分类号 0175.2 密级 限制

UDC 004.72

学 位 论 文

基于 Spark 的大规模图数据中的点对 SimRank 相似度计算

（题名和副题名）

高兴坤

（作者姓名）

指导教师姓名、职务、职称、学位、单位名称及地址 唐杰 副教授

南京大学计算机科学与技术系 南京市栖霞区仙林大道 163 号 210023

申请学位级别 硕士 专业名称 计算机科学与技术

论文提交日期 2018 年 5 月 10 日 论文答辩日期 2018 年 6 月 1 日

学位授予单位和日期

答辩委员会主席： 张三丰 教授

评阅人： 阳顶天 教授

张无忌 副教授

黄裳 教授

郭靖 研究员

南京大學

研 究 生 毕 业 论 文

(申请硕士学位)

论 文 题 目 基于 Spark 的大规模图数据中的点对 SimRank 相似度计算

作 者 姓 名 高兴坤

学科、专业方向 计算机科学与技术

研 究 方 向 大数据与分布式计算

指 导 教 师 唐杰 副教授

2018 年 5 月 13 日

学　　号：MG1533012

论文答辩日期：2018 年 6 月 1 日

指 导 教 师：　　（签字）

SimRank Computation on Large Graphs based on Spark

by

GAO Xing-Kun

Supervised by

Associate Professor Tang Jie

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
Master
in
Computer Science and Techonology

Department of Computer Science and Technology
Nanjing University

May 20, 2018

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于 Spark 的大规模图数据中的点对 SimRank 相似度计算

计算机科学与技术 专业 2015 级硕士生姓名：高兴坤

指导教师（姓名、职称）：唐杰 副教授

摘　要

复杂网络的研究可上溯到 20 世纪 60 年代对 ER 网络的研究。90 年后代随着 Internet 的发展，以及对人类社会、通信网络、生物网络、社交网络等各领域研究的深入，发现了小世界网络和无尺度现象等普适现象与方法。对复杂网络的定性定量的科学理解和分析，已成为如今网络时代科学研究的一个重点课题。

在此背景下，由于云计算时代的到来，本文针对面向云计算的数据中心网络基础设施设计中的若干问题，进行了几方面的研究。……

关键词： 小世界理论；网络模型；数据中心

南京大学研究生毕业论文英文摘要首页用纸

THESIS: SimRank Computation on Large Graphs based on Spark

SPECIALIZATION: Computer Science and Techonology

POSTGRADUATE: GAO Xing-Kun

MENTOR: Associate Professor Tang Jie

Abstract

dsafsdf

keywords: Small World, Network Model, Data Center

目　　次

绪论

研究背景与意义

图（graph）作为一种能够刻画实体之间关系的抽象结构，在很多领域中有着重要的应用。例如，电子商务中商品与用户之间的交易行为，科研著作中各类论文之间的相互参考引用，Web 中个网页之间的链接，以及社交网络中用户之间的交流传播等等行为，本质上都可以通过基于图来建模。从图中挖掘出深层次的信息也成了学术界尤其是产业界的一个重要课题。

相似性度量，即比较不同对象之间的相似度，是一个经典的课题。在推荐系统【】领域，人们关心哪几类商品一定程度上是相似的；在信息检索【】领域，人们关心如何从 WWW 找出相似的网页或文章，并对其聚类；在实体解析领域，人们关心如何从从复杂网络中推测出不同角色是否在真实世界代表着同一个真实实体。可以看出，现实生活中对相似度的计算无处不在。

而为了准确度量出“相似性”这一概念，学者们提出了各式各样的相似度指标。其中，SimRank 是近年来比较流行的一种度量图中节点相似性的模型，与传统方法不同的是，它考虑了整个图的拓扑结构，其基本思想与著名的网页排序算法 PageRank【】有相似之处：如果两个顶点的邻居顶点点非常相似，那么这两个顶点也相似。SimRank 相似度可以通过随机游走模型解释，拥有坚实的理论基础，近年来在很多应用中得到了广泛的应用。

传统的 SimRank 计算，可以归为三类问题：1）单顶点对 (single-pair) 的相似度计算；2）单源点 (single-source) 的相似度计算；3）全节点对 (all-pair) 的相似度计算。这三个计算任务的复杂度依次增加，由计算图中某一对顶点的相似度，到图中某顶点到其他所有顶点的相似度，再到计算图中所有顶点对的相似度。正因为 SimRank 试图充分利用整个图结构的拓扑信息，它的计算具有较高的时间复杂度和空间复杂度。然而随着互联网技术的发展，人类已经跨入大数据时代。社交网络用户呈现出爆发性地增长，电子商务变得日益普及，万维网上网络数据也不断增多，现实生活中图数据的规模越来越大。传统的 SimRank 相似度计算方法已经无法适用与大规模的图分析任务。

另一方面，在大数据时代诸如 MapReduce、Spark 等通用式的分布式计算平台正在日益流行。这些计算平台充分发挥了分布式集群的威力，使得计算能力在横向上扩展（Scale Out）而不是在纵向上提高（Scale Up），使得原来难以企及的计算任务变得可能。基于这些平台，用户可以快速部署出自己的解决方案，使用其提供的编程接口专注于自己的计算任务，而无需关系分布式平台底层的复杂网络通信、资源分配、资源调度问题。

因此，基于此类分布式计算平台设计出高效的分布式 SimRank 计算方法，具有现实意义。目前现有的分布式计算方法主要是基于矩阵计算，而对于顶点数超过百万的大图，其邻接矩阵的元素规模超过万亿，面对如此庞大的输入规模传统基于矩阵计算的方法根本无法满足需求。面向日益增长的大规模图数据处理的需求，设计新的分布式 SimRank 相似度计算方法就显得非常必要。

研究现状

SimRank 计算研究现状

图划分研究现状

技术背景

Spark 背景介绍

Aparch Spark 是由美国 UC Berkeley AMP Lab 开发的通用分布式计算引擎，目前它已经成为 Apache Software Foundation 下以及同类大数据开源项目中最受关注的项目之一。Spark 遵循类似于 MapReduce 的编程范式，但是在执行效率上，Spark 改进了 Hadoop[14] 批处理框架在迭代计算与交互式处理方面的不足，引入了 RDD（弹性分布式数据集）的概念，允许将计算的中间结果保存在内存之中而无需写入磁盘，大大改善了迭代计算的效率，应用程序的性能得到数十倍甚至百倍的提升；在用户易用性上，Spark 支持 Java, Scala, Python 等主流编程语言，提供了更加丰富的编程 API；在通用性上，Spark 针对不同的开发需求提供了更高阶的库，包括 SQL 查询，流式计算，机器学习算法以及图计算等等，开发者可以根据自己的需要，单独或组合使用这些库来处理复杂的数据分析任务。目前，Spark 已经成长为包含 Spark SQL，Spark

