# 2023-2024 学年秋季学期

# 《计算思维实训（2）》(0830A031)

# 课程报告

| 成绩<br>(百分制) | |
|---|---|

| | | | |
|---|---|---|---|
| 学　　号 | 22121799 | 学　　院 | 计算机工程与科学学院 |
| 姓　　名 | 金义杰 | 手工签名 | |
| 报告题目 | 无尽度网络的小世界特性探索 | | |
| 实训报告成绩 50% | 实训过程描述：清晰、全面。（5%） | | |
| | 报告主要部分：1、计算思维实例叙述清楚、有意义；2、分析到位；3、思路独特或有创意；4、关键实现技术描述清楚详细；5、内容丰富；6、不直接粘贴所有代码；7、代码有注释或说明。（25%） | | |
| | 实训收获体会：感受真实、深刻，建议有价值。（10%） | | |
| | 书写格式：书写规范、用词正确、无明显的错别字，图表、代码清晰规范，格式协调、效果好，参考文献书写、引用规范合理。（10%） | | |
| 工作实绩 50% | 1、平时表现、进步情况（25%），2、工作实绩（25%） | | |

**教师对该生工作实绩简要评述：**

教师签名：

日期：　　年　　月　　日

# Part 1. 本学期实训过程概述

## 1. 实训总体概况、实训过程

### 1.1 总体概况

本次实训我基于上学期的无尺度网络建模的经验，针对本学期的研究内容，重新进行了无尺度网络的建模，使用了更高级的数据结构，并实现了更完善的网络设计，完成了对无尺度网络的小世界特性的分析。

### 1.2 实训过程

1. 重览论文、加深理解：仔细查看了小世界网络的论文资料，并通过各种渠道加深理解小世界网络的拓扑特性，并将其特性可能在无尺度网络上的表现一一列举。
2. 初步尝试：一开始是在网上找了相应解析，妄图通过一两个公式实现复杂的功能，但尝试发现并不可行，同时，先前网络结构的设计的弊端也相应体现（如设计成有向图，无法有效进行深度优先查找遍历）。
3. 重新设计：在认识到网络的缺陷之后，我选择重新设计网络结构，此次设计保证了一个无向图的结构，而由于内存分配的问题我依旧摒弃了指针的设计，同时内存占用的问题也让我不得不放弃原先一直准备做的邻接表的设计。（在完成相应设计后，开 100000 顶点的网络，其邻接矩阵就直接占满了 64G 内存，触发内存分配错误），当然，在重新设计网络结构之后，我依旧统计了其度分布特性，用以确定其确实是一个无尺度网络。
4. 核心思考：由于图结构的复杂性，对其遍历需要我在算法上有更深的理解，因此我花费了一定的时间理解图的基本存储结构和传统的搜索、最优化算法，其中就包括针对二元最短路径的弗洛伊德算法，并基于深度优先搜索进行了细节查找的实现，利用该优先搜索算法，我也利用其同步实现了聚类系数的统计记录。
5. 书写报告与总结：此次完成任务相对顺利，遇到了相应困难，我通过搜索资料和请教老师、实验室的学长学姐快速突破了知识难点并快速搭建全新模型得出优质结果。总结整段实训经历，我受益颇多，并且我相信，这段时间的复杂网络科学的入门研究，对我后续进行人工智能入门研究、分析理解复杂社会哲学关系上都有极大的帮助。

# 2. 对计算思维实训的认识与体会

## 2.1 实训认识

我对本次计算思维实训的认识在于，其使得我对算法的认识更深了一步，实训任务完成后我发现，过去在我心中那些遥不可及的复杂算法，其实也并没有那么难以理解。其实我稍微划以时间，解决它并不是天大的难事，我认为这个是我们计算思维的一个很重要的点，那就是敢于钻研、沉于钻研、乐于钻研，这才是我们学习计算机的乐趣所在。

计算机是一个庞大的领域，每个领域都有其特殊的知识，其实每一个小的知识都不算复杂，可当我们将其串在一起形成一个复杂的体系系统之后，它才称之为计算机科学，而我们研究者所要做的就是一步步地学习、攻克那一个个小的要点，循序渐进下，窥得其全貌并不是遥不可及的梦想。

## 2.2 实训体会

本次实训我针对需求完成的任务，结合先前的无尺度网络建模经验，重新进行了无尺度网络的数据结构的设计，遵循了基本的图数据结构，选用更优的随机数生成器，完成一个基于大规模复杂无向图的无尺度网络的设计，结合了图、最优化理论、并行计算技术完成了更多的复杂操作。这让我成功接触了计算机的多个领域的多个小的方面的知识，提升了我的知识广度，为我后续学习拓宽知识面提供了理论基础。

基于上次实训的经验，我完全重新设计了网络结构，使其更符合无向图的特征并可进行深度优先搜索，实现各项最短路径算法规划。虽然邻接矩阵依旧没有进行实现，但我在学习的过程中已经明晰了其原理，且完成了相应设计，但其内存占用过大，我不得不将其摒弃。这次重新实现网络，提升了我的工程能力，这是一个计算机人必备的能力。

本次实训我还学习了一些多线程并行计算的实现方式（利用 OpenMP），这个方法对我的网络生成和最短路径的计算带来了极大的帮助。我初步认识了并行计算，为我后续进行深入研究打下基础。

借助对无尺度网络的理解和建模过程的锻炼，我已经对网络科学有了一些初步的认识，再结合先前对数字逻辑、OOP 和算法的理解，我融会贯通，初步理解了图结构，状态机和动态规划的理论，这一部分的理解促使我开始阅览卷积神经网络相关论文，并着手开始照猫画虎搭建一些经典的卷积神经网络。

# Part 2. 综合实训报告

# 无尺度网络的小世界特性探索

22121799 金义杰

计算机工程与科学学院

**摘要：**本文围绕无尺度网络与小世界模型展开研究，介绍了它们在复杂网络领域中的重要性和特性。基于先前研究成果，重新优化了无尺度网络的构建算法，添加并行机制。通过 C++代码设计和实验数据分析，验证了无尺度网络的小世界特性，并对其平均最短路径和聚类系数进行了深入讨论。研究结果表明，无尺度网络具有小世界网络的特性，为复杂网络研究入门研究提供了一定参考。

**关键词：**复杂网络科学；无尺度网络; 小世界网络；图论；最优化理论；

## 1 无尺度网络与小世界模型

无尺度网络与小世界模型是复杂网络领域中的重要研究方向。在过去的几十年中，人们对网络拓扑结构进行了深入的探索和理解，从而提出了不同的网络模型。

在过去，ER 随机网络被广泛应用于描述网络结构，即节点之间的连接是随机发生的。然而，随着对真实世界网络的研究，人们发现 ER 随机网络并不能完全反映现实网络的特性。因此，研究者开始寻找更适合描述真实网络的模型。

小世界网络模型是在这一背景下提出的[3]。在小世界网络中，节点之间的连接不再是完全随机的，而是具有一定的规律性。通过选择性地连接邻近的节点，小世界网络展现出了“六度分隔”或“小世界”效应：即通过很少的中间节点，即可将一个节点与另一个节点联系起来。

