图论研究：利用 Python 解决图问题

曾理 李乘黄 单赢宇 吕佳颖

（北京理工大学计算机学院 计算机科学与技术专业）

姓名	学号	联系方式	专业
曾理（组长）	1120212666	15882032796	计算机科学与技术（徐特立英才班）
李乘黄	1120211158	18810008652	
单赢宇	1120212931	13818601183	
吕佳颖	1120213567	18950065499	

摘要：提供一组使用 Python 和 Networkx[1] 库编写的代码，用于研究图论中各种性质. 通过该代码，可以从邻接矩阵，邻接表，度数列等来源产生图对象，并能绘制图形并标记其中的特定对象. 可以计算图上的最小生成树，最短路径，欧拉回路，哈密顿回路，以及对图进行染色等。研究成果包括解决部分综合性问题，如中国邮政员问题和拉姆齐定理等

关键词: 离散数学 | 图论 | Python | Networkx

Graph theory research: using Python to solve graph problems

ZENG li, Li Cheng-huang, SHAN Ying-yu, LYU Jia-ying

(Computer science and technology, School of computer science,
Beijing University of Technology)

Abstract: This study provides a set of code written using the Python and Networkx libraries, for investigating various properties in graph theory. Through this code, graph objects can be generated from sources such as adjacency matrices, adjacency lists, and degree sequences, and can be plotted and marked with specific objects. It allows for the calculation of minimum spanning trees, shortest paths, Eulerian cycles, Hamiltonian cycles, and graph coloring. The research includes solving some composite problems such as Chinese Postman problem and Königsberg bridge problem.

Keywords: Discrete Mathematics | Graph Theory | Python | Networkx

目录

介绍

在离散数学中，图论是一个重要的分支，在许多方面都有所应用。图论与计算机结合最著名的例子应该是四色定理的证明，计算机帮助数学家完成了遍历工作。本研究旨在解决图的输入、图的基本操作和图的可视化问题，并尝试用这些函数解决较为复杂的问题。

为了解决这些问题，我们使用了 Python 代码和 Networkx 库。Python 易于编写，有着十分强大的第三方库支持。Networkx 库是一个用于创建、操作和研究复杂网络的 Python 库。它具有丰富的网络分析工具和算法，能够方便地处理图论问题。

使用 Python 和 Networkx 库能够有效地解决图论问题，并且在研究中进行图可视化，使得研究结果更加全面，更加具有可视性.。通过研究离散数学科目中的图论部分并编写相关代码，我们可以更好地了解图论，并为解决实际问题提供有力工具。

考虑到图论的复杂性，不同性质的图需要不同函数来处理，因此实际函数可能有微调。

准备工作如下：
1.下载 networks 库
2.启动 python，导入 networks 模块
如：>>> import networkx as nx

基本操作

创建图

创建图主要分为三个步骤：首先，对输入进行处理，处理输入并转换成一个列表。然后，将所得的数据处理成图的邻接矩阵。最后，将传入的邻接矩阵转换为 networkx 图对象。
以邻接表为例：

对于输入的处理：

```
# 输入一个邻接表，返回一个邻接矩阵
def list_to_matrix():
    # 从键盘接受用户输入
    input_data = ''
    while True:
        line = input()
        if line:
            input_data += line + '\n'
        else:
            break

    ret = adjacency_list(input_data)

    return ret
```

Code-1

将输入的邻接表转换为邻接矩阵：

```
# 传入一个邻接表，返回一个邻接矩阵
def adjacency_list(input_data):
    # 生成邻接表的字典
    temp_adjacency_list = {i: list(map(int, line.split()[1:])) for i, line in
enumerate(input_data.strip().split('\n'))}
    # 生成一个 nxn 矩阵
    num_nodes = len(temp_adjacency_list)
    matrix = [[0] * num_nodes for _ in range(num_nodes)]
    # 对矩阵赋值
    for i, neighbors in temp_adjacency_list.items():
        for j in neighbors:
            matrix[i][j - 1] = 1
    return matrix
```

Code-2

将邻接矩阵处理成 networkx 图对象：

```
# 将邻接矩阵转换为图对象
def matrix_to_graph(matrix):
```

```
    # 将矩阵转换为 numpy 矩阵
    np_matrix = np.matrix(matrix)

    # 使用 from_numpy_matrix 函数返回图对象
    G = nx.from_numpy_matrix(np_matrix)

    return G
```