Streaming, MLlib, GraphX 等多个子项目的完善的生态系统。使用 Spark，用户只需调用编程接口来实现自己的数据计算任务，而无需关心数据的具体分布、并行调度策略、集群节点之间的数据传送与错误恢复等复杂的底层细节。凭借其高效、用户友好、通用的优点，Spark 在学术界和工业界得到了广泛的实际应用。 一个典型的 Spark 集群通常包括以下几个组件：1)Driver Program,

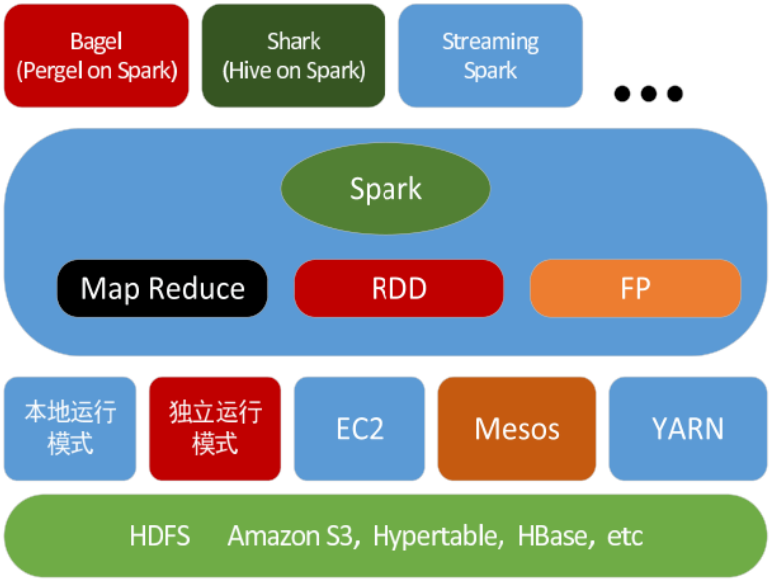


图 1-1: Spark 系统架构

每一个 Spark 应用程序都包含一个 driver program，它执行用户的 main 函数，并负责在集群调度所有的并行化操作。当应用程序刚从客户端提交上来的时候，Driver Program 需要连接到负责分配整个集群上各种资源的 Cluster Manager，并向其请求集群上能够使用的 Executor。然后，Driver Program 会把应用程序的 Jar 包发送给这些 Executor，计算过程中会将计算 Task 分配给这些 Executor。2)Cluster Manager, 它是向集群申请各种资源的一个额外的服务。Spark 支持 Mesos, YARN 等等。3)Executor, 它是运行在集群中每一个 Worker Node 上的一个进程，负责本节点上数据的读写以及各种计算。

Spark 数据模型

Spark 采用了一种被称为弹性分布式数据集（RDDs, Resilient Distributed Datasets）的分布式数据抽象，它支持工作集的重用，同时拥有非循环数据流模型（acyclic data flow model）的优点，即数据的负载均衡、位置感知和容错机制。同时 RDD 是一种受到限制的抽象，这体现在它是只读的，并且只

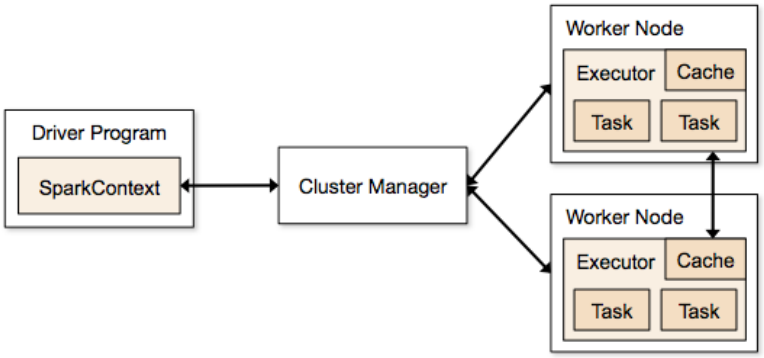


图 1-2: Spark 系统架构

能由已有的分布式数据集或其他 RDD 执行转换操作来创建。通过递归式地保存自己与其 ParentRDD 之间的转换记录 (lineage)，当发生数据分区丢失时 RDD 可以利用这些记录重新进行计算，快速地恢复数据。RDD 的缺点是它只能提供粒度较粗的转换 (coarse-grained transformation) 并且它是只读的，这对于一些需要频繁更新数据集的一部分的应用，可能会带来性能上的不足。矩阵计算就是一个需要频繁更新矩阵本身的操作，因此实现算法时要格外注意如何解决这个问题。RDD 可以调用的操作包括转换 (transformation) 和动作 (action) 两类。每一个转换操作都会基于调用它的一个或多个 parentRdd 产生一个新的 RDD，所有的转换操作都是懒惰 (lazy evaluation) 的，这意味着用户在一个 RDD 上调用一个转换操作并不会立即执行，Spark 只是把它当作 matadata 记住了这些操作。只有当该 RDD 执行动作操作时，Spark 才会利用这些 metatada 自后往前地追踪到当前 RDD 最古老的 parentRDD，然后从前往后顺序计算这些 RDD 之间的转换。Spark 提供了多种多样的转换，例如 map, mapPartitions, union, join, filter 等等。动作 (action) 指的是需要向 driver program 返回计算结果或需要将数据导出至存储系统的一类操作。每一个动作都会生成一个 Job，对数据进行计算并将计算结果返回给 driver。Spark 同样提供了多种多样的动作, 例如 reduce，collect, count，saveAsTextFile 等等。