然而，小世界网络模型忽略了节点度的差异性[1]。节点度指的是每个节点所拥有的边的数量。实际上，在许多现实网络中，存在少数高度连接的超级节点（Hubs），以及大量度较低的普通节点。为了更好地描述这种特性，无尺度网络模型被引入。

无尺度网络模型着重考虑了节点度的分布。通过研究真实网络的数据，研究者发现网络节点的度分布呈现出幂律分布的特点，即存在少量高度连接的超级节点和大量度较低的普通节点。这种幂律分布形成了无尺度网络的特性。

实际上，无尺度网络在实现极大创新的同时，同时也隐性地保有了很多 WS 小世界网络模型的特征，本次实验的目的就是探索出无尺度网络的小世界特性，主要关注其两项参数：顶点的平均聚类系数和两两之间的平均最短距离。

## 2 网络再建模 **（仅进行核心功能讲解，完整代码在附录中）**

网络的建模基于 BA 的 Scale-Free Networks 论文 [1][2]。选择使用 C++ 进行建模 [5][6]。

## 2.1 核心结构设计

### 2.1.1 顶点结构设计

```
struct vertex {
    vector<int> o_idx; // 出边索引
    vector<int> i_idx; // 入边索引
    int idx;           // 顶点编号
    int degree;        // 顶点度数

    // 构造函数，用于初始化顶点的出边索引和度数
    vertex(int edges = 0) : degree(0) {
        o_idx.resize(edges, 0); // 根据传入的 edges 参数初始化出边索引向量
    }
};
```

1. 定义了一个结构体 **vertex**，用于表示图中的顶点。
2. **vector<int> o_idx** 定义了一个名为 **o_idx** 的成员变量，类型为 **vector<int>**，用于存储顶点的出边索引。
3. **vector<int> i_idx** 定义了一个名为 **i_idx** 的成员变量，类型为 **vector<int>**，用于存储顶点的入边索引。
4. **int idx** 定义了一个名为 **idx** 的成员变量，类型为 **int**，用于存储顶点的下标编号。
5. **int degree** 定义了一个名为 **degree** 的成员变量，类型为 **int**，用于存储顶点的度数。
6. 构造函数：定义了一个构造函数 **vertex**，用于初始化顶点的出边索引和度数。
   1）构造函数接受一个默认参数 **edges**，用于指定顶点的边数。
   2）在构造函数体内，首先将顶点的度数 **degree** 初始化为 0。
   3）然后通过调用 **resize** 函数，将出边索引向量 **o_idx** 的大小设置为 **edges**，并且初始值为 0。

### 2.1.2 无尺度网络类设计

```
class SFN {
private:
    default_random_engine engine; // 随机数生成器
    uniform_real_distribution<double> prob_distribution; // 均匀分布随机数生成器
    uniform_int_distribution<int> index_distribution; // 均匀分布整数随机数生成器

    vector<vertex> graph; // 存储图的顶点
```

```
 8.
 9. public:
10.     SFN(int link_edges = 0, int init_vertexes = 0, int num_vertexes = 0);
11.     void __init_mem_space__();
12.     void __generation__();
13.     void __link_vertex__(int a, int b, int &cnt);
14.
15.     void floyd(vector<int> &dis);
16.     int find(int target, int idx);
17.     double coefficient();
18.
19.     void save_distribution();
20.
21.     double linking_prob();
22.     double be_linked_prob(const vertex& vx);
23.     int random_index();
24.};
```

1. 在私有部分（private）：

   1）**default_random_engine engine**：默认的随机数生成器，用于生成随机数。

   2）**uniform_real_distribution<double> prob_distribution**：均匀分布随机数生成器，用于生成均匀分布的随机实数。

   3）**uniform_int_distribution<int> index_distribution**：均匀分布整数随机数生成器，用于生成均匀分布的随机整数。

   4）**vector<vertex> graph**：存储图的顶点的向量容器，用于存储图中的顶点信息。

2. 在公有部分（public）：

   1）**SFN(int link_edges = 0, int init_vertexes = 0, int num_vertexes = 0)**：构造函数，用于初始化 SFN 类的实例，可以指定连接边数、初始顶点数和顶点总数的参数。

   2）**void __init_mem_space__( )**：初始化内存空间的方法，用于初始化内存空间。

   3）**void __generation__( )**：生成方法，用于执行生成操作。

   4）**void __link_vertex__(int a, int b, int &cnt)**：连接顶点的方法，接受两个顶点和一个计数器作为参数。

   5）**void floyd(vector<int> &dis)**：Floyd 算法，用于计算最短路径。

   6）**int find(int target, int idx)**：查找方法，用于查找特定目标。

   7）**double coefficient( )**：系数计算方法，用于计算系数。

   8）**void save_distribution( )**：保存分布的方法，用于保存分布信息。

9）**double linking_prob( )**：连接概率方法，用于计算连接概率。

10）**double be_linked_prob(const vertex& vx)**：被连接概率方法，接受一个顶点参数，用于计算被连接概率。

11）**int random_index( )**：随机索引方法，用于生成随机索引。

## 2.2 核心函数解析

### 2.2.1 构造函数

```
1. // 构造函数
2. SFN::SFN(int link_edges, int init_vertexes, int num_vertexes):
engine(chrono::steady_clock::now().time_since_epoch().count()) {
3.     edge = link_edges; // 边数
4.     init = scale = init_vertexes; // 初始顶点数
5.     total = num_vertexes; // 总顶点数
6.     __init_mem_space__(); // 初始化存储空间
7.     __generation__(); // 生成无标度网络
8. }
```

1. **SFN::SFN(int link_edges, int init_vertexes, int num_vertexes): engine(chrono::steady_clock::now().time_since_epoch().count())**: 定义了 **SFN** 类的构造函数，函数名为 **SFN**，参数列表包括 **link_edges**、**init_vertexes** 和 **num_vertexes**。在初始化列表中，使用 **chrono** 库获取当前时间的时间戳作为随机数引擎 **engine** 的种子。
2. **edge = link_edges** 将参数 **link_edges** 的值赋给类成员变量 **edge**，表示边的数量。**init = scale = init_vertexes** 将参数 **init_vertexes** 的值同时赋给 **init** 和 **scale** 两个类成员变量，表示初始顶点数。**total = num_vertexes** 将参数 **num_vertexes** 的值赋给类成员变量 **total**，表示总顶点数。
3. 调用类的私有成员函数**__init_mem_space__( )**，用于初始化存储空间。
4. 调用类的私有成员函数**__generation__( )**，用于生成无标度网络。