Code-3

绘制图

color 参数用于判断是否要对图染色，如果是，那么得到的图中点将会被染上不同的颜色，且保证是最优的染色方案之一。

pos 参数用于确定图中点的位置，在后文求平面展开的部分会有具体的运用

```
# 绘制图
def plot_graph(G, color=False, pos=None):
    """
      绘制给定的图像。

      参数:
      G -- 一个 networkx 中的图对象
      color -- 可选参数，如果设置为 True，则染色
      pos -- 可选参数，表示节点位置的字典
      """
    # 使用 pos 参数绘制点
    if pos is None:
        pos = nx.spring_layout(G)
    nx.draw_networkx(G, pos, with_labels=True)

    # 如果需要，染色
    if color:
        colors = {0: 'red', 1: 'blue', 2: 'green', 3: 'yellow', 4: 'purple'}
        Colors = nx.greedy_color(G)
        nx.draw_networkx_nodes(G, pos, node_color=[colors[Colors[n]] for n in G.nodes()])

    plt.show()
```

Code-4

如果图是一张带权图，且需要在图中展示边权，可以使用以下代码：

```
# 绘制带权图
def show_weighted_graph(G):
    """
    绘制带权图。
```

```
    """
    # 计算点的位置
    pos = nx.spring_layout(G)

    # 绘制边
    nx.draw(G, pos, with_labels=True, node_size=300)

    # 添加边权标签
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

    # 显示图像
    plt.show()
```

Code-5

求解基本问题

求得最小生成树

我们可以使用 Kruskal 算法解决这一问题，在 networkx 库中也有相应的函数。受篇幅所限，此处给出调用函数的实现：

```
def minimum_spanning_tree(G):
    """
    返回最小生成树。

    参数:
    G -- 一个 networkx 中的图对象

    返回值:
    一个 networkx 中的最小生成树图对象
    """
    # 计算最小生成树
    mst = nx.minimum_spanning_tree(G, algorithm="kruskal")

    # 返回最小生成树
    return mst
```

Code-6

其中，algorithm 参数表示使用的算法名称，也可以根据树的性质选择 *prim* 算法或是 *boruvka* 算法。

求得两点间最短路径

同上，使用库中函数实现：

```
# 获得两点间最短路径
def shortest_path(G, source, target):
    """
    返回从源点到目标点的最短路径以及经过的路径。

    参数:
    G -- 一个 networkx 中的图对象
    source -- 源点
    target -- 目标点

    返回值:
    一个包含最短路径长度和路径节点的元组
    """
    # 计算最短路径长度
    length = nx.shortest_path_length(G, source, target)

    # 计算最短路径
    path = nx.shortest_path(G, source, target)

    # 返回最短路径长度和路径节点
    return length, path
```

Code-7

求得一个图的补图

仍然调用库函数实现：

```
# 获得一个图的补图
def complement(G):
    """
    返回给定图的补图。

    参数:
    G -- 一个 networkx 中的图对象

    返回值:
    一个 networkx 中的补图
    """
    # 计算补图
    H = nx.complement(G)

    # 返回补图
    return H
```

Code-8

查阅源码，可以发现核心语句是：

```
R.add_edges_from(
        ((n, n2) for n, nbrs in G.adjacency() for n2 in G if n2 not in nbrs if n != n2)
    )
```

Code-9

也即：对图 G 中的每两个点，如果它们在原始图中不相邻，那么在补图中相邻。

欧拉图、哈密顿图

同样调用库函数：

```
 # 计算欧拉回路
    circuit = nx.eulerian_circuit(G, start)
...
 # 计算哈密顿回路
    circuit = nx.hamiltonian_circuit(G, start)
```

Code-10

标注一个图的子图

先确定位置，再依次绘制原图和子图。子图用不同的颜色标注出来。

```
# 标注出一个图的子图
def show_subgraph(G, subG):
    """
    绘制图 G，并用红色标注出子图 subG。
    """
    # 计算点的位置
    pos = nx.spring_layout(G)

    # 绘制边
    nx.draw(G, pos, with_labels=True, node_size=300)

    # 绘制子图的边
    nx.draw(subG, pos, with_labels=True, node_size=300, edge_color='red')

    # 显示图像
    plt.show()
```