Spark 常用编程接口

Spark 提供的编程接口，主要包括两类：转化和动作。其中，典型的转化包括以下几类：

1. `map` 一个 RDD 可以通过 map 将 RDD 中的每一个元素通过用户指定的函数转化为另一种类型的元素，也就是将原来的 RDD 转变为指定类型的

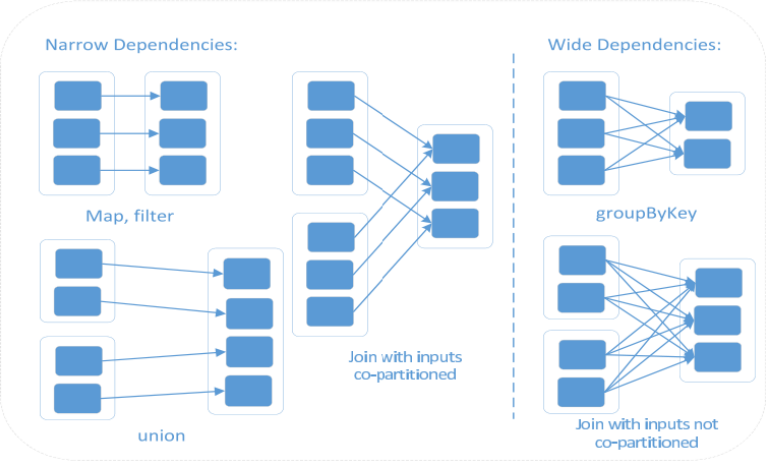


图 1-3: Spark 系统架构

RDD。map 函数接受的参数是一个函数算子。

2. `mapValues` 与 map 类似，mapValues 用于类型为 (key, value) 的 PairRDD。不同的是 mapValues 不改变元素的 key 而只改变元素的 value，因此新的 RDD 会保留原 RDD 的 partitioner。
3. `mapPartitions` 与 map 类似，它的第一个参数是一个函数，但 mapPartitions 是对原 RDD 的每一个 partition 而不是每一个 element 实施这个函数；它的第二个参数是一个 Boolen 类型，当原 RDD 的元素类型为 (key, value) 键值对，并且 mapPartitions 操作不会改变元素的 key 时，将第二个参数设为 true 告诉 Spark 不要改变 RDD 的 partitioner。
4. `filter` 一个 RDD 可以通过 filter 对 RDD 中的所有元素按照一定条件进行过滤, 过滤后的元素组成一个新的 RDD。因此，filter 函数接受的参数是一个返回类型为 Boolean 的函数，这个函数指明一个元素能不能通过筛选。
5. `join` 两个元素类型为 (key, value) 的 PairRDD 可以通过 join 操作将双方 key 相同的元素合并，返回一个新的 RDD。
6. `union` 两个 RDD 可以通过 union 操作合并为一个 RDD。
7. `reduceByKey` 一个 PairRDD 调用这个操作可以将原 RDD 中 key 相同的若干元素规约为一个元素，返回一个新的 RDD。它的第一个参数是一个函数，这个函数相当于元素之间的运算符；第二个可选的参数可以指定新 RDD 的 partition 个数。类似于 Hadoop 中的 combiner，spark 在 shuffle 之前会首先对每个计算节点本地的元素先进行合并，从而减小 shuffle 的开销。

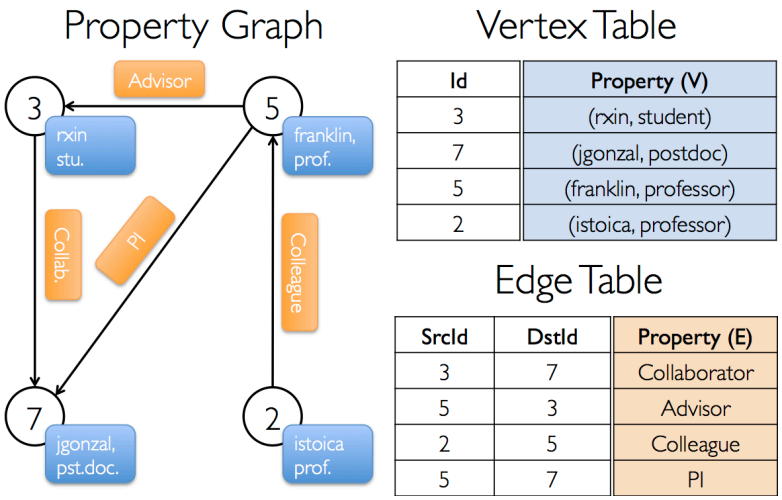
典型的动作包括以下几类:

1. `reduce` 它的参数是一个满足交换律、结合律的二元运算符，原 RDD 的所有元素经过这个运算符计算，返回一个最终值。
2. `collect` 它将 RDD 中所有元素从集群上以一个数组的形式返回到 Driver Program。collect 是一个开销非常大的操作，只适合在小数据集上调用。

还有一种比较特殊的广播操作。通常来说，当集群中的某个节点计算过程中发现需要使用用户自定义的变量时，会向 driver program 请求将该变量传送到该节点，每一个节点单独地对本地的变量副本进行计算，计算过程中对该副本做出的改动并不会反映到 driver program。Spark 提供了一种广播变量（broadcast variable），它允许开发者主动地将可能在在节点用到的变量广播到集群中的每个节点上，而不用在程序执行过程中当特定节点发现要使用这个变量再从 driver program 发送到这个节点上。Spark 在传送广播变量时使用了一些高效的算法，减少了广播过程中的通信开销。文献 [15] 表明，在进行迭代计算时，广播变量能极大地提高整体性能。

GraphX 背景介绍

GraphX[todo] 是 Spark 生态圈中的一个核心组件，主要用于图并行计算。它提供对图计算和图挖掘简洁易用的而丰富的接口，极大的方便了对分布式图处理的需求 GraphX 通过引入一个新的图抽象来拓展 Spark 中的 RDD: Resilient Distributed Property Graph，一种点和边都带属性的有向多重图这里的属性 (property) 指的是用户自己定义的描述边或定点某些性质的对象。一个典型的属性图中包含了描述节点信息的 VertexRDD 和描述边信息的 EdgeRDD。对属性图的所有操作，最终都会转换成其关联的 VertexRDD 和的 EdgeRDD 上的相关操作。这样对一个图计算任务，最终在逻辑上，等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性: Immutable、Distributed 和 Fault-Tolerant，其中最关键的是 Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

图 1-4: 一个简单的属性图。左图为图的拓扑信息及其关联的属性；右图为该属性图底层存储的 RDD

GraphX 编程范式

本文主要研究工作

分析已有的计算方法我们发现，传统的单机 SimRank 计算方法对于大规模的输入图无能无力，而现有的分布式算法基本也采用了矩阵计算的思路，而对于一个顶点数目为百万级别的图，其矩阵的元素基本在万亿级别以上。面对如此大的计算规模，即便分布式处理也无能为力。在此背景下，本文针对大规模图数据中点对 SimRank 相似度的分布式计算问题，进行了下列即方面的研究。本文的创造性研究成果具体如下：

