2018 年 12 月

哈爾濱工業大學

大数据计算基础

题　　目：多源的特定属性的社区查找算法设计(64)

专　　业：大数据专业

学　　号：1160300610

姓　　名：李思睿

课程类别：必修

目录

1 问题描述

1.1 问题描述

题目编号：64

题目名称：大规模数据图上的多源的特定属性的社区查找算法设计

题目难度：【1.1】

问题描述：

在社交网络中，用户相当于每一个点，用户之间通过互相的关注关系构成了整个网络的结构，在这样的网络中，有的用户之间的连接较为紧密，有的用户之间的连接关系较为稀疏，在这样的网络中，连接较为紧密的部分可以被看成一个社区，其内部的节点之间有较为紧密的连接，而在两个社区间则相对连接较为稀疏，这便称为社区结构。网络图中的社区结构反映了网络中的个体行为的局部性特征以及其相互之间的关联关系，研究网络中的社区对理解整个网络的结构和功能起到至关重要的作用，并且可帮助我们分析及预测整个网络各元素间的交互关系。由于社区结构的重要性，人们围绕它开展了许多研究工作，特定属性的社区查找便是其中之一。给定网络图 G（顶点带有属性信息）、顶点集合 QV 和属性集合 QA，特定属性的社区查找的目标是挖掘出网络图中包含 QV 且具有 QA 属性特征的社区结构。特定属性的社区查找能很好地处理用户的个性化需求，帮助用户找寻所需的社区结构。不过，需要注意的是，若用户同时想查找多个社区结构或多个用户同时想查找各自所需的社区结构，面对这种多源的特定属性的社区查找问题，现有的特定属性的社区查找算法只能一个个按顺序处理，这种串行处理方式很费时并不实用，因此，基于分布式的多源的特定属性的社区查找算法研究就显得非常迫切和重要。

(1)算法设计部分：请设计一个分布式的多源的特定属性的社区查找算法。

(2)实验部分：请使用 Hadoop/Spark 系统实现该算法，并使用实验数据：SNAP（Stanford Network Analysis Project）中的 Orkut 大规模数据图，测试算法性能，获取如下信息：

社区发现结果的准确性（将实验结果与真实社区结构进行比较，使用 F-Score 测量指标对算法准确性进行评估）

算法运行时间其中 Orkut 数据集链接：http://snap.stanford.edu/data/

1.2 问题的背景/意义

问题背景：

复杂网络是复杂系统的抽象，显示中许多复杂系统都可以用复杂网络的相关特性进行描述和分析。图，网络中的节点表示系统中的个体，边表示个体之间的关系。如：社会关系网络就是其中的典型网络图之一，对复杂网络的研究一直是许多领域的研究热点，其中社区结构是复杂网络中的一个普遍特征，整个网络是由许多个社区组成的。“社会网络”这一概念的兴起，源于其对社会互动的恰当描述。如果将咖啡馆里的人、一起工作的同事或者在互联网上互动的人认为是一个有边界社会群体，就会错误地认为他们是互相认识的，而对共同群体有归属感。然而事实上人们实在不断地进入和退出社会网络，而这种社会网络又具有复杂的结构。在很多现实世界网络中，都包含社区结构。发现网络社区是网络科学的根本问题，近年来引起了广泛的关注。

另一个相关但不相同的问题是社区查找，其目标是找到包含查询节点的最可能的社区。社区发现和社区查找最主要的区别就是社区发现问题通过优化一些预定义的标准来识

别网络中的所有社区，而社区查找是社区发现问题的依赖于查询的变体，其目标为挖掘出网络图中包含 QV 且具有 QA 属性特征的社区结构，也就是找到一个包含查询节点的密集子图问题。

基于上面的两个情况的叙述上，就出现了若用户同时想查找多个社区结构或多个用户同时想查找各自所需的社区结构，面对这种多源的特定属性的社区查找问题，现有的特定属性的社区查找算法只能一个个按顺序处理，这种串行处理方式很费时并不实用，因此，基于分布式的多源的特定属性的社区查找算法研究就显得非常迫切和重要。因此，就有了现在的大规模数据图上的多源的特定属性的社区查找算法。

研究意义：

网络图中的社区结构反映了网络中的个体行为的局部性特征以及其相互之间的关联关系，研究网络中的社区对理解整个网络的结构和功能起到至关重要的作用，并且可帮助我们分析及预测整个网络各元素间的交互关系。

社区发现以及社区查找，可以更准确的定位社会群体，以便用于广告的精准投放，推荐商品，推荐好友等等；并且还可以在学术网络中一句指定研究主题寻找权威作者、在商务网络中个针对特定产品查找营销群体等，还可以在复杂的学术网络中划分出关系紧密的作者群体，在多种人际关系中分析出志同道合的朋友圈等。

1.3 问题的技术难点

难点一：

首先在由于要做的是多源的特定属性的社区查找，因此采用先找到所有的社区结构，在进行查找。因此就出现了如何高效地进行社区发现的问题。由于在大规模的图结构中，社区的分布可能没有任何的规则，并且社区之间还有可能发生一定程度上的重叠现象，如何处理这些棘手的问题都是设计社区发现算法时的难点。

难点二：

在进行完社区发现之后，另一项主要内容就是如何进行社区查找，找到满足要求的包含特定点的社区结构，并且还要满足进行多源特定属性查找的特点，即满足不同用户同时查找满足他们查找需求的社区结构。

难点三：

以上的两个主要问题以及所要设计的算法都要实现所谓的分布式处理，要将他们放到现有的大数据分布式处理系统中进行运行，因此需要设计出满足分布式要求的基于大数据的社区发现和社区查找算法，这也是此问题的一大难点之一。

2 基于的系统/算法

2.1 系统架构/算法原理

系统架构：

(本节简单描述所使用的系统架构，详细的系统架构介绍见下面小节)

本实验采用支持数据密集型分布式应用的软件框架 Apache Hadoop，通过伪分布式来实现分布式并行运行，并且利用其上的 MapReduce 分布式计算框架来实现并行编程，将程序放到集群中的任一节点上进行执行，其次利用了 Hadoop 所提供的分布式文件系统来存储计算节点的数据，将程序的输入文件以及程序的输出文件都放到 HDFS 上进行存储。

算法原理：

（本节只简单描述算法的基本原理，详细的并行 map-reduce 算法设计见下节）

①首先采用了两种算法进行对大规模图的**社区发现**工作。

第一个算法是采用并行的基于密度的社区发现扫描算法 **PSCAN**，其基本算法原理为通过每次扫描图中一条边上的两个端点，并通过给定的阈值来判断这两个点之间的相似度是否满足阈值要求，以此来判断是否属于同一个社区当中（若属于则保留当前边，若不属于则删除当前边）。因此最后通过边相连的各个顶点就属于一个社区。（详细的并行算法设计见下文）

第二个算法是采用并行的基于派系过滤的社区发现算法 **CPM**，其基本原理为首先通过寻找网络中的极大完全子图，然后利用这些完全子图来寻找 K-派系的连通子图（即 K-派系社区），不同的 K 值对应不同的社区结构。找到所有的 K-派系社区之后，就可以判断每两个 K-派系社区指向共享的节点的数目时候大于等于 K-1，以此来得到 K-派系连接社区。（详细的并行算法设计见下文）。

②其次采用了一种算法来进行社区发现之后的**社区查找**工作。

社区查找算法是并行的基于倒排索引的多源社区查找算法 **IICS**，其基本原理为首先通过已经划分好了的社区结构，为社区中的每个节点都建立一个顶点到其所在社区的倒排索引，然后通过遍历索要查询的顶点集中的顶点来确定他们的公共社区，公共社区即为最后的目标社区。

2.2 使用这个系统/算法的原因

使用 hadoop 系统架构的原因：

首先问题要求是基于大规模数据的图处理问题，因此自然要使用分布式架构系统来解决问题提高问题的处理效率，其次在问题描述中要求实现用户同时想查找多个社区结构或多个用户同时想查找各自所需的社区结构，因此面对这种多源的特定属性的社区查找问题如果一个一个按需串行处理很费时并且不实用，所以选用 hadoop 这种分布式的系统架构是非常必要的。

设计使用 SCAN 算法的原因：

首先明确 PSCAN 算法的特点：

①PSCAN 算法是基于密度的扫描的线性复杂度算法，需要将图中的所有边都要遍历一遍来确定一条边的两个端点是否满足要求属于同一个社区。②除此之外，PSCAN 算法还有一个重要特点，就是其社区发现的结果的各个社区之间是不存在社区重叠的，即每一个点只能属于一个社区。③而且，PSCAN 算法需要选用多个阈值来确定聚类效果最好的阈值设定。

综上所述，设计 PSCAN 算法的原因在于其在小规模的图结构中常常表现的非常出色，因为小规模图中的边数相对较少，程序的运行时间较短，所以可以接受进行阈值的多次设定来取最佳值；其次，小规模的社区结构图中常常不会出现重叠社区的情况，因此 PSCAN 算法的聚类性能较好。（用于对比算法的性能）

设计使用 CPM 算法的原因：

首先明确 CPM 算法的特点：

①CPM 算法是基于派系过滤的社区发现算法，它并不需要遍历所有的边结构。②CPM 算法中派系之间的连接是可以重叠的，因此聚类的结果中各个社区之间可能存在重叠现象。③CPM 算法是基于完全子图的选取的。

综上所述，使用 CPM 算法的原因在于它更实用大规模的图结构，因为大规模的图结构当中通常是完全子图比较多的网络，即密集网络。CPM 的效率取决于寻找所有极大完全子图的效率，尽管寻找所有极大完全子图是 NP 完全问题，但在真实网络中是非常快的。

设计使用 IICS 算法的原因：

IICS 算法是基于倒排索引的对已有社区进行查找的算法，算法目标明确且易于实现，这一点在并行处理多个用户的查询时尤为重要，通过建立倒排索引，来使得不同用户查询

的不同社区之间的并行处理不会交叉影响，这是使得算法能应用于并行处理程序的重要条件。

2.3 如何使用此系统/算法

使用 hadoop 分布式系统架构：

（本节只阐述 hadoop 系统架构在本实验中的使用，其搭建过程和细节见下面小节）

对于 MapReduce 计算变成框架的使用，通过编写系统架构所提供的系统框架，只需要实现 map-reduce 架构下的 map 函数和 reduce 函数即可，并且此架构对程序员隐藏了系统级的运行细节，我们只需要处理算法的逻辑实现即可，而不必考虑系统执行细节的问题。

对于 HDFS 分布式文件系统的使用，通过一些常用的 HDFS 的 shell 命令来完成一些列对文件的下载、上传、移动、删除等常用操作，并且还可以利用 HDFS 的 web 界面来进行对分布式文件系统上的文件的操作。

