx 库函数, 编写如下的 Python 程序:

Python 程序

```

import numpy as np
import networkx as nx

List = [(0, 1, 1), (0, 2, 2), (0, 4, 7), (0, 6, 4), (0, 7, 8), (1, 2, 2),
        (1, 3, 3), (1, 7, 7), (2, 3, 1), (2, 4, 5), (3, 4, 3), (3, 5, 6),

```



```
(4, 5, 4), (4, 6, 3), (5, 6, 6), (5, 7, 4), (6, 7, 2)]

G = nx.Graph()
G.add_weighted_edges_from(List)
A = nx.to_numpy_matrix(G, nodelist=range(8)) # 导出邻接矩阵
np.savetxt("Pdata10_6.txt", A)
p = nx.dijkstra_path(G, source=3, target=7, weight='weight') # 求最短路径

d = nx.dijkstra_path_length(G, 3, 7, weight="weight") # 求最短距离
print("最短路径为: ", p, "\n最短距离为: ", d)
```

运行结果

最短路径为: [3, 4, 6, 7]

最短距离为: 8

求得的最短路径为: $v_3 \rightarrow v_4 \rightarrow v_6 \rightarrow v_7$; 最短距离为 8.

注: 在利用 networkx 库函数计算时, 如果两个顶点之间没有边, 对应的邻接矩阵元素为 0, 而不是像数学理论上对应的邻接矩阵元素为 ∞ . 下面同样约定算法上的数学邻接矩阵和 networkx 库函数调用时的邻接矩阵是不同的.

10.2.2 每对顶点间的最短路算法

利用 Dijkstra 算法, 当然还可以寻求赋权图中所有顶点对之间的最短路. 具体方法是: 每次以不同的顶点作为起点, 用 Dijkstra 算法求出从该起点到其余顶点的最短路, 反复执行 $n-1$ (n 为顶点个数) 次这样的操作, 就可得到每对顶点之间的最短路. 但这样做需要大量的重复计算, 效率不高. 为此, R. W. Floyd 另辟蹊径, 于 1962 年提出了一个直接寻求任意两顶点之间最短路的算法.

对于赋权图 $G = (V, E, A_0)$, 其中顶点集 $V = \{v_1, \dots, v_n\}$, 邻接矩阵

$$A_0 = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix},$$

这里

$$a_{ij} = \begin{cases} v_i \text{ 与 } v_j \text{ 之间边的权值,} & v_i \text{ 与 } v_j \text{ 之间有边,} \\ \infty, & v_i \text{ 与 } v_j \text{ 之间无边,} \end{cases} \quad (i \neq j),$$

$$a_{ii} = 0, \quad i = 1, 2, \dots, n.$$

对于无向图, A_0 是对称矩阵, $a_{ij} = a_{ji}$, $i, j = 1, 2, \dots, n$.



Floyd 算法是一个经典的动态规划算法, 其基本思想是递推产生一个矩阵序列 $A_1, A_2, \dots, A_k, \dots, A_n$, 其中矩阵 $A_k = (a_k(i, j))_{n \times n}$, 其第 i 行第 j 列元素 $a_k(i, j)$ 表示从顶点 v_i 到顶点 v_j 的路径上所经过的顶点序号不大于 k 的最短路径长度.

计算时用迭代公式

$$a_k(i, j) = \min(a_{k-1}(i, j), a_{k-1}(i, k) + a_{k-1}(k, j)),$$

k 是迭代次数, $i, j, k = 1, 2, \dots, n$.

最后, 当 $k = n$ 时, A_n 即是各顶点之间的最短距离值.

如果在求得两点间的最短距离时, 还要求得两点间的最短路径, 需要在上面距离矩阵 A_k 的迭代过程中, 引入一个路由矩阵 $R_k = (r_k(i, j))_{n \times n}$ 来记录两点间路径的前驱后继关系, 其中 $r_k(i, j)$ 表示从顶点 v_i 到顶点 v_j 的路径经过编号为 $r_k(i, j)$ 的顶点.

路径矩阵的迭代过程如下.

(1) 初始时

$$R_0 = O_{n \times n}.$$

(2) 迭代公式为

$$R_k = (r_k(i, j))_{n \times n},$$

其中

$$r_k(i, j) = \begin{cases} k, & a_{k-1}(i, j) > a_{k-1}(i, k) + a_{k-1}(k, j), \\ r_{k-1}(i, j), & \text{否则.} \end{cases}$$

直到迭代到 $k = n$, 算法终止.

查找 v_i 到 v_j 最短路径的方法如下.

若 $r_n(i, j) = p_1$, 则点 v_{p_1} 是顶点 v_i 到顶点 v_j 的最短路的中间点, 然后用同样的方法再分头查找. 若

(1) 向顶点 v_i 反向追踪得: $r_n(i, p_1) = p_2, r_n(i, p_2) = p_3, \dots, r_n(i, p_s) = 0$;

(2) 向顶点 v_j 正向追踪得: $r_n(p_1, j) = q_1, r_n(q_1, j) = q_2, \dots, r_n(q_t, j) = 0$;

则由点 v_i 到 v_j 的最短路径为: $v_i, v_{p_s}, \dots, v_{p_2}, v_{p_1}, v_{q_1}, v_{q_2}, \dots, v_{q_t}, v_j$.

networkx 求所有顶点对之间最短路径的函数为

`shortes_path(G, source=None, target=None, weight=None, method='dijkstra')`

返回值是可迭代类型, 其中 `method` 可以取值 'dijkstra', 'bellman-ford'.

networkx 求所有顶点对之间最短距离的函数为

`shortest_path_length(G, source=None, target=None, weight=None, method='dijkstra')`

返回值是可迭代类型, 其中 `method` 可以取值 'dijkstra', 'bellman-ford'.

例 10.7: (续例 10.5) 求图 10.7 所示的图 G 中所有顶点对之间的最短距离.



Python 程序

```
import numpy as np

def floyd(graph):

    m = len(graph)
    dis = graph
    path = np.zeros((m, m)) # 路由矩阵初始化
    for k in range(m):
        for i in range(m):
            for j in range(m):
                if dis[i][k] + dis[k][j] < dis[i][j]:
                    dis[i][j] = dis[i][k] + dis[k][j]
                    path[i][j] = k
    return dis, path

inf = np.inf
a = np.array([[0, 1, 2, inf, 7, inf, 4, 8], [1, 0, 2, 3, inf, inf, inf, 7],
              [2, 2, 0, 1, 5, inf, inf, inf], [inf, 3, 1, 0, 3, 6, inf, inf],
              [7, inf, 5, 3, 0, 4, 3, inf], [inf, inf, inf, 6, 4, 0, 6, 4],
              [4, inf, inf, inf, 3, 6, 0, 2], [8, 7, inf, inf, inf, 4, 2, 0]]) # 输入邻接矩阵
dis, path = floyd(a)
print("所有顶点对之间的最短距离为:\n", dis, '\n', "路由矩阵为:\n", path)
```

运行结果

所有顶点对之间的最短距离为:

```
[[0. 1. 2. 3. 6. 9. 4. 6.] [1. 0. 2. 3. 6. 9. 5. 7.] [2. 2. 0. 1. 4. 7. 6. 8.] [3. 3. 1.
0. 3. 6. 6. 8.] [6. 6. 4. 3. 0. 4. 3. 5.] [9. 9. 7. 6. 4. 0. 6. 4.] [4. 5. 6. 6. 3. 6. 0. 2.]
[6. 7. 8. 8. 5. 4. 2. 0.]]
```

路由矩阵为:

```
[[0. 0. 0. 2. 3. 3. 0. 6.] [0. 0. 0. 0. 3. 3. 0. 0.] [0. 0. 0. 0. 3. 3. 0. 6.] [2. 0. 0.
0. 0. 0. 4. 6.] [3. 3. 3. 0. 0. 0. 0. 6.] [3. 3. 3. 0. 0. 0. 0. 0.] [0. 0. 0. 4. 0. 0. 0. 0.]
[6. 0. 6. 6. 6. 0. 0. 0.]]
```

例 10.8: (续例 10.5) 求图 10.7 所示的图 G 中所有顶点对之间的最短距离和最短路径。直接调用 networkx 库函数, 编写的程序如下:



Python 程序

```
import numpy as np
import networkx as nx
a = np.loadtxt("Pdata10_6.txt")
G = nx.Graph(a) # 利用邻接矩阵构造赋权无向图
d = nx.shortest_path_length(G, weight='weight') # 返回值是可迭代类型
Ld = dict(d) # 转换为字典类型
print("顶点对之间的距离为:", Ld) # 显示所有顶点对之间的最短距离
print("顶点0到顶点4的最短距离为:", Ld[0][4]) # 显示一对顶点之间的最短距离

m, n = a.shape
dd = np.zeros((m, n))
for i in range(m):
    for j in range(n):
        dd[i, j] = Ld[i][j]
print("顶点对之间最短距离的数组表示为:\n", dd) # 显示所有顶点对之间最短距离

np.savetxt('Pdata10_8.txt', dd) # 把最短距离数组保存到文本文件中
p = nx.shortest_path(G, weight='weight') # 返回值是可迭代类型
dp = dict(p) # 转换为字典类型
print("\n顶点对之间的最短路径为:", dp)
print("顶点 0 到顶点 4 的最短路径为:", dp[0][4])
```

运行结果

顶点对之间的距离为:

```
{0: {0: 0, 1: 1.0, 2: 2.0, 3: 3.0, 6: 4.0, 4: 6.0, 7: 6.0, 5: 9.0},
 1: {1: 0, 0: 1.0, 2: 2.0, 3: 3.0, 6: 5.0, 4: 6.0, 7: 7.0, 5: 9.0},
 2: {2: 0, 3: 1.0, 0: 2.0, 1: 2.0, 4: 4.0, 6: 6.0, 5: 7.0, 7: 8.0},
 3: {3: 0, 2: 1.0, 1: 3.0, 4: 3.0, 0: 3.0, 5: 6.0, 6: 6.0, 7: 8.0},
 4: {4: 0, 3: 3.0, 6: 3.0, 5: 4.0, 2: 4.0, 7: 5.0, 1: 6.0, 0: 6.0},
 5: {5: 0, 4: 4.0, 7: 4.0, 3: 6.0, 6: 6.0, 2: 7.0, 1: 9.0, 0: 9.0},
 6: {6: 0, 7: 2.0, 4: 3.0, 0: 4.0, 1: 5.0, 5: 6.0, 3: 6.0, 2: 6.0},
 7: {7: 0, 6: 2.0, 5: 4.0, 4: 5.0, 0: 6.0, 1: 7.0, 3: 8.0, 2: 8.0}}
```

顶点 0 到顶点 4 的最短距离为: 6.0

顶点对之间最短距离的数组表示为:

```
[[0. 1. 2. 3. 6. 9. 4. 6.]
 [1. 0. 2. 3. 6. 9. 5. 7.]
 [2. 2. 0. 1. 4. 7. 6. 8.]
 [3. 3. 1. 0. 3. 6. 6. 8.]
```



[6. 6. 4. 3. 0. 4. 3. 5.]

[9. 9. 7. 6. 4. 0. 6. 4.]

[4. 5. 6. 6. 3. 6. 0. 2.]

[6. 7. 8. 8. 5. 4. 2. 0.]]

顶点对之间的最短路径为:

{0: {0: [0], 1: [0, 1], 2: [0, 2], 4: [0, 2, 3, 4], 6: [0, 6], 7: [0, 6, 7], 3: [0, 2, 3], 5: [0, 2, 3, 5]},

1: {1: [1], 0: [1, 0], 2: [1, 2], 3: [1, 3], 7: [1, 7], 4: [1, 3, 4], 6: [1, 0, 6], 5: [1, 3, 5]},

2: {2: [2], 0: [2, 0], 1: [2, 1], 3: [2, 3], 4: [2, 3, 4], 5: [2, 3, 5], 6: [2, 0, 6], 7: [2, 0, 6, 7]},

3: {3: [3], 1: [3, 1], 2: [3, 2], 4: [3, 4], 5: [3, 5], 0: [3, 2, 0], 7: [3, 4, 6, 7], 6: [3, 4, 6]},

4: {4: [4], 0: [4, 3, 2, 0], 2: [4, 3, 2], 3: [4, 3], 5: [4, 5], 6: [4, 6], 1: [4, 3, 1], 7: [4, 6, 7]},

5: {5: [5], 3: [5, 3], 4: [5, 4], 6: [5, 6], 7: [5, 7], 0: [5, 3, 2, 0], 2: [5, 3, 2], 1: [5, 3, 1]},

6: {6: [6], 0: [6, 0], 4: [6, 4], 5: [6, 5], 7: [6, 7], 1: [6, 0, 1], 2: [6, 0, 2], 3: [6, 4, 3]},

7: {7: [7], 0: [7, 6, 0], 1: [7, 1], 5: [7, 5], 6: [7, 6], 4: [7, 6, 4], 3: [7, 6, 4, 3], 2: [7, 6, 0, 2]}}

顶点 0 到顶点 4 的最短路径为: [0, 2, 3, 4]

求得顶点对之间的最短距离矩阵为

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 6 & 9 & 4 & 6 \\ 1 & 0 & 2 & 3 & 6 & 9 & 5 & 7 \\ 2 & 2 & 0 & 1 & 4 & 7 & 6 & 8 \\ 3 & 3 & 1 & 0 & 3 & 6 & 6 & 8 \\ 6 & 6 & 4 & 3 & 0 & 4 & 3 & 5 \\ 9 & 9 & 7 & 6 & 4 & 0 & 6 & 4 \\ 4 & 5 & 6 & 6 & 3 & 6 & 0 & 2 \\ 6 & 7 & 8 & 8 & 5 & 4 & 2 & 0 \end{bmatrix}.$$

求得的顶点对之间的最短路径这里就不一一列举了.