1. 针对单源点 SimRank 相似度的计算，提出一种基于随机游走模型的分布式计算方法。整个算法包含三个步骤：首先生成随机游走，再对游走进行匹配计算其对应概率，最后汇总得出相似度。本文设计了一系列优化方法来加速算法过程，包括减少随机游走的数量，使用更紧凑的数据结构来压缩中间数据，以及通过动态规划的技巧加速随机游走的匹配。本文还对算法的复杂度给出了详尽的分析。
2. 针对大规模图数据，提出一种分布式图划分方法。该算法能够在保留图中稠密子结构的同时，对图中顶点进行尽量均匀的划分，同时最小化划分之间边的数量（或权重）。实验表明该分割方法具有较高的效率和划分质量。
3. 基于上诉的图划分方法，进而提出一种计算所有顶点之间 SimRank 相似度

的分布式近似计算方法。该方法使用分而治之的思想，将所有点对的计算问题划分为若干个子问题，最后再基于子问题的解聚合出初始问题的解。具体地说，算法首先对图进行划分，然后对每个分块内的顶点对计算局部相似度，然后将各个分块抽象为新的顶点形成一个粗化的图并依次计算各个分块之间的相似度，最后通过局部相似度和分块相似度共同估算出所有点对的全局相似度。实验表明，我们的方法在大大加速了计算速度的同时，也最大程度地保留了计算结果的准确率。

4. 对上述计算方法，我们分别在流行的通用计算平台 Spark 上的给出其分布式实现，并使用真实的数据集在验证算法的精度、效率，以及可扩展性。

论文结构

本文一共 5 个章节，每个章节的安排如下：

第一章：绪论。分别论述了研究背景和意义、研究现状以及现阶段研究尚存在的有待改进的问题、技术背景、本文的主要研究工作和文章总的框架结构。

第二章：分布式单源点 SimRank 相似度计算方法。该章首先对 SimRank 构建基于随机游走的模型，然后提出算法的框架以及算法中的若干优化思路，接着基于 Spark 平台实现算法，最后用实验验证算法各方面的性能。

第三章：分布式图划分方法。该章首先分析了典型的图划分过程的目标和难点，然后对比社区发现与图划分这两个任务的异同，并提出使用基于模块度的社区发现算法来进行图划分任务。接着提出算法的主要框架，包括塌缩、初始化分、恢复三个步骤。我们基于 Spark 平台给出算法的总体实现，最后通过实验验证算法各方面的性能。

第四章：分布式全点对 SimRank 相似度计算方法。首先将全点对的相似度计算划分为两个层面的相似度：分块内相似度和分块间相似度。然后分别给出了这两个相似度的具体计算方法。接着提出了基于上述两个相似度估算全局相似度的方法。我们基于 Spark 平台给出算法的总体实现，最后通过实验验证算法各方面的性能。

第五章：总结与展望。总结本文所阐述的所有算法，并对下一步工作做出展望。

SimRank 原理及技术背景

SimRank

为了方便描述，首先给出一些通用的符号定义。本文使用关系 (V,E) 表示一个图，其中 V 图的节点集合，$E \subseteq V \times V$ 是图中边的集合。n 和 m 分别是图中节点的个数、边的个数。节点 u 称作节点 v 的入邻点（或出邻点），当且仅当 (u,v)(或 (v,u)) 是 G 中的一条边。节点 u 的所有入邻点用符号 $I(u) = \{v : (v,u) \in E\}$ 表示，所有出邻点用符号 $O(u) = \{v : (u,v) \in E\}$ 表示。图中所有节点的平均入度（同样也是出度）用符号 d 表示。节点 u,v 的间的相似性由 $s(u,v)$ 表示，相应的，所有点对的相似度可以写作 $n \times n$ 的相似度矩阵 S，并且有 $s(u,v) = s_{uv}$。

SimRank 是一个基于结构上下文的相似度模型，它的核心思想是：如果两个节点的邻居节点非常相似，那么这两个节点也相似。节点 u,v 间的相似值用数学公式如下表达：

$$s(u,v) = \begin{cases} 1, & u = v; \\ \dfrac{c}{|I(u)||I(v)|} \displaystyle\sum_{u' \in I(u), v' \in I(v)} s(u',v'), & u \neq v. \end{cases} \tag{2-1}$$

其中，$0 < c < 1$ 衰减系数，用以提高临近结构对最终相似性的贡献权重，而降低更远结构的贡献权重。文献【1】证明了，上面的式子总是存在唯一的解，并且基于 SimRank 的定义，提出了一种基于矩阵乘法的迭代算法。设 S^k 是相似度矩阵 S 在第 k 轮迭代中的计算结果，S^0 为矩阵初始值，并且如果 $u = v$，有 $S_{uv} = 1$，否则 $S_{uv} = 0$。则矩阵 S^{k+1} 的计算方式如下：

$$S_{uv}^{k+1} = \begin{cases} 1, & u = v; \\ \dfrac{c}{|I(u)||I(v)|} \displaystyle\sum_{u' \in I(u), v' \in I(v)} S_{u'v'}^{k+1}, & u \neq v. \end{cases} \tag{2-2}$$

文献【1】已经证明，$\lim_{k \to \infty} S_{uv}^k = s(u,v)$。可以看出，该算法的时间复杂度为 $O(n^2)$，空间复杂度为 $O(kn^2d^2)$ time.

SimRank 的另一个模型是基于随机游走模型。节点序列 $W = v_0v_1v_2\dots v_l$ 如果对任意 $0 \le i \le l-1$ 都满足 (v_i, v_{i+1}) 都是 G 中的边，则称其为 G 中的一个游走。如果 W 还满足以下的 Markov 条件，即下式对所有的 $i \ge 1$, $v_0, v_1, \dots, v_i \in V$ 成立：

$$\begin{aligned} &Pr(X_i = v_i | X_0 = v_0, \dots, X_{i-1} = v_{i-1}) \\ &\quad = Pr(X_i = v_i | X_{i-1} = v_{i-1}) \end{aligned} \tag{2-3}$$

其中，X_i 表示在第 i 部，游走正好到达的节点。对任意 $u, v \in V$，$Pr(X_i = v | X_{i-1} = u)$ 表示在第 $i-1$ 步到达节点 u 的随机游走将在第 i 步到达节点 v 的条件概率。对于 SimRank 问题，每一条随机游走从某个节点出发，然后顺着图 G 中的逆变，每一个步骤随机走向该节点的某一个入邻点，并且特别的，转移概率定义为

$$Pr(X_i = v_i | X_{i-1} = v_{i-1}) = \begin{cases} \frac{1}{|I(v_{i-1})|}, & (v_i, v_{i-1}) \in E; \\ 0, & otherwise. \end{cases} \tag{2-4}$$