使用 SCAN 算法、CPM 算法、IICS 算法：

通过设计算法，实现算法的重要函数，以及调用编写的处理输入文件函数，来使用这三个算法，并以此来解决实际问题。

3 系统设计/算法设计

3.1 系统架构/算法原理的描述

系统架构的描述：

采用 Hadoop 系统架构（如图 1）进行实现，并且利用其中的 **MapReduce** 分布式计算框架进行分布式编程；使用 **HDFS** 作为分布式文件系统来进行数据的存放以及读取。

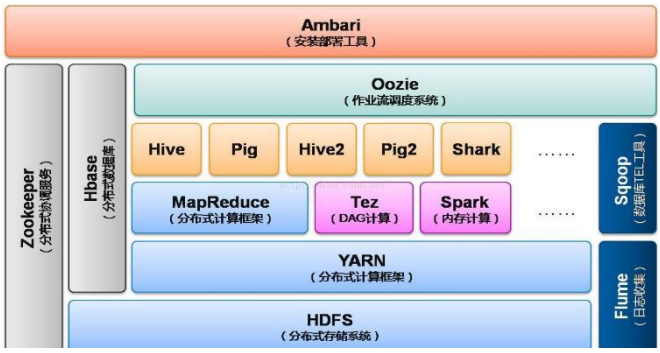


Figure1（Hadoop2.0 的架构图）

①HDFS 分布式文件系统（如图 2）主要由这几部分构成，分别是：NameNode、Secondary NameNode、DataNode、Client。

(1)NameNode 为主 Master，只有一个，负责管理 HDFS 的名称空间以及数据块的映射信息，配置副本策略，处理客户端的读写请求；

(2)Secondary NameNode 为 NameNode 的热备份，定期合并 fsimage 和 fsedits，并将其推送给 NameNode；当活跃的主节点出现故障时，快速切换为新的活跃的主节点。

(3)DataNode 有多个，负责存储实际的数据块，并且执行数据块的读和写。

(4)Client 负责与 NameNode 进行交互，获取文件位置信息；与 DataNode 进行交互，读取或者写入数据；管理 HDFS 以及访问 HDFS。

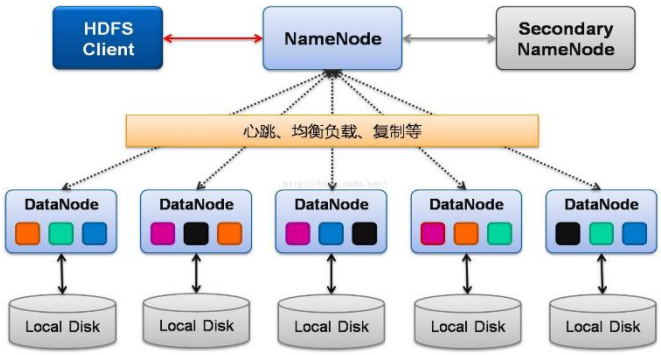


Figure2（HDFS 的架构图）

②Hadoop 的 MapReduce 采用 Master/Slave（M/S）架构（如图 3），主要包括的组件有：Client、JobTracker、TaskTracker 以及 Task。

(1)Client，用户编写的 MapReduce 程序通过 Client 提交到 JobTracker 端；同时，用户可通过 Client 提供的一些接口查看作业运行状态。在 Hadoop 内部用“作业”（Job）来表示 MapReduce 程序。一个 MapReduce 程序可对应若干作业，而每个作业会被分解成若干个 Map/Reduce 任务（Task）。

(2)JobTracker 主要负责资源监控和作业调度。JobTracker 监控所有 TaskTracker 与作业的健康状况，一旦发现失败情况后，其会将相应的任务转移到其他节点；同时 JobTracker 会跟踪任务的执行进度、资源使用量等，并将这些信息高速给任务调度器。

(3)TaskTracker 会周期性地通过 Heartbeat 将本节点上资源的使用情况和任务的运行进度汇报给 JobTracker，同时接受 JobTracker 发送过来的命令并执行相应的操作。TaskTracker 使用“slot”来划分本节点上的资源量。“slot”代表计算资源，如 CPU，内存等等。一个 Task 获取到一个 slot 到才有机会运行，而 Hadoop 调度器的作用就是将各个 TaskTracker 上的空闲 slot 分配给 Task 使用。

(4)Task 分为 Map Task 和 Reduce Task 两种，均由 Task Tracker 启动。分片 split 的数目决定 Map Task 的数目，每个分片会交由一个 Map Task 进行处理。

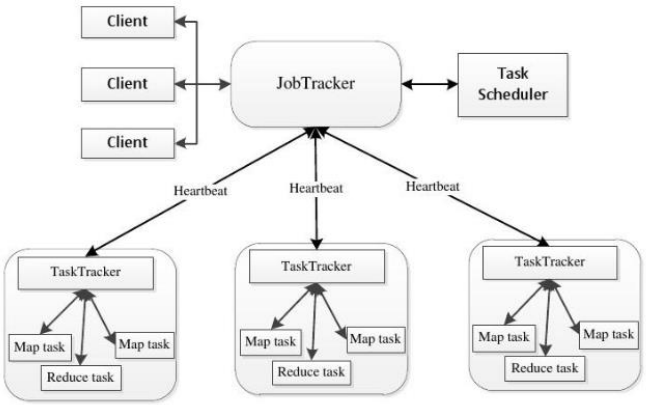


Figure3（MapReduce 架构）

算法原理及实现的详细描述：

①PSCAN 算法（非重叠社区发现算法）

PSCAN 算法原理：PSCAN 是一种采用并行的基于密度的社区发现扫描算法，并且另一个重要特点为它是一种非重叠的社区发现算法，这也就意味着算法的执行结果会产生一些列的没有公共节点的社区结构。其基本算法原理为通过每次扫描图中一条边上的两个端点，并通过给定的阈值来判断这两个点之间的相似度是否满足阈值要求，以此来判断是否属于同一个社区当中（若属于则保留当前边，若不属于则删除当前边）。因此最后通过边相连的各个顶点就属于一个社区。

当设计成并行程序的时：

Map/Reduce 函数的基本原理：每个 map 任务都会接收到一个节点以及此节点的邻居节点信息，map 函数将输出的 key 值设定为包含输入键值的其中一条边的信息，输出的 value 值设定为输入键值的邻居信息。根据上面的描述可知，Reduce 函数经过 mapper 的 shuffle 阶段，每个 reduce 任务的输入的 key 值为：一条边信息；输入的 value 值为：所有这条边的其中一个节点的邻居节点集合的一个迭代器。因此在 reduce 任务中，我们可以轻易地通过遍历迭代器中的集合信息来求得边的两个端点的相似性（在本实验中，节点相似性度量采用余弦相似性 Salton），因此就可以通过判断两个节点之间的相似性是否达到阈值要求来决定是否删除这两个节点之间的边。最终，所有有边相连的节点构成一个社区，自然地，社区之间不会出现重叠现象。

②CPM 算法（重叠社区发现算法）

CPM 算法原理：CPM 算法是采用派系过滤的社区发现算法，它与 PSCAN 算法不同，它是一种重叠社区的发现算法，因此算法的执行结果是一系列可能相互重叠甚至相互包含的社区结构。其基本原理为首先通过寻找网络中的极大完全子图，然后利用这些完全子图来寻找 K-派系的连通子图（即 K-派系社区），不同的 K 值对应不同的社区结构。找到所有的 K-派系社区之后，就可以判断每两个 K-派系社区指向共享的节点的数目时候大于等于 K-1，以此来得到 K-派系连接社区。

当设计成并行程序时，CPM 算法可以分为三个阶段来进行，三个阶段分别是：寻找极大完全子图、寻找 K 派系社区、寻找所有的 K 派系连接社区。

<1>寻找极大完全子图阶段：

Map/Reduce 函数的基本原理：Map 函数利用图中的邻接表的信息进行处理来输出键值对，其中输出键值对的 key 值为邻居节点的某一节点，value 值为该行文件的内容即一个节点的邻接表信息，然后经过 mapper 的 shuffle 处理之后，就得到了该目标节点的邻居以及邻居的邻居等信息。Reduce 函数输入为经过 mapper 的 shuffle 阶段输出结果，输入的键值对的 key 值为其中的某个节点，value 值为该节点的所有邻居节点，以及那些邻居节点的邻居节点集合的迭代器，因此 reduce 函数掌握了某个节点、其邻居节点以及邻居节点的邻居节点的所有信息，然后函数利用回溯法从目标节点出发，按一定的策略（在此选取在根节点的邻居中与其他邻居邻接最多的节点进行搜索的策略）进行搜索建立解空间树，递归搜索满足约束条件的解，解空间即为所有的极大完全子图。

<2>寻找 K 派系社区阶段（此阶段是 K 派系连接阶段的预处理阶段）：

Map/Reduce 函数的基本原理：Map 函数利用上一步已经找到的所有极大完全子图来进行划分任务，将任务变为键值对的形式，map 函数的划分规则为找到完全子图中的的最小值点作为之后进行合并的 key 值，value 值为对应的完全子图的信息。Reduce 函数将有 mapper 任务讲过 shuffle 过程传递过来的键值对进行处理，其中 key 值为某个最小值节点，value 值为包含这个最小值节点的所有完全子图。然后利用 K 派系连接原则，计算这些完全子图中满足公共节点数目大于最小集合数目-1 的集合，将其进行合并，（注意在此处并没有使用严格意义上的 K-1 条件来进行派系合并，因为此步骤是 K 派系连接阶段的预处理步骤，存在很多的极大完全子图点的数量很小，因此需要先将这一部分顶点进行处理，而对于那些已经存在了的很大的完全子图来说，在 min-1 的条件下很难进行合并，因此可以通过等待一轮，在第三步进行 K-1 派系合并）。最后输出所有合并后的派系。

<3>寻找所有的 K 派系连接社区阶段：

Map/Reduce 函数的基本原理：Map 函数读入已经进过第二步预处理后的所有 K 派系社区，派系数量已经大大减少，以此 Map 函数将每个派系中的每个点做为 key 值，派系内容作为 value 值进行 shuffle 过程。Reduce 函数对所有包含某个公共节点的派系进行合并检

查，判断是否其中某些社区满足公共节点数目大于等于为 K-1 的要求，若满足，则输出可以合并的信息。最后 Reduce 函数输出了所有可以进行合并的派系的序号。

③IICS 算法（多源的社区查找算法）

IICS 算法原理：其基本原理为首先通过已经划分好了的社区结构，为社区中的每个节点都建立一个顶点到其所在社区的倒排索引，然后通过遍历索要查询的顶点集中的顶点来确定他们的公共社区，公共社区即为最后的目标社区。

当设计成并行程序时，IICS 算法可以分为两个阶段来进行，两个阶段分别是：建立倒排索引的预处理阶段、社区查找阶段。

<1>建立倒排索引的预处理阶段

Map/Reduce 函数的基本原理：Map 函数的输入文件的每一行内容都是利用前面讲述的算法进行社区发现之后的社区信息，其中包括社区的 ID，以及社区内部的节点信息。Map 函数通过遍历社区内部的所有节点，每个节点到其社区内容的倒排索引。Reduce 函数之间将倒排索引的信息进行输出即可。

<2>社区查找阶段

Map/Reduce 函数的基本原理：Map 函数通过读取节点到社区的倒排索引信息来确定用户的请求，将一个请求信息作为 key 值进行输出，遍历请求节点集合通过倒排索引来找出其社区，作为 value 值进行输出。Reduce 函数在经过 mapper 的 shuffle 过程的合并之后，就得到了某个请求顶点集合中，所有顶点分别的应的包含此顶点的社区，通过求这些社区集合的交集，最终就得到了索要查询的社区。

3.2 系统架构框图/算法实现及伪代码

系统架构框图：

见上一小节中系统架构的描述（在此不再赘述）。

算法的详细实现及伪代码：

①PSCAN 算法实现及伪代码：

Map 函数的实现：map 函数的输入是经过预处理的图文件（图的邻接表），其中每个 map 函数只读取文件中的一行，通过":"进行分割来确定节点信息以及节点的邻居信息，然后遍历邻居集合，来求出所有邻接的边做为 key 值输出，然后让邻居集合作为 value 输出.