10.2.3 最短路应用范例

例 10.9: (设备更新问题) 某种工程设备的役龄为 4 年, 每年年初都面临着是否更新的问题: 若卖旧买新, 就要支付一定的购置费用; 若继续使用, 则要支付更多的维护费用, 且使用年限越长维护费用越多. 若役龄期内每年的年初购置价格、当年维护费用及年



末剩余净值如表 10.1 所示. 请为该设备制订一个 4 年役龄期内的更新计划, 使总的支付费用最少.

表 10.1: 相关费用数据

年份	1	2	3	4
年初购置价格/万元	25	26	28	31
当年维护费用/万元	10	14	18	26
年末剩余净值/万元	20	16	13	11

可以把这个问题化为图论中的最短路问题.

构造赋权有向图 $D = (V, A, W)$, 其中顶点集 $V = \{v_1, v_2, \dots, v_5\}$, 这里 v_i ($i = 1, 2, 3, 4$) 表示第 i 年年初的时刻, v_5 表示第 4 年年末的时刻, A 为弧的集合, 邻接矩阵 $W = (w_{ij})_{5 \times 5}$, 这里 w_{ij} 为第 i 年年初至第 j 年年初 (或 $j-1$ 年年末) 期间所支付的费用, 计算公式为

$$w_{ij} = p_i + \sum_{k=1}^{j-i} a_k - r_{j-i},$$

其中 p_i 为第 i 年年初的购置价格, a_k 为使用到第 k 年当年的维护费用, r_i 为使用 i 年旧设备的出售价格 (残值). 则邻接矩阵

$$W = \begin{bmatrix} 0 & 15 & 33 & 54 & 82 \\ \infty & 0 & 16 & 34 & 55 \\ \infty & \infty & 0 & 18 & 36 \\ \infty & \infty & \infty & 0 & 21 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}.$$

那么制订总的支付费用最小的设备更新计划, 就是在图 10.8 所示的有向图 D 中求从顶点 v_1 到顶点 v_5 的费用最短路.

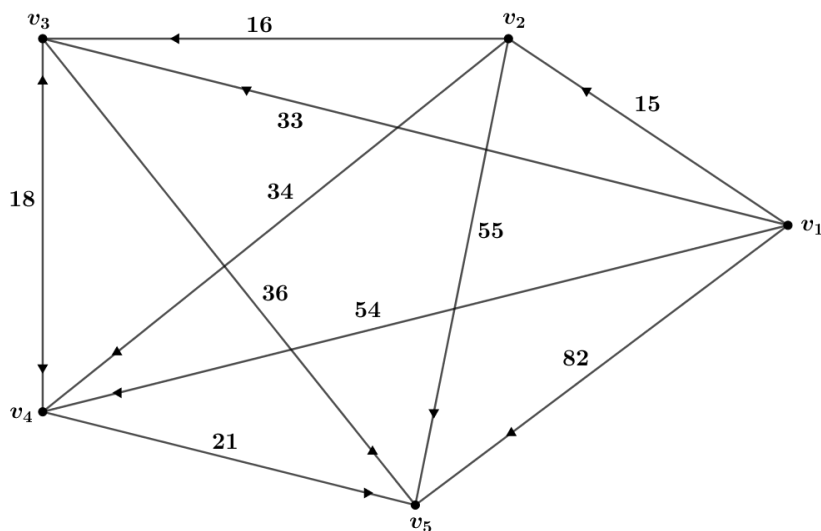


图 10.8



利用 Dijkstra 算法, 使用 Python 软件, 求得 v_1 到 v_5 的最短路径为 $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5$, 最短路径的长度为 67. 即设备更新计划为第 1 年年初买进新设备, 使用到第 1 年年底, 第 2 年年初购进新设备, 使用到第 2 年年底, 第 3 年年初再购进新设备, 使用到第 4 年年底.

计算及画图的 Python 程序如下:

Python 程序

```
import numpy as np
import networkx as nx
import pylab as plt

p = [25, 26, 28, 31]
a = [10, 14, 18, 26]
r = [20, 16, 13, 11]

b = np.zeros((5, 5)) # 邻接矩阵 (非数学上的邻接矩阵) 初始化
for i in range(5):
    for j in range(i + 1, 5):
        b[i, j] = p[i] + np.sum(a[0:j - i]) - r[j - i - 1]
print(b)
G = nx.DiGraph(b)
p = nx.dijkstra_path(G, source=0, target=4, weight='weight') # 求最短路径;

print("最短路径为:", np.array(p) + 1) # python下标从0开始
d = nx.dijkstra_path_length(G, 0, 4, weight='weight') # 求最短距离
print("所求的费用最小值为:", d)
s = dict(zip(range(5), range(1, 6))) # 构造用于顶点标注的标号字典
plt.rc('font', size=16)
pos = nx.shell_layout(G) # 设置布局
w = nx.get_edge_attributes(G, 'weight')
nx.draw(G, pos, font_weight='bold', labels=s, node_color='r')
nx.draw_networkx_edge_labels(G, pos, edge_labels=w)
path_edges = list(zip(p, p[1:]))
nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='r', width=3)

plt.savefig("figure10_9.png", pdi=500)
plt.show()
```



运行结果

[[0. 15. 33. 54. 82.] [0. 0. 16. 34. 55.] [0. 0. 0. 18. 36.] [0. 0. 0. 0. 21.] [0. 0. 0. 0. 0.]]

最短路径为: [1 2 3 5]

所求的费用最小值为: 67.0

重心问题 有些公共服务设施 (例如邮局、学校等) 的选址, 要求设施到所有服务对象点的距离总和最小. 一般要考虑人口密度问题, 或者全体被服务对象来往的总路程最短.

例 10.10: 某矿区有六个产矿点, 如图 10.9 所示. 已知各产矿点每天的产矿量 (标在图 10.9 的各顶点旁) 为 q_i ($i = 1, 2, \dots, 6$), 现要从这六个产矿点选一个来建造选矿厂, 问应选在哪个产矿点, 才能使各产矿点所产的矿石运到选矿厂所在地的总运力 ($t \cdot \text{km}$) 最小.

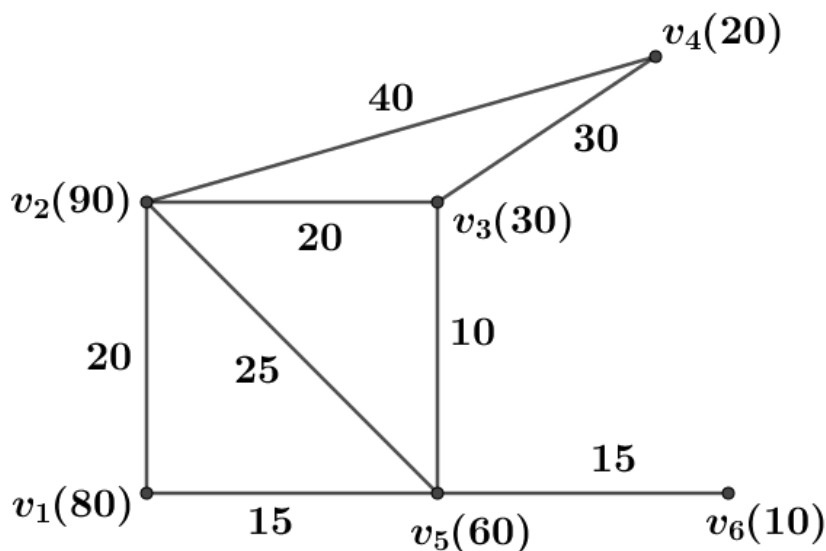


图 10.9: 各产矿点分布图

令 d_{ij} ($i, j = 1, 2, \dots, 6$) 表示顶点 v_i 与 v_j 之间的距离. 若选矿厂设在 v_i 并且各产矿点到选矿厂的总运力为 m_i , 则确定选矿厂的位置就转化为求 m_k , 使得 $m_k = \min_{1 \leq i \leq 6} m_i$.

由于各产矿点到选矿厂的总运力依赖于任意两顶点之间的距离, 即任意两顶点之间最短路的长度, 因此可首先利用 Dijkstra (或 Floyd) 算法求出所有顶点之间的最短距离, 然后计算出顶点 v_i 作为设立选矿厂时各产矿点到 v_i 的总运力

$$m_i = \sum_{j=1}^6 q_j d_{ij}, \quad i = 1, 2, \dots, 6.$$

具体的计算结果见表 10.2.



表 10.2: 各顶点对之间的最短距离和总运力计算数据

产矿点	v_1	v_2	v_3	v_4	v_5	v_6	总运力 m_i
v_1	0	20	25	55	15	30	4850
v_2	20	0	20	40	25	40	4900
v_3	25	20	0	30	10	25	5250
v_4	55	40	30	0	40	55	11850
v_5	15	25	10	40	0	15	4700
v_6	30	40	25	55	15	0	8750

最后利用 $m_5 = \min_{1 \leq i \leq 6} m_i$, 求得 v_5 为设置选矿厂的位置.

计算的 Python 程序如下:

Python 程序

```
import numpy as np
import networkx as nx

List = [(1, 2, 20), (1, 5, 15), (2, 3, 20), (2, 4, 40), (2, 5, 25), (3, 4, 30),
        (3, 5, 10), (5, 6, 15)]
G = nx.Graph()
G.add_nodes_from(range(1, 7))
G.add_weighted_edges_from(List)
c = dict(nx.shortest_path_length(G, weight='weight'))
d = np.zeros((6, 6))
for i in range(1, 7):
    for j in range(1, 7):
        d[i - 1, j - 1] = c[i][j]
print('各顶点对之间的最短距离: \n', d)
q = np.array([80, 90, 30, 20, 60, 10])
m = d @ q # 计算运力, 这里使用矩阵乘法
mm = m.min() # 求运力的最小值
ind = np.where(m == mm)[0] + 1 # python下标从0开始, np.where返回值为元组
print("运力 m=", m, '\n最小运力 mm=', mm, "\n选矿厂的设置位置为: ", ind)
```

运行结果

各顶点对之间的最短距离:

```
[[ 0. 20. 25. 55. 15. 30.] [20. 0. 20. 40. 25. 40.] [25. 20. 0. 30. 10. 25.] [55. 40.
30. 0. 40. 55.] [15. 25. 10. 40. 0. 15.] [30. 40. 25. 55. 15. 0.]]
```



运力 $m = [4850. 4900. 5250. 11850. 4700. 8750.]$

最小运力 $mm = 4700.0$

选矿厂的设置位置为: [5]

10.3 最小生成树算法及其 networkx 实现

树 (tree) 是图论中非常重要的一类图, 它非常类似于自然界中的树, 结构简单, 应用广泛, 最小生成树问题则是其中的经典问题之一. 在实际应用中, 许多问题的图论模型都是最小生成树, 如通信网络建设、有线电视铺设、加工设备分组等.

10.3.1 基本概念

定义 10.10: 树

连通的无圈图称为树.



例如, 图 10.10 给出的 G_1 是树, 但 G_2 和 G_3 则不是树.

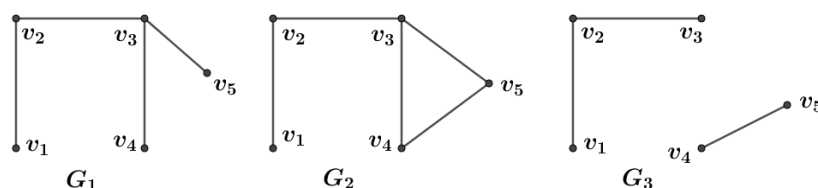


图 10.10: 树与非树

定理 10.2

设 G 是具有 n 个顶点 m 条边的图, 则以下命题等价.