### 2.2.2 初始化函数、生成函数、连接函数

```
1. // 初始化存储空间
 2. void SFN::__init_mem_space__() {
 3.     cout << "Initialization Started" << endl;
 4.     for (int i = 0; i < init; ++i) {
```

```
        vertex vx;
        vx.idx = i;     // 记录顶点对应下标值
        graph.push_back(vx);    // 将顶点连入图的存储空间中
    }
    cout << "Initialization Finished" << endl;
}

// 生成无标度网络
void SFN::__generation__() {
    cout << "Generation Started" << endl;
    #pragma omp parallel for num_threads(2)
    for (int i = init; i < total; ++i) {
        int cnt = 0;
        unordered_map<int, int> hash;
        #pragma omp parallel for num_threads(4)
        while (cnt < edge) {
            int idx = random_index();   // 随机生成下标值
            if (hash.count(idx) != 0) continue;
            double rand_prob = linking_prob();  // 随机生成一个浮点数判断连接与否
            double linked_prob = be_linked_prob(graph[idx]);    // 生成对应顶点的被连接概率
            if (total_degree < 10) {    // 前十个度数的时候为了防止网络失活进入死循环，直接可以连接
                __link_vertex__(i, idx, cnt);   // 连接函数
                hash[idx] = 0;  // 给哈希表对应下标位置赋一个值，用以保证不会重链接
            } else if (rand_prob < linked_prob) {
                __link_vertex__(i, idx, cnt);
                hash[idx] = 0;  // 给哈希表对应下标位置赋一个值，用以保证不会重链接
            }
        }
        // 打印百分比进度条
        int progress = ((i - init + 1) * 100) / (total - init);
        cout << "Progress: [" << string(progress / 2, '=') << ">" << string(50 - progress /
2, ' ') << "] " << progress << "%" << "\r";
        cout.flush();
    }
    cout << endl;
    cout << "Generation Finished" << endl;
}

// 连接顶点
void SFN::__link_vertex__(int a, int b, int &cnt) {
    // 构造一个有 edge 个连出边的顶点
    vertex vx(edge);
    // vx 的连出边向量加上 b，graph[b]的连入边向量加上 a
    vx.o_idx.push_back(b);
```

```
48.     graph[b].i_idx.push_back(a);
49.     // vx 度加 1，graph[b]的度加 1
50.     vx.degree++;
51.     graph[b].degree++;
52.     // 更新 cnt 的值，用于判断内层循环退出与否，度数每次连边加 2
53.     total_degree += 2;
54.     cnt++;
55.     // 当 cnt 等于 edge 时，代表连边完成，将顶点放入图结构中，并增大 scale
56.     if(cnt == edge) {
57.         vx.idx = a;     // 记录顶点在图存储空间中的下标值
58.         graph.push_back(vx);    // 存储进入
59.         scale++;        // 增添网络规模
60.     }
61. }
```

**1. SFN::__init_mem_space__( )：**

1）**for (int i = 0; i < init; ++i)** ：通过 for 循环，对于每一个 i 从 0 到 init-1，执行以下操作：

2）**vertex vx**：声明一个名为 vx 的变量，类型为 vertex，这里假设 vertex 是一个结构体或类，用来表示图中的顶点。

3）**vx.idx = i**：给 vx 的成员变量 idx 赋值为 i，记录顶点对应的下标值。

4）**graph.push_back(vx)**：将顶点 vx 连入图的存储空间中，这里假设 graph 是一个存储顶点的容器，使用 push_back 方法将 vx 加入容器。

**2. void SFN::__generation__()：**

1）**#pragma omp parallel for num_threads(2)**：使用 **OpenMP** 并行化指令，将循环并行化，指定使用 2 个线程并行执行。

2）**for (int i = init; i < total; ++i)**：循环遍历从 **init** 到 **total-1** 的索引 **i**，表示要生成的顶点的范围。

3）**int cnt = 0**：初始化计数器 **cnt**，用于记录已连接的边数。

4）**unordered_map<int, int> hash**：创建一个无序映射 **hash**，用于记录已连接的顶点，避免重复连接。

5）**#pragma omp parallel for num_threads(4)**：使用 **OpenMP** 并行化指令，将循环并行化，指定使用 4 个线程并行执行。

6）**while (cnt < edge)**：当已连接的边数小于指定的边数时，执行以下循环。

7）**int idx = random_index( )**：生成一个随机下标值 idx，用于选择要连接的顶点。

8）**if (hash.count(idx) != 0) continue**：判断生成的下标值是否已存在于 **hash** 中，如果是，则跳过继续下一次循环，避免重复连接。

9）**double rand_prob = linking_prob( )**：生成一个随机浮点数 **rand_prob**，用于判断是否连接顶点。

10）**double linked_prob = be_linked_prob(graph[idx])**：根据给定顶点的被连接概率计算得到 linked_prob，用于判断是否连接该顶点。

11）**if (total_degree < 10)**：如果总度数小于 10，表示图中只有少量顶点，为了避免网络失活进入死循环，直接连接顶点。

12）**__link_vertex__(i, idx, cnt)**：调用连接函数 **__link_vertex__()**，将当前顶点和选择的顶点作为参数进行连接，并更新计数器 **cnt**。

13）**hash[idx] = 0**：将已连接的顶点存入 **hash** 中，用作记录。

14）**(rand_prob < linked_prob)**：如果随机概率小于被连接概率，则连接顶点。

15）**__link_vertex__(i, idx, cnt)**：同样调用连接函数 **__link_vertex__()**，将当前顶点和 1）选择的顶点作为参数进行连接，并更新计数器 **cnt**。

12）**hash[idx] = 0**：同样将已连接的顶点存入 **hash** 中。

13）**int progress = ((i - init + 1) * 100) / (total - init)**：计算生成进度的百分比。

**3. void SFN::link_vertex(int a, int b, int &cnt)：**

1）**vertex vx(edge)**：创建了一个名为 **vx** 的对象，该对象属于 **vertex** 类，并传入了一个参数 **edge**。

2）**vx** 的连出边向量加上 **b**，**graph[b]**的连入边向量加上 **a**：将顶点 **vx** 的出边向量

中添加了顶点 **b**，同时将顶点 **b** 的入边向量中添加了顶点 **a**。

3）**vx.o_idx.push_back(b)**：将顶点 **b** 添加到顶点 **vx** 的出边向量中。

4）**graph[b].i_idx.push_back(a)**：将顶点 **a** 添加到顶点 **b** 的入边向量中。

5）**vx.degree++**：将顶点 **vx** 的度数增加 **1**。

6）**graph[b].degree++**：将顶点 **b** 的度数增加 **1**。

7）**total_degree += 2**：将总度数 **total_degree** 增加 **2**，用于判断内层循环是否退出。

8）**cnt++**：将计数器 **cnt** 增加 **1**。

9）**if(cnt == edge)**：如果计数器 **cnt** 等于 **edge**，即连边操作完成。

10）**vx.idx = a**：将顶点 **vx** 在图存储空间中的下标值设置为 **a**。

11）**graph.push_back(vx)**：将顶点 **vx** 存储到图结构中。

12）**scale++**：增加网络规模 **scale** 的值。

## 2.3验证新网络模型的无尺度特性

### 2.3.1 数据生成、读取和分析

**C++**进行度分布数据生成，代码如下，按照格式以度数为下标，下标对应元素值为属于该度数顶点数量，存入一个 **csv** 文件中

```
// 存储度分布
void SFN::save_distribution() {
    ofstream ofile("degree_distribution.csv");
    if(ofile.is_open() == false)
        return;
    int max = graph[0].degree;
    for(int i = 0; i < graph.size(); ++i)
        if(graph[i].degree > max)
            max = graph[i].degree;
    vector<int> degrees(max + 1, 0);
    for(int i = 0; i < graph.size(); ++i)
        degrees[graph[i].degree]++;

    for(int i = 0; i < degrees.size(); ++i)
```

```
        ofile << i << ", " << degrees[i] << endl;
    ofile.close();
}
```