PSCAN_Mapper 1: 获取边信息和端点的邻居信息

Input: key值:文件内容的偏移量
value值:文件每一行的内容（一个节点node的邻接表）

Output: key值:一条边的信息
value值:这条边中某个端点的邻居信息

```
1  node ←按照格式分割value得到主节点;
2  neighbors[ ] ←按照格式分割value得到邻居集合;
3  for each neighbor in neighbors do
4      if node <neighbor then
5          edge ← (node , neighbor);
6      else
7          edge ← (neighbor , node);
8      end
9      Emit(edge,neighbors);
10 end
```

Figure4（PSCAN 算法的 Mapper 函数伪代码）

Reduce 函数的实现：通过 reduce 函数输入的 value 值（迭代器），来对这条边两个端点的邻居信息进行迭代，通过 Salton 相似性度量规则，通过遍历其中一个邻居集合的节点信，通过 contains()函数求出两个集合的公共节点数目 common，然后在调用 size()函数来求得两个集合的大小，最后通过公式：共同邻居数比上他们各自节点度之积的平方根，来求得相似性度量的指标，并用它来和预先设定的阈值进行比较，若大于阈值，则通过 context()函数输出这条边的信息，否则直接跳过执行。

PSCAN_Reducer 2: 基于相似度扫描，操作图结构

```
Input: key值:一条边的信息
       value值:此条边当中各个端点所包含邻居信息的迭代器
Output: key值:null
        value值:边信息
for each neighbors in value.next() do
    set1 ← neighbors;
end
for each neighbors in value.next() do
    set2 ← neighbors;
end
sim ← calSim(set1,set2);
if sim >阈值 then
    Emit(null,key);
end
/* 计算相似度函数 */
Function calSim (set1 , set2)
    for each i in set1 do
        if set2包含i then
            common+1;
        end
    end
    sim ← common/(set1.size * set2.size);
    return sim;
end
```

Figure5（PSCAN 算法的 Reducer 伪代码）

②CPM 算法实现及伪代码：

<1>寻找极大完全子图阶段：

Map 函数的实现：输入的文件为经过预处理后的图文件信息，每一行都是一个节点的邻接表信息，Map 函数通过遍历 node 节点的邻居集合，将其中的某个邻居点节点作为 key 值进行输出，node 节点的邻接表信息作为 value 值进行输出。

Clique_Mapper 1: 获取节点的邻居以及二跳邻居信息

```
Input: key值:文件内容的偏移量
       value值:文件每一行的内容（一个节点node的邻接表）
Output: key值:node邻居中的某个节点nodeKey
        value值:node的邻接表（邻居中不包含nodeKey）
node ←按照格式分割value得到主节点;
neighbors[ ] ←按照格式分割value得到邻居集合;
for each neighbor in neighbors[ ] do
    value.neighbors去掉neighbor节点;
    Emit(neighbor,value);
end
```

Figure6（CPM 算法第一步的 Mapper 伪代码）

Reduce 函数的实现：由前面讲述的原理可知，reduce 函数的输入的键值对的 key 值为其中的某个节点，value 值为该节点的所有邻居节点，以及那些邻居节点的邻居节点集合的迭代器。然后首先现将 key 值加入到 root 集合当中作为当前的根节点，然后获取其邻居集合以及邻居的邻居集合（二跳邻居集合），然后开始应用回溯法建立递归树，首先找出与其邻居集合中与其他邻居邻接最多的节点 next（以此来作为递归搜索的条件）。然后找到与根节点邻接但不与 next 邻接的节点集合 cand，然后开始将 next 节点加入到 root 集合当中，将 next 节点标记为已经访问过，开始递归调用算法，以 next 节点为根节点递归搜索。递归的终止条件为当 cand 集合为空时，即不存在与 root 邻接但不与 next 邻接的节点，此时正是极大完全子图所具有的的特征，因此找到了一个极大完全子图，递归终止。

```
Clique_Reducer 2: 寻找图中的所有极大完全子图
  Input: key值:node邻居中的某个节点nodeKey
         value值:node的邻接表（邻居中不包含nodeKey）
  Output: key值:null
          value值:图中的所有极大完全子图
1  root ← key;
2  Rootlist.add(key);
3  while value.hasnext do
4      neighbor ← 按照格式分割value得到nodeKey的邻居节点;
5      directNeighbors.add(neighbor);
6      secondNeighbors[ ] ← 按格式分割value得到nodeKey邻居节点的
         邻居节点集合;
7      dict ← 建立neighbor到secondNeighbors的映射;
8      cliques[ ] ← buildTree(root, directNeighbors, set);
9      for each clique in cliques do
10         Emit(null,clique);
11     end
12     Rootlist.remove(最后一个根节点);
13 end
14 /* 回溯法递归求解极大完全子图函数 */
15 Function buildTree (root, neighbors, hasVisited)
16     res[];
17     if neighbors is empty then
18         clique ← printClique();
19         if clique is not empty then
20             res[].add(clique);
21         end
22     else
23         /*找出与根节点的邻居节点链接最多的邻居节点*/
24         next ← maxNeighbor(neighbors);
25         cand[ ] ← neighbors/dict.get(next); /*求差集*/
26         while true do
27             cand.remove(next);
28             Rootlist.add(next);
29             if next is not visited then
30                 set[next]标记为visited;
31                 res[ ] ← buildTree(next, neighbors∩dict.get(next), set);
32             end
33             if cand is empty then
34                 Rootlist.remove(最后一个节点);
35                 break;
36             else
37                 next ← maxNeighbor(cand);
38                 Rootlist.remove(最后一个节点);
39             end
40         end
41         return res[ ];
42     end
43 end
```

Figure7（CPM 算法的第一步的 Reducer 伪代码）

<2>寻找 K 派系社区阶段（此阶段是 K 派系连接阶段的预处理阶段）：

Map 函数的实现：输入的文件为讲过步骤一的图中的所有极大完全子图，文件的每一行内容都一个完全子图的信息。Map 函数通过遍历完全子图中的节点，将值最小的节点作为 key 值进行输出，将包含最小节点的完全子图信息作为 value 值进行输出。

```
KMerge_Mapper 3: 获取派系中的最小点及其所在派系
  Input: key值:文件内容的偏移量
         value值:文件每一行的内容（图中的极大完全子图（派系））
  Output: key值:某个派系中的最小值节点nodeKey
          value值:nodeKey所在派系信息
1  clique ← value;
2  minNode ← min(clique) /*选出派系中的最小值节点*/
3  Emit(minNode,value);
```

Figure8（CPM 算法第二步的 Mapper 伪代码）

Reduce 函数的实现：输入的 key 值为某几个完全子图的中的公共最小点，value 值为包含 key 值的所有完全子图的迭代器。首先，通过遍历迭代器将所有完全子图都提取出来放入到集合当中。新建一个集合 mergedClique 来存放合并后的派系，通过遍历每个完全子图，通过判断完全子图是否可以与 mergedClique 集合中的派系通过集合间的 addAll()函数进行合并（合并条件为：若两个集合的公共元素数量大于等于二者最小集合元素数量-1，就进行合并）。最后，对所有合并的集合按次序进行输出，输出的 key 值为按自然数生成的派系 ID 号，value 值为合并后的 K 派系社区的内容。

```
KMerge_Reducer 4: 获取派系中的最小点及其所在派系
  Input: key值:某几个派系中的公共最小点nodeKey
         value值:包含nodeKey的所有派系
  Output: key值:合并后的派系的ID
          value值:合并后的派系
1  cliques[ ] ← value;
2  mergedClique[ ]; /*合并后的派系集合*/
3  for each clique in cliques do
4      /* 判断clique能否与mergedClique中某个派系进行合并 */
5      for each temp in mergedClique do
6          if contain(temp, clique) then
7              index = clique的序号
8          end
9      end
10     index ← contains(mergedClique, clique);
11     if index == -1 then
12         mergedClique[].add(clique);
13     else
14         mergedClique.get(index).addAll(clique);
15     end
16 end
17 for each clique in mergedClique do
18     ID++;
19     Emit(ID, clique);
20 end
21 /* 判断派系之间能否合并的函数 */
22 Function isAdjacent (set1, set2)
23     for each i in set1 do
24         if set2包含i then
25             common+1;
26         end
27     end
28     min ← min(set1.size(),set2.size());
29     if common≥min-1 then
30         return true;
31     else
32         return false;
33     end
34 end
```

Figure 9（CPM 算法的第二部 Reducer 伪代码）

<3>寻找所有的 K 派系连接社区阶段：

Map 函数的实现：输入的文件的每一行内容为派系的 ID 号和派系的顶点信息。Map 通过遍历一个派系中所包含的每个顶点，将其作为输出的 key 值，将输入的 value 作为输出的 value 值，进行输出。

```
KAllCombined_Mapper 5: 获取派系ID以及派系内容
  Input: key值:文件内容的偏移量
         value值:文件每一行的内容（CliqueID+派系的内容）
  Output: key值:派系中的某个节点nodeKey
          value值:（CliqueID+派系的内容）
1  clique ← 按照格式分割value得到社区信息;
2  for each node in clique do
3      Emit(node, value);
4  end
```

