算法期末大作业实验报告

#2022201895 于佳鑫

实验目的

本实验的目的是实现复杂网络中的几种常用算法，包括计算图的基础指标、最密子图、k-core分解、k-clique最密子图等。此外，还将实现静态和交互式图的可视化，支持节点和边的样式设置，以及良好的交互性（如缩放、平移、布局调整等）。

整体框架介绍

本项目使用Python实现，项目代码结构如下：

文件结构

```
LAB/
|
├── data/                                  # 存放图数据的文件夹
|
├── graph_lib/                             # 实现图算法的核心库
|   ├── __init__.py                        # 初始化文件
|   ├── algorithms.py                      # 各种图算法的实现
|   ├── graph.py                           # 图的基本操作和算法实现
|   ├── io.py                              # 图的输入输出功能
|   └── visualization.py                   # 图的可视化功能
|
├── lib/                                   # 其他库文件
|
├── outputs/                               # 输出结果的文件夹
|   ├── bron_kerbosch_cliques/             # Bron-Kerbosch算法的输出结果
|   ├── densest_subgraph_approx/           # 贪心算法求解最密子图的输出结果
|   ├── densest_subgraph_exact/            # 精确算法求解最密子图的输出结果
|   ├── graph/                             # 图的基本操作输出结果
|   ├── k_clique_densest/                  # k-clique最密子图的输出结果
|   ├── k_core/                            # k-core分解的输出结果
|   ├── visualization/                     # 静态可视化的输出结果
|   └── visualization_interactive/         # 交互式可视化的输出结果
```

```
|
├── install.ipynb                    # 安装依赖的Jupyter Notebook文件
├── main.py                          # 主程序文件
└── README                           # 项目简介
```

数据集介绍

由于Python运行效率和设备的限制以及本人能力有限，尽管对算法进行了多次优化，但是在运行静态图像时仍然存在运行时间很长的问题。所以我选用了一些更小的数据集来进行算法的验证。

```
1. karate.txt:   Nodes: 34      Edges: 78
2. fb-pages.txt: Nodes: 620     Edges: 2083
3. tvshow.txt:   Nodes: 3891    Edges: 17221
4. CondMat.txt:  Nodes: 23133   Edges: 93439
```

示例使用方法

运行示例

1. **安装依赖**:
   打开 `install.ipynb` 文件，并运行其中的单元格，安装所需的Python包。
2. **运行主程序**:
   使用命令行运行 `main.py` 文件，提供所需的命令行参数。示例如下：
   - 计算并保存图的密度和平均度：

     ```
     python main.py input.txt --density --average_degree
     ```

   - 计算并保存 k-core 分解结果：

     ```
     python main.py input.txt --k_cores
     ```

   - 计算并保存最密子图（精确算法）：

     ```
     python main.py input.txt --densest_subgraph exact
     ```

   - 生成并保存静态图的可视化结果：

     ```
     python main.py input.txt --visualize --layout circular --node_color red
     --node_size 600 --edge_color blue --font_size 12
     ```

   - 生成并保存交互式图的可视化结果：

     ```
     python main.py input.txt --visualize_interactive --node_color green --
     node_size 15 --edge_color gray
     ```

实验步骤

1. **实现基础图操作和指标计算**
    - **图的加载与基本操作**：
        - 图的加载：使用 NetworkX 库从文件中读取图数据，构建图对象。实现从文件加载节点和边的方法，支持去除重边和自环。
        - 基本操作：实现添加边、删除边、获取节点和边等基本操作。
    - **计算图的密度和平均度**：
        - 图的密度：图的密度定义为边数与最大可能边数之比。实现计算图密度的方法。
        - 图的平均度：图的平均度是所有节点度数的平均值。实现计算图平均度的方法。
2. **实现 k-core 分解算法**
    - **算法思路**：
        - k-core 是图中一个最大子图，其中每个节点的度数至少为 k。利用 NetworkX 的 core_number 函数计算图中每个节点的 coreness 值。
        - 遍历所有节点，根据每个节点的 coreness 值进行 k-core 分解。
    - **结果保存**：
        - 将计算得到的每个节点的 coreness 值保存到文件中，文件格式如下：