相应地，一条随机游走的概率，定义为：

$$Pr(W) = \prod_{i=1}^{l} Pr(X_i = v_i | X_{i-1} = v_{i-1}) \tag{2-5}$$

现在假设图 G 中有两条随机游走以相同速度分别从节点 u 和节点 v 同时出发，每次都顺着图中的逆边移动，也就是从当前所在节点移动到下一个入邻点，并且这两条随机游走在同一个节点 x 相遇并且为初次相遇，我们就把这两条终止在节点 x 的游走称作“匹配游走”。每一对匹配游走的长度就等于随机游走的步骤数。我们基于此定义相遇概率：

$$Pr\Big((W_u, W_v)\Big) = Pr(W_u) Pr(W_v) \tag{2-6}$$

文献【1】表明，节点 u, v 之间的相似度 $s(u, v)$ 等于服从上面概率分布的若干匹配游走的均值：

$$s(u, v) = \sum_{W_u, W_v} c^l Pr\Big((W_u, W_v)\Big) \tag{2-7}$$

其中，(W_u, W_v) 是一对任意的匹配游走，l 是游走的长度，c 是前文定义的衰减系数。

分布式单源点相似度计算方法

概述

相关工作

为了方便叙述，我们用 n 表示输入图的节点个数，用 m 表示图的边的个数，用 d 表示图中节点的平均度数。对于迭代算法，使用 k 表示算法需要迭代的次数。

基于矩阵乘法的计算方法。Jeh 和 Widom 提出了第一个基于矩阵乘法的 SimRank 迭代计算方法，改算法计算全点对的相似性，其复杂度为 $O(kn^2d^2)$.[9] 随后通过剪纸、局部访问等技术在矩阵乘法层面将算法的时间复杂度提升到 $O(kn^2d)$。【10】使用了快速矩阵乘法来加速计算过程。文献【12】进一步将算法的时间复杂度提高到 $O(kn^2d')$, 其中 $d' < d$。文献【13】提出了一种基于 Keonecker 乘积和矩阵奇异值分解（SVD）的非迭代算法，该算法首先在 $O(r^4N^2)$ 时间内计算一些出辅助矩阵，其中 r 是输入图的领接矩阵的秩，然后再在 $O(r^4N^2)$ 时间内获取给定源点与其他所有节点的相似性。文献【14】使用了 GPU 来加速矩阵的计算速度。对于单点对的 SimRank 计算，文献【11】给出了一个时间复杂度为 $O(kd^2 \cdot \min\{n^2, d^k\}$ 的算法。文献【15】进一步地通过使用概率方法将时间复杂度提高到 $O(km^2 - m)$。

基于随机游走的计算方法。图中节点 v,u 的相似性，可以解释为：以节点 u 和节点 v 分别为初始点，同步移动的随机游走第一次在某个节点相遇的概率的期望值。文献【16】给出了第一个基于随机游走的算法，该算法首先建立一个大小为 $O(nN)$ 的 N 个随机游走的“指纹”索引，然后基于索引查询单点对的相似性。文献【17】提出一种在随机游走中使用采样技巧计算单点对相似性，该方法在允许一定精度损失的情况下提高了计算效率。文献【18】研究了图中 top-k 个最相似点对的查询问题，其中 k 往往是一个非常小的值。该方法把单点对相似查询问题转变为规模为 $G \times G$ 的乘积图上的查询问题。

分布式计算方法。文献【19】提出了一种基于增量变化的 $Delta - SimRank$

算法。该算法发现，SimRank 可以改写为一种迭代的增量计算方式，即迭代过程中，不直接改变点对之间的相似性而是计算相对于上一轮迭代的增量。该方法充分利用了迭代计算过程中很多点对间相似性增量为 0 的事实，减小了计算过程中的数据传输量。文献【20】提出了一种基于 Spark 平台的分布式计算方法，该方法打破了计算点对相似性之间的递归依赖关系：首先线下计算一个不变矩阵 D，然后线上根据 D 使用蒙特卡洛方法快速给出查询。然而，D 的计算被当做一个解线性方程组的过程，这在分布式环境下非常低效，因此虽然算法线上查询的效率很高，但线下预处理的时间开销非常巨大。

基于以上的 SimRank 定义以及其基于随机游走模型的解释，计算单源点相似度 $s(u,*)$ 是非常直接的。直觉上讲，$s(u,*)$ 的计算过程可以分解为对 G 中所有节点 v 计算 $s(u,v)$ 的过程。而要计算 $s(u,v)$，我们首先要找出分别从节点 u 和节点 v 出发的所有匹配游走，然后聚合这些游走计算出它们的相遇概率。注意到想要枚举出任意长度（包括正无穷）的随机游走显然是不可行的，并且显然考虑的长度越长，最后计算得到的相似性越精确。现实应用中，我们考虑长度为 k 的游走得到的数值可以满足一般应用。

算法中心思想

文献【11】给出了一个计算单点对 SimRank 相似度的 spSimRank 算法。算法的核心思想是两个随机冲浪者分别从节点 u,v 出发，每一步跟随所在节点的入边；每经过一个节点 t，原来的随机游走会产生 $|I(t)|$ 个新的游走。这样的话经过 k 步游走之后，总共会产生 $O(d^k)$ 个不同的游走，每条随机游走的长度不超过 k。这些游走使用了一个叫做 Path-Tree 的数据结构来保存。然后对两个 path-Tree 中的所有随机游走进行匹配过程，找出其中的匹配游走。显然与基于矩阵迭代计算的计算方式相比，该算法不需要考虑图中所有的节点。如果我们从这个算法出发，可以轻易得出一个通过暴力枚举求单源点 SimRank 相似度的算法。

然而，直接通过多次调用 spSimRank 来计算单源点相似性有以下的缺点：

1. 总共生成了 $O(nd^k)$ 个游走，但其中大多数的游走都不能匹配到主游走
2. 尽管 Path-Tree 这种数据结构已经非常高效，但是考虑到会生成 n 个 Path-Tree，空间复杂度仍然比较高。在分布式环境下，因为需要频繁交换节点的局部拓扑信息，这回引起较高的网络开销。

Algorithm 3.1 Naive Single-source SimRank:nssSimRank

```
1: procedure SingleSourceSimRank(G, u, k)
2:     for l = 1 to k do
3:         W_u[l] ← all walks of length l starting from u;
4:     for v ∈ V(G) do
5:         s(u, v) ← spSimRank(G, W_u[], v, k);
6:     return s(u, ∗).
7: procedure spSimRank(G, W_u[], v, k)
8:     s(u, v) ← 0;
9:     for l = 1 to k do
10:        W_v[l] ← all walks of length l starting from v;
11:        s_l(u, v) ← 0;
12:        for w_u in W_u[l] do
13:            for w_v in W_v[l] do
14:                if w_v and w_u first meet at index l then
15:                    add score(w_u, w_v) to s_l(u, v);
16:                                          ▷ According to Eq. (2-7)
17:        add s_l(u, v) to s(u, v);
18:    return s(u, v).
```