Figure10（CPM 算法第二步 Mapper 伪代码）

Reduce 函数的实现：输入的 key 值为某个顶点，value 值为包含这个顶点的所有派系的迭代器。首先通过迭代器，将所有派系放入集合当中，然后通过两层循环来遍历所有的派系社区，通过判断每两个派系之间是否满足合并要求来进行合并（合并条件为：若两个集合的公共元素个数大于等于 K-1 个），若满足要求，则之间将可以合并的两个派系的 ID 进行输出；若不满足，则直接跳过处理下一个集合。

```
KAllCombined_Reducer 6: 找出所有能合并的派系
  Input: key值:派系中的某个顶点nodeKey
         value值:包含这个顶点的所有派系（CliqueID+派系的内容）
  Output: key值:null
          value值:能合并的两个派系的ID
1  cliqueID ← 按照格式分割value得到社区编号信息;
2  clique ← 按照格式分割value得到社区信息;
3  for each clique1 in cliques do
4      for each clique2 in cliques do
5          if isAdjacent(clique1, clique2) then
6              id1 ← min(clique1ID, clique2ID);
7              id2 ← max(clique1ID, clique2ID);
8              Emit(null,(id1,id2));
9          end
10     end
11 end
```

```
12 /* 判断派系之间能否合并的函数 */
13 Function isAdjacent (set1, set2)
14     for each i in set1 do
15         if set2包含i then
16             common+1;
17         end
18     end
19     K ← 4;
20     if common≥K-1 then
21         return true;
22     else
23         return false;
24     end
25 end
```

Figure11（CPM 算法的第三部 Reducer 伪代码）

③IICS 算法实现及伪代码：

<1>建立倒排索引的预处理阶段

Map 函数的实现：输入的文件内容为社区发现之后的所有社区的信息，包括社区的 ID 以及社区内节点的信息。Map 函数简单的遍历一个社区内的所有节点，将每个节点作为 key 值，将社区的信息作为 value 值输出。

```
InvertedIndex_Mapper 1: 建立节点到社区的倒排索引
  Input: key值:文件内容的偏移量
         value值:文件每一行的内容（节点到社区的倒排索引）
  Output: key值:用户的请求标签
          value值:识别后的对应文件的一行内容（节点到社区的倒
          排索引）
1  community[ ] ←按照格式分割value得到邻居集合;
2  for each node in community do
3      Emit(node,value);
4  end
```

Figure12（IICS 算法第一步的 Mapper）

Reduce 函数的实现：经过 mapper 的 shuffle 过程之后，输入到 reducer 的 key 值为这区中的某个节点，value 值为包含该节点的所有社区的迭代器，reduce 函数不做任何处理，直接通过 context()函数将建立好的倒排索引信息写入文件。

<2>社区查找阶段

Map 函数的实现：输入的文件内容为每个节点到其所属社区的倒排索引内容，map 函数所属的类中，定义了一个用户输入的查询数据，其中不同用户之间的查询用换行“\n”分割，同一用户的不同查询用分号“;”分割，以此来模拟多源的社区查找。Map 函数首先通过换行符来对请求进行划分并作为外层循环来遍历每个用户的不通请求，然后在用分号进行划分，在内层循环来遍历一个用户的不同请求。在内层循环中判断如果用户请求的节点在当前的 reducer 任务执行的社区中，则将用户请求标志信息作为 key 值，对应的社区内容作为 value 值进行输出，若不存在直接跳过。

```
IICS_Mapper 2: 分割识别多个用户的多个查询集合
  Input: key值:文件内容的偏移量
         value值:文件每一行的内容（聚类后的社区）
  Output: key值:社区中的一个节点node
          value值:对应文件的一行内容（聚类后的社区）
1 node ←按照格式分割value得到node节点;
2 userInput[ ] ← 按照格式分割input得到每个用户的请求;
3 for each userRequest in userInput do
4     nodeInput[ ] ← 按照格式分割userRequest得到当前用户的不同
       请求集合;
5     for each temp in nodeInput do
6         if ContainTarget(node, temp) then
7             message ← 编写用户请求标签;
8             Emit(message, value);
9         end
10    end
11 end
12 Function ContainTarget (node , set)
13    for each temp in set do
14        if node == temp then
15            return true;
16        end
17    end
18    return false;
19 end
```

Figure13（IICS 算法第三步 Reducer 伪代码）

Reduce 函数的实现：经过 mapper 任务的 shuffle 阶段，reduce 函数的输入 key 值为用户的请求标签，value 值为包含用户请求的节点集合的社区集合的迭代器。首先 reduce 函数先迭代出一个社区，之后的每一次迭代的社区都与第一个社区求交集，最后产生的结果即为索要查询的社区，然后见用户的请求标签作为输出的 key 值，产生的目标社区作为 value 值进行输出。（若交集为空，则输入空集合）。

```
IICS_Reduce 3: 处理单个用户单个的查询请求
  Input: key值:用户的请求标签
         value值:包含用户请求节点集合的倒排索引的迭代器
  Output: key值:用户的请求标签
          value值:满足用户查询请求的社区ID
1 set1 ← 获取value中的第一个集合;
2 for value.hasnext() do
3     set1 ← set1∩value.next();
4 end
5 Emit(key, set1);
```

Figure14（IICS 算法第二部 Reducer 伪代码）

（备注：对于 PSCAN 算法和 CPM 算法最后的运行结果，虽然已经得到了社区结果，但需要进行一步社区格式的转化，以便测试性能指标，以及数据的预处理和抽样，具体实现较为简单，不属于核心算法，在此不做描述）。

4 实验流程

4.1 系统搭建

本次实验重要在 linux 系统下完成，通过在 linux 系统下搭建 hadoop 来完成实验。

首先通过在 Windows 操作系统下使用开源虚拟机软件 VirtualBox 安装 Ubuntu。Ubuntu 系统的安装和配置在此不详细叙述。

下面主要详细介绍 hadoop 的安装及配置过程:

①安装 Java 环境。

为了方便起见直接安装 OpenJDk1.7，通过命令 sudo apt-get install openjdk-7-jre openjdk-7-jdk。安装完成之后，需要为 java 配置相应的环境变量，通过 vim ~/.bashrc 中增加 export JAVA_HOME=JDK 安装路径，来为 java 配置环境变量，最后通过 source ~/.bashrc 来使得配置生效。

②安装 Hadoop2.8.5（本人选用）。

首先到官网下载相应版本的安装包，然后通过命令 sudo tar -zxf ~/下载/hadoop-2.6.0.tar.gz -C /usr/local 将安装包解压到用户的本地目录上面去，并且通过命令 sudo chown -R hadoop ./hadoop 来修改文件夹的权限。

③配置 Hadoop 的模式（本人配置了单机模式已经伪分布式模式）

Hadoop 的默认模式为非分布式模式（本地模式），无需进行其他配置即可运行，在非分布式即单 Java 进程，方便进行调试。

Hadoop 的伪分布式配置，Hadoop 可以在单节点上通过伪分布式的方式运行，Hadoop 进程以分离的 Java 进程来运行，节点既作为 NameNode 也作为 DataNode，同时，读取 HDFS 中的文件。Hadoop 的配置文件位于/hadoop/etc/hadoop/中，伪分布式需要修改 2 个配置文件 core-site.xml 和 hdfs-site.xml。Hadoop 的配置文件时 xml 格式，每个配置以声明 property 的 name 和 value 的方式来实现。

首先修改配置文件./etc/hadoop/core-site.xml，将当中添加配置信息:

```
<configuration>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>file:/usr/local/hadoop/tmp</value>
        <description>Abase for other temporary directories.
        </description>
    </property>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

（Figure15 配置文件 core-site.xml）

其中第一个配置项为配置 hadoop 的临时存放目录，不过若没有配置 hadoop.tmp.dir 参数，则默认使用的临时目录为 /tmp/hadoo-hadoop，而这个目录在重启时有可能被系统清理掉，导致必须重新执行 format 才行。（也可以换到本地的某个文件夹下，避免有可能因设备问题而使得数据丢失）第二个配置项为 HDFS 文件系统的默认存放路径，其中 9000 为 fileSystem 的默认端口号。

然后修改配置文件 hdfs-site.xml，将当中添加配置信息：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/local/hadoop/tmp/dfs/data</value>
    </property>
</configuration>
```

（Figure16 配置文件 hdfs-site.xml）

其中配置了 HDFS 中主节点 namenode 的文件存放路径以及数据节点 datanode 的文件存放路径。（※在本次实验过程中，突然发现 HDFS 的文件系统空间不足，主节点 namenode 进入了 safe 模式，不能对 HDFS 文件系统中的任何文件进行操作，已经了文件系统的扩充，扩充的主要步骤就是要对此配置文件进行修改，改变主节点和数据节点的存储路径，在此我又给虚拟机额外分配了一块 30G 磁盘，并修改了配置文件）

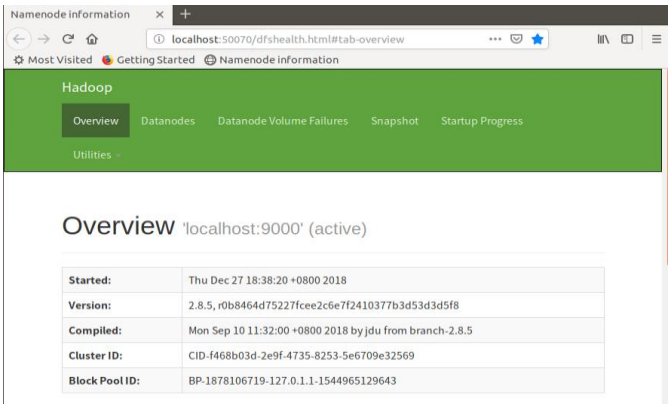
```
<property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/home/hadoop/sda3/hadoop/tmp/dfs/name</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>file:/home/hadoop/sda3/hadoop/tmp/dfs/data</value>
</property>
```

（Figure17 修改后的配置文件）

配置完成后，执行 NameNode 的格式化./bin/hdfs namenode –format。

接着就可以开启 NameNode 和 DataNode 守护进程./sbin/start-dfs.sh。

成功启动后，可以访问 Web 界面 http://localhost:50070 来查看 NameNode 和 DataNode 的信息，还可以在线查看 HDFS 中的文件。



（Figure18HDFS 的 web 界面）

④配置 eclipse 运行 map-reduce 程序

（在配置 eclipse 之前必须确保已经开启了 Hadoop）

首先需要先下载 hadoop-eclipse-plugin 的插件，并将其复制到 Eclipse 安装目录的 plugins 文件夹中，运行 eclipse –clean 来重启 eclipse 即可。

启动 eclipse 之后，就可以在左侧的 project Explorer 中看到 DFS Locations 了，然后选择 Windows 菜单下的 Preference，此时窗口会多出一个 Hadoop Map/Reduce 的选项，点击之后，选择 Hadoop 的安装目录即可。然后切换到 Map/Reduce 开发视图。