            ```
            运行时间
            节点 coreness[节点]
            ```

3. **实现最密子图算法**
    - **精确算法**：
        - **算法思路**：
            - 使用二分搜索结合最大流算法求解最密子图。最密子图是边密度最高的子图。
            - 首先通过二分搜索确定最大密度值。然后构建一个流网络，通过最大流算法Dicnic验证该密度值是否可行。
        - **步骤**：
            1. 初始化图的度数范围作为二分搜索的左右边界。
            2. 在每次迭代中，计算中间值作为候选密度。
            3. 构建流网络，判断该密度值是否可行。
            4. 根据判断结果更新二分搜索的边界，直至收敛。
    - **贪心算法**：
        - **算法思路**：
            - 贪心算法通过反复删除当前度数最小的节点来近似求解最密子图。每次删除一个节点后，重新计算图的密度。
        - **步骤**：
            1. 复制原始图作为工作图。
            2. 在每次迭代中，删除度数最小的节点。
            3. 更新图的密度，记录当前最优密度及对应的子图。
            4. 直至图中没有节点为止，输出密度最大的子图。
    - **结果保存**：
        - 将计算得到的最密子图及其密度保存到文件中，文件格式如下：

```
运行时间
密度
子图中的节点列表
```

4. **实现 k-clique 最密子图的启发式算法**
   - **算法思路**：
     - k-clique 是指图中的一个完全子图，包含 k 个节点。使用启发式算法计算密度最大的 k-clique 子图。
     - 算法通过多次迭代，更新节点值来近似找到密度最大的 k-clique 子图。每次迭代中，对所有 k-clique 进行遍历，找到最小值节点并更新其值。
     - 经过多次迭代后，根据节点值提取密度最大的子图。
   - **步骤**：
     1. 初始化所有节点的值为 0。
     2. 进行若干次迭代：
        - 每次迭代中，遍历所有 k-clique，找到其中值最小的节点并更新其值。
        - 记录每个节点的值。
     3. 归一化节点值，按节点值排序，提取前 k 个节点构成子图。
     4. 计算并输出该子图的密度。
   - **结果保存**：
     - 将计算得到的 k-clique 最密子图及其密度保存到文件中，文件格式如下：

       ```
       运行时间
       密度
       子图中的节点列表
       ```

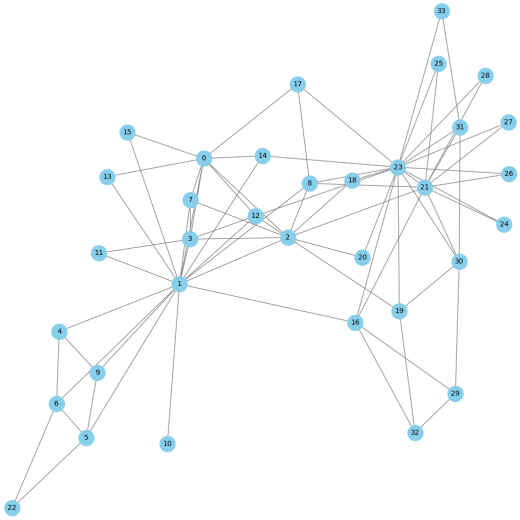
5. **实现图的静态和交互式可视化**
   - **静态可视化**：
     - **思路**：
       - 使用 Matplotlib 和 NetworkX 实现静态图的可视化。
       - 支持不同的布局方式（如 spring, circular, shell 等），并允许用户设置节点和边的样式（颜色、大小、标签等）。
     - **步骤**：
       1. 根据用户选择的布局方式计算节点的位置。
       2. 设置节点和边的颜色、大小等样式参数。
       3. 使用 Matplotlib 绘制图形，保存并展示图像。
   - **交互式可视化**：
     - **思路**：
       - 使用 Pyvis 实现交互式图的可视化。Pyvis 允许在浏览器中查看和操作图，支持缩放、平移和布局调整等功能。
     - **步骤**：
       1. 创建 Pyvis 网络对象，设置图的基本属性（如大小、背景色、字体颜色等）。
       2. 添加节点和边，设置其样式参数。
       3. 启用交互功能，如物理引擎、缩放和平移等。
       4. 生成 HTML 文件并保存。
6. **设计并执行实验**