随后利用 **python** 进行数据分析，调用 **pandas** 库和 **matplotlib** 中的 **pyplot** 库

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

df = pd.read_csv("output\degree_distribution.csv", header=None, names=['degree', 'num'])
df = df[df['num'] != 0]
df = np.log(df)
plt.scatter(x = df['degree'], y = df['num'], alpha = 0.7, color = "teal")
plt.grid()
plt.xlabel("Degree of Vertexes")
plt.ylabel("Number of Corresponding Vertex")
plt.show()
```

度分布图像如下：

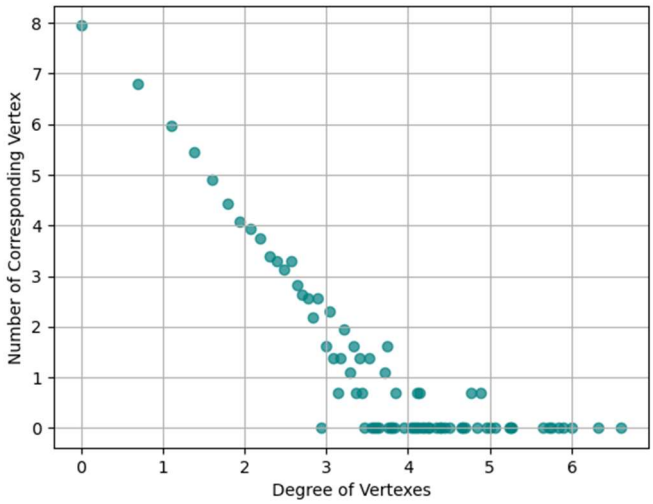


图 1.　5000 顶点网络度分布图像

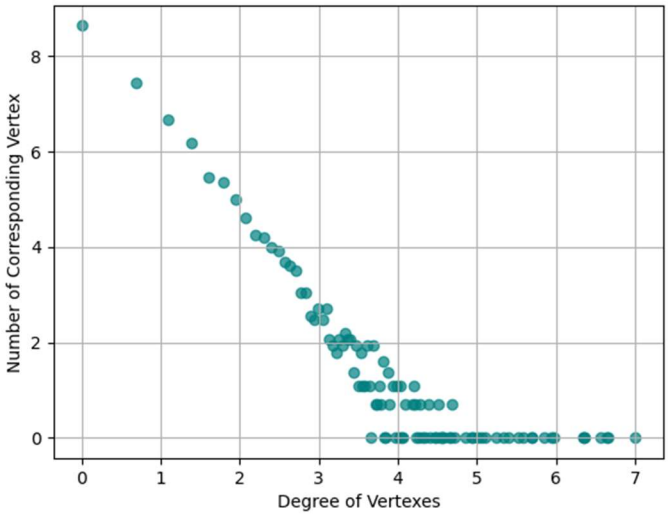


图 2.　10000 顶点网络度分布图像

可见，新建立的无尺度网络度分布符合幂律分布，具有显著的无尺度特性。

# 3　平均最短路径与聚类系数

## 3.1　算法选择

结合图的结构特性以及无尺度网络的结构特性，即无向图的特性，进行后续分析。在计算平均最短路径的时候，是任意两个节点之间的最短路径，这是二元最短路径问题，参考弗洛伊德算法部分特性进行解决，使用深度优先搜索算法作为核心方法。计算聚类系数时，找到邻居间仍为邻居的顶点数量，结合聚类系数计算公式[1]计算得到每个顶点的聚类系数。

### 3.1.1 弗洛伊德算法

弗洛伊德算法（Floyd-Warshall Algorithm）是一种用于解决所有点对最短路径问题的动态规划算法。该算法可以在有向图或无向图中使用，但不能包含负权边。

算法的基本思想是，对于图中任意两个顶点 i 和 j，以 k 作为中间节点，如果从 i 到 k 的路径比从 i 到 j 的路径短，则将 i 到 k 的路径加上 k 到 j 的路径得到一条从 i 到 j 的

[1] Ci = Ei / (ki * (ki - 1) / 2): 此处 Ei 和 ki * (ki - 1) / 2 分别为顶点 i 的 ki 个邻居顶点之间存在的边数（邻居直接也为邻居）和理论可存在的最大边数。

短路径。在执行过程中，通过不断更新每个节点之间的最短路径来逐步求解所有点对之间的最短路径。

弗洛伊德算法的时间复杂度为 O(n^3)，其中 n 为图的节点数。尽管它的时间复杂度较高，但该算法在需要求解所有点对之间的最短路径时非常有效。这也是我选择弗洛伊德算法的原因，思路清晰，问题适用。

### 3.1.2 深度优先搜索

深度优先搜索（Depth-First Search，DFS）是一种用于遍历或搜索图或树的算法。该算法通过深入到图的深处，并尽可能地探索每个分支，直到无法继续前进时才返回上一级。首先，选择一个起始节点作为当前节点，并将其标记为已访问。随后，访问当前节点，并对其进行相应的操作。再从当前节点出发，按照某种顺序选择一个未访问的相邻节点作为下一个当前节点。如果所有相邻节点都已被访问，则回溯到上一级节点，即返回上一个当前节点。再从当前节出发，递归重复操作。

深度优先搜索使用递归或栈来实现。当使用递归时，算法会自动维护一个隐式的栈，记录当前节点以及需要回溯的节点。当使用显式的栈时，则是模拟递归过程。

深度优先搜索算法在图和树的遍历查找中相当常用。

## 3.2 算法设计

```
1. // Floyd 算法计算最短距离
2. void SFN::floyd(vector<int> &dis) {
3.     int count = 0;
4.     #pragma omp parallel for num_threads(8) // 使用 OpenMP 并行化，使用 8 个线程并行计算
5.     for(int i = 0; i < graph.size(); ++i) { // 遍历所有节点
6.         for(int j = 0; j < graph.size() && j != i; ++j) { // 遍历除自身以外的所有节点
7.             int flag = 0;
8.             for(int k = 0; k < graph[i].o_idx.size(); ++k) { // 如果节点 i 直接连接节点 j，距离为
1
9.                 if(graph[i].o_idx[k] == j) {
10.                    dis.push_back(1);
11.                    flag = 1; // 标记已找到直接连接的情况
12.                    break;
13.                }
14.            }
15.            if(flag) continue; // 如果找到直接连接的情况，跳过后面的循环
```