3. 在最后的额匹配过程，统一长度的任意随机游走是以暴力方法匹配的，时间复杂度较高

算法的优化

减少随机游走的数量

如果直接使用 spSimRank 算法，总共会生成 $O(nd^k)$ 条随机游走，如果可以将系数 n 减少到某个值 c 使得 $C \ll n$，那么游走的数量会极大地减少。

定义 3-1 (倒叙游走)　我们直接定义随机游走的系列的倒叙序列为倒序游走。

显然，原始的随机游走与其倒序游走存在一一对应关系。注意到随意游走是在 G 中跟随入边生成的，相应的，倒序游走跟随出边生成。并且，倒序游走

$x, w'_1, \ldots, w'_2, w'_2, \ldots, w'_1, x, \ldots$

图 3-1: The left are two walks starting from u and v respectively, first meeting at x; the right are reversed walks starting from x, passing u and v respectively.

(a) $N_G(v,4)$. (b) Path-tree of $N_G(v,4)$.

图 3-2: (a) A neighborhood of v. (b) The corresponding Path-Tree representation of the neighborhood.

的转移概率需要调整为

$$Pr(X_i = v|X_{i-1} = v_{i-1}) = \begin{cases} \frac{1}{|I(v_i)|}, & (v_{i-1}, v_i) \in E; \\ 0, & otherwise. \end{cases} \tag{3-1}$$

基于倒序游走的定义，可以观察到有下列现象：

事实 3-1 假设两个分别从节点 u, v 开始的随机游走 W_u, W_v 经过 l 步后在节点 x 相遇，那么如果我们从节点 x 出发生成所有长度为 l 的倒序游走，W_u, W_v 的倒序游走一定也在其中

事实 3-2 假设 $W_u = uw_1w_2 \ldots w_l$ 是一条主游走，则所有有可能与 W_u 在 l 步相遇的从游走，只能在 $\{w_1, w_2, \ldots, w_l\}$ 中的任意一点相遇。也就是说，设 Nei 是节点 u 在 k 部可达的点集，那么只要生成从 Nei 出发的倒序游走就可以计算 $s(u, *)$。

以上两个事实的正确性是显然的。可以看出，总共需要生成的倒序游走的数量大约在 $O(|Nei|d^k)$ 而不是暴力算法的 $O(nd^k)$。事实上，$|Nei|$ 的复杂度为 $O(d^k)$，远比图 G 中点数目小，并且独立于 G 真实大小。

压缩中间数据的表示

相比于尽管生成游走的减少了很多，但是 $O(d^k)$ 复杂度依旧很高。而这些游走序列之间会有很多前缀子序列高度重合，如果单独地保存每一条游走，那么整个存储会有较大的冗余。TODO 为了解决多个游走序列共享很多前缀

子序列，对 $|Nei|$ 中的每一个节点 v，我们并不使用任何特别设计的数据结构单独存储从 v 的所有随机游走，而是直接使用从 v 开始跟随逆边 k 可达的一个邻域 $N_G(v,k)$。严格的说，$N_G(v,k)$ 是节点从 v 跟随逆边 k 步可达的节点集合在 G 中的生成子图。例如，在图【4】中，展示了 v 的一个邻域。他所对应的 path-tree 显示在右图中，灰色的节点是我们欲查询的节点 u。可以看到，$N_G(v,k)$ 自己本身就是 Path-Tree 的一种压缩表示。显然，我们在这一步中没有显式地表示出每一条游走，是因为在分布式环境中生成游走的过程需要在不同的计算节点上传输 G 中的局部拓扑信息，如果中间传输数据量过大，那么网络开销会极大地影响算法的最终性能。我们在下面会详细叙述即使没有显式表示每一条游走，算法仍然可以高效地计算出最终结果。

使用动态规划技巧加速随机游走的匹配

当 Nei 中的每个节点 v 收集到了从它自己开始的倒序游走 $N_G(v,k)$ 之后，这些从游走需要与主随机游走 $N_G(u,k)$ 进行匹配，从而得到最后的结果。由于代表所有主游走的 $N_G(u,k)$ 只是一个很小的信息 $(O(d^k))$，我们可以把它预先广播到分布式集群中的每一个节点上。而所有的从游走 $N_G(v,k)$ 按照 v 作为 key 分不到不同的节点上，然后每个计算节点直接在本地进行匹配过程。正因为这样，算法中最耗时的匹配部分是分布到每个计算节点上进行的，这是我们设计该算法的主要目的。

为了方便叙述，我们以在节点 v 第一次相遇的随机游走为例。注意整个匹配过程中，我们只对与主游走在 v 相遇，但是之后再也不和主游走有任何共同节点的逆序游走感兴趣。例如，在3-2b中，路径 vau 和 vbe 是一对对 $s(u,e)$ 有意义的匹配游走。同理，和 vau 能匹配上的随机游走还包括 vau 和 vcf。但是，vad 和 vau 不是一对合法的匹配游走，因为它们首次在节点 a 而不是 v 首次相遇。实际上，vad 和 vau 同样对 $s(d,u)$ 的计算做出了贡献，但是在分布式环境下，这个计算过程是由形如图3-2b、但 root 节点为 a 的 Path-Tree 引导的。这个计算过程也应当发生在该 Path-Tree 的计算节点上。基于以上的简单观察，可以发现以下事实:

事实 3-3　如果我们的查询节点 u 处于某个 Path-Tree 的某一层里，那么那一层的其他任一满足 $LCA(u,w)=v$ 的节点 w 都会对节点 u,w 的相似度 $s(u,w)$ 的计算做出大小为 $c^l Pr(W_u)Pr(W_w)$ 的贡献，其中 LCA 指的是最近公共祖先

(Lowest Common Ancestor)，W_u, W_w 是由对应 Path-Tree 展开的 root-to-node 路径。

我们使用 DFS 算法来搜索整个领域 Nei_v。 在 DFS 过程中，每一条 root-to-node 路径的概率被记录下来。当 DFS 的深度达到 l 时，算法停止，对应逆序游走的概率被保存在一个 HashMap 中。当我们再次匹配长度为 $l+1$ 的主游走时，就不再需要从节点 v 开始我们的 DFS 过程，因为该 Path-Tree 中长度小于等于 l 的逆序游走的概率之前已经全部计算过了。例如，在图3-2b中，能和游走 $vbdu$ 匹配的唯一游走是 $vbef$，而计算 $vbef$ 的概率时，我们只需要从节点 e 开始 DFS 过程，因为 vbe 的概率在之前搜索 vae 的匹配游走时已经计算被保存过了。