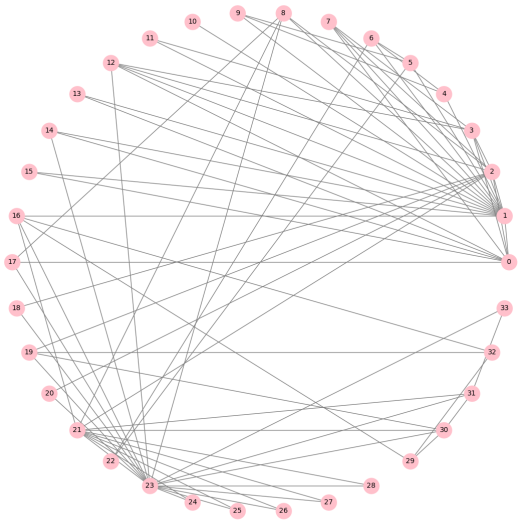
- **加载样例图数据**：
  - 从 data 文件夹中加载样例图数据，构建图对象。
- **执行各类算法**：
  - 依次执行图的基础操作、k-core 分解、最密子图算法和 k-clique 最密子图算法。
  - 记录每个算法的运行时间和计算结果。
- **生成并保存可视化结果**：
  - 生成图的静态和交互式可视化结果，并保存到 outputs 文件夹中。
- **记录运行时间和计算结果**：
  - 将各类算法的运行时间和计算结果保存到对应的输出文件中。
  - 分析结果，验证算法的正确性和效率。