然后建立与 Hadoop 集群的连接，点击 Eclipse 软件右下角的 Map/Reduce Locations 面板，在面板中选择 New Hadoop Loction，在弹出来的 General 面板中，General 的设置要与 Hadoop 的设置一致，在伪分布式的情况下，填写 localhost，设置 fs.defaultFS 为 hdfs://localhost:9000，并且要将 DFS Master 的 Port 更改为 9000。Map/Reduce(V2) Master 的 port 用默认的即可。Advanced parameters 选项面板是对 Hadoop 参数进行配置，实际上就是填写 Hadoop 的配置项（/usr/local/hadoop/etc/hadoop 中的配置文件）。最后，点击 Finish，Map/Reduce Location 就创建好了。然后就可以创建 Map-Reduce 程序了。

最后，在运行 Map-Reduce 程序之前，还需要执行一项重要的操作，就是将 /usr/local/hadoop/etc/hadoop 中修改过的配置文件（core-site.xml、hdfs-site.xml），以及 log4j.property 复制到程序项目的源文件夹 src 中，其中前两个文件使用于让程序跑在伪分布式环境下的，最后一个 log4j 文件适用于记录程序中的输出日志。

至此，本次实验的系统就搭建完毕了！

4.2 算法设计过程

（本小节只讲述算法的设计过程，具体实现细节见前面有关小节）

首先，判断题目中要实现的是多源的特定属性的社区查找算法，这其中就包含了好几个过程，首先要做的就是对大规模的图进行聚类操作，因为对与多个用户的查找，每一次单独处理会很不切实际，因此要进行聚类找出所有的社区结构；

其次，在进行社区发现的过程中，首先想到的就是通过遍历图中所有的边，来判断边的两个端点的相似度，来依次确定图中的所有定点是否在一个社区当中，因此就使用的 PSCAN 基于扫描的社区发现算法，算法在设计成并行程序的时候，需要考虑的就是怎么将遍历计算所有节点见得相似度并行开来，因此想到的是在 mapper 任务中按照 key 值为边信息进行分类聚合，然后在 reducer 任务中计算节点 相似度。但不难发现，此算法的时间复杂度较高，而且需要设置合适的阈值才能得到较好的结果。

因此又使用了第二个算法，基于派系过滤的社区发现算法 CPM 算法，此算法很好的解决了 PSCAN 的瓶颈，它能够处理大规模的图聚类问题，并且不需要寻找合适的阈值进行搜索，并且能发现图中的重叠社区。在设置成并行程序时，基于算法的原理，需要先寻找到图中的所有极大完全子图，然后通过派系合并的条件并行地将所有派系进行合并，最终得到所有的派系社区。

最后，在对大规模图进行聚类之后，要做的就是给定多个用户的不同请求信息，对社区进行相应的查找。这里的重点是进行多源的查找，所以必须要放到分布式系统上来运行。对于这种情况，设计了一种基于倒排索引的社区查找算法，由于用户给定的请求是一个顶点集合，要求请求的社区必须包含顶点集合，因此建立了一个图中节点到社区的倒排索引，然后通过将不同用户的不通请求进行分割，来实现多源的社区查找。

4.3 算法/系统模块输入数据示例截图及描述

（数据集有多个，在此选用数据集 Football.txt（其中包含 115 个节点，1226 条边）来作为示例，所有的数据集在下面小节展示）

①算法 PSCAN 的输入数据（一部分）

输入的数据为一个 Football 的图信息，文件内容的每一行都为一个节点的邻接表项，通过冒号进行分割，冒号之前为目标节点，冒号之后为该目标节点所邻接的所有节点。↓

```
88:46,49,53,58,67,69,73,83,107,110,114,89
89:1,25,33,37,45,55,88,99,105,103,109
110:13,24,46,49,53,67,73,80,83,88,114
111:7,8,21,23,46,51,64,68,77,78,108
112:44,52,57,63,66,75,91,92,96,97
113:17,20,28,59,65,70,76,87,95,96
114:15,46,49,53,58,67,73,83,88,104,110
90:0,5,8,11,23,24,28,50,69
```

②CPM 算法的输入数据（和 PSCAN 的格式一样）

在此不再赘述。

③IICS 算法的输入数据（一部分）

其中输入数据文件的每一行都是一个社区结构，前面的是社区的 ID，后面的是社区中所包含的节点的内容。↓

```
0   [0, 33, 1]
1   [0, 16, 67, 4, 23, 104, 41, 9, 90, 93]
2   [1, 33, 37, 103, 25, 89, 105, 109, 45]
3   [50, 69, 24, 90, 11, 28]
4   [106, 92, 15]
5   [18, 19, 36]
6   [80, 82, 19, 35, 101, 55, 91, 44, 29, 30, 94, 79]
7   [64, 32, 2, 100, 6, 39, 106, 60, 13, 14, 15, 47]
```

（Figure20IICS 算法的输入数据）

该数据为用户的输入数据，其中不同用户之间的查询用换行符隔开，一个用户所要查找的不同属性信息用分号隔开。

```
"16,12;42,39"+"\n"+
"50,20,17;1,7"+"\n"+
"4,11,12,6;37,30";
```

（Figure20IICS 算法的输入数据）

4.4 算法/系统模块输出结果示例截图及描述

①算法 PSCAN 的输出数据（一部分）

（由于 PSCAN 需要找到最合适的阈值 alpha，因此算法要运行多次，在此选用 alpha=0.6 进行展示，alpha 属于[1,9]）

该数据是没有经过处理后的社区，也就是主算法的运行结果。其中每个一行都一条边。所有有边相连的点构成了一个社区。↓

```
0,104
0,16
0,23
0,4
0,41
0,9
0,93
1,103
1,105
```

（Figure21alpha=0.6 的输出数据）

该数据是经过处理的方便进行性能评价的输入数据，其中每一行数据都是一个社区。↓

```
1   [81, 98, 84, 5, 10, 107]
2   [0, 16, 4, 23, 104, 41, 9, 93]
3   [1, 33, 37, 103, 105, 25, 89, 109, 45]
4   [32, 64, 2, 100, 6, 39, 106, 60, 13, 15, 47]
5   [51, 68, 21, 22, 7, 8, 108, 77, 78, 111]
6   [49, 114, 67, 83, 53, 88, 73, 110, 46]
7   [48, 66, 86, 57, 75, 91, 44, 92]
```

（Figure22alpha=0.6 的经过处理的输出数据）

该数据是算法根据 F-Score 指标计算的结果。后面两行，等号前面为标准社区和求解社区的对应关系，等号右面二者之间的准确率和召回率。↓

```
F-Score: 0.8506587258617578
{4->3 =1.0, 8->11 =0.83, 5->4 =1.0, 3->13 =0.8, 1->2 =1.0, 2->1 =0.0,
{4->3 =1.0, 8->11 =1.0, 5->4 =1.0, 3->13 =1.0, 1->2 =1.0, 2->1 =0.0,
```

（Figure23PSCAN 算法的性能指标）

②CPM 算法的输出数据（一部分）

该数据是没有经过最后处理的主算法执行结果，该数据的每一行是两个社区的 ID 号，用逗号进行分割，表明两个派系之间可以进行合并。↓

```
4,48
8,18
8,19
8,58
8,9
11,18
18,19
18,58
```

（Figure24CPM 算法的输出数据）

该数据是经过处理的方便进行性能评价的输入数据，其中每一行数据都是一个社区。↓

```
0   [0, 33, 1]
1   [0, 16, 67, 4, 23, 104, 41, 9, 90, 93]
2   [1, 33, 37, 103, 25, 89, 105, 109, 45]
3   [50, 69, 24, 90, 11, 28]
4   [106, 92, 15]
5   [18, 19, 36]
6   [80, 82, 19, 35, 101, 55, 91, 44, 29, 30, 94, 79]
7   [64, 32, 2, 100, 6, 39, 106, 60, 13, 14, 15, 47]
8   [23, 78, 111]
```

（Figure24CPM 算法经过处理的输出数据）

该数据是算法根据 F-Score 指标计算的结果。后面两行，等号前面为标准社区和求解社区的对应关系，等号右面二者之间的准确率和召回率。↓

```
F-Score: 0.9226715307924787
{4->2 =1.0, 2->25 =0.8, 6->23 =1.0, 12->21 =1.0, 1->1 =1.0, 7->32 =1.0, 3->6 =1.0, 1
{4->2 =1.0, 2->25 =1.0, 6->23 =0.75, 12->21 =1.0, 1->1 =0.8, 7->32 =0.58, 3->6 =0.83
```

（Figure25CPM 算法的性能指标）

该数据是该算法各个小部分之间的时间指标：↓

```
社区划分算法运行时间:6.249秒
社区格式转化运行时间:0.313秒
计算聚类评价指标运行时间:0.425秒
程序运行总时间:6.987秒
```

（Figure26CPM 算法的时间性能指标）

③IICS 算法的输出数据（一部分）

该数据为不同用户的不同请求的查询结果。↓

```
The 0th User || The 0th request:    [1]
The 0th User || The 1th request:    []
The 1th User || The 0th request:    []
The 1th User || The 1th request:    [3, 9]
```

（Figure27IICS 算法的输出结果）

```
程序运行总时间:4.774秒
```

（Figure28IICS 算法的时间性能指标）

5 实验

5.1 测试环境

实验系统采用 Utuntu 18.04 虚拟机

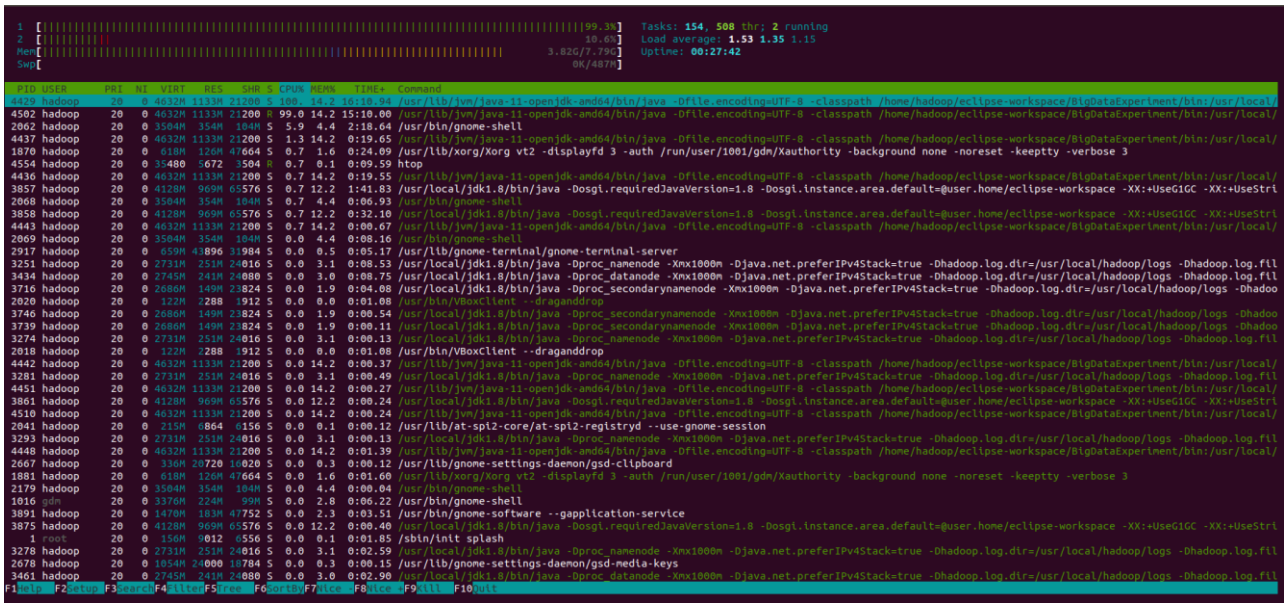
CPU 采用 Intel(R) Core(TM) i7-8750H CPU@2.20GHz