因此为了克服多个游走之间共享了很多重复前缀这一问题，我们把匹配游走这一问题看做是一个动态规划 (Dynamic Programming) 问题，通过 Memorization 的技巧来降低匹配的时间复杂度。算法的具体细节在3.2中列出。程序 LevelMatch 展示了一个 Path-Tree 中的根节点 v 如何寻找一个长度为 l 的主游走。各参数的意义如下：W_l 是长度为 l 的主游走的集合；N 是从节点 v 开始展开的领域；$M_{l'}$ 保存了之前调用 LevelMatch 函数来匹配长度为 l' 主游走得到的匹配概率，具体地说它是一个 (key,value) 形式的 HashMap, 其中 key 是 v 的邻居节点，表示该 k-v 对保存的是以 Path-Tree 中第二层节点为根节点的哪一棵子树的信息，value 是一串元素，每个元素记录着以 key 为根节点的子树中所有的游走的终止节点以及对应游走的概率；类似的 M_l 是一个将要被填充的空的 HashMap，保存匹配长度为 l 的游走的匹配概率；最后一个参数 δ 是一个概率阈值，其具体的含义将在下一节给出。我们首先循环 W_l 中的每一条主游走 p，并且知道该主游走在 Path-Tree 中的哪一棵子树中（2-3 行），如果 M_l 中没有记录该子树中匹配游走的概率，就开始 DFS 过程（4 行）。我们先检查 $W_{\square}l\prime\square$ 中是否有记录该子树的信息，如果有的话就直接从记录的游走的终止节点开始调用 DFS(5-6 行），否则就需要从该子树的根节点开始 DFS。程序 DFS 就是一个常见的 DFS 过程。通过对所有的 $W_l, (l \le k)$，所有游走的匹配概率都可以高效地计算出来。

Algorithm 3.2 Dynamic Programming Path Matching

```
1: procedure LevelMatch(W_l, N, v, M_{l'}, M_l, δ)
2:     for p ∈ W_l do
3:         br ← second last vertex of p;
4:         for t ∈ (v_N.neighbors − br) & t ∉ M_l do
5:             if !M_{l'}.contains(t) then
6:                 DFS(N, t, l, M_l, δ, t, 1, 1);
7:             else
8:                 for w ∈ M_{l'} do
9:                     for nei ∈ w_N.neighbors do
10:                        DFS(N, nei, l, M_l, δ, br, w.mul, l');
11:    return M_l.
12: procedure DFS(N, v, l, M, δ, br, mul, depth)
13:    mul ← mul * v_N.indegree;
14:    if mul > δ then
15:        return;                                  ▷ Early termination.
16:    if depth = l then
17:        add (v, mul) to M(br);                   ▷ Record probability.
18:    else
19:        for nei ∈ v_N.neighbors do
20:            DFS(N, nei, l, M, δ, br, mul, depth + 1);
21:    return M.
```

使用概率阈值剔除极小的概率

很多现实的大图都是 scale-free【2 1】的，这意味这图中的极小比例的节点会有极高的度数。我们的算法的性能与节点的度数紧密相关，因为在生产游走过程中每条游走每经过一个度数为 d 的节点，都会在那个节点分裂为 d 条更长的游走。因此，我们使用一个阈值 δ 来删除那些包含多个拥有极高度数的节点的游走，因为根据公式【TODO】，这些游走对的概率非常之小，对最终的计算精度基本可以忽略不计。δ 的取值应当注意在算法的精度与时间、空间复杂度取得平衡，更大的 δ 意味着更低的精度，当时算法整体的时间复杂度和空间复杂度更低，反之反是。

算法的复杂度分析

我们综合分析一下算法的复杂度。源点 u 可达的节点的数目大约为 $O(d^k)$，而对于其中的每一个可达节点，都拥有一个大小为 $O(d^k)$，隐式包含 $O(d^k)$ 条逆序游走信息的领域，因此，总的空间复杂度为 $O(d^{2k})$。因为这个过程中所有产生的游走都需要通过网络传输，所以通信开销也是 $O(d^{2k})$。在对游走进行匹配时，对每一个领域，大约有 $O(d^k)$ 个游走被匹配了，所以总的计算复杂度为 $O(d^{2k})$。可以看到，算法总的复杂度与整个图的规模 $O(n+m)$ 没有以来关系，所以我们的算法是非常高效的。

基于 Spark 平台的算法实现

基于 Spark 平台的算法实现可以概括为 4 个步骤:

1. 找出从顶点 u 出发顺着入边经过 k 部可达的邻域 Nei，同时记录所有的主游走。然后将所有的主游走广播到集群所有计算节点上。
2. 从 Nei 中每一个顶点 v 出发，顺着出边找出其 k 步可达的邻域，这些邻域看做一个以 v 为 key 的 key-value 对，散布在集群中。
3. 对上面每一个 v 的邻域，和第一步中的 Nei 进行匹配，计算匹配概率。
4. 聚合所有的匹配概率得出最终的相似度 $s(u,*)$。

上述四个步骤中第 1 个和第 2 个很相似，区别只是要获得邻域的顶点的数量（1 vs $|Nei|$）。因此我们重点描述第 2、3、4 步骤在 Spark 平台上的实现。

邻域收集步骤

代码3.3展示了生成邻域的具体细节。算法总共接受 3 个参数：*edgeRDD* 表示从 HDFS 读入的、每个元素代表图中一条边的 RDD，*Nei* 是要生成邻域的出发顶点集合，k 是游走的最大长度。首先我们把图从按照边存储的格式变为邻接表这种格式，并存储在 *graphRDD* 中（第 2-3 行），然后我们从 *graphRDD* 生成 *nbrhdRDD*（第 4 行），这个 RDD 里面存放着所有的邻域。*nbrRDD* 表示所有邻域的最外面一层，初始化时它就是集合 *Nei* 的直接邻居顶点（第 5 行）。接下来算法以迭代方法不断纳入新的边对 *nbrhdRDD* 进行扩张，即把之前邻域最外围的顶点 *nbrRDD* 的邻居节点纳入到邻域中。每一次

扩张也代表着游走的最大长度增加了一步，这个操作是通过 Spark 中的连接（join）操作完成的。每次扩张时通过 *nbrRDD* 与 *graphRDD* 做连接操作得到最新的最外围顶点（第 7-8 行），再通过 *nbrhdRDD* 与 *nbrRDD* 做连接操作把这些顶点纳入到邻域中（第 9-10 行）。