实验结果及分析

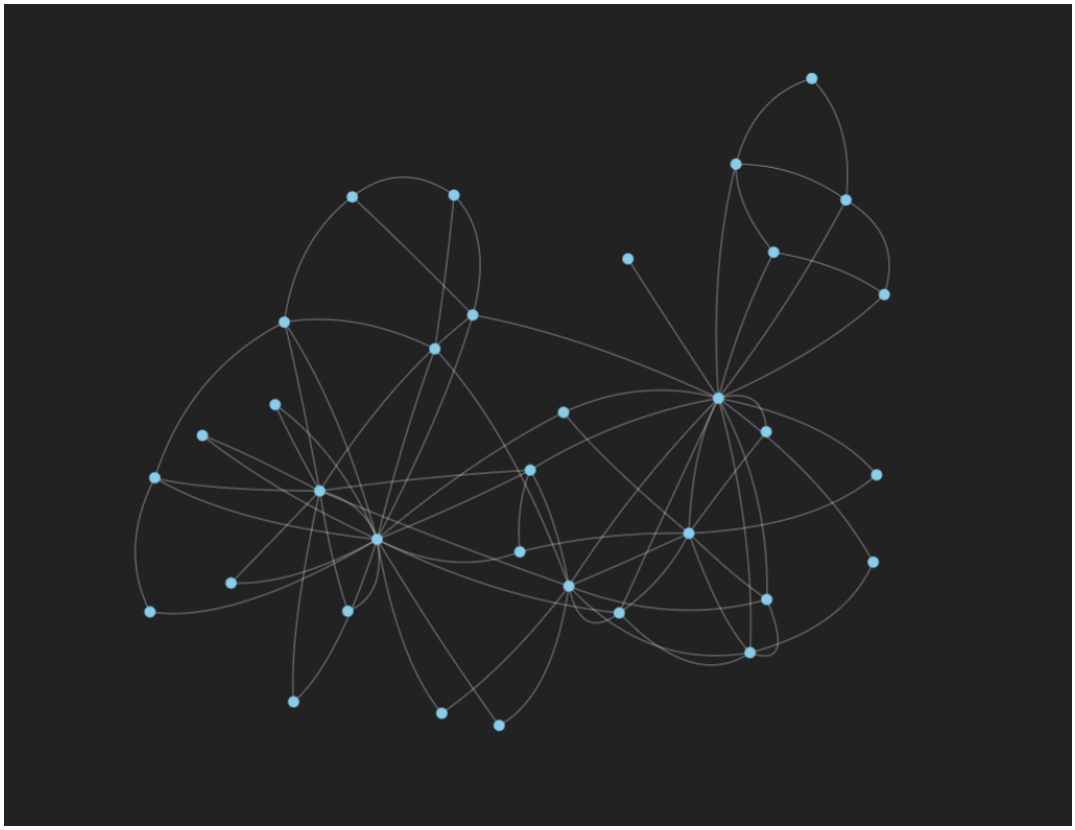
实验结果全部存放于outputs文件夹中。

1. **基础指标计算**
   - 计算了样例图的密度和平均度，验证了基础操作的正确性。
2. **k-core 分解结果**
   - 通过 k-core 分解算法计算得到每个节点的 coreness，并将结果保存到文件中。
   - 结果展示了图中节点的 k-core 结构，验证了算法的正确性和效率。
3. **最密子图算法结果**
   - 精确算法和贪心算法分别计算得到样例图的最密子图，并将结果保存到文件中。
   - 通过比较两种算法的运行时间和结果，验证了精确算法的准确性和贪心算法的效率。
4. **k-clique 最密子图结果**
   - 通过启发式算法计算得到 k=4 时的最密子图，并将结果保存到文件中。
   - 结果展示了算法在不同迭代次数下的收敛性和计算效率。
5. **静态和交互式可视化结果**
   - 使用 Matplotlib 生成了静态图的可视化结果，展示了不同布局和样式设置的效果。





- 使用 Pyvis 生成了交互式图的可视化结果，支持缩放、平移和布局调整，提高了用户体验。（可动态移动调整节点位置，缩放大小，平移等）



部分命令行结果如下：

```
& python main.py .\data\karate.txt --bron_kerbosch_cliques 4
最大k-cliques (k=4): [{31, 21, 30, 23}]
Bron-Kerbosch最大cliques结果已保存到:
outputs/bron_kerbosch_cliques\karate_bron_kerbosch_cliqurborbosch_cliques_202406
27173533.txt
& python main.py .\data\karate.txt --densest_subgraph approx
最密子图结果已保存到:
outputs/densest_subgraph_approx\karate_densest_subgraph_approx_20240627173600.tx
t
& python main.py .\data\karate.txt --densest_subgraph exact
最密子图结果已保存到:
outputs/densest_subgraph_exact\karate_densest_subgraph_exact_20240627173610.txt
& python main.py .\data\karate.txt --k_clique_densest 4
k-clique最密子图结果已保存到:
outputs/k_clique_densest\karate_k_clique_densest_4_20240627173707.txt
& python main.py .\data\karate.txt --k_cores
k-core分解结果已保存到: outputs/k_core\karate_k_core_20240627173733.txt
& python main.py .\data\karate.txt --visualize_interactive
图的交互式可视化结果已保存到:
outputs/visualization_interactive\karate_visualization_interactive_2024062717375
1.html
& python main.py .\data\karate.txt --visualize --node_size 7 --edge_color grey -
-font_size 7
图的静态可视化结果已保存到:
outputs/visualization\karate_visualization_20240627173811.png
& python main.py .\data\karate.txt --visualize --node_size 7 --edge_color grey -
-font_size 7 --layout circ

& python main.py .\data\karate.txt --density --average_degree
Graph Density: 0.1354723707664884
Average Degree: 4.470588235294118
```

结论

通过本实验，成功实现了几种常用的图算法，并通过静态和交互式可视化展示了算法的结果。结果表明，各种算法在样例图数据上的表现符合预期。通过实验分析，可以得出以下结论：

1. 精确算法在计算最密子图时具有高精度，但计算时间较长。
2. 贪心算法计算最密子图效率高，但结果可能不够精确。
3. 启发式算法在计算 k-clique 最密子图时，具有较好的收敛性和计算效率。
4. 交互式可视化工具增强了对复杂网络结构的理解和分析能力，提供了良好的用户体验。