实验为 linux 虚拟机分配了 8G 内存

虚拟机的磁盘空间为 50G

（使用 TestExperimentData100.txt（其中包含了 24044 个节点，770600 条边）作为环境测试的测试文件进行测试）

利用 htop 工具查看 CPU 和内存的使用情况：↓



（Figure29IICS 算法的时间性能指标）

CPU 使用情况：我为虚拟机配置了两个 CPU，该进程的 CPU 占用率接近 100%（实际为 99%，其中 S 代表休眠进程，R 代表正在运行的进程）

内存占用情况：该进程占用了 4632M 的虚拟内存，1133M 的物理内存，21200M 的共享内存，占用了 14.2%的物理内存和虚拟内存。

利用 df 指令查看磁盘使用情况：↓

```
hadoop@bigdata:~$ df
文件系统           1K-块      已用      可用 已用% 挂载点
udev             4054052         0   4054052    0% /dev
tmpfs             816832      1580    815252    1% /run
/dev/sda5       20016976  11819928   7157176   63% /
tmpfs            4084152         0   4084152    0% /dev/shm
tmpfs               5120         4      5116    1% /run/lock
tmpfs            4084152         0   4084152    0% /sys/fs/cgroup
/dev/loop10        14976     14976         0  100% /snap/gnome-logs/45
/dev/loop4         35584     35584         0  100% /snap/gtk-common-themes/319
/dev/loop2         91648     91648         0  100% /snap/core/6130
/dev/loop1         91648     91648         0  100% /snap/core/6034
/dev/loop6         90368     90368         0  100% /snap/core/5897
/dev/loop3          3840      3840         0  100% /snap/gnome-system-monitor/51
/dev/loop0         14848     14848         0  100% /snap/gnome-logs/37
/dev/loop8         13312     13312         0  100% /snap/gnome-characters/103
LinuxShareFile 317448188 34196460 283251728   11% /mnt/Windows_Share
/dev/loop7          2432      2432         0  100% /snap/gnome-calculator/180
/dev/loop11         2304      2304         0  100% /snap/gnome-calculator/260
/dev/loop12        35456     35456         0  100% /snap/gtk-common-themes/818
/dev/loop5          3840      3840         0  100% /snap/gnome-system-monitor/57
/dev/loop13       144128    144128         0  100% /snap/gnome-3-26-1604/74
/dev/loop14        13312     13312         0  100% /snap/gnome-characters/139
/dev/loop9        144384    144384         0  100% /snap/gnome-3-26-1604/70
/dev/sda3       30831588   6858688  22383656   24% /home/hadoop/sda3
tmpfs             816828        28    816800    1% /run/user/121
tmpfs             816828        28    816800    1% /run/user/1001
/dev/sr0           56654     56654         0  100% /media/hadoop/VBox_GAs_5.2.22
```

（Figure30 磁盘使用情况）

磁盘占用情况：其中/dev/sda3 是我为 hadoop 文件系统分配的磁盘，一共 30G，使用了 24%，其中包含了 map-reduce 程序计算的中间结果以及 hdfs 分布式文件系统的存储。

5.2 数据集

实验所给数据集：http://snap.stanford.edu/data/中的 Orkut 大规模数据图。

数据集来源及意义：

数据集名称	数据集来源	数据集意义
club.txt	第三方网站的聚类常用数据集	空手道俱乐部
dolphin.txt	第三方网站的聚类常用数据集	海豚群落
football.txt	第三方网站的聚类常用数据集	足球俱乐部
science.txt	第三方网站的聚类常用数据集	世界科学家分布
TestExperimentData10.txt	在 SNAP 中的 Orkut 大规模图 TOP5000 社区中随机抽样 10 个	Orkut 大规模图的抽样
TestExperimentData50.txt	在 SNAP 中的 Orkut 大规模图 TOP5000 社区中随机抽样 50 个	Orkut 大规模图的抽样
TestExperimentData100.txt	在 SNAP 中的 Orkut 大规模图 TOP5000 社区中随机抽样 100 个	Orkut 大规模图的抽样
TestExperimentData500.txt	在 SNAP 中的 Orkut 大规模图 TOP5000 社区中随机抽样 500 个	Orkut 大规模图的抽样

数据集内容及属性：

数据集名称	顶点数	边数	行数	每一行的含义
club.txt	34	154	78	一条无向边
dolphon.txt	62	318	159	一条无向边
football.txt	115	1226	613	一条无向边
science.txt	1461	5484	2742	一条无向边
TestExperimentData10.txt	1064	22696	1064	一个邻接表表项
TestExperimentData50.txt	8129	225468	8129	一个邻接表表项
TestExperimentData100.txt	24044	770600	24044	一个邻接表表项
TestExperimentData500.txt	106855	3968220	106855	一个邻接表表项

（备注：所有数据中都是无向边，但在程序中通过来回的有向边实现无向边；后面四个抽样的数据每一行之所以是一个邻接表表项，是因为在抽样的过程中，实现了对数据的预处理，在没经过预处理之前每一行的内容也是一条无向边。）

5.3 测试算法/系统的效果

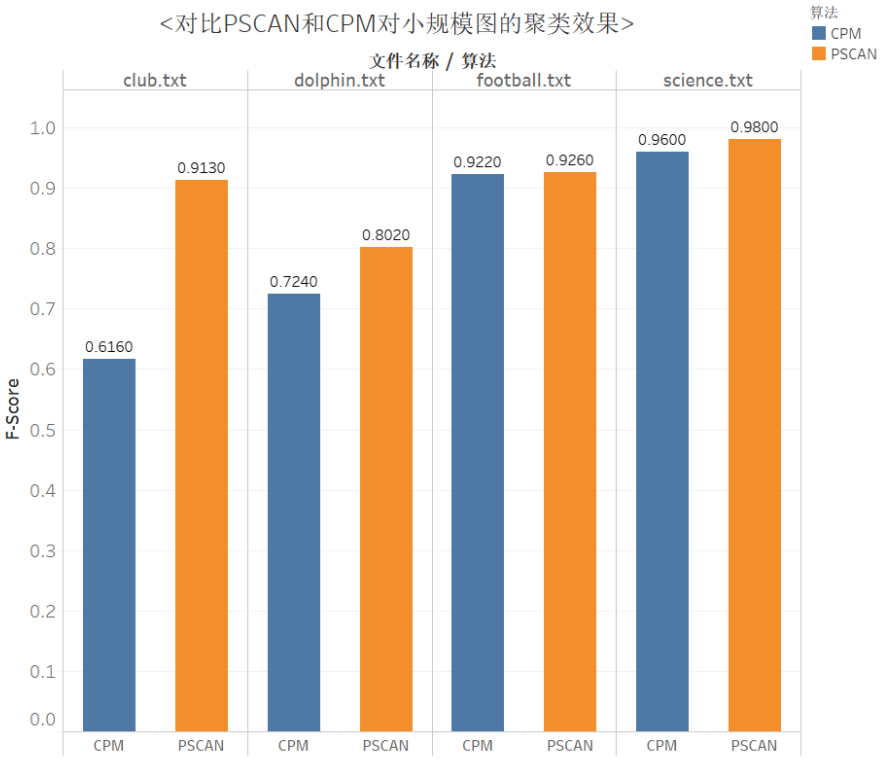
聚类效果的评价采用 F-Score 指标，其计算公式为：

$$F\_Score = (1+\beta^2)\cdot\frac{Precision\cdot \mathrm{Re}call}{\beta^2\cdot Precision+\mathrm{Re}call}$$

其中：Precision 为准确率，Recall 为精确率，β 用于调整召回率和准确率对指标的影响程度，当认为召回率更重要些时，β 值大于 1；准确率更重要时，β 值小于 1。（由于实验是一个多类别问题，所以相关指标的计算通过一对多的方式进行拆分，即把某一个社区中的点看为正例，其余全部看为反例，然后依次将每个社区都轮流设为正例，最后求得指标的均值。）

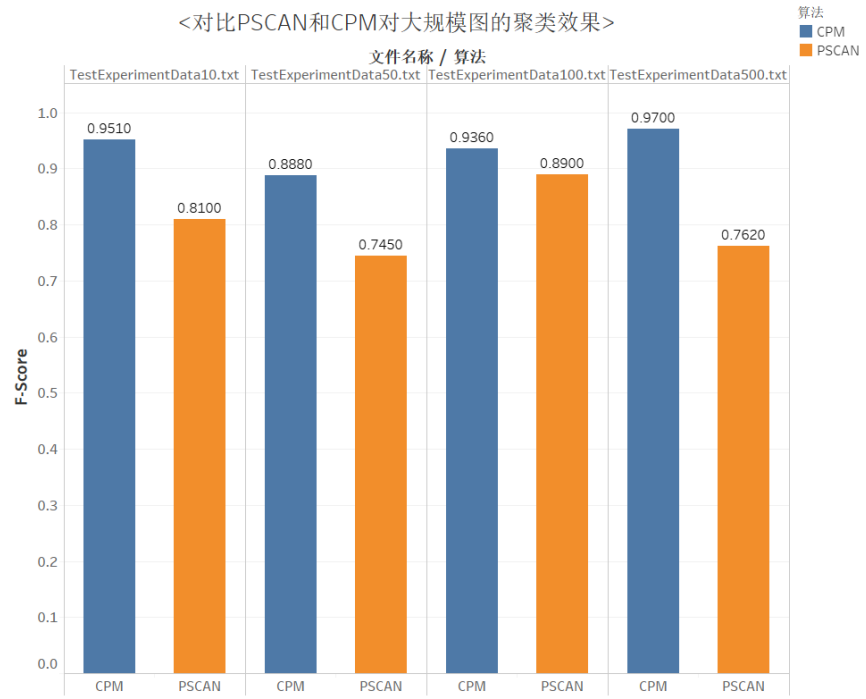
①首先对比了一下 PSCAN 和 CPM 两种社区发现算法分别在小规模图上和发规模图上的聚类效果，并通过 F-Score 指标进行展示。其中图表的横坐标为不同文件下的不同算法，纵坐标为 F-Score 指标。