- (1) 图 G 是树;
- (2) 图 G 中任意两个不同顶点之间存在唯一的路;
- (3) 图 G 连通, 删除任一条边均不连通;
- (4) 图 G 连通, 且 $n = m + 1$;
- (5) 图 G 无圈, 添加任一条边可得唯一的圈;
- (6) 图 G 无圈, 且 $n = m + 1$.



定义 10.11: 生成树

若图 G 的生成子图 H 是树, 则称 H 为 G 的生成树或支撑树.



一个图的生成树通常不唯一.

定理 10.3

连通图的生成树一定存在.



证明： 给定连通图 G , 若 G 无圈, 则 G 本身就是自己的生成树. 若 G 有圈, 则任取 G 中一个圈 C , 记删除 C 中一条边后所得之图为 G' . 显然 G' 中圈 C 已经不存在, 但 G' 仍然连通. 若 G' 中还有圈, 再重复以上过程, 直至得到一个无圈的连通图 H . 易知 H 是 G 的生成树. \square

定理 10.3 的证明方法也是求生成树的一种方法, 称为“破圈法”.

定义 10.12: 最小生成树

在赋权图 G 中, 边权之和最小的生成树称为 G 的最小生成树.



一个简单连通图只要不是树, 其生成树一般不唯一, 而且非常多. 一般地, n 个顶点的完全图, 其不同生成树的个数为 n^{n-2} . 因而, 寻求一个给定赋权图的最小生成树, 一般是不能用枚举法的. 例如, 20 个顶点的完全图有 20^{18} 个生成树, 20^{18} 有 24 位. 所以, 通过枚举求最小生成树是无效的算法, 必须寻求有效的算法.

10.3.2 求最小生成树的算法

对于赋权连通图 $G = (V, E, W)$, 其中 V 为顶点集合, E 为边的集合, W 为邻接矩阵, 这里顶点集合 V 中有 n 个顶点, 构造它的最小生成树. 构造连通图最小生成树的算法有 Kruskal 算法和 Prim 算法.

1. Kruskal 算法

Kruskal 算法思想: 每次将一条权最小的边加入子图 T 中, 并保证不形成圈. Kruskal 算法如下:

- (1) 选 $e_1 \in E$, 使得 e_1 是权值最小的边.
- (2) 若 e_1, e_2, \dots, e_i 已选好, 则从 $E - \{e_1, e_2, \dots, e_i\}$ 中选取 e_{i+1} , 使得
 - (i) $\{e_1, e_2, \dots, e_i, e_{i+1}\}$ 中无圈,
 - (ii) e_{i+1} 是 $E - \{e_1, e_2, \dots, e_i\}$ 中权值最小的边.
- (3) 直到选得 e_{n-1} 为止.

例 10.11: 用 Kruskal 算法求如图 10.11 所示连通图的最小生成树.

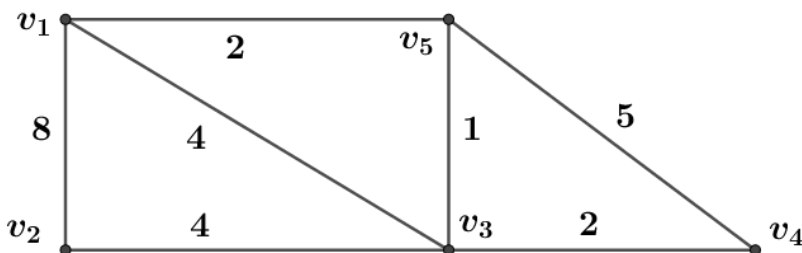


图 10.11: 树与非树

解： 首先将给定图 G 的边按照权值从小到大进行排序, 如表 10.3 所列.



表 10.3: 按照权值排列的边数据

边	(v_3, v_5)	(v_1, v_5)	(v_3, v_4)	(v_1, v_3)	(v_2, v_3)	(v_4, v_5)	(v_1, v_2)
取值	1	2	2	4	4	5	8

其次, 依照 Kruskal 算法的步骤, 迭代 4 步完成最小生成树的构造. 按照边的排列顺序, 前三次取定

$$e_1 = (v_3, v_5), \quad e_2 = (v_1, v_5), \quad e_3 = (v_3, v_4).$$

由于下一个未选边中的最小权边 (v_1, v_3) 与已选边 e_1, e_2 构成圈, 所以排除. 第 4 次选 $e_4 = (v_2, v_3)$, 得到图 10.12 所示的树就是图 G 的一棵最小生成树, 它的权值是 9.

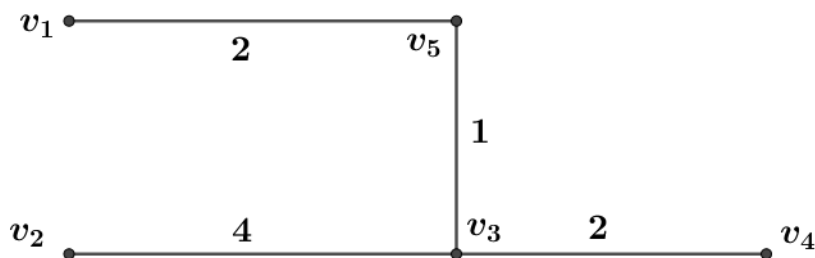


图 10.12: 生成的最小生成树

□

2. Prim 算法

设置两个集合 P 和 Q , 其中 P 用于存放 G 的最小生成树中的顶点, 集合 Q 存放 G 的最小生成树中的边. 令集合 P 的初值为 $P = \{v_1\}$ (假设构造最小生成树时, 从顶点 v_1 出发), 集合 Q 的初值为 $Q = \emptyset$ (空集). Prim 算法的思想是, 从所有 $p \in P, v \in V - P$ 的边中, 选取具有最小权值的边 pv , 将顶点 v 加入集合 P 中, 将边 pv 加入集合 Q 中, 如此不断重复, 直到 $P = V$ 时, 最小生成树构造完毕, 这时集合 Q 中包含了最小生成树的所有边.

Prim 算法如下:

(1) $P = \{v_1\}, Q = \emptyset$;

(2) while $P \neq V$

找最小边 pv , 其中 $p \in P, v \in V - P$;

$$P = P + \{v\};$$

$$Q = Q + \{pv\};$$

end

例 10.12: (续例 10.11) 用 Prim 算法求图 10.11 所示连通图的最小生成树.

解: 按照 Prim 算法的步骤, 迭代 4 步完成最小生成树的构造.

(0) 第 0 步初始化, 顶点集 $P = \{v_1\}$, 边集 $Q = \emptyset$;



- (1) 第 1 步, 找到最小边 (v_1, v_5) , $P = \{v_1, v_5\}$, $Q = \{(v_1, v_5)\}$;
 - (2) 第 2 步, 找到最小边 (v_3, v_5) , $P = \{v_1, v_3, v_5\}$, $Q = \{(v_1, v_5), (v_3, v_5)\}$;
 - (3) 第 3 步, 找到最小边 (v_3, v_4) , $P = \{v_1, v_3, v_4, v_5\}$, $Q = \{(v_1, v_5), (v_3, v_5), (v_3, v_4)\}$;
 - (4) 第 4 步, 找到最小边 (v_2, v_3) , $P = \{v_1, v_2, v_3, v_4, v_5\}$, $Q = \{(v_1, v_5), (v_3, v_5), (v_3, v_4), (v_2, v_3)\}$
- 最小生成树构造完毕. □

10.3.3 用 networkx 求最小生成树及应用

networkx 求最小生成树函数为 `minimum_spanning_tree`, 其调用格式为

`T = minimum_spanning_tree (G, weight='weight', algorithm='kruskal')`

其中 G 为输入的图, `algorithm` 的取值有三种字符串: 'kruskal', 'prim' 或 'boruvka', 缺省值为 'kruskal'; 返回值 T 为所求得的最小生成树的可迭代对象.

例 10.13: (续例 10.11) 利用 networkx 的 Kruskal 算法求例 10.11 的最小生成树.

Python 程序

```
import numpy as np
import networkx as nx
import pylab as plt
L = [(1, 2, 8), (1, 3, 4), (1, 5, 2), (2, 3, 4), (3, 4, 2), (3, 5, 1),
     (4, 5, 5)]
b = nx.Graph()
b.add_nodes_from(range(1, 6))
b.add_weighted_edges_from(L)
T = nx.minimum_spanning_tree(b) # 返回可迭代对象
w = nx.get_edge_attributes(T, 'weight') # 提取字典数据
TL = sum(w.values()) # 计算最小生成树的长度
print("最小生成树为:", w)
print("最小生成树的长度为: ", TL)
pos = nx.shell_layout(b)
nx.draw(T, pos, node_size=280, with_labels=True, node_color='r')
nx.draw_networkx_edge_labels(T, pos, edge_labels=w)
plt.show()
```

运行结果

最小生成树为: (1, 5): 2, (2, 3): 4, (3, 5): 1, (3, 4): 2

最小生成树的长度为: 9

所求的最小生成树如图 10.13 所示.



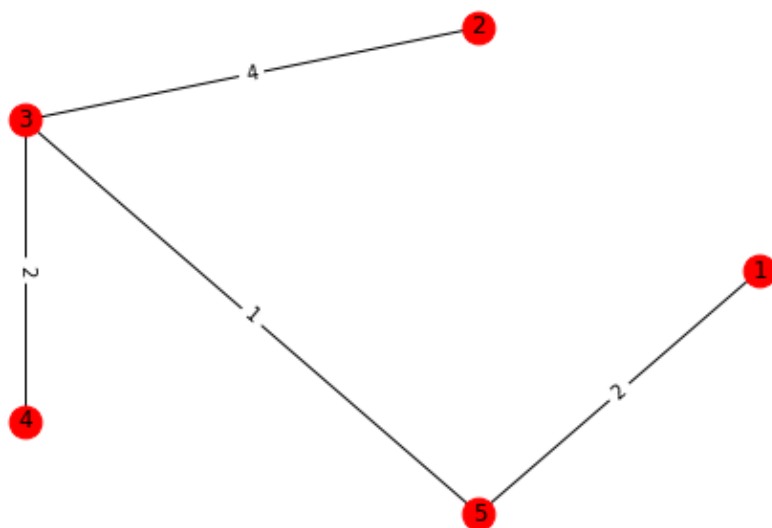


图 10.13: 最小生成树

例 10.14: 已知 8 口油井, 相互之间的距离如表 10.4 所列. 已知 1 号油井离海岸最近, 为 5 n mile (1n mile = 1.852 km). 问从海岸经 1 号油井铺设油管将各油井连接起来, 应如何铺设使油管长度最短.

表 10.4: 各油井间距离

(单位: n mile)

	2	3	4	5	6	7	8
1	1.3	2.1	0.9	0.7	1.8	2.0	1.5
2		0.9	1.8	1.2	2.6	2.3	1.1
3			2.6	1.7	2.5	1.9	1.0
4				0.7	1.6	1.5	0.9
5					0.9	1.1	0.8
6						0.6	1.0
7							0.5

这是一个求最小生成树的问题, 利用 Python 程序求解如下:

Python 程序

```
import numpy as np
import networkx as nx
import pandas as pd
import pylab as plt

a = pd.read_excel("Pdata10_14.xlsx", header=None)
b = a.values
```



```

b[np.isnan(b)] = 0
c = np.zeros((8, 8)) # 邻接矩阵初始化
c[0:7, 1:8] = b # 构造图的邻接矩阵
G = nx.Graph(c)
T = nx.minimum_spanning_tree(G) # 返回可迭代对象
d = nx.to_numpy_matrix(T) # 返回最小生成树的邻接矩阵
print("邻接矩阵 c=\n", d)
W = d.sum() / 2 + 5 # 求油管长度
print("油管长度 W=", W)
s = dict(zip(range(8), range(1, 9))) # 构造用于顶点标注的标号字典
plt.rc('font', size=16)
pos = nx.shell_layout(G)
nx.draw(T, pos, node_size=280, labels=s, node_color='r')
w = nx.get_edge_attributes(T, 'weight')
nx.draw_networkx_edge_labels(T, pos, edge_labels=w)
plt.savefig('figure10_14.png')
plt.show()

```

运行结果

邻接矩阵 c=

```

[[0. 0. 0. 0. 0.7 0. 0. 0. ]
 [0. 0. 0.9 0. 0. 0. 0. 0. ]
 [0. 0.9 0. 0. 0. 0. 0. 1. ]
 [0. 0. 0. 0. 0.7 0. 0. 0. ]
 [0.7 0. 0. 0.7 0. 0. 0. 0.8]
 [0. 0. 0. 0. 0. 0. 0.6 0. ]
 [0. 0. 0. 0. 0. 0.6 0. 0.5]
 [0. 0. 1. 0. 0.8 0. 0.5 0. ]]

```

油管长度 W= 10.2.