进一步的工作可以包括对更多样例数据进行测试，优化算法性能，以及增强可视化工具的功能。

通过本次实验，我更加了解了一个Python包的整体框架的构建，同时对图算法有了更深的了解。

代码展示

以下为实验中主要代码的实现：

```
import networkx as nx
import time
import matplotlib.pyplot as plt
from itertools import combinations
from pyvis.network import Network
from matplotlib.colors import Normalize
import matplotlib.cm as cm
from networkx.algorithms.flow import maximum_flow

class Graph:
    def __init__(self):
        self.graph = nx.Graph()
        self.node_map = {}  # 用于存储顶点映射
        self.reverse_map = {}  # 用于存储反向映射关系

    def add_edge(self, u, v):
        self.graph.add_edge(u, v)

    def remove_edge(self, u, v):
        self.graph.remove_edge(u, v)

    def nodes(self):
        return self.graph.nodes()

    def edges(self):
        return self.graph.edges()

    # 获取图的密度
    def density(self):
        return nx.density(self.graph)

    # 获取图的平均度
    def average_degree(self):
        degrees = dict(self.graph.degree()).values()
        return sum(degrees) / len(degrees) if len(degrees) > 0 else 0
```

```
    # 计算k-core分解并保存到文件
    def k_cores(self, output_path):
        start_time = time.time()
        cores = nx.core_number(self.graph)
        end_time = time.time()
        runtime = end_time - start_time

        try:
            with open(output_path, 'w') as f:
                f.write(f"{runtime:.4f}s\n")
                for node in sorted(cores):
                    original_node = self.reverse_map.get(node, node)
                    f.write(f"{original_node} {cores[node]}\n")
        except IOError as e:
            print(f"保存k-core分解结果时出现错误: {e}")
        except Exception as e:
            print(f"保存k-core分解结果时出现未知错误: {e}")

    def visualize(self, output_path, node_color='skyblue', node_size=500,
edge_color='green', font_size=10, with_labels=True, layout='spring'):
        pos = self._get_layout(layout)

        # 创建图形
        plt.figure(figsize=(12, 12))
        # 优化布局
        if layout == 'spring':
            pos = nx.spring_layout(self.graph, k=0.1, iterations=50)
        elif layout == 'circular':
            pos = nx.circular_layout(self.graph)
        elif layout == 'shell':
            pos = nx.shell_layout(self.graph)
        elif layout == 'spectral':
            pos = nx.spectral_layout(self.graph)
        elif layout == 'kamada_kawai':
            pos = nx.kamada_kawai_layout(self.graph)
        else:
            pos = nx.spring_layout(self.graph)  # 默认使用 spring 布局
        # 节点颜色优化
        if isinstance(node_color, list):
            norm = Normalize(vmin=min(node_color), vmax=max(node_color))
            cmap = cm.viridis
            node_color = [cmap(norm(value)) for value in node_color]
        # 边颜色和宽度优化
        if isinstance(edge_color, list):
            norm = Normalize(vmin=min(edge_color), vmax(max(edge_color))
            cmap = cm.plasma
            edge_color = [cmap(norm(value)) for value in edge_color]

        edge_width = [self.graph[u][v].get('weight', 1.0) for u, v in
self.graph.edges()]
        # 绘制图形
        nx.draw(self.graph, pos, with_labels=with_labels, node_size=node_size,
node_color=node_color, edge_color=edge_color, width=edge_width,
font_size=font_size)
        # 去除背景网格
```

```
        plt.axis('off')
        # 保存和展示图形
        plt.savefig(output_path)
        plt.show()

    def _get_layout(self, layout):
        if layout == 'spring':
            return nx.spring_layout(self.graph)
        elif layout == 'circular':
            return nx.circular_layout(self.graph)
        elif layout == 'random':
            return nx.random_layout(self.graph)
        elif layout == 'shell':
            return nx.shell_layout(self.graph)
        elif layout == 'spectral':
            return nx.spectral_layout(self.graph)
        else:
            raise ValueError(f"Unsupported layout: {layout}")

    # 交互式可视化图
    def visualize_interactive(self, output_file='graph.html', node_color='blue',
node_size=10, edge_color='gray'):
        net = Network(height='750px', width='100%', bgcolor='#222222',
font_color='white')
        net.barnes_hut()
        for node in self.graph.nodes:
            net.add_node(node, label=str(node), color=node_color,
size=node_size)
        for edge in self.graph.edges:
            net.add_edge(edge[0], edge[1], color=edge_color)
        net.show_buttons(filter_=['physics'])
        net.save_graph(output_file)

    def exact_densest_subgraph(self):
        def get_density(subgraph):
            nodes = subgraph.nodes()
            edges = subgraph.edges()
            return len(edges) / len(nodes)

        def binary_search_densest_subgraph():
            def check_density(graph, threshold):
                G = graph.copy()
                source, sink = 'source', 'sink'
                node_list = list(G.nodes())  # Copy the nodes list to avoid
modifying during iteration
                for node in node_list:
                    G.add_edge(source, node, capacity=len(G.edges(node)))
                    G.add_edge(node, sink, capacity=threshold)