Code-11

判断图是否是平面图，并求得平面展开

```
# 判断图是否是平面图，返回平面图的坐标
def get_coordinates(G):
    if nx.check_planarity(G)[0]:
        # 如果是平面图，使用 planar_layout 函数获取坐标
        pos = nx.planar_layout(G)
        return pos
```

```
    else:
        # 如果不是平面图，返回 None
        return None
```

Code-12

可能的返回值 pos 是一个字典，标注了每个点的位置。依据这些位置来做出图，即可得到图的平面展开。

求解特殊问题

中国邮政员问题

先找到图中所有度为奇数的点，构成一个点集。不断地从这个点集中寻找两点间最短路径并添加到图中，同时在点集中删去这条最短路径的端点。重复直至点集为空，这样原图中就不存在奇数度顶点，也即原图成为欧拉图。

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


# 在给定图中找到 V 中任意两点的最短路径的最小值。
def min_shortest_path(G, V):
    """
    在给定图中找到 V 中任意两点的最短路径的最小值。

    参数：
    G -- 一个 NetworkX 图对象
    V -- G 中的顶点列表

    返回值：
    一个包含起点，终点和元组列表表示的最短路径边的三元组
    """
    min_length = float('inf')
    min_path = (None, None, [])
    for u in V:
        for v in V:
            if u != v:
                length, path = nx.single_source_dijkstra(G, u, v, weight='weight')
                if length < min_length:
                    min_length = length
                    min_path = (u, v, [(path[i], path[i + 1]) for i in range(len(path) -
1)])
    return min_path
```

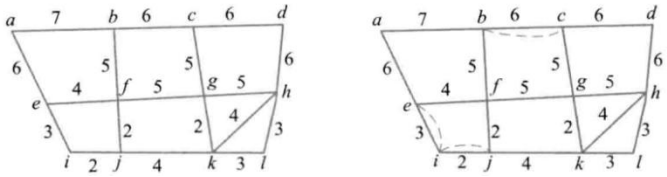
```
# 将给定的图 G 转换为多重图，并添加 paths 中的所有边
def add_paths(G, paths):
    """
    将给定的图 G 转换为多重图，并添加 paths 中的所有边
    """
    MG = nx.MultiGraph()
    MG.add_nodes_from(G.nodes())
    MG.add_weighted_edges_from(G.edges(data='weight'))
    for path in paths:
        for edge in path:
            MG.add_edge(edge[0], edge[1], weight=G[edge[0]][edge[1]]['weight'])
    return MG


def chinese_postman(G, start=None):
    """
    解决中国邮政员问题，传入图 G，返回邮政员的路径和总长
    """
    # 奇点集
    odd_vertices = []
    # 确定图是否有欧拉回路
    if not nx.is_eulerian(G):
        # 标识奇点
        odd_vertices = [v for v in G.nodes() if G.degree(v) % 2 == 1]
    # 奇点之间的最短路径集合
    paths = []
    # 当图中存在奇点时
    while odd_vertices:
        # 计算奇点集中两个点之间的最短路径
        u, v, path = min_shortest_path(G, odd_vertices)
        # 添加最短路径到集合中
        paths.append(path)
        # 从奇点集中删除已经连接的点
        odd_vertices.remove(u)
        odd_vertices.remove(v)

    # 在原图中添加所有奇点之间的最短路径，得到多重图 MG
    MG = add_paths(G, paths)

    # 在多重图 MG 中求出欧拉回路
    cp_path = list(nx.eulerian_path(MG, start))
    # 计算欧拉回路的总长
    cp_length = sum(MG[cp_path[i][0]][cp_path[i][1]][0]['weight'] for i in
```

```
range(len(list(cp_path))))
    # 返回欧拉回路路径和总长
    return cp_path, cp_length




if __name__ == "__main__":
    G = graph_with_weight()
    p, l = chinese_postman(G, 'a')
    print(convert_to_string(p), l, sep='\n')
    show_weighted_graph(G)
```

Code-13

以《离散数学》上例题为例：

例 15.7 邮递员负责的街区如图 15.12(a)所示,长度单位是百米,邮局位于 a 处. 试设计邮递员的最短投递路线.



Graph-1

这个问题描述的是这样一张图：

a b 7　b c 6　c d 6　a e 6　e f 4　b f 5　f g 5　c g 5　g h 5
d h 6　e i 3　i j 2　f j 2　j k 4　g k 2　k h 4　k l 3　h l 3
（分别表示边的起点、终点、边权）
将此图输入，得到结果：