求得的油管长度的最小值为 10.2n mile. 所求的最小生成树如图 10.14 所示.



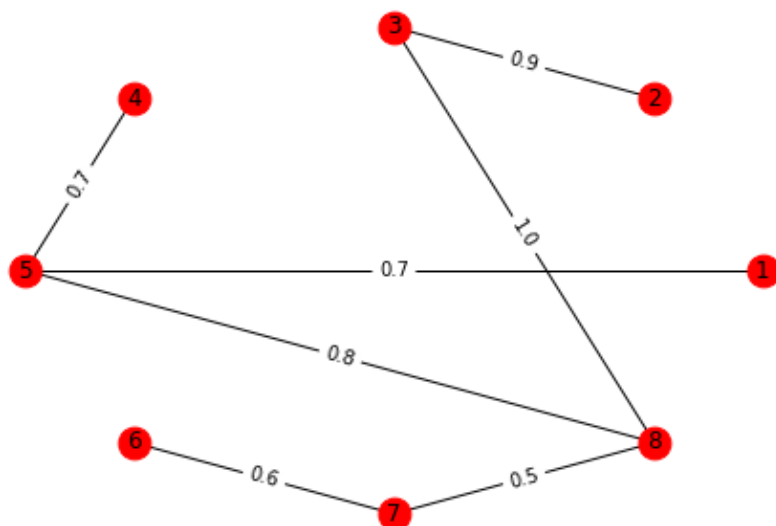


图 10.14: 油井连接示意图

10.4 匹配问题

定义 10.13

在图 $G = (V, E)$ 中, 若 $M \subset E, \forall e_i, e_j \in M, e_i$ 与 e_j 无公共端点 ($i \neq j$), 则称 M 为图 G 中的一个对集; M 中的一条边的两个端点叫做在对集 M 中相配; M 中的端点称为被 M 许配; G 中每个顶点皆被 M 许配时, M 称为完美对集; G 中已无使 $|M'| > |M|$ 的对集 M' , 则 M 称为最大对集; 若 G 中有一轨, 其边交替地在对集 M 内外出现, 则称此轨为 M 的交错轨, 交错轨的起止顶点都未被许配时, 此交错轨称为可增广轨。



若把可增广轨上在 M 外的边纳入对集, 把 M 内的边从对集中删除, 则被许配的顶点数增加 2, 对集中的“对儿”增加一个。

1957 年, 贝尔热 (Berge) 得到最大对集的充要条件:

定理 10.4

M 是图 G 中的最大对集当且仅当 G 中无 M 可增广轨。



1935 年, 霍尔 (Hall) 得到下面的许配定理:

定理 10.5: 许配定理

G 为二分图, X 与 Y 是顶点集的划分, G 中存在把 X 中顶点皆许配的对集的充要条件是, $\forall S \subset X$, 有 $|N(S)| \geq |S|$, 其中 $N(S)$ 是 S 中顶点的邻集。



由上述定理可以得出:

推论 若 G 是 k 次 ($k > 0$) 正则二分图, 则 G 有完美对集. 所谓 k 次正则图, 即每个顶点皆为 k 度的图。



由此推论得出下面的婚配定理:

定理 10.6: 婚配定理

每个姑娘都结识 $k(k \geq 1)$ 个小伙子, 每个小伙子都结识 k 个姑娘, 则每个姑娘都能和她认识的一个小伙子结婚, 并且每个小伙子也能和他认识的一个姑娘结婚.



人员分派问题等实际问题可以化成对集来解决.

10.4.1 人员分派问题

工作人员 x_1, x_2, \dots, x_n 去做 n 件工作 y_1, y_2, \dots, y_n , 每人适合做其中一件或几件, 问能否每人都有一份适合的工作? 如果不能, 最多几人可以有适合的工作?

这个问题的数学模型是: $G = (V, E)$ 是二分图, 顶点集划分为 $V = X \cup Y, X = \{x_1, \dots, x_n\}, Y = \{y_1, \dots, y_n\}$, 当且仅当 x_i 适合做工作 y_j 时, $x_i y_j \in E$, 求 G 中的最大对集.

解决这个问题可以利用 1965 年埃德蒙兹 (Edmonds) 提出的匈牙利算法.

匈牙利算法

- (1) 从 G 中任意取定一个初始对集 M .
- (2) 若 M 把 X 中的顶点皆许配, 停止, M 即完美对集; 否则取 X 中未被 M 许配的一顶点 u , 记 $S = \{u\}, T = \emptyset$.
- (3) 若 $N(S) = T$, 停止, 无完美对集; 否则取 $y \in N(S) - T$.
- (4) 若 y 是被 M 许配的, 设 $yz \in M, S = S \cup \{z\}, T = T \cup \{y\}$, 转 (3); 否则, 取可增广轨 $P(u, y)$, 令 $M = (M - E(P)) \cup (E(P) - M)$, 这里 $E(P)$ 表示增广轨 P 上的边, 转 (2).

把以上算法稍加修改就能够用来求二分图的最大完美对集.

10.4.2 最优分派问题

在人员分派问题中, 工作人员适合做的各项工作效益未必一致, 需要制订一个分派方案, 使公司总效益最大.

这个问题的数学模型是: 在人员分派问题的模型中, 图 $G = (V, E, W)$ 为赋权图, 每边加了权 $w(x_i y_j) \geq 0$, 表示 x_i 干 y_j 工作的效益, 求赋权图 G 的权最大的完美对集.

解决这个问题可以用库恩-曼克莱斯 (Kuhn-Munkres) 算法. 为此, 要引入可行顶点标号与相等子图的概念.

定义 10.14


在赋权二分图 $G = (V, E, W)$ 中, 若映射 $l: V(G) \rightarrow \mathbb{R}$, 满足 $\forall x \in X, y \in Y$,

$$l(x) + l(y) \geq w(xy),$$



则称 l 是二分图 G 的可行顶点标号. 令

$$E_l = \{xy \mid xy \in E(G), l(x) + l(y) = w(xy)\},$$

称以 E_l 为边集的 G 的生成子图为相等子图, 记作 G_l . 

可行顶点标号是存在的. 例如

$$l(x) = \max_{y \in Y} w(xy), \quad x \in X; \quad l(y) = 0, \quad y \in Y.$$

定理 10.7

G_l 的完美对集即为 G 的权最大的完美对集. 

Kuhn-Munkres 算法

- (1) 选定初始可行顶点标号 l , 确定 G_l , 在 G_l 中选取一个对集 M .
- (2) 若 X 中顶点皆被 M 许配, 停止, M 即 G 的权最大的完美对集; 否则, 取 G_l 中未被 M 许配的顶点 u , 令 $S = \{u\}, T = \emptyset$.
- (3) 若 $N_{G_l}(S) \supset T$, 转 (4); 若 $N_{G_l}(S) = T$, 取

$$\alpha_l = \min_{x \in S, y \notin T} \{l(x) + l(y) - w(xy)\},$$

$$\bar{l}(v) = \begin{cases} l(v) - \alpha_l, & v \in S, \\ l(v) + \alpha_l, & v \in T, \\ l(v), & \text{其他.} \end{cases}$$

$$l = \bar{l}, \quad G_l = G_{\bar{l}}.$$

- (4) 选 $N_{G_l}(S) - T$ 中一顶点 y , 若 y 已被 M 许配, 且 $yz \in M$, 则 $S = S \cup \{z\}$, $T = T \cup \{y\}$, 转 (3); 否则, 取 G_l 中一个 M 可增广轨 $P(u, y)$, 令

$$M = (M - E(P)) \cup (E(P) - M),$$

转 (2). 其中, $N_{G_l}(S)$ 是 G_l 中 S 的相邻顶点集.

例 10.15: 假设要分配 5 个人做 5 项不同工作, 每个人做不同工作产生的效益由邻接矩阵

$$W = (w_{ij})_{5 \times 5} = \begin{bmatrix} 3 & 5 & 5 & 4 & 1 \\ 2 & 2 & 0 & 2 & 2 \\ 2 & 4 & 4 & 1 & 0 \\ 0 & 2 & 2 & 1 & 0 \\ 1 & 2 & 1 & 3 & 3 \end{bmatrix}$$



表示, 即 w_{ij} ($i, j = 1, 2, \dots, 5$) 表示第 i 个人干第 j 项工作的效益, 试求使效益达到最大的分配方案.

解: 构造赋权图 $G = (V, E, \tilde{W})$, 顶点集 $V = \{v_1, v_2, \dots, v_{10}\}$, 其中 v_1, v_2, \dots, v_5 分别表示 5 个人, $v_6, v_7, v_8, v_9, v_{10}$ 分别表示 5 项工作, 邻接矩阵为

$$\tilde{W} = \begin{bmatrix} \mathbf{O} & \mathbf{W} \\ \mathbf{O} & \mathbf{O} \end{bmatrix}_{10 \times 10},$$

则问题归结为求赋权图 G 的权最大的完美对集. □

Python 程序

```
import numpy as np
import networkx as nx
from networkx.algorithms.matching import max_weight_matching
a = np.array([[3, 5, 5, 4, 1], [2, 2, 0, 2, 2], [2, 4, 4, 1, 0],
              [0, 2, 2, 1, 0], [1, 2, 1, 3, 3]])
b = np.zeros((10, 10))
b[0:5, 5:] = a
G = nx.Graph(b)
s0 = max_weight_matching(G) # 返回值为 (人员, 工作) 的集合, 0~4为人员, 5
                             ~9为工作
s = [sorted(w) for w in s0]
L1 = [x[0] for x in s]
L1 = np.array(L1) + 1 # 人员编号
L2 = [x[1] for x in s]
L2 = np.array(L2) - 4 # 工作编号
c = a[L1 - 1, L2 - 1] # 提取对应的效益
d = c.sum() # 计算总的效益
print("工作分配对应关系为: \n人员编号: ", L1)
print("工作编号: ", L2)
print("总的效益为: ", d)
```

运行结果

工作分配对应关系为:

人员编号: [3 2 5 4 1]

工作编号: [3 1 5 2 4]

总的效益为: 15

即分配第一个人去做第 4 项工作, 第二个人去做第 1 项工作, 第三个人去做第 3 项工作, 第四个人去做第 2 项工作, 第五个人去做第 5 项工作, 总的效益为 15.



10.5 最大流与最小费用流问题

10.5.1 网络最大流问题

许多系统包含了流量问题. 例如, 公路系统中有车辆流, 控制系统中有信息流, 供水系统中有水流, 金融系统中有现金流等.

图 10.15 是联结某产品产地 v_1 和销地 v_6 的交通网, 每一弧 (v_i, v_j) 代表从 v_i 到 v_j 的运输线, 产品经这条弧由 v_i 输送到 v_j , 弧旁的数字表示这条运输线的最大通过能力. 产品经过交通网从 v_1 输送到 v_6 . 现在要求制定一个运输方案使从 v_1 运到 v_6 的产品数量最多.

图 10.16 给出了一个运输方案, 每条弧旁的数字表示在这个方案中, 每条运输线上的运输数量. 这个方案使 8 个单位的产品从 v_1 运到 v_6 , 在这个交通网上输送量是否还可以增多, 或者说这个运输网络中, 从 v_1 到 v_6 的最大输送量是多少呢? 本节就是要研究类似这样的问题.

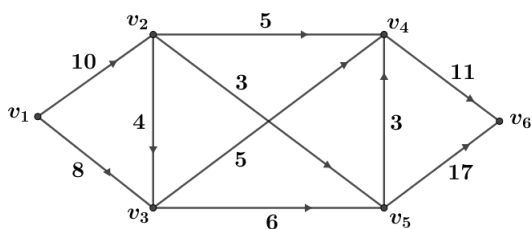


图 10.15

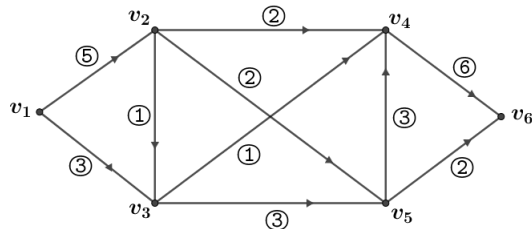


图 10.16

10.5.2 基本概念与基本定理

1. 网络与流

定义 10.15: 网络

给一个有向图 $D = (V, A)$, 在 V 中指定了一点称为发点 (记为 v_s), 而另一点称为收点 (记为 t), 其余的点叫中间点. 对于每一个弧 $(v_i, v_j) \in A$, 对应有一个 $c(v_i, v_j) \geq 0$ (或简写为 c_{ij}), 称为弧的容量. 通常我们就把这样的 D 叫做一个网络. 记作 $D = (V, A, C)$.