                flow_value, _ = maximum_flow(G, source, sink,
flow_func=nx.algorithms.flow.edmonds_karp)
                return flow_value < len(G.edges)

            degrees = [deg for node, deg in self.graph.degree()]
            left, right = min(degrees
```

```
), max(degrees)
            while right - left > 1e-5:
                mid = (left + right) / 2
                if check_density(self.graph, mid):
                    right = mid
                else:
                    left = mid

            return left

        max_density = binary_search_densest_subgraph()
        densest_subgraph = None

        for size in range(1, len(self.graph.nodes()) + 1):
            for subset in combinations(self.graph.nodes(), size):
                subgraph = self.graph.subgraph(subset)
                if get_density(subgraph) == max_density:
                    densest_subgraph = subgraph
                    break

        return densest_subgraph, max_density

    def greedy_densest_subgraph(self):
        graph = self.graph.copy()
        best_density = 0
        best_subgraph = graph.copy()

        while graph.number_of_nodes() > 0:
            density = nx.density(graph)
            if density > best_density:
                best_density = density
                best_subgraph = graph.copy()

            min_degree_node = min(graph.nodes, key=graph.degree)
            graph.remove_node(min_degree_node)

        return best_subgraph, best_density

    # 计算最密子图并保存到文件
    def densest_subgraph(self, output_path, method="exact"):
        start_time = time.time()

        if method == "exact":
            subgraph, max_density = self.exact_densest_subgraph()
        elif method == "approx":
            subgraph, max_density = self.greedy_densest_subgraph()
        else:
            raise ValueError("Unsupported method. Use 'exact' or 'approx'.")

        end_time = time.time()
        runtime = end_time - start_time

        try:
            with open(output_path, 'w') as f:
                f.write(f"{runtime:.4f}s\n")
```

```
                f.write(f"density {max_density:.4f}\n")
                if subgraph is not None:
                    for node in sorted(subgraph.nodes()):
                        original_node = self.reverse_map.get(node, node)
                        f.write(f"{original_node} ")
                    f.write("\n")
        except IOError as e:
            print(f"保存最密子图结果时出现错误: {e}")
        except Exception as e:
            print(f"保存最密子图结果时出现未知错误: {e}")

    # k-clique分解
    def bron_kerbosch(self, r, p, x, cliques):
        if not p and not x:
            cliques.append(r)
            return
        for v in list(p):
            self.bron_kerbosch(r | {v}, p & set(self.graph.neighbors(v)), x &
set(self.graph.neighbors(v)), cliques)
            p.remove(v)
            x.add(v)
    def find_maximal_cliques(self):
        cliques = []
        self.bron_kerbosch(set(), set(self.graph.nodes()), set(), cliques)
        return cliques
    def k_clique_decomposition(self, k):
        maximal_cliques = self.find_maximal_cliques()
        k_cliques = [clique for clique in maximal_cliques if len(clique) == k]
        return k_cliques

    # 使用启发式算法计算k-clique最密子图
    def k_clique_densest_subgraph(self, k, iterations=1000):
        r = {node: 0 for node in self.graph.nodes}
        for _ in range(iterations):
            s = r.copy()
            for clique in self._find_k_cliques(k):
                min_node = min(clique, key=lambda node: s[node])
                r[min_node] += 1
        for node in r:
            r[node] /= iterations
        densest_subgraph, density = self._extract_densest_subgraph(r, k)
        return densest_subgraph, density
    def _find_k_cliques(self, k):
        # 使用networkx查找所有k-cliques
        cliques = [clique for clique in nx.find_cliques(self.graph) if
len(clique) == k]
        return cliques
    def _extract_densest_subgraph(self, r, k):
        sorted_nodes = sorted(r, key=r.get, reverse=True)
        subgraph_nodes = sorted_nodes[:k]
        subgraph = self.graph.subgraph(subgraph_nodes)
        return subgraph, nx.density(subgraph)
    # 查找并保存k-clique最密子图
    def find_k_clique_densest_subgraph(self, k, output_path, iterations=1000):
        start_time = time.time()
```

```
        subgraph, density = self.k_clique_densest_subgraph(k, iterations)
        end_time = time.time()
        runtime = end_time - start_time
        try:
            with open(output_path, 'w') as f:
                f.write(f"{runtime:.4f}s\n")
                f.write(f"{density:.4f}\n")
                for node in sorted(subgraph.nodes()):
                    original_node = self.reverse_map.get(node, node)
                    f.write(f"{original_node} ")
                f.write("\n")
        except IOError as e:
            print(f"保存k-clique最密子图结果时出现错误: {e}")
        except Exception as e:
            print(f"保存k-clique最密子图结果时出现未知错误: {e}")
```