```
            for(int k = 0; k < graph[i].o_idx.size(); ++k) {
                int num = find(j, graph[i].o_idx[k]); // 调用 find 函数找到节点 i 到节点 j 的最短距离
                if(num >= 0)
                    dis.push_back(num); // 将距离加入 dis 数组
            }
        }
        #pragma omp atomic // 使用原子操作保证 count 的线程安全
        count++;
        // 打印百分比进度条
        int progress = (count * 100) / graph.size();
        cout << "Progress: [" << string(progress / 2, '=') << ">" << string(50 - progress / 2, ' ') << "] " << progress << "%" << "\r";
        cout.flush(); // 将输出立即刷新到控制台
    }
    cout << endl; // 打印换行符，避免在进度条之后输出其他内容
}
```

1. **int count = 0**：定义一个整型变量 **count**，用于记录当前处理的节点数。

2. **#pragma omp parallel for num_threads(8)**：使用 **OpenMP** 并行化，将接下来的 **for** 循环并行化，在 8 个线程上进行计算。

3. **for(int i = 0; i < graph.size(); ++i)**：外层循环，遍历所有节点。

4. **for(int j = 0; j < graph.size() && j != i; ++j)**：内层循环，遍历除自身以外的所有节点。

5. **int flag = 0**：定义一个标志变量 **flag**，用于标记是否找到直接连接的情况。

6. **for(int k = 0; k < graph[i].o_idx.size(); ++k)**：内层循环，遍历节点 **i** 的所有出连节点。

7. **if(graph[i].o_idx[k] == j)**：判断节点 **i** 是否直接连接节点 **j**。

8. **dis.push_back(1)**：将距离 1 加入到 **dis** 数组中。

9. **flag = 1**：标记已找到直接连接的情况。

10. **break**：跳出内层循环。

11. **if(flag) continue**：如果找到直接连接的情况，跳过后面的循环。

12. **for(int k = 0; k < graph[i].o_idx.size(); ++k)**：内层循环，遍历节点 **i** 的所有出连节点。

13. **int num = find(j, graph[i].o_idx[k])**：调用 **find** 函数找到节点 **i** 到节点 **j** 的最短距离。

14. **if(num >= 0)**：判断找到的最短距离是否有效。

15. **dis.push_back(num)**：将最短距离加入 **dis** 数组。

16. **#pragma omp atomic**：使用原子操作保证 **count** 的线程安全。

17. **count++**：对 **count** 进行自增操作。

18. **int progress = (count * 100) / graph.size()**：计算进度百分比。

```
1. int SFN::find(int target, int idx) {
2.     if(target == idx) { // 如果目标节点等于当前节点，直接返回 1
3.         return 1;
4.     } else {
5.         vector<int> visited(graph.size(), 0); // 标记是否访问过某个节点
6.         vector<int> stack; // 使用栈来实现深度优先搜索
7.         vector<int> steps(graph.size(), 0); // 记录每个节点的步数
8.         stack.push_back(idx); // 将起始节点加入栈中
9.         steps[idx] = 1; // 起始节点的步数为 1
10.        while (!stack.empty()) { // 当栈不为空时循环
11.            int current = stack.back(); // 取出栈顶元素
12.            stack.pop_back(); // 弹出栈顶元素
13.            visited[current] = 1; // 标记该节点已访问过
14.            for (int i = 0; i < graph[current].o_idx.size(); ++i) { // 遍历当前节点的所有相邻节
点
15.                int next = graph[current].o_idx[i];
16.                if (next == target) { // 如果找到目标节点，返回目标节点的步数
17.                    return steps[current] + 1;
18.                }
19.                if (!visited[next]) { // 如果相邻节点未访问过，将其加入栈中，并记录其步数
20.                    stack.push_back(next);
21.                    steps[next] = steps[current] + 1;
22.                }
23.            }
24.        }
25.        return -1; // 如果未找到路径，返回-1
26.    }
27. }
```

1. **if(target == idx) { return 1; }**如果目标节点等于当前节点，直接返回 1，表示目标节点就是当前节点，步数为 1。

2. **else:** 否则，需要进行深度优先搜索来查找最短路径。

3. **vector<int> visited(graph.size(), 0):** 创建一个大小为图节点数量的 **visited** 数组，用于标记是否访问过某个节点。

4. **vector<int> stack:** 创建一个空的栈 **stack**，用于实现深度优先搜索。

5. **vector<int> steps(graph.size(), 0):** 创建一个大小为图节点数量的 **steps** 数组，用于记录每个节点的步数。

6. **stack.push_back(idx):** 将起始节点 idx 加入栈中。

7. **steps[idx] = 1:** 将起始节点的步数设置为 1。

8. **while (!stack.empty()) :** 进入一个 **while** 循环，当栈不为空时循环执行深度优先搜索。

9. **int current = stack.back():** 取出栈顶元素作为当前节点 **current**。

10. **stack.pop_back():** 弹出栈顶元素。

11. **visited[current] = 1:** 标记当前节点已经访问过。

12. **for (int i = 0; i < graph[current].o_idx.size(); ++i):** 遍历当前节点的所有相邻节点，进行深度优先搜索。

13. **int next = graph[current].o_idx[i]:** 获取当前节点相邻节点的索引。

14. **if (next == target) { return steps[current] + 1; }:** 如果找到目标节点，直接返回目标节点的步数。

15. **if (!visited[next]):** 如果相邻节点未访问过，将其加入栈中，并记录其步数。

16. **return -1:** 如果未找到路径，返回-1。

```
1. double SFN::coefficient() {
2.     double Csum = 0;
3.     for(int i = 0; i < graph.size(); ++i) {
4.         double Ci = 0;
5.         int ki = graph[i].o_idx.size() + graph[i].i_idx.size();
6.
7.         // 计算邻居节点之间仍未邻居节点的个数之和
8.         for(int j = 0; j < graph[i].o_idx.size(); ++j) {
```

```
            int neighbor1 = graph[i].o_idx[j];
            for(int k = j+1; k < graph[i].o_idx.size(); ++k) {
                int neighbor2 = graph[i].o_idx[k];
                if(find(neighbor2, neighbor1) == -1) {
                    Ci++;
                }
            }
        }
        Csum += Ci / (ki * (ki - 1) / 2); // 将 Ci 除以邻居节点对的总数，并累加到 Csum 中
    }
    return Csum / graph.size(); // 返回平均聚集系数
}
```

首先定义了一个名为 **coefficient** 的函数，返回类型为 **double**。 声明并初始化变量 **Csum**，用于保存累加的聚集系数总和。使用循环遍历图中的每个节点。定义并初始化变量 **Ci**，用于计算当前节点的聚集系数。计算当前节点的度，即邻居节点的数量。进入内部循环，遍历当前节点的出边邻居节点。在内部循环中，计算当前邻居节点之间仍未邻居节点的个数，即满足条件的节点对数目。获取当前邻居节点的索引。

进入第二个内部循环，从当前邻居节点的下一个节点开始遍历。获取第二个邻居节点的索引。如果第二个邻居节点不是第一个邻居节点的邻居，则满足条件，递增 **Ci**。将计算结果累加到 **Ci** 中。外部循环结束后，将 **Ci** 除以邻居节点对的总数，并累加到 **Csum** 中。返回值为平均聚集系数，即将 **Csum** 除以图中节点的数量。

## 3.3 实验数据及其分析

```
Link_edges_of_vertexes: 3
Init_vertexes: 3
Num_vertexes: 1000
Initialization Started
Initialization Finished
Generation Started
Progress: [==================================================>] 100%
Generation Finished

Calculating Least Distance
Progress: [==================================================>] 100%
Finished
Sum: 6484 Average: 6.484
Baseline: 9.96578