所谓网络上的流, 是指定义在弧集合 A 上的一个函数 $f = \{f(v_i, v_j)\}$, 并称 $f(v_i, v_j)$ 为弧 (v_i, v_j) 上的流量 (有时也简记作 f_{ij}).

例如, 图 10.15 就是一个网络, 指定 v_1 是发点, v_6 是收点, 其他的点是中间点. 弧旁的数字为 c_{ij} .

图 10.16 所示的运输方案, 就可看做是这个网络上的一个流, 每个弧上的运输量就是该弧上的流量, 即 $f_{12} = 5, f_{24} = 2, f_{13} = 3, f_{34} = 1$ 等.

2. 可行流与最大流



在运输网络的实际问题中可以看出, 对于流有两个明显的要求: 一是每个弧上的流量不能超过该弧的最大通过能力 (即弧的容量); 二是中间点的流量为零. 因为对于每个点, 运出这点的产品总量与运进这点的产品总量之差, 是这点的净输出量, 简称为是这点的流量; 由于中间点只起转运作用, 所以中间点的流量必为零. 易见发点的净流出量和收点的净流入量必相等, 也是这个方案的总输送量. 因此有:

定义 10.16: 可行流

满足下述条件的流 f 称为可行流:

(1) 容量限制条件: 对每一弧 $(v_i, v_j) \in A$,

$$0 \leq f_{ij} \leq c_{ij}.$$

(2) 平衡条件:

对于中间点: 流出量等于流入量, 即对每个 i ($i \neq s, t$) 有

$$\sum_{(v_i, v_j) \in A} f_{ij} - \sum_{(v_j, v_i) \in A} f_{ji} = 0.$$

对于发点 v_s , 记

$$\sum_{(v_s, v_j) \in A} f_{sj} - \sum_{(v_i, v_s) \in A} f_{is} = v(f).$$

对于收点 v_t , 记

$$\sum_{(v_t, v_j) \in A} f_{tj} - \sum_{(v_j, v_t) \in A} f_{jt} = -v(f).$$

式中, $v(f)$ 称为这个可行流的流量, 即发点的净输出量 (或收点的净输入量). 

可行流总是存在的. 比如令所有弧的流量 $f_{ij} = 0$, 就得到一个可行流 (称为零流). 其流量 $v(f) = 0$.

最大流问题就是求一个流 $\{f_{ij}\}$ 使其流量 $v(f)$ 达到最大, 并且满足

$$\begin{aligned} 0 \leq f_{ij} \leq c_{ij} & \quad (v_i, v_j) \in A \\ \sum f_{ij} - \sum f_{ji} = \begin{cases} v(f), & (i = s) \\ 0, & (i \neq s, t) \\ -v(f), & (i = t) \end{cases} \end{aligned}$$

最大流问题是一个特殊的线性规划问题. 即求一组 $\{f_{ij}\}$, 在满足上述条件下使 $v(f)$ 达到极大. 将会看到利用图的特点, 解决这个问题的方法较之线性规划的一般方法要方便、直观得多.

3. 增广链



若给一个可行流 $f = \{f_{ij}\}$, 我们把网络中使 $f_{ij} = c_{ij}$ 的弧称为饱和弧, 使 $f_{ij} < c_{ij}$ 的弧称为非饱和弧. 使 $f_{ij} = 0$ 的弧称为零流弧, 使 $f_{ij} > 0$ 的弧称为非零流弧.

在图 10.16 中, (v_5, v_4) 是饱和弧, 其他的弧为非饱和弧. 所有弧都是非零流弧.

若 μ 是网络中联结发点 v_s 和收点 v_t 的一条链, 我们定义链的方向是从 v_s 到 v_t , 则链上的弧被分为两类: 一类是弧的方向与链的方向一致, 叫做前向弧. 前向弧的全体记为 μ^+ . 另一类弧与链的方向相反, 称为后向弧. 后向弧的全体记为 μ^- .

图 10.15 中, 在链 $\mu = (v_1, v_2, v_3, v_4, v_5, v_6)$ 上,

$$\mu^+ = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_5, v_6)\},$$

$$\mu^- = \{(v_5, v_4)\}.$$

定义 10.17: 增广链

设 f 是一个可行流, μ 是从 v_s 到 v_t 的一条链, 若 μ 满足下列条件, 称之为 (关于可行流 f 的) 增广链.

在弧 $(v_i, v_j) \in \mu^+$ 上, $0 \leq f_{ij} < c_{ij}$, 即 μ^+ 中每一弧是非饱和弧.

在弧 $(v_i, v_j) \in \mu^-$ 上, $0 < f_{ij} \leq c_{ij}$, 即 μ^- 中每一弧是非零流弧.



图 10.16 中, 链 $\mu = (v_1, v_2, v_3, v_4, v_5, v_6)$ 是一条增广链. 因为 μ^+ 和 μ^- 中的弧满足增广链的条件. 比如:

$$(v_1, v_2) \in \mu^+, \quad f_{12} = 5 < c_{12} = 10,$$

$$(v_5, v_4) \in \mu^-, \quad f_{54} = 3 > 0.$$

4. 截集与截量

设 $S, T \subset V, S \cap T = \emptyset$, 我们把始点在 S 中, 终点在 T 中的所有弧构成的集合, 记为 (S, T) .

定义 10.18: 截集

给网络 $D = (V, A, C)$, 若点集 V 被剖分为两个非空集合 V_1 和 \bar{V}_1 , 使 $v_s \in V_1, v_t \in \bar{V}_1$, 则把弧集 (V_1, \bar{V}_1) 称为是 (分离 v_s 和 v_t 的) 截集.



显然, 若把某一截集的弧从网络中丢去, 则从 v_s 到 v_t 便不存在路. 所以, 直观上说, 截集是从 v_s 到 v_t 的必经之道.

定义 10.19: 截量

给一截集 (V_1, \bar{V}_1) , 把截集 (V_1, \bar{V}_1) 中所有弧的容量之和称为这个截集的容量 (简称为截量), 记为 $c(V_1, \bar{V}_1)$, 即

$$c(V_1, \bar{V}_1) = \sum_{(v_i, v_j) \in (V_1, \bar{V}_1)} c_{ij}.$$



不难证明, 任何一个可行流的流量 $v(f)$ 都不会超过任一截集的容量. 即

$$v(f) \leq c(V_1, \bar{V}_1).$$

显然, 若对于一个可行流 f^* , 网络中有一个截集 (V_1^*, \bar{V}_1^*) , 使 $v(f^*) = c(V_1^*, \bar{V}_1^*)$, 则 f^* 必是最大流, 而 (V_1^*, \bar{V}_1^*) 必定是 D 的所有截集中, 容量最小的一个, 即最小截集.

定理 10.8

可行流 f^* 是最大流, 当且仅当不存在关于 f^* 的增广链.



证明: 若 f^* 是最大流, 设 D 中存在关于 f^* 的增广链 μ , 令

$$\theta = \min \left\{ \min_{\mu^+} (c_{ij} - f_{ij}^*), \min_{\mu^-} f_{ij}^* \right\}.$$

由增广链的定义, 可知 $\theta > 0$, 令

$$f_{ij}^{**} = \begin{cases} f_{ij}^* + \theta, & (v_i, v_j) \in \mu^+ \\ f_{ij}^* - \theta, & (v_i, v_j) \in \mu^- \\ f_{ij}^*, & (v_i, v_j) \notin \mu \end{cases}$$

不难验证 $\{f_{ij}^{**}\}$ 是一个可行流, 且 $v(f^{**}) = v(f^*) + \theta > v(f^*)$. 这与 f^* 是最大流的假设矛盾.

现在设 D 中不存在关于 f^* 的增广链, 证明 f^* 是最大流. 我们利用下面的方法来定义 V_1^* :

令 $v_s \in V_1^*$,

若 $v_i \in V_1^*$, 且 $f_{ij}^* < c_{ij}$, 则令 $v_j \in V_1^*$;

若 $v_i \in V_1^*$, 且 $f_{ji}^* > 0$, 则令 $v_j \in V_1^*$;

因为不存在关于 f^* 的增广链, 故 $v_t \notin V_1^*$.

记 $\bar{V}_1^* = V \setminus V_1^*$, 于是得到一个截集 (V_1^*, \bar{V}_1^*) . 显然必有

$$f_{ij}^* = \begin{cases} c_{ij} & (v_i, v_j) \in (V_1^*, \bar{V}_1^*) \\ 0 & (v_i, v_j) \in (\bar{V}_1^*, V_1^*) \end{cases}$$

所以 $v(f^*) = c(V_1^*, \bar{V}_1^*)$. 于是 f^* 必是最大流. 定理得证. □

由上述证明中可见, 若 f^* 是最大流, 则网络中必存在一个截集 (V_1^*, \bar{V}_1^*) , 使

$$v(f^*) = C(V_1^*, \bar{V}_1^*).$$

于是有如下重要的结论:

最大流量最小截量定理: 任一个网络 D 中, 从 v_s 到 v_t 的最大流的流量等于分离 v_s, v_t 的最小截集的容量.

定理 10.8 为我们提供了寻求网络中最大流的一个方法. 若给了一个可行流 f ; 只要判断 D 中有无关于 f 的增广链. 如果有增广链, 则可以按定理 10.8 前半部证明中的办法, 改进 f , 得到一个流量增大的新的可行流. 如果没有增广链, 则得到最大流. 而



利用定理 10.8 后半部证明中定义 V_1^* 的办法, 可以根据 v_t 是否属于 V_1^* 来判断 D 中有无关于 f 的增广链.

实际计算时, 用给顶点标号的方法来定义 V_1^* . 在标号过程中, 有标号的顶点表示是 V_1^* 中的点, 没有标号的点表示不是 V_1^* 中的点. 一旦 v_t 有了标号, 就表明找到一条增广链; 如果标号过程进行不下去, 而 v_t 尚未标号, 则说明不存在增广链, 于是得到最大流. 而且同时也得到一个最小截集.

10.5.3 寻求最大流的标号法

从一个可行流出发 (若网络中没有给定 f , 则可以设 f 是零流), 经过标号过程与调整过程.

1. 标号过程

在这个过程中, 网络中的点或者是标号点 (又分为已检查和未检查两种), 或者是未标号点. 每个标号点的标号包含两部分: 第一个标号表明它的标号是从哪一点得到的, 以便找出增广链; 第二个标号是为确定增广链的调整量 θ 用的.

标号过程开始, 总先给 v_s 标上 $(0, +\infty)$, 这时 v_s 是标号而未检查的点, 其余都是未标号点. 一般地, 取一个标号而未检查的点 v_i , 对一切未标号点 v_j :

(1) 若在弧 (v_i, v_j) 上, $f_{ij} < c_{ij}$, 则给 v_j 标号 $(v_i, l(v_j))$. 这里 $l(v_j) = \min [l(v_i), c_{ij} - f_{ij}]$. 这时点 v_j 成为标号而未检查的点.

(2) 若在弧 (v_j, v_i) 上, $f_{ji} > 0$, 则给 v_j 标号 $(-v_i, l(v_j))$, 这里 $l(v_j) = \min [l(v_i), f_{ji}]$. 这时点 v_j 成为标号而未检查的点.

于是 v_i 成为标号而已检查过的点. 重复上述步骤, 一旦 v_t 被标上号, 表明得到一条从 v_s 到 v_t 的增广链 μ , 转入调整过程.

若所有标号都是已检查过的, 而标号过程进行不下去时, 则算法结束, 这时的可行流就是最大流.

2. 调整过程

首先按 v_t 及其他点的第一个标号, 利用“反向追踪”的办法, 找出增广链 μ . 例如设 v_t 的第一个标号为 v_k (或 $-v_k$), 则弧 (v_k, v_t) (或相应地 (v_t, v_k)) 是 μ 上的弧. 接下来检查 v_k 的第一个标号, 若为 v_i (或 $-v_i$), 则找出 (v_i, v_k) (或相应地 (v_k, v_i)). 再检查 v_i 的第一个标号, 依此下去, 直到 v_s 为止. 这时被找出的弧就构成了增广链 μ_0 . 令调整量 θ 是 $l(v_t)$, 即 v_t 的第二个标号. 令

$$f'_{ij} = \begin{cases} f_{ij} + \theta, & (v_i, v_j) \in \mu^+, \\ f_{ij} - \theta, & (v_i, v_j) \in \mu^-, \\ f_{ij}, & (v_i, v_j) \notin \mu. \end{cases}$$



去掉所有的标号, 对新的可行流 $f' = \{f'_{ij}\}$, 重新进入标号过程.