通过对图 31 的分析可以发现，在小规模图上 PSCAN 算法发挥的更好，对社区划分的更加精准，而 CPM 算法则发挥的不太稳定，只对于某些特定的数据集发挥较好。



（Figure31 小规模图聚类效果对比）

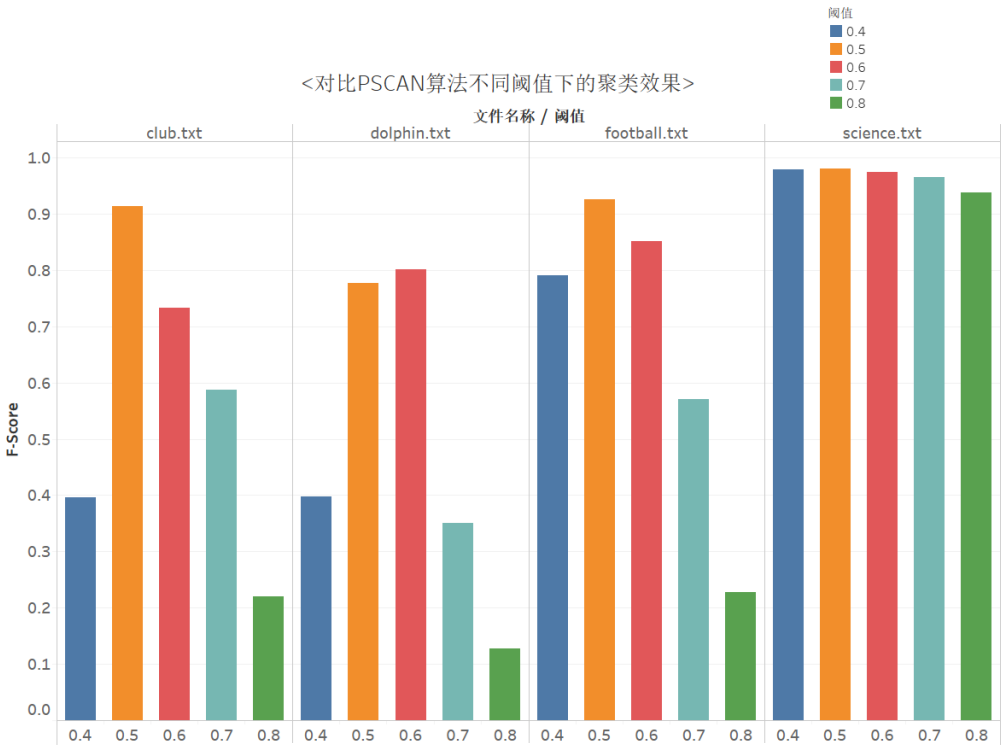
通过对图 32 的分析可以发现，在较大规模的图上 CPM 算法相较于 PSCAN 算法凸显出较大优势，算法的效果明显好于 PSCAN 算法，这样证实了我们对算法设计的分析。



（Figure32 大规模图聚类效果对比）

②对比分析 PSCAN 算法在运行时，不同阈值对社区发现结果的影响。同样算法结果的评价指标用 F-Score 进行衡量。其中图标的横坐标为不同文件下的不同阈值，纵坐标为 F-Score 指标。

通过对图 33 的分析可以发现，阈值的选择对 PSCAN 算法的影响很大（science 文件有些例外，原因是由于它代表了世界科学家的分布，所有社区之间间隔较远，而且干扰较少，而社区内部节点关系及其密切），但通常阈值选取 0.5-0.6 能取得较好的效果。

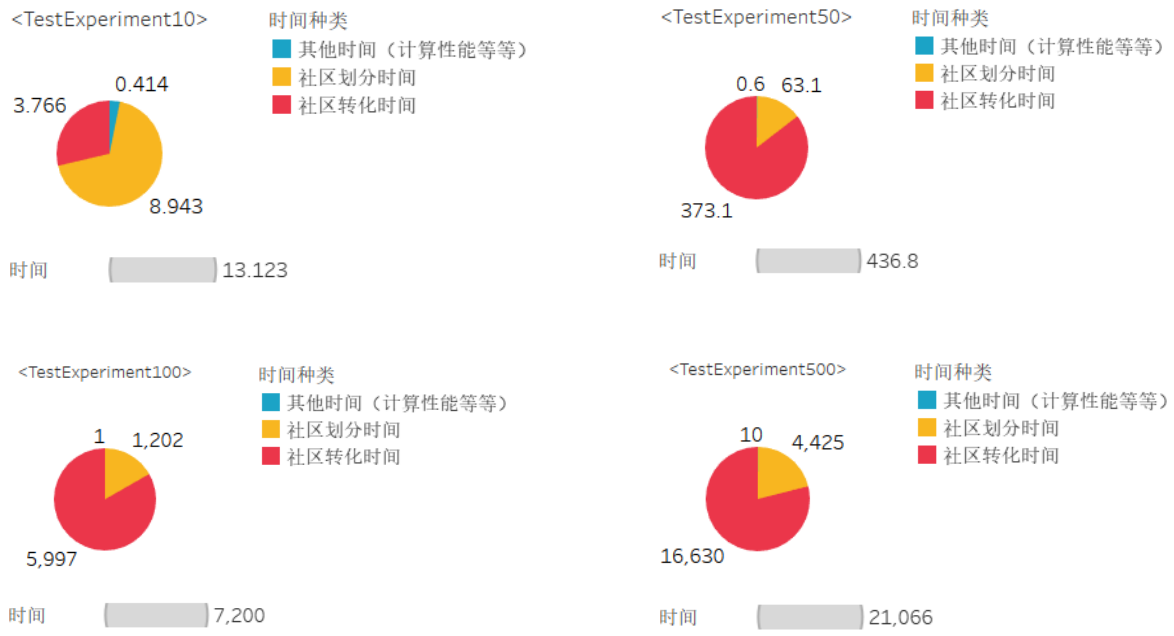


（Figure33 不同阈值下 PSCAN 算法聚类效果对比）

5.4 测试算法/系统的性能

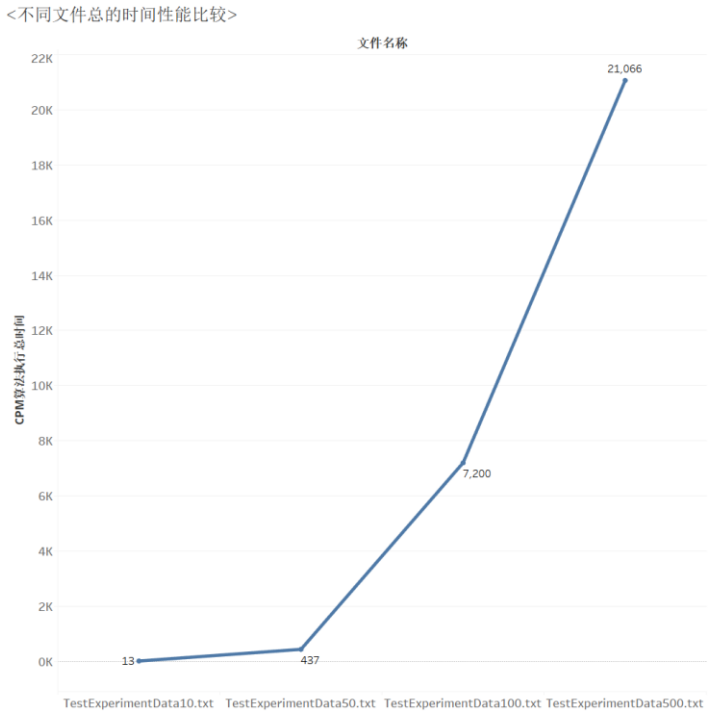
①测试 CPM 社区发现算法的各个阶段的时间性能情况，其中总的算法执行时间包括：社区发现（划分）算法的执行时间，社区转化的时间，以及其他时间（计算性能指标的时间），通过对大的图文件进行相关的指标分析，以饼状图的方式展示出来。

通过对图 34 的观察，我们不难发现随着大规模图的增大，其他时间所占的比例逐渐减小，并且时间性能较高，没有耗费大量时间；而社区划分时间的比例也逐渐减小，时间增长的速度可以接受，然而社区转化的时间随着图规模的增大逐渐变成了时间耗费的主要来源，并且时间耗费较多，性能一般。因此在社区转化算法还有进一步改进的空间。



（Figure34 各个部分时间性能分析）

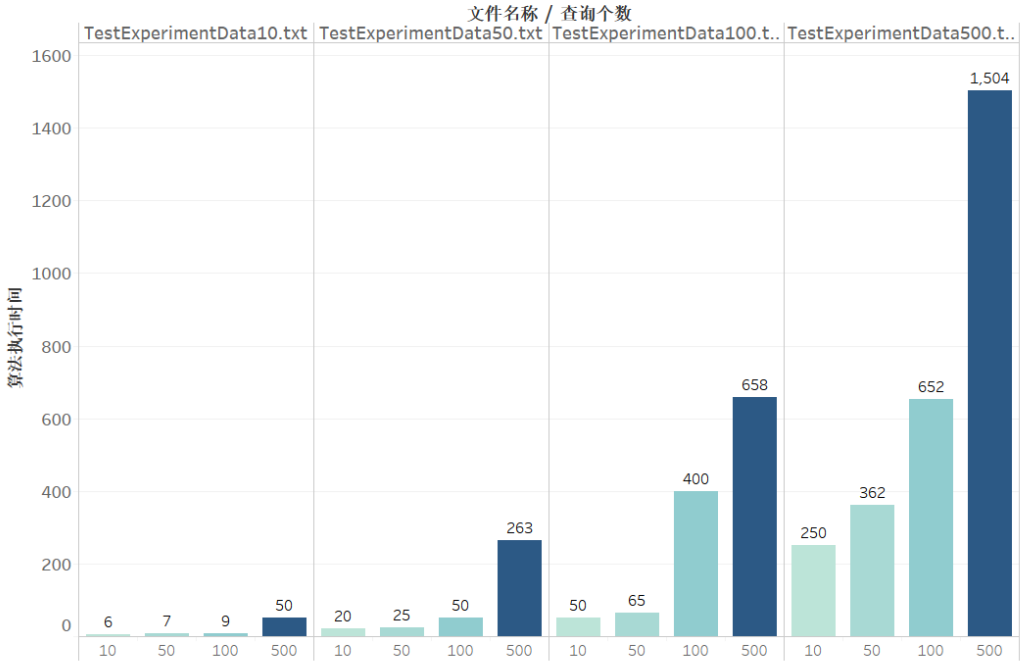
②通过对图 35 观察 CPM 算法的总时间变化趋势，来分析算法的时间性能。



（Figure35 总时间变化趋势）

③图 36 对 IICS 算法进行时间性能的分析，通过对不同文件以及查询个数的不同进行横向和纵向的对比，不难发现 IICS 算法的时间性能较高。随着图大小的不断提高，时间增长幅度逐渐增大，但相对于其预处理过程（CPM 算法）的执行时间，其总的时间大小可以接受。