Calculating Cluster Coefficient
Cluster Coefficient: 0.358099

Elapsed time: 7 seconds.
```

图 3. 1000 顶点平均最短路径、聚类系数数据

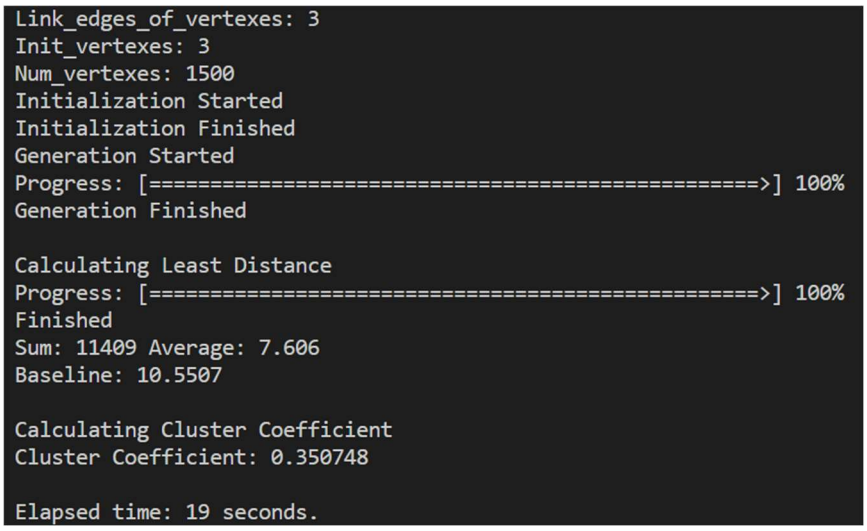


图 4. 1500 顶点平均最短路径、聚类系数数据

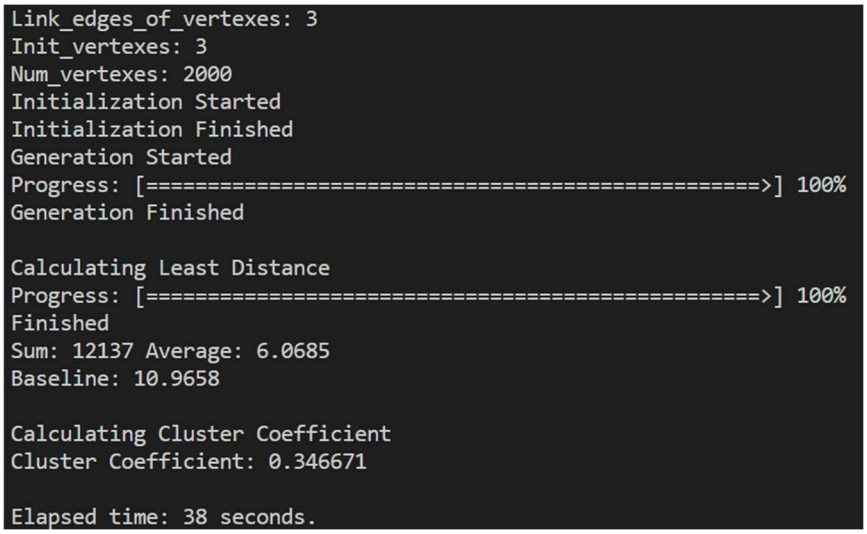


图 5. 2000 顶点平均最短路径、聚类系数数据

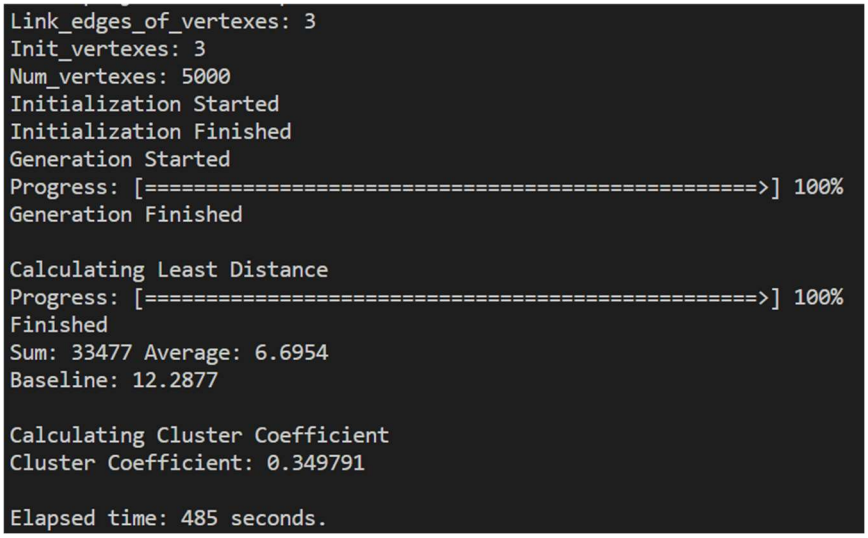


图 6. 5000 顶点平均最短路径、聚类系数数据

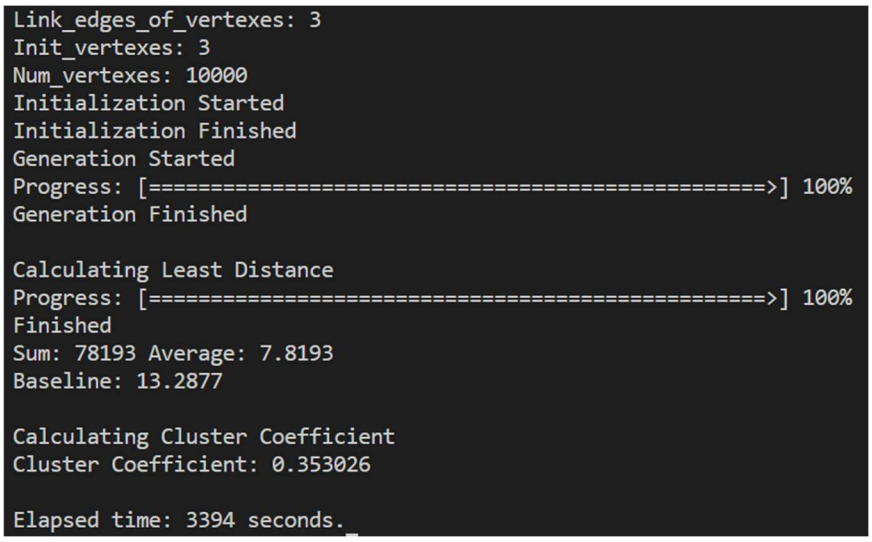


图 7. 10000 顶点平均最短路径、聚类系数数据

随即网络的理论最短距离是 ln（总定点数）/ ln（平均度数），而实际最短距离始终少于理论距离，说明无尺度网络并不属于纯随机网络，而其又大于 0，因此属于半随机网络，而小世界网络也属于这一类网络（最早 WS 便是给规则网络加以部分随机特性得出的小世界网络模型）。

而其聚类系数依旧在 0 到 1 之间，主要呈现在 0.34 到 0.36 之间，聚类系数较高，而一般的小世界网络的聚类系数也在 0.1 到 0.5 之间。

结合两个数据可以推断出，无尺度网络具有小世界网络的特性。

# 4 总结

本次实训，我在上学期的实训基础上，删掉了实现网络可视化的代码和很大部分的冗余代码，并优化了网络结构，真正构建了一个无向图结构，顶点使用两个向量作为出入边下标的存储结构，实现了顶点的双指向功能。

依靠这些全新的设计，我更快地实现了网络的构建，并加上 **OpenMP** 进行并行加速，实现了更快的网络生成速度。代码中给控制台加上了进度条，实现了更好的进度可视化。使用了更好的随机数生成器，而非古老的 **ctime** 中的随机数生成器，生成质量更佳的随机数，确保网络的质量。

学习了图的结构、图的各项最短路径算法和优先搜索算法，并着重学习了弗洛伊德算法和深度优先搜索算法，顺利实现了小世界特性的研究探索工作。

这次实训依旧有很多遗憾，比如超出内存范围的邻接矩阵让我不得不放弃矩阵的优雅解决方案，而且如果可以使用 **CUDA** 来实现无尺度网络的生成，定会更快更好，但是现在我的知识储备不足以支持我完成该项工作，对系统结构的理解的不足阻止了我的这种想法，希望后续的学习可以让我拥有这项能力。

正如无尺度网络的特性一样，足够多的边带给一个顶点在度数上的超群会给其带来越发多的边，富者愈富正如这个道理，但是，这个“富”并不一定是金钱上的富有，我们在学习的过程中，勇于探索新的知识，扩展知识面和知识深度（连接更多的顶点，得到更大的度数），打破之前自己的定式，挑战自己当前的能力极限（使得被连接概率变大），才能在短时间内得到长足的进步（获得更多的连接，从而获得更大的度数，度数增长从而呈现幂律分布），最后成为那个佼佼者，即网络中的 **Hub** 存在。

但同时，我们也不能忽略一些可以取得的巧劲，闷头干从来不是一个 **Hub** 应该做的事情，**Hub** 往往高瞻远瞩，他们本就邻居众多，他们也从不拘泥于自己身边的那些顶点，喜欢多去和外部的顶点接触，因为往往他们仅仅与下一个顶点相差 1、2 步罢了。多看看前方，也许你人生中的那个贵人就在你前方两个顶点处，只是你们中间隔着一个碍眼的顶点让你难以与之接触，只要愿意伸出手去探寻，成功或许就在一念之间（小世界特性）。

# 附录

**C++建模代码（核心功能讲解在正文中）**