例 10.16: 用标号法求图 10.17 所示网络的最大流. 弧旁的数是 (c_{ij}, f_{ij}) .

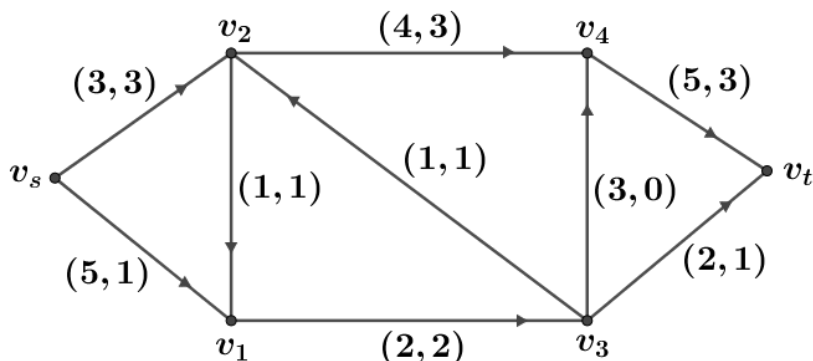


图 10.17: 赋权有向图

解: (1) 标号过程

① 首先给 v_s 标上 $(0, +\infty)$;

② 检查 v_s , 在弧 (v_s, v_2) 上, $f_{s2} = c_{s2} = 3$, 不满足标号条件. 弧 (v_s, v_1) 上, $f_{s1} = 1$, $c_{s1} = 5$, $f_{s1} < c_{s1}$, 则 v_1 的标号为 $(v_s, l(v_1))$, 其中

$$l(v_1) = \min[l(v_s), (c_{s1} - f_{s1})] = \min[+\infty, 5 - 1] = 4;$$

③ 检查 v_1 , 在弧 (v_1, v_3) 上, $f_{13} = 2$, $c_{13} = 2$, 不满足标号条件.

在弧 (v_2, v_1) 上, $f_{21} = 1 > 0$, 则给 v_2 记下标号为 $(-v_1, l(v_2))$, 这里

$$l(v_2) = \min[l(v_1), f_{21}] = \min[4, 1] = 1;$$

④ 检查 v_2 , 在弧 (v_2, v_4) 上, $f_{24} = 3$, $c_{24} = 4$, $f_{24} < c_{24}$, 则给 v_4 标号 $(v_2, l(v_4))$, 这里

$$l(v_4) = \min[l(v_2), (c_{24} - f_{24})] = \min[1, 1] = 1.$$

在弧 (v_3, v_2) 上, $f_{32} = 1 > 0$, 给 v_3 标号: $(-v_2, l(v_3))$, 这里

$$l(v_3) = \min[l(v_2), f_{32}] = \min[1, 1] = 1;$$

⑤ 在 v_3, v_4 中任选一个进行检查. 例如, 在弧 (v_3, v_t) 上, $f_{3t} = 1$, $c_{3t} = 2$, $f_{3t} < c_{3t}$, 给 v_t 标号为 $(v_3, l(v_t))$, 这里

$$l(v_t) = \min[l(v_3), (c_{3t} - f_{3t})] = \min[1, 1] = 1,$$

因 v_t 有了标号, 故转入调整过程.

(2) 调整过程

按点的第一个标号找到一条增广链, 如图 10.18 中双箭头线表示.



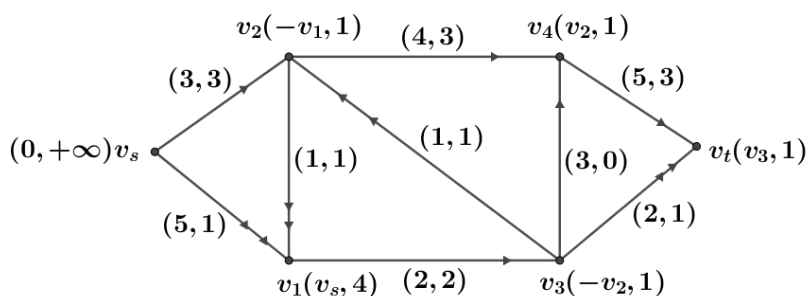


图 10.18

易见

$$\mu^+ = \{(v_s, v_1), (v_3, v_t)\},$$

$$\mu^- = \{(v_2, v_1), (v_3, v_2)\}.$$

按 $\theta = 1$ 在 μ 上调整 f .

$$\mu^+ \text{ 上: } f_{s1} + \theta = 1 + 1 = 2,$$

$$f_{3t} + \theta = 1 + 1 = 2.$$

$$\mu^- \text{ 上: } f_{21} - \theta = 1 - 1 = 0,$$

$$f_{32} - \theta = 1 - 1 = 0.$$

其余的 f_{ij} 不变.

调整后得如图 10.19 所示的可行流, 对这个可行流进入标号过程, 寻找增广链.

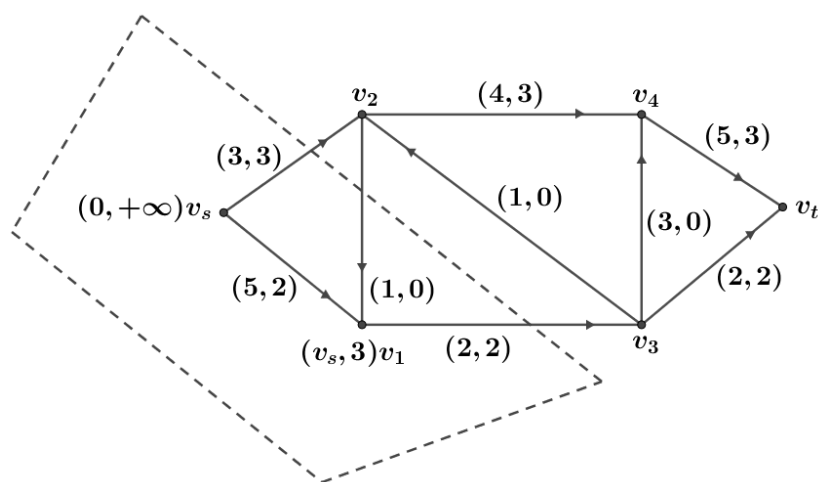


图 10.19

开始给 v_s 标以 $(0, +\infty)$, 于是检查 v_s , 给 v_1 标以 $(v_s, 3)$, 检查 v_1 , 弧 (v_1, v_3) 上, $f_{13} = c_{13}$, 弧 (v_2, v_1) 上, $f_{21} = 0$, 均不符号条件, 标号过程无法继续下去, 算法结束.

这时的可行流 (图 10.19) 即为所求最大流. 最大流量为

$$\nu(f) = f_{s1} + f_{s2} = f_{4t} + f_{3t} = 5.$$

与此同时可找到最小截集 (V_1, \bar{V}_1) , 其中 V_1 为标号点集合, \bar{V}_1 为未标号点集合. 弧集合 (V_1, \bar{V}_1) 即为最小截集. 上例中, $V_1 = \{v_s, v_1\}$, $\bar{V}_1 = \{v_2, v_3, v_4, v_t\}$, 于是 $(V_1, \bar{V}_1) = \{(v_s, v_2), (v_1, v_3)\}$ 是最小截集, 它的容量也是 5. \square



由上述例子可见, 用标号法找增广链以求最大流的结果, 同时得到一个最小截集. 最小截集容量的大小影响总的输送量的提高. 因此, 为提高总的输送量, 必须首先考虑改善最小截集中各弧的输送状况, 提高它们的通过能力. 另一方面, 一旦最小截集中弧的通过能力被降低, 就会使总的输送量减少.

将例 10.16 用 networkx 求从 v_s 到 v_t 的网络最大流.

Python 程序

```
import numpy as np
import networkx as nx
import pylab as plt

L = [(1, 2, 5), (1, 3, 3), (2, 4, 2), (3, 2, 1), (3, 5, 4), (4, 3, 1),
      (4, 5, 3), (4, 6, 2), (5, 6, 5)]

G = nx.DiGraph()
for k in range(len(L)):
    G.add_edge(L[k][0] - 1, L[k][1] - 1, capacity=L[k][2])
value, flow_dict = nx.maximum_flow(G, 0, 5)
print("最大流的流量为: ", value)
print("最大流为: ", flow_dict)

n = len(flow_dict)
adj_mat = np.zeros((n, n), dtype=int)
for i, adj in flow_dict.items():
    for j, weight in adj.items():
        adj_mat[i, j] = weight
print("最大流的邻接矩阵为: \n", adj_mat)
ni, nj = np.nonzero(adj_mat) # 非零弧的两端点编号
key = range(n)
s = ['v' + str(i + 1) for i in range(n)]
s = dict(zip(key, s)) # 构造用于顶点标注的字符字典
plt.rc('font', size=16)
pos = nx.shell_layout(G) # 设置布局
w = nx.get_edge_attributes(G, 'capacity')
nx.draw(G, pos, font_weight='bold', labels=s, node_color='r')
nx.draw_networkx_edge_labels(G, pos, edge_labels=w)
path_edges = list(zip(ni, nj))
nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='r', width=3)

plt.show()
```



运行结果

最大流的流量为：5

最大流为：{0: {1: 2, 2: 3}, 1: {3: 2}, 2: {1: 0, 4: 3}, 3: {2: 0, 4: 0, 5: 2}, 4: {5: 3}, 5: {}}

最大流的邻接矩阵为：

[[0 2 3 0 0 0]

[0 0 0 2 0 0]

[0 0 0 0 3 0]

[0 0 0 0 0 2]

[0 0 0 0 0 3]

[0 0 0 0 0 0]]

最大流的示意图如图 10.20 所示.

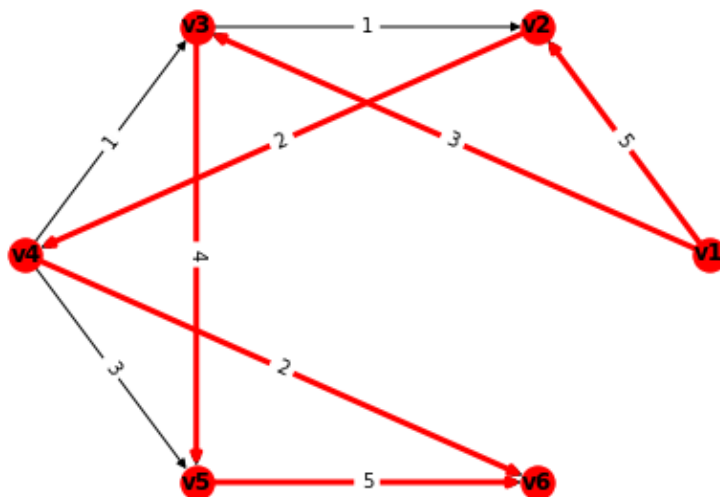


图 10.20

由字典的第一个元素 0 : {1 : 2, 2 : 3}, 得知 $f_{s2} = 2, f_{s3} = 3$; 由第二个元素 1 : {3 : 2}, 知 $f_{24} = 2$; 由第三个元素 2 : {1 : 0, 4 : 3} 知, $f_{32} = 0, f_{35} = 3$; 由第四个元素 3 : {2 : 0, 4 : 0, 5 : 2}, 知 $f_{43} = 0, f_{45} = 0, f_{4t} = 2$; 由第五个元素 4 : {5 : 3} 知, $f_{5t} = 3$.

10.5.4 最小费用流问题

在许多实际问题中, 往往还要考虑网络上流的费用问题. 例如, 在运输问题中, 人们总是希望在完成运输任务的同时, 寻求一个使总的运输费用最小的运输方案. 设 f_{ij} 为弧 (v_i, v_j) 上的流量, b_{ij} 为弧 (v_i, v_j) 上的单位费用, c_{ij} 为弧 (v_i, v_j) 上的容量, 则



最小费用流问题可以用如下的线性规划问题描述:

$$\begin{aligned} \min \quad & \sum_{(v_i, v_j) \in A} b_{ij} f_{ij}, \\ \text{s.t.} \quad & \begin{cases} \sum_{j: (v_i, v_j) \in A} f_{ij} - \sum_{j: (v_j, v_i) \in A} f_{ji} = \begin{cases} v, & i = s, \\ -v, & i = t, \\ 0, & i \neq s, t \end{cases} \\ 0 \leq f_{ij} \leq c_{ij}, \quad \forall (v_i, v_j) \in A. \end{cases} \end{aligned}$$

当 $v = \text{最大流 } v_{\max}$ 时, 本问题就是最小费用最大流问题; 如果 $v > v_{\max}$, 本问题无解.