<社区查找时间对比>



（Figure36 社区查找算法的时间性能）

6 结论

本次实验完成的是对大规模数据图上的多源的特定属性的社区查找问题，尝试采用了 PSCAN 和 CPM 两个算法对数据图进行了聚类，之后使用了基于倒排索引的社区查找算法进行了多源的社区查找。

通过对实验结果的分析以及性能指标的评价，不难发现，在社区发现（聚类）的实现过程中，基于扫描的 PSCAN 算法在小规模（即顶点数、边数、平均聚类系数、图密度、网络直径都较小）的图上能发挥出较好的效果，社区划分较为准确，并且能在有限的时间内完成对最佳阈值的查找即最佳社区划分的选择，总体效果不错；但由于其算法的特定属性，需要遍历图中的所有边，并且需要尝试多次才能找到最佳的阈值，因此它在大规模的图上进行聚类效果不佳而且效率较低。

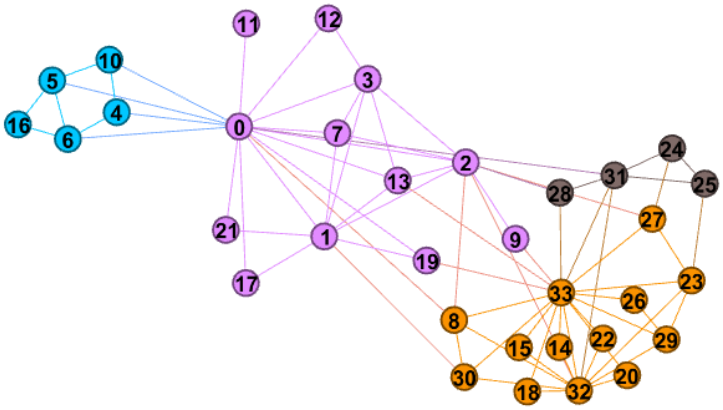
然而，基于派系过滤的 CPM 算法在大规模（即顶点数、边数、平均聚类系数、图密度、网络直径都较大）的图上能发挥出较好的效果，它与 PSCAN 算法不同，它能查找到所有的重叠社区，这在大规模图中是非常重要的，因为大规模的图中常常存在着重叠的社区结构。除此之外，CPM 算法不需要找到最佳的阈值，因此在时间效率上有很大的进步；同样 CPM 算法在小规模的图上表现的不尽人意，因为在小规模图中，完全子图的个数较小，图的密度较低，导致 CPM 算法没有很大的发挥空间。并且 CPM 算法对参数的设置较高，CPM 算法对某些及其离散的图（如 Science 图文件）需要在第一步的时候，将两个顶点的完全子图不能进行过滤，而一般的图需要对两个节点的完全子图进行过滤；其次对于较小的图结构来说（如 Football，Dolphin，Club 等）在进行寻找 K 派系连接社区阶段（第三阶段）用两个极大完全子图顶点个数-1（即 min-1）这个限制条件来筛选可以进行 K 派系社连接的社区，而对于大规模的图结构来说，则需要通过 K-1 这个限制条件来筛选可以进行 K 派系连接的社区。综上，CPM 算法尽管不需要进行繁琐的迭代过程，但是需要进行精确的参数设定，并且更为重要的是它能对大规模的图结果进行非常出色的社区查找工作。

在进行社区发现之后，采用的基于倒排索引的 IICS 社区查找算法，这种算法由于是建立在进行社区发现之后，并且算法的主要时间耗费在建立倒排索引上，因此算法的时间复杂度主要取决于进行社区发现的时间。并且 IICS 算法不存在准确度的问题，算法通过将不同用户的不同请求进行分离，保证能到遍历到所有的解空间，实现查找多个用户所要查找的不同社区结构。因此 IICS 算法可以再 CPM 算法进行处理过之后，即在找到所有的社区结构之后，较为轻松地进行社区查找工作，并且算法的时间效率相对较高。

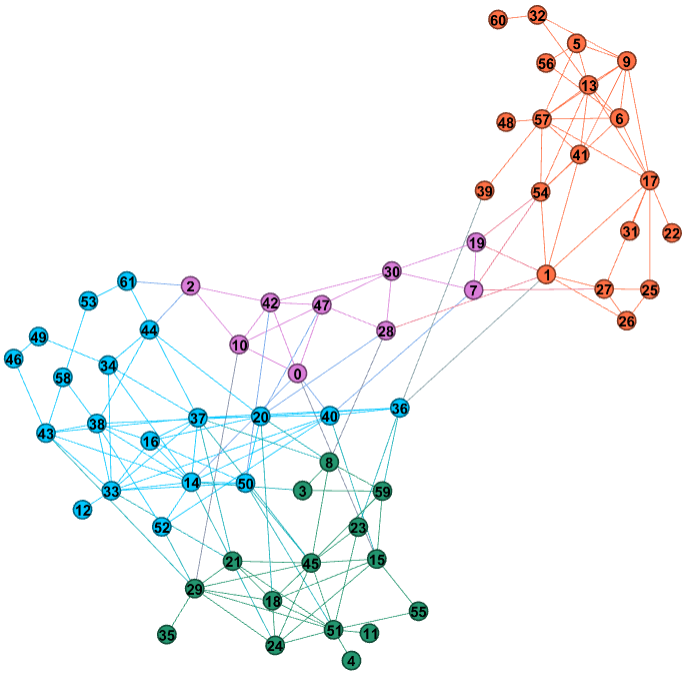
7 附录

对小文件的目标聚类效果进行展示，可用于对比分析。

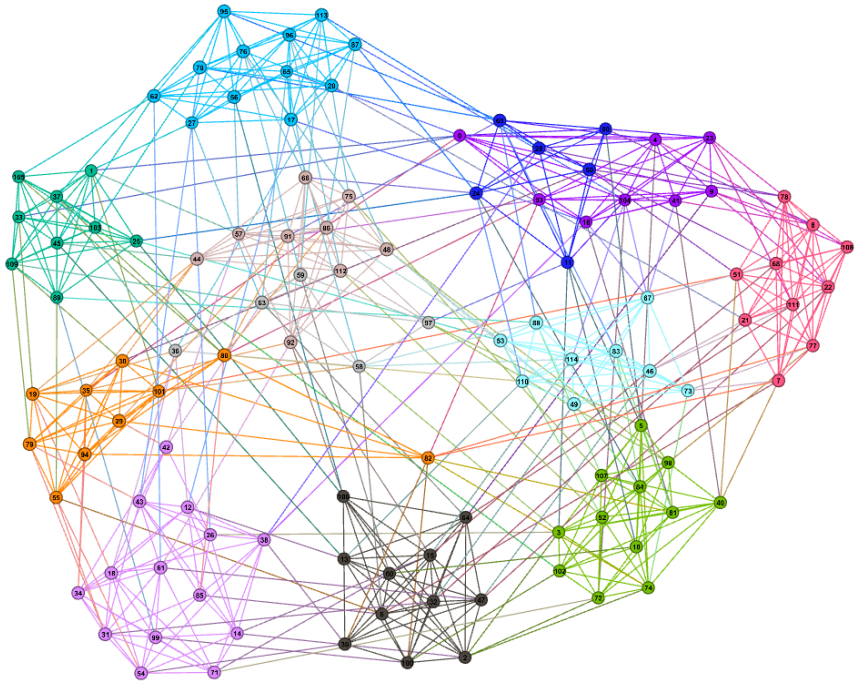
Club 图文件的聚类效果:



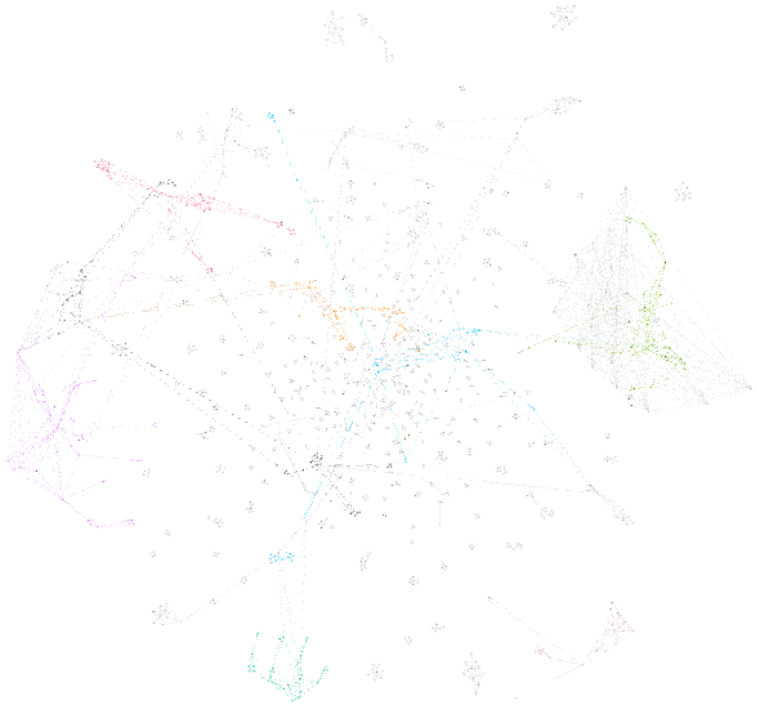
Dolphin 图文件的聚类效果:



Football 图文件的聚类效果：



Science 文件的聚类效果：



8 参考文献

[1] 施栓,肖仰华,温文灏,等.基于 Mapreduce 的大规模社会网络提取方法研究[J].计算机应用研究,2011,28(1):145-148.

[2] 高海艳,窦永香,齐艺兰.利用交互历史进行 P2P 知识共享社区发现的研究[J].现代图书情报技术,2013,29(9):93-98.

[3] Fkake G W,Lawrence S,Giles C L.Efficient identification of web communities[C]

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

[5] 吴祖峰,王鹏飞,秦志光,等.改进的 Louvain 社团划分算法[J].电子科技大学学报,2013,1:024

[6] 陈东明,刘健,王东琦,等.基于 MapReduce 的分布式网络数据聚类算法[J].计算机工程,2013,39(7):76-32

[7] 程鹏, 基于 Hadoop 平台的社区发现算法研究[D].东北大学,2011