```
abcbfeijfgcdhgkhlkjiea
89
```

结果总长度与书中给出的结果一致。具体路径有所不同，但不难验证也是原图的一条欧拉回路。

拉姆齐定理

在K_6中，用黑白两种颜色对边染色，必有黑色或白色的三阶完全图。
该命题等价于：对于K_6的任何子图 subG，必有K_3是 subG 的子图或K_3是 subG 补图的子图

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

def is_subgraph(G1, G2):
    # 检查 G2 是否是 G1 的子图
```

```
    return nx.algorithms.isomorphism.GraphMatcher(G1, G2).subgraph_is_isomorphic()


def all_subgraphs(G):
    """
    生成给定图 G 的所有子图的列表。
    """
    # 获取 G 的所有边的子集
    edge_subsets = itertools.chain.from_iterable(
        itertools.combinations(G.edges(), r) for r in range(len(G.edges()) + 1))
    # 创建子图列表
    subgraphs = []
    for edges in edge_subsets:
        # 从边创建子图
        subgraph = nx.Graph()
        subgraph.add_edges_from(edges)
        # 添加来自 G 的节点
        subgraph.add_nodes_from(G.nodes())
        # 将子图添加到列表中
        subgraphs.append(subgraph)

    return subgraphs


# 不需要输入
if __name__ == "__main__":
    G_6 = nx.complete_graph(6)
    G_3 = nx.complete_graph(3)
    subgraphs = all_subgraphs(G_6)

    for i in subgraphs:
        # 如果子图 i 和 i 的补图都不包含 3 阶完全图
        if not (is_subgraph(i, G_3) or is_subgraph(nx.complement(i), G_3)):
            print('wrong')
            break
    else:
        print('r(3,3)=6')
```

Code-14

原理上，这段代码可以用于验证 r(4,4)，但所需时间较长。

哈夫曼编码

实现了哈夫曼树的构建和编码生成。

首先，函数 build_huffman_tree()接收一个参数 freq，freq 是一个字典，存储了各个字符的频率。它首先将字符和其对应频率封装成元组，并存入堆中。然后不断地从堆中取出两个元素，将它们合并成一个新的元素，直到堆只剩下一个元素为止。最后返回这个堆中的唯一元素，即哈夫曼树。

接着，函数 generate_huffman_code()接收两个参数，一个是哈夫曼树，另一个是前缀。递归地遍历哈夫曼树，如果当前结点是一个元组，则继续遍历它的左右子树，并在前缀上加上"0"和"1"。如果当前结点是一个字符，则将该字符和前缀添加到字典 huffman_code 中。这样就得到了各个字符的哈夫曼编码。

```
def build_huffman_tree(freq):
    heap = [(freq[w], w) for w in freq]
    heapq.heapify(heap)

    while len(heap) > 1:
        f1, c1 = heapq.heappop(heap)
        f2, c2 = heapq.heappop(heap)
        heapq.heappush(heap, (f1 + f2, (c1, c2)))

    return heap[0][1]


def generate_huffman_code(tree, prefix=""):
    if isinstance(tree, tuple):
        generate_huffman_code(tree[0], prefix + "0")
        generate_huffman_code(tree[1], prefix + "1")
    else:
        huffman_code[tree] = prefix
```

Code-15

一个测试用例是：

```
# Example usage
freq = {'a': 6, 'b': 3, 'c': 8, 'd': 2, 'e': 10, 'f': 4}
tree = build_huffman_tree(freq)
huffman_code = {}
generate_huffman_code(tree)
print(huffman_code)
```