1961 年, Busacker 和 Gowan 提出了一种求最小费用流的迭代法. 其步骤如下:

(1) 求出从发点到收点的最小费用通路 $\mu(v_s, v_t)$.

(2) 对该通路 $\mu(v_s, v_t)$ 分配最大可能的流量:

$$\bar{f} = \min_{(v_i, v_j) \in \mu(v_s, v_t)} \{c_{ij}\},$$

并让通路上的所有边的容量相应减少 \bar{f} . 这时, 对于通路上的饱和边, 其单位流费用相应改为 ∞ .

(3) 作该通路 $\mu(v_s, v_t)$ 上所有边 (v_i, v_j) 的反向边 (v_j, v_i) . 令

$$c_{ji} = \bar{f}, \quad b_{ji} = -b_{ij}.$$

(4) 在这样构成的新网络中, 重复上述步骤 (1), (2), (3), 直到从发点到收点的全部流量等于 v 为止.

例 10.17: (续例 10.16) 如图 10.15 所示带有运费的网络, 求从 v_s 到 v_t 的最小费用最大流, 其中弧上权重的第 1 个数字是网络的容量, 第 2 个数字是网络的单位运费.

Python 程序

```
import numpy as np
import networkx as nx

L = [(1, 2, 5, 3), (1, 3, 3, 6), (2, 4, 2, 8), (3, 2, 1, 2), (3, 5, 4, 2),
      (4, 3, 1, 1), (4, 5, 3, 4), (4, 6, 2, 10), (5, 6, 5, 2)]

G = nx.DiGraph()
for k in range(len(L)):
    G.add_edge(L[k][0] - 1, L[k][1] - 1, capacity=L[k][2], weight=L[k][3])

mincostFlow = nx.max_flow_min_cost(G, 0, 5)
```



```

print("所求流为：", mincostFlow)
mincost = nx.cost_of_flow(G, mincostFlow)
print("最小费用为：", mincost)
flow_mat = np.zeros((6, 6), dtype=int)
for i, adj in mincostFlow.items():
    for j, f in adj.items():
        flow_mat[i, j] = f
print("最小费用最大流的邻接矩阵为：\n", flow_mat)

```

运行结果

所求流为：0: 1: 2, 2: 3, 1: 3: 2, 2: 1: 0, 4: 4, 3: 2: 1, 4: 1, 5: 0, 4: 5: 5, 5:

最小费用为：63

最小费用最大流的邻接矩阵为：

[[0 2 3 0 0 0]

[0 0 0 2 0 0]

[0 0 0 0 4 0]

[0 0 1 0 1 0]

[0 0 0 0 0 5]

[0 0 0 0 0 0]]

10.6 旅行商问题

10.6.1 旅行商问题

旅行商问题，即 TSP 问题 (Travelling Salesman Problem)，又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访 n 个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

例 10.18: 旅行商要从 10 个城市中的某一个城市出发，去其他 9 个城市旅行，要求每个城市到达一次，走完一轮后回到原来城市，问他应如何选择旅行路线，使总路程最短？其中，10 个城市间的相互距离如表 10.5 所示：



表 10.5: 10 个城市间的距离数据

	1	2	3	4	5	6	7	8	9	10
1	0	7	4	5	8	6	12	13	11	18
2	7	0	3	10	9	14	5	14	17	17
3	4	3	0	5	9	10	21	8	27	12
4	5	10	5	0	14	9	10	9	23	16
5	8	9	9	14	0	7	8	7	20	19
6	6	14	10	9	7	0	13	5	25	13
7	12	5	21	10	8	13	0	23	21	18
8	13	14	8	9	7	5	23	0	18	12
9	11	17	27	23	20	25	21	18	0	16
10	18	17	12	16	19	13	18	12	16	0

解： 记城市间距离用矩阵 D 表示, d_{ij} 表示城市 i 到城市 j 的距离, 设 0-1 矩阵 X 表示经过各城市的路线, 其中

$$x_{ij} = \begin{cases} 1, & \text{若路线包括从城市 } i \text{ 到城市 } j, \\ 0, & \text{否则.} \end{cases}$$

考虑到可行路线上每个城市后只有一个城市, 则

$$\sum_{j=1, j \neq i}^n x_{ij} = 1, \quad i = 1, 2, \dots, n.$$

每个城市前只有一个城市, 则

$$\sum_{i=1, i \neq j}^n x_{ij} = 1, \quad j = 1, 2, \dots, n.$$

只有上述约束条件不能避免在一次遍历中产生多于一个互不相通回路. 为此引入额外变量 u_i ($i = 1, 2, \dots, n$), 附加以下充分约束条件:

$$u_i - u_j + nx_{ij} \leq n - 1, \quad 1 < i \neq j \leq n.$$

该约束可保证城市不构成回路. 事实上,

若城市 i, j 构成回路, 则 $x_{ij} = 1, x_{ji} = 1$, 故 $u_i - u_j \leq -1$ 且 $u_j - u_i \leq -1$, 从而 $0 \leq -2$, 矛盾;

若城市 i, j, k 构成回路, 则 $x_{ij} = 1, x_{jk} = 1, x_{ki} = 1$, 故 $u_i - u_j \leq -1, u_j - u_k \leq -1, u_k - u_i \leq -1$, 从而 $0 \leq -3$, 矛盾; 依次类推.



综上得到优化模型:

$$\begin{aligned} \min Z &= \sum_{i=1}^n \sum_{j=1}^n d_{ij} x_{ij} \\ \text{s.t.} &\begin{cases} \sum_{j=1, j \neq i}^n x_{ij} = 1, \quad i = 1, 2, \dots, n, \\ \sum_{i=1, i \neq j}^n x_{ij} = 1, \quad j = 1, 2, \dots, n, \\ u_i - u_j + nx_{ij} \leq n-1, \quad 1 < i \neq j \leq n, \\ x_{ij} = 0 \text{ 或 } 1, \quad i, j = 1, 2, \dots, n, \\ u_i \in \mathbb{R}, \quad i = 1, 2, \dots, n. \end{cases} \end{aligned}$$

用线性规划法借助 Lingo 软件求解 (略). □

用 Python 求解

Python 程序

```
import numpy as np
from python_tsp.exact import solve_tsp_dynamic_programming

distance_matrix = np.array([ [0, 7, 4, 5, 8, 6, 12, 13, 11, 18],
[7, 0, 3, 10, 9, 14, 5, 14, 17, 17],
[4, 3, 0, 5, 9, 10, 21, 8, 27, 12],
[5, 10, 5, 0, 14, 9, 10, 9, 23, 16],
[8, 9, 9, 14, 0, 7, 8, 7, 20, 19],
[6, 14, 10, 9, 7, 0, 13, 5, 25, 13],
[12, 5, 21, 10, 8, 13, 0, 23, 21, 18],
[13, 14, 8, 9, 7, 5, 23, 0, 18, 12],
[11, 17, 27, 23, 20, 25, 21, 18, 0, 16],
[18, 17, 12, 16, 19, 13, 18, 12, 16, 0] ])
permutation, distance = solve_tsp_dynamic_programming(distance_matrix)
print('最短路径:', permutation)
print('最短距离:', distance)
```

运行结果

最短路径: [0, 3, 2, 1, 6, 4, 5, 7, 9, 8]

最短距离: 77

10.6.2 比赛项目排序问题

在运动比赛中, 为了使比赛公平、公正、合理地举行, 一个基本要求是: 在比赛项目排序过程中, 尽可能使每个运动员不连续参加 2 项比赛, 以便运动员恢复体力, 发挥



正常水平.

现有某小型运动会的比赛报名表, 有 14 个比赛项目, 40 名运动员参加比赛. 表中第 1 行表示 14 个比赛项目, 第 1 列表示 40 名运动员, 表中 “#” 号位置表示运动员参加此项比赛. 建立此问题的数学模型, 并且合理安排比赛项目顺序, 使连续参加两项比赛的运动员人次尽可能的少.

若 1050 名运动员参加 60 个项目, 又将如何?

表 10.6: 某小型运动会比赛报名表 (部分)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		#	#						#				#	
2								#			#	#		
3		#		#						#				
4			#					#				#		

将表格中的 # 替换为 1, 空白替换为 0, 得到 40×14 的数据表, 存为 table1.txt.

该题目与旅行商问题类似, 即求一条遍历 14 个项目的路线, 使得总路程最短. 可考虑用旅行商问题的算法解决. 问题的关键在于怎么构造项目之间的距离矩阵 D ?

若项目 i 和项目 j 相邻, 将同时参加这两个项目的人数, 作为 i 和 j 的距离 d_{ij} . 由于开始项目和结束项目没有连接, 可考虑引入虚拟项目 15, 并规定虚拟项目与各项目的距离都为 0.

上述数据表增加一个 15 列 (元素全部为零), 用矩阵 $A_{40 \times 15} = (a_{ij})$ 表示, 其中

$$a_{ij} = \begin{cases} 1, & \text{第 } i \text{ 个人参加项目 } j, \\ 0, & \text{第 } i \text{ 个不参加项目 } j. \end{cases}$$

则

$$\begin{aligned} d_{ij} &= 0, \quad i = j, \quad i, j = 1, 2, \dots, 14, 15, \\ d_{ij} &= \sum_{k=1}^{40} a_{ki} a_{kj}, \quad i \neq j, \quad i, j = 1, 2, \dots, 14, 15, \\ d_{i,15} &= d_{15,i} = 0, \quad i = 1, 2, \dots, 14, 15. \end{aligned}$$

即

$$D = B - \Lambda,$$

其中, $B = A^T A = (b_{ij})$, $\Lambda = \text{diag}(b_{11}, b_{22}, \dots, b_{15,15})$.

用 Python 求解

Python 程序



```
import numpy as np
from python_tsp.exact import solve_tsp_dynamic_programming

a = np.loadtxt('baoming1.txt') # 导入原始报名数据
b = np.hstack((a, np.zeros((40, 1)))) # 增加一个虚拟项目 15, 即原数据在
                                      # 最后一列增加零列

d = np.dot(b.T, b)
D = d - np.diag(np.diagonal(d)) # 项目之间的距离矩阵

permutation, distance = solve_tsp_dynamic_programming(D)
print('项目排序:', permutation)
print('项目距离:', distance)
```

运行结果

项目排序: [0, 4, 6, 10, 2, 5, 1, 8, 3, 12, 9, 11, 13, 14, 7]

项目距离: 2.0



10.7 PageRank 算法

PageRank 算法是基于网页链接分析对关键字匹配搜索结果进行处理的. 它借鉴传统引文分析思想: 当网页甲有一个链接指向网页乙, 就认为乙获得了甲对它贡献的分值, 该值的多少取决于网页甲本身的重要程度, 即网页甲的重要性越大, 网页乙获得的贡献值就越高. 由于网络中网页链接的相互指向, 该分值的计算为一个迭代过程, 最终网页根据所得分值进行检索排序. 互联网是一张有向图, 每一个网页是图的一个顶点, 网页间的每一个超链接是图的一条弧, 邻接矩阵 $\mathbf{B} = (b_{ij})_{N \times N}$, 如果从网页 i 到网页 j 有超链接, 则 $b_{ij} = 1$, 否则为 0. 记矩阵 \mathbf{B} 的行和为 $r_i = \sum_{j=1}^N b_{ij}$, 它表示页面 i 发出的链接数目. 假如在上网时浏览页面并选择下一个页面的过程, 与过去浏览过哪些页面无关, 而仅依赖于当前所在的页面. 那么这一选择过程可以认为是一个有限状态、离散时间的随机过程, 其状态转移规律用 Markov 链描述. 定义矩阵 $\mathbf{A} = (a_{ij})_{N \times N}$ 如下

$$a_{ij} = \frac{1-d}{N} + d \frac{b_{ij}}{r_i}, \quad i, j = 1, 2, \dots, N$$

其中 d 是模型参数, 通常取 $d = 0.85$, \mathbf{A} 是 Markov 链的转移概率矩阵, a_{ij} 表示从页面 i 转移到页面 j 的概率. 根据 Markov 链的基本性质, 对于正则 Markov 链存在平稳分布 $\mathbf{x} = [x_1, \dots, x_N]^T$, 满足

$$\mathbf{A}^T \mathbf{x} = \mathbf{x}, \quad \sum_{i=1}^N x_i = 1$$

\mathbf{x} 表示在极限状态 (转移次数趋于无限) 下各网页被访问的概率分布, Google 将它定义为各网页的 PageRank 值. 假设 \mathbf{x} 已经得到, 则它按分量满足方程

$$x_k = \sum_{i=1}^N a_{ik} x_i = (1-d) + d \sum_{i: b_{ik}=1} \frac{x_i}{r_i}$$

网页 i 的 PageRank 值是 x_i , 它链出的页面有 r_i 个, 于是页面 i 将它的 PageRank 值分成 r_i 份, 分别“投票”给它链出的网页. x_k 为网页 k 的 PageRank 值, 即网络上所有页面“投票”给网页 k 的最终值. 根据 Markov 链的基本性质还可以得到, 平稳分布 (即 PageRank 值) 是转移概率矩阵 \mathbf{A} 的转置矩阵 \mathbf{A}^T 的最大特征值 ($= 1$) 所对应的归一化特征向量.