```cpp
#include <fstream>
#include <vector>
#include <random>
#include <chrono>
#include <iostream>
#include <unordered_map>
#include <omp.h>
using namespace std;

static int total_degree = 0;
static int scale = 0;
static int edge = 0;
static int total = 0;
static int init = 0;

struct vertex {
    vector<int> o_idx; // 出边索引
    vector<int> i_idx; // 入边索引
    int idx;           // 顶点编号
    int degree;        // 顶点度数

    // 构造函数，用于初始化顶点的出边索引和度数
    vertex(int edges = 0) : degree(0) {
        o_idx.resize(edges, 0); // 根据传入的 edges 参数初始化出边索引向量
```

```
25.     }
26. };
27.
28. class SFN {
29. private:
30.     default_random_engine engine; // 随机数生成器
31.     uniform_real_distribution<double> prob_distribution; // 均匀分布随机数生成器
32.     uniform_int_distribution<int> index_distribution; // 均匀分布整数随机数生成器
33.
34.     vector<vertex> graph; // 存储图的顶点
35.
36. public:
37.     SFN(int link_edges = 0, int init_vertexes = 0, int num_vertexes = 0);
38.     void __init_mem_space__();
39.     void __generation__();
40.     void __link_vertex__(int a, int b, int &cnt);
41.
42.     void floyd(vector<int> &dis);
43.     int find(int target, int idx);
44.     double coefficient();
45.
46.     void save_distribution();
47.
48.     double linking_prob();
49.     double be_linked_prob(const vertex& vx);
50.     int random_index();
51. };
52.
53. // 构造函数
54. SFN::SFN(int link_edges, int init_vertexes, int num_vertexes):
engine(chrono::steady_clock::now().time_since_epoch().count()) {
55.     edge = link_edges; // 边数
56.     init = scale = init_vertexes; // 初始顶点数
57.     total = num_vertexes; // 总顶点数
58.     __init_mem_space__(); // 初始化存储空间
59.     __generation__(); // 生成无标度网络
60. }
61.
62. // 初始化存储空间
63. void SFN::__init_mem_space__() {
64.     cout << "Initialization Started" << endl;
65.     for (int i = 0; i < init; ++i) {
66.         vertex vx;
67.         vx.idx = i;     // 记录顶点对应下标值
```

```
68.         graph.push_back(vx);    // 将顶点连入图的存储空间中
69.     }
70.     cout << "Initialization Finished" << endl;
71. }
72.
73. // 生成无标度网络
74. void SFN::__generation__() {
75.     cout << "Generation Started" << endl;
76.     #pragma omp parallel for num_threads(2)
77.     for (int i = init; i < total; ++i) {
78.         int cnt = 0;
79.         unordered_map<int, int> hash;
80.         #pragma omp parallel for num_threads(4)
81.         while (cnt < edge) {
82.             int idx = random_index();   // 随机生成下标值
83.             if (hash.count(idx) != 0) continue;
84.             double rand_prob = linking_prob();  // 随机生成一个浮点数判断连接与否
85.             double linked_prob = be_linked_prob(graph[idx]);    // 生成对应顶点的被连接概率
86.             if (total_degree < 10) {   // 前十个度数的时候为了防止网络失活进入死循环，直接可以连接
87.                 __link_vertex__(i, idx, cnt);   // 连接函数
88.                 hash[idx] = 0;  // 给哈希表对应下标位置赋一个值，用以保证不会重链接
89.             } else if (rand_prob < linked_prob) {
90.                 __link_vertex__(i, idx, cnt);
91.                 hash[idx] = 0;  // 给哈希表对应下标位置赋一个值，用以保证不会重链接
92.             }
93.         }
94.         // 打印百分比进度条
95.         int progress = ((i - init + 1) * 100) / (total - init);
96.         cout << "Progress: [" << string(progress / 2, '=') << ">" << string(50 - progress /
2, ' ') << "] " << progress << "%" << "\r";
97.         cout.flush();
98.     }
99.     cout << endl;
100.     cout << "Generation Finished" << endl;
101. }
102.
103. // 连接顶点
104. void SFN::__link_vertex__(int a, int b, int &cnt) {
105.     // 构造一个有 edge 个连出边的顶点
106.     vertex vx(edge);
107.     // vx 的连出边向量加上 b，graph[b]的连入边向量加上 a
108.     vx.o_idx.push_back(b);
109.     graph[b].i_idx.push_back(a);
110.     // vx 度加 1，graph[b]的度加 1
```