Code-16

示例

最小生成树

一张带权图，计算出它的最小生成树并标记出来。
（在此题中，使用虚线来标注最小生成树）

以例 16.3(a)为例：

输入的图：

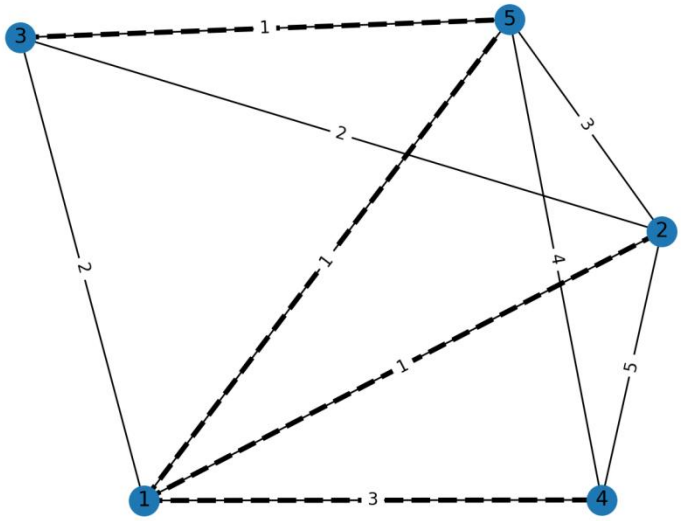
1 2 1　2 3 2　1 3 2　2 4 5　2 5 3　4 5 4　3 5 1　1 4 3　1 5 1

使用的代码：

```
G = graph_with_weight()
minG = minimum_spanning_tree(G)
show_subgraph(G, minG)
```

Code-17

得到的最小生成树：



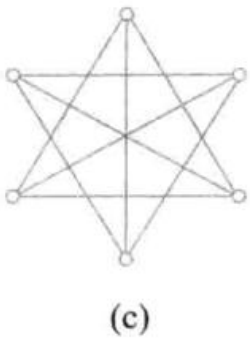
Graph-2

平面嵌入

根据度序列生成一张图，(需要保证生成的图唯一)。然后返回它的平面嵌入并绘制

以习题 17.2(c)为例：

原题图：



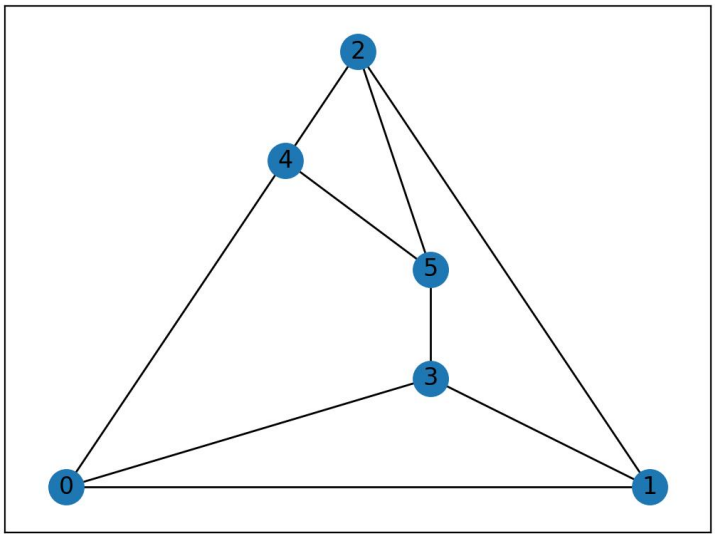
Graph-3

不难发现，每个点的度数都是 3，因此从给定的度序列就可以生成对应的图。
使用的代码：

```
    degree = [3, 3, 3, 3, 3, 3]
    G = generate_simple_graph(degree)  # 通过度序列得到简单图
    pos = get_coordinates(G)
    plot_graph(G, pos=pos)
```

Code-18

得到的图：



Graph-4

不难验证，此图即是原图的一个平面嵌入。

结束语

本文通过编程实现了图论中的重要算法和方法，并综合运用这些方法解决了部分具体问题。在实现过程中，我们采用了模块化的编程思想，使得代码具有较高的可移植性和可组合性，为进一步的研究与应用奠定了基础。由于时间和篇幅的限制，本文中的代码仅涵盖了图论中的部分基本概念，在使用时可能需要进行一定的调整。在未来的研究中，我们将继续深入探索图论中更为复杂的问题，并寻求优化代码的时空复杂度。篇幅所限，本文中所有代码仅是示例代码片段，完整代码可在“https://github.com/zeta-zl/Discrete-Mathematics”中查询。

参考文献

[1]Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics,

and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008