例 10.18 已知一个 $N = 6$ 的网络如图 10.16 所示, 求它的 PageRank 取值. 解相



应的邻接矩阵 B 和 Markov 链状态转移概率矩阵 A 分别为

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$A = \begin{bmatrix} 0.025 & 0.875 & 0.025 & 0.025 & 0.025 & 0.025 \\ 0.025 & 0.025 & 0.45 & 0.45 & 0.025 & 0.025 \\ 0.025 & 0.025 & 0.025 & 0.3083 & 0.3083 & 0.3083 \\ 0.875 & 0.025 & 0.025 & 0.025 & 0.025 & 0.025 \\ 0.025 & 0.025 & 0.025 & 0.025 & 0.025 & 0.875 \\ 0.875 & 0.025 & 0.025 & 0.025 & 0.025 & 0.025 \end{bmatrix}$$

计算得到该 Markov 链的平稳分布为

$$x = [0.2675, 0.2524, 0.1323, 0.1697, 0.0625, 0.1156]^T$$

这就是 6 个网页的 PageRank 值, 其柱状图如图 10.17 所示. 编号 1 的网页的 PageRank 值最高, 编号 5 的网页的 PageRank 值最低, 网页的 PageRank 值从大到小的排序依次为 1, 2, 4, 3, 6, 5.

10.8 复杂网络简介

本节主要介绍复杂网络的一些统计性质, 并利用 networkx 计算这些统计性质.

10.8.1 复杂网络初步介绍

复杂网络 (complex network) 是指具有自组织、自相似、吸引子、小世界、无标度中部分或全部性质的网络.

简而言之复杂网络是呈现高度复杂性的网络. 其复杂性主要表现在以下几个方面:

- (1) 结构复杂: 表现在节点数目巨大, 网络结构呈现多种不同特征.
- (2) 网络进化: 表现在节点或连接的产生与消失. 例如万维网, 网页随时可能出现链接或断开, 导致网络结构不断发生变化.
- (3) 连接多样性: 节点之间的连接权重存在差异, 且有可能存在方向性.
- (4) 动力学复杂性: 节点集可能属于非线性动力学系统, 例如, 节点状态随时间发生复杂变化.
- (5) 节点多样性: 复杂网络中的节点可以代表任何事物, 例如, 人际关系构成的复杂网络节点代表单独个体, 万维网组成的复杂网络节点可以表示不同网页.



(6) 多重复杂性融合: 即以上多重复杂性相互影响, 导致更为难以预料的结果. 例如, 设计一个电力供应网络需要考虑此网络的进化过程, 其进化过程决定网络的拓扑结构. 当两个节点之间频繁进行能量传输时, 他们之间的连接权重会随之增加, 通过不断的学习与记忆逐步改善网络性能. 目前, 复杂网络研究的内容主要包括: 网络的几何性质、网络的形成机制、网络演化的统计规律、网络上的模型性质, 以及网络的结构稳定性、网络的演化动力学机制等问题. 其中在自然科学领域, 网络研究的基本测度包括: 度 (degree) 及其分布特征、度的相关性、集聚程度及其分布特征、最短距离及其分布特征、介数 (betweenness) 及其分布特征、连通集团的规模分布.

复杂网络一般具有以下特性:

(1) 小世界. 它以简单的措辞描述了大多数网络尽管规模很大但是任意两个节点间却有一条相当短的路径的事实. 例如, 在社会网络中, 人与人相互认识的关系很少, 但是却可以找到很远的无关系的其他人. 正如麦克卢汉所说, 地球变得越来越小, 变成一个地球村, 也就是说, 变成一个小世界.

(2) 集群. 例如, 社会网络中总是存在熟人圈或朋友圈, 其中每个成员都认识其他成员. 集聚程度的意义是网络集团化的程度; 这是一种网络的内聚倾向. 连通集团概念反映的是一个网络中各集聚的小网络分布和相互联系的情况. 例如, 它可以反映这个朋友圈与另一个朋友圈的相互关系. (3) 幂律 (power law) 度分布. 度指的是网络中某个节点与其他节点关系 (用网络中的边表示) 的数量; 度的相关性指节点之间关系的联系紧密性. 无标度网络 (scale-free network) 的特征主要集中反映了集聚的集中性. 实际网络都兼有确定和随机两大特征, 确定性的法则或特征通常隐藏在统计性质中.

10.8.2 复杂网络的统计描述

人们在刻画复杂网络结构的统计特性上提出了许多概念和方法, 其中包含: 节点的度和度分布、平均路径长度、聚类系数.

1. 节点的度和度分布

节点 v_i 的度 k_i 定义为与该节点连接的边数. 直观上看, 一个节点的度越大就意味着这个节点在某种意义上越“重要”. 定义 10.16 网络中所有节点 v_i 的度 k_i 的平均值称为网络的平均度, 记为 $\langle k \rangle$, 即

$$\langle k \rangle = \frac{1}{N} \sum_{i=1}^N k_i$$

无向无权图的邻接矩阵 $\mathbf{A} = (a_{ij})_{N \times N}$ 与节点 v_i 的度 k_i 的函数关系很简单: 邻接矩阵二次幂 $\mathbf{A}^2 = (a_{ij}^{(2)})_{N \times N}$ 的对角线元素 $a_{ii}^{(2)}$ 就等于节点 v_i 的度, 即

$$k_i = a_{ii}^{(2)}$$

实际上, 无向无权图的邻接矩阵 \mathbf{A} 的第 i 行或第 i 列元素之和也是度, 从而无向无权



网络的平均度就是 A^2 的对角线元素之和除以节点数, 即

$$\langle k \rangle = \text{tr}(A^2) / N$$

式中, $\text{tr}(A^2)$ 表示矩阵 A^2 的迹 (trace), 即对角线元素之和. 网络中节点的度分布情况可用分布函数 $P(k)$ 来描述, $P(k)$ 表示的是网络中度为 k 的节点在整个网络中所占的比率, 也就是说, 在网络中随机抽取到度为 k 的节点的概率为 $P(k)$. 一般地, 可以用一个直方图来描述网络的度分布 (degree distribution) 性质. 对于规则的网格来说, 由于所有的节点具有相同的度, 所以其度分布集中在一个单一尖峰上, 是一种 Delta 分布. 网络中的任何随机化倾向都将使这个尖峰的形状变宽. 完全随机网络 (completely stochastic network) 的度分布近似为泊松分布, 其形状在远离峰值 $\langle k \rangle$ 处呈指数下降. 这意味着当 $k > \langle k \rangle$ 时, 度为 k 的节点实际上是不存在的. 因此, 这类网络也称为均匀网络 (homogeneous network). 近几年的大量研究表明, 许多实际网络的度分布明显地不同于泊松分布. 特别地, 许多网络的度分布可以用幂律形式 $P(k) \propto k^{-\gamma}$ 来更好地描述.

2. 平均路径长度网络中两个节点 v_i 和 v_j 之间的距离 d_{ij} 定义为连接这两个节点的最短路径上的边数, 它的倒数 $1/d_{ij}$ 称为节点 v_i 和 v_j 之间的效率, 记为 ε_{ij} . 通常效率用来度量节点间的信息传递速度. 当 v_i 和 v_j 之间没有路径连通时, $d_{ij} = \infty$, 而 $\varepsilon_{ij} = 0$. 网络中任意两个节点之间的距离的最大值称为网络的直径, 记为 D , 即

$$D = \max_{1 \leq i < j \leq N} d_{ij}$$

其中, N 为网络节点数. 定义 10.17 网络的平均路径长度 L 定义为任意两个节点之间的距离的平均值, 即

$$L = \frac{1}{C_N^2} \sum_{1 \leq i < j \leq N} d_{ij}$$

3. 聚类系数

在你的朋友关系网络中, 你的两个朋友很可能彼此也是朋友, 这种属性在复杂网络理论中称为网络的聚类特性. 一般地, 假设网络中的一个节点 v_i 有 k_i 条边将它和其他节点相连, 这 k_i 个节点就称为节点 v_i 的邻居. 显然, 在这 k_i 个节点之间最多可能有 $C_{k_i}^2$ 条边. 定义 10.18 节点 v_i 的 k_i 个邻居节点之间实际存在的边数 E_i 和总的可能的边数 $C_{k_i}^2$ 之比就定义为节点 v_i 的聚类系数 C_i , 即

$$C_i = \frac{E_i}{C_{k_i}^2}$$

从几何特点看, (10.8) 式的一个等价定义为

$$C_i = \frac{\text{与节点 } v_i \text{ 相连的三角形的数量}}{\text{与节点 } v_i \text{ 相连的三元组的数量}} = \frac{n_1}{n_2},$$

其中, 与节点 v_i 相连的三元组是指包括节点 v_i 的三个节点, 并且至少存在从节点 v_i 到其他两个节点的两条边 (图 10.18). 下面讨论如何根据无权无向图的邻接矩阵 A 来



求节点 v_i 的聚类系数 C_i . 显然, 邻接矩阵二次幂 A^2 的对角元素 $a_{ii}^{(2)}$ 表示的是与节点 v_i 相连的边数, 也就是节点 v_i 的度 k_i . 而邻接矩阵三次幂 A^3 的对角线元素 $a_{ii}^{(3)}$ 表示的是从节点 v_i 出发经过三条边回到节点 v_i 的路径数, 也就是与节点 v_i 相连的三角形数的两倍 (正向走和反向走). 因此, 由聚类系数的表达式 (10.9) 可知

$$C_i = \frac{n_1}{n_2} = \frac{2n_1}{2C_{k_i}^2} = \frac{2n_1}{k_i(k_i-1)} = \frac{a_{ii}^{(3)}}{a_{ii}^{(2)}(a_{ii}^{(2)}-1)}$$

整个网络的聚类系数 C 就是所有节点 v_i 的聚类系数 C_i 的平均值, 即

$$C = \frac{1}{N} \sum_{i=1}^N C_i$$

显然, $0 \leq C \leq 1$. 当所有的节点均为孤立节点, 即没有任何连接边时, $C = 0$; $C = 1$ 当且仅当网络是全局耦合的, 即网络中任意两个节点都直接相连. 对于一个含有 N 个节点的完全随机的网络, 当 N 很大时, $C = O(N^{-1})$. 而许多大规模的实际网络都具有明显的聚类效应, 它们的聚类系数尽管远小于 1 但却比 $O(N^{-1})$ 大得多. 事实上, 在很多类型的网络中, 随着网络规模的增加, 聚类系数趋向于某个非零常数, 即当 $N \rightarrow \infty$ 时, $C = O(1)$. 这意味着这些实际的复杂网络并不是完全随机的, 而是在某种程度上具有类似于社会关系网络中“物以类聚, 人以群分”的特性. 例 10.19 计算图 10.19 所示简单无向网络的直径 D 平均路径长度 L 和聚类系数.

解首先构造图 10.19 对应的图 $G = (V, E)$ 的邻接矩阵

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

然后应用 Floyd 算法求出任意节点之间的最短距离. 其中的最大距离为网络直径, 网络直径 $D = d_{16} = d_{36} = 3$, 把所有节点对之间的距离求和 (只需求对应矩阵上三角元素的和), 再除以 C_6^2 , 求得网络的平均路径长度 $L = 1.6$. 下面以节点 v_2 的聚类系数计算为例, 节点 v_2 与 4 个节点相邻, 这 4 个节点之间可能存在的最大边数为 $C_4^2 = 6$, 而这 4 个节点之间实际存在的边数为 3, 由定义可得

$$C_2 = \frac{3}{C_4^2} = \frac{1}{2}$$

同理可求得其他节点的聚类系数为

$$C_1 = 1, \quad C_3 = \frac{2}{3}, \quad C_4 = \frac{1}{3}, \quad C_5 = \frac{2}{3}, \quad C_6 = 0$$

整个网络的聚类系数 $C = \frac{19}{36}$.