```
111.     vx.degree++;
112.     graph[b].degree++;
113.     // 更新 cnt 的值，用于判断内层循环退出与否，度数每次连边加 2
114.     total_degree += 2;
115.     cnt++;
116.     // 当 cnt 等于 edge 时，代表连边完成，将顶点放入图结构中，并增大 scale
117.     if(cnt == edge) {
118.         vx.idx = a;     // 记录顶点在图存储空间中的下标值
119.         graph.push_back(vx);    // 存储进入
120.         scale++;        // 增添网络规模
121.     }
122. }
123. 
124. // Floyd 算法计算最短距离
125. void SFN::floyd(vector<int> &dis) {
126.     int count = 0;
127.     #pragma omp parallel for num_threads(8) // 使用 OpenMP 并行化，使用 8 个线程并行计算
128.     for(int i = 0; i < graph.size(); ++i) { // 遍历所有节点
129.         for(int j = 0; j < graph.size() && j != i; ++j) { // 遍历除自身以外的所有节点
130.             int flag = 0;
131.             for(int k = 0; k < graph[i].o_idx.size(); ++k) { // 如果节点 i 直接连接节点 j，距离
为 1
132.                 if(graph[i].o_idx[k] == j) {
133.                     dis.push_back(1);
134.                     flag = 1; // 标记已找到直接连接的情况
135.                     break;
136.                 }
137.             }
138.             if(flag) continue; // 如果找到直接连接的情况，跳过后面的循环
139.             for(int k = 0; k < graph[i].o_idx.size(); ++k) {
140.                 int num = find(j, graph[i].o_idx[k]); // 调用 find 函数找到节点 i 到节点 j 的最短
距离
141.                 if(num >= 0)
142.                     dis.push_back(num); // 将距离加入 dis 数组
143.             }
144.         }
145.         #pragma omp atomic // 使用原子操作保证 count 的线程安全
146.         count++;
147.         // 打印百分比进度条
148.         int progress = (count * 100) / graph.size();
149.         cout << "Progress: [" << string(progress / 2, '=') << ">" << string(50 - progress /
2, ' ') << "] " << progress << "%" << "\r";
150.         cout.flush(); // 将输出立即刷新到控制台
151.     }
```

```
152.     cout << endl; // 打印换行符，避免在进度条之后输出其他内容
153. }
154. 
155. int SFN::find(int target, int idx) {
156.     if(target == idx) { // 如果目标节点等于当前节点，直接返回 1
157.         return 1;
158.     } else {
159.         vector<int> visited(graph.size(), 0); // 标记是否访问过某个节点
160.         vector<int> stack; // 使用栈来实现深度优先搜索
161.         vector<int> steps(graph.size(), 0); // 记录每个节点的步数
162.         stack.push_back(idx); // 将起始节点加入栈中
163.         steps[idx] = 1; // 起始节点的步数为 1
164.         while (!stack.empty()) { // 当栈不为空时循环
165.             int current = stack.back(); // 取出栈顶元素
166.             stack.pop_back(); // 弹出栈顶元素
167.             visited[current] = 1; // 标记该节点已访问过
168.             for (int i = 0; i < graph[current].o_idx.size(); ++i) { // 遍历当前节点的所有相邻
节点
169.                 int next = graph[current].o_idx[i];
170.                 if (next == target) { // 如果找到目标节点，返回目标节点的步数
171.                     return steps[current] + 1;
172.                 }
173.                 if (!visited[next]) { // 如果相邻节点未访问过，将其加入栈中，并记录其步数
174.                     stack.push_back(next);
175.                     steps[next] = steps[current] + 1;
176.                 }
177.             }
178.         }
179.         return -1; // 如果未找到路径，返回-1
180.     }
181. }
182. 
183. double SFN::coefficient() {
184.     double Csum = 0;
185.     for(int i = 0; i < graph.size(); ++i) {
186.         double Ci = 0;
187.         int ki = graph[i].o_idx.size() + graph[i].i_idx.size();
188. 
189.         // 计算邻居节点之间仍未邻居节点的个数之和
190.         for(int j = 0; j < graph[i].o_idx.size(); ++j) {
191.             int neighbor1 = graph[i].o_idx[j];
192.             for(int k = j+1; k < graph[i].o_idx.size(); ++k) {
193.                 int neighbor2 = graph[i].o_idx[k];
194.                 if(find(neighbor2, neighbor1) == -1) {
```

```
                    Ci++;
                }
            }
        }
        Csum += Ci / (ki * (ki - 1) / 2); // 将 Ci 除以邻居节点对的总数，并累加到 Csum 中
    }
    return Csum / graph.size(); // 返回平均聚集系数
}

// 存储度分布
void SFN::save_distribution() {
    ofstream ofile("degree_distribution.csv");
    if(ofile.is_open() == false)
        return;
    int max = graph[0].degree;
    for(int i = 0; i < graph.size(); ++i)
        if(graph[i].degree > max)
            max = graph[i].degree;
    vector<int> degrees(max + 1, 0);
    for(int i = 0; i < graph.size(); ++i)
        degrees[graph[i].degree]++;

    for(int i = 0; i < degrees.size(); ++i)
        ofile << i << ", " << degrees[i] << endl;
    ofile.close();
}

// 生成连接概率
double SFN::linking_prob() {
    prob_distribution = uniform_real_distribution<double>(0.0, 1.0);
    return prob_distribution(engine);
}

// 生成被连接概率
double SFN::be_linked_prob(const vertex& vx) {
    return (double)vx.degree / total_degree;
}

// 随机生成下标值
int SFN::random_index() {
    index_distribution = uniform_int_distribution<int>(0, scale - 1);
    return index_distribution(engine);
}

```

```cpp
// 主函数
int main() {
    int m, m0, t;
    cout << "Link_edges_of_vertexes: "; cin >> m;
    cout << "Init_vertexes: ";          cin >> m0;
    cout << "Num_vertexes: ";           cin >> t;
    auto start_time = chrono::steady_clock::now();

    SFN sfn(m, m0, t);

    sfn.save_distribution(); // 存储度分布

    cout << endl << "Calculating Least Distance" << endl;
    vector<int> dis;
    sfn.floyd(dis); // 计算最短距离
    int sum = 0;
    for(int i = 0; i < dis.size(); ++i)
        sum += dis[i];
    cout << "Finished" << endl;
    cout << "Sum: " << sum << " Average: " << (double)sum / t << endl;
    cout << "Baseline: " << log(t) / log(2) << endl;

    cout << endl << "Calculating Cluster Coefficient" << endl;
    double coef = sfn.coefficient(); // 计算聚集系数
    cout << "Cluster Coefficient: " << coef << endl;

    auto end_time = chrono::steady_clock::now();
    auto elapsed_time = chrono::duration_cast<chrono::seconds>(end_time -
start_time).count();

    cout << endl << "Elapsed time: " << elapsed_time << " seconds." << endl;

    return 0;
}
```

Python 数据分析代码 （单个脚本，用于分析新建网络是否具有无尺度特性）

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

```

```
df = pd.read_csv("output\degree_distribution.csv", header=None, names=['degree', 'num'])
df = df[df['num'] != 0]
df = np.log(df)
plt.scatter(x = df['degree'], y = df['num'], alpha = 0.7, color = "teal")
plt.grid()
plt.xlabel("Degree of Vertexes")
plt.ylabel("Number of Corresponding Vertex")
plt.show()
```

# 参考文献


[1] Barabási A L, Albert R, Jeong H. Mean-field theory for scale-free random networks[J]. Physica A: Statistical Mechanics and its Applications, 1999, 272(1-2): 173-187.

[2] Barabási A L, Bonabeau E. Scale-free networks[J]. Scientific american, 2003, 288(5): 60-69.

[3] Watts D J, Strogatz S H. Collective dynamics of 'small-world'networks[J]. nature, 1998, 393(6684): 440-442.

[4] McKinney W. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython[M]. " O'Reilly Media, Inc.", 2012.

[5] Meyers S. Effective C++: 55 specific ways to improve your programs and designs[M]. Pearson Education, 2005.

[6] Stroustrup B. The C++ programming language[M]. Pearson Education, 2013.