Algorithm 3.3 Collect Neighborhood

```
1: procedure COLLECT(edgeRDD, Nei, k)
2:     graphRDD ← edgeRDD
3:         .reduceByKey().cache();
4:     nbrhdRDD ← graphRDD.map().cache();
5:     nbrRDD ← graphRDD.filter().cache();
6:     for l = 2 to k do
7:         nbrRDD ← nbrRDD
8:             .join(graphRDD).map();
9:         nbrhdRDD ← nbrhdRDD
10:            .leftOuterJoin(nbrRDD).map();
11:    return nbrhdRDD.
```

匹配游走步骤

当所有邻域全部生成完毕后，我们开始匹配主游走和从游走。具体细节如算法3.4所示。程序接受 4 个参数：*nbrhdRDD* 表示程序3.3的返回结果；*MW* 表示被广播到集群各个计算节点上的所有主游走，其数据结构是一个由 $key-value$ 对组成的 HashMap，其中 *key* 是每个主游走的终止顶点（因此 $key \in Nei$），*value* 是所有以 *key* 终点的随机游走；剩下两个参数 u 和 k 分别是查询节点和游走的最大长度。对 *nbrhdRDD* 中的每个元素 $(v, nbrhd)$，我们调用算法3.2计算 *nbrhd* 中所有游走的匹配概率。注意到 *nbrhdRDD* 中的所有元素以 v 为 *key* 被 Spark 默认的 Partitioner 均匀分布在集群中，因此整个计算过程是分布式的。计算出的结果被组装成一个列表（第 7 行），然后经 flagmap 操作发射出去（第 8-9 行），最后经由 reduceByKey 操作将属于不同 v 的相似度 $s(u,v)$ 聚集起来，并送至 Driver Program（第 11-12 行）。

Algorithm 3.4 Compute SimRank

```
1: procedure COMPUTE(nbrhdRDD, MW, u, k)
2:     simRankRDD ← hbrhdRDD
3:         .flatMap((v, nbrhd) ⇒ {
4:             create MP;
5:             for W_l in MW do
6:                 M ← LEVELMATCH(W_l, ...);
7:                 extend MP with all elemetns in M;
8:             for (v, score(u, v)) in MP do
9:                 yield (v, score(u, v));
10:        })
11:    s(u, ∗) ← simRankRDD.reduceByKey().collect();
12:    return s(u, ∗).
```

实验评估

实验环境

所有的实验在一个由 6 个硬件完全相同的计算节点组成的集群上完成，每台计算节点处理器为 12 核的 Intel Xeon E-2650，频率为 2.1GHz，内存为 64GB，硬盘为 1TB。计算节点之间由千兆网卡连接。所有的节点上运行 Ubuntu 16.04 操作系统。Spark 运行版本为 1.6.2，底层分布式文件系统 HDFS 的版本号为 2.6.0。所有的 6 个节点都配置为 slave 节点，其中一个节点被二外配置为 master 节点。在 Spark 运行时，我们为其分配了 10GB 的内存。

实验数据集

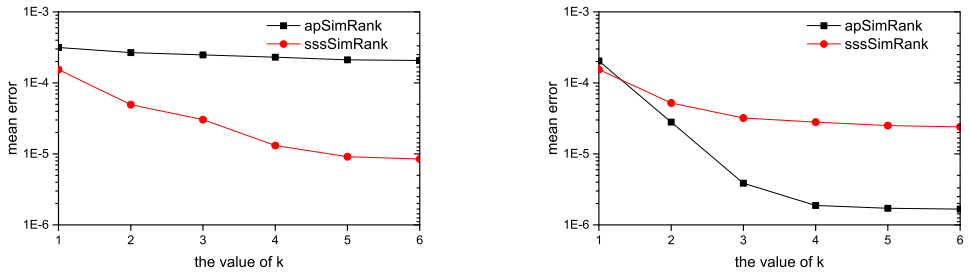
我们一共使用了 5 个真实的图数据来评估算法的性能。数据集的具体细节如表3-1。每个图一开始为普通的文本形式，每一行代表这图中的一条边。在开始实验前，所有的数据集都预先上传到分布式文件系统 HDFS 上。

实验参数设置

根据文献文献【9】中的描述，通常算法所需要的最大迭代次数 k 是由衰减系数 c 和算法的预期精度决定的。如果希望最终误差 $s^*(u,v) - s^k(u,v)$ 小于某个 ϵ，那么 k 至少要被设置为 $k = \lfloor \log_c \epsilon \rfloor$。在我们的实验中，我们选取

表 3-1: 数据集描述

数据集	顶点数	边数	顶点平均度数	大小
p2p-gnutella08 ①	6,301	20,777	3.29	215.2KB
wiki-vote ②	7,115	103.689	14.57	1.1MB
eu-2005 ③	862,664	19,235,140	22.29	256.4MB
ljournal-2008 ④	5,363,260	79,023,142	14.73	1.2GB
arabic-2005 ⑤	22,744,080	639,999,458	28.14	10.9GB



(a) p2p-gnutella08 (b) wiki-vote

图 3-3: apSimRank 和 sssSimRank 的相似性误差随迭代次数的变化曲线比较

$\epsilon = 0.01$，因为这样的精度可以满足大多数实际应用。c 被设置为 0.5，相应的，k 被设置为 $k = 6$。

此外，用于过滤掉极小概率的游走的阈值被设置为 $\delta = 0.002$。主要到这里的 δ 指的是一个随机游走的概率阈值，根据 2-6对于一对匹配好的游走，其对应的匹配概率相应的变成 δ^2。如果再考虑到因子 c^l，那么这个概率是极端小的，完全可以忽略。

算法精度的测试采用的方法是多次实验取平均值，每次实验时随机选取图中的某个顶点为查询顶点。具体的，对与小图（大小 <10MB），我们重复实验 100 次计算其平均值；对于更大的图，重复次数设为 1000。

算法有效性

我们首先比较我们的算法和全点对算法的精度和收敛速度。我们选取的精度指标为平均误差 (mean error)，即 $ME = \frac{1}{n}\sum_{v \in V} \left| s(u,v) - s^k(u,v) \right|$，其中 $s(u,v)$ 为根据 2-2迭代计算至完全收敛的真实相似性，$s^k(u,v)$ 为我们算法迭代 k 次后的相似性。我们在两个小图上进行实验。其中 p2p-gnutella08 是个比较稀疏的

图，平均顶点度数 $d = 3.29$，而 wiki-vote 是一个更加稠密的图，平均顶点度数为 $d = 15.57$。图 ??为最终的比较结果。从图中可以看到，全点对算法和我们的算法在 6 次迭代以内精度都得到了收敛，我们的算法 sssSimRank 有更好的收敛速度，三次迭代之后平均误差就在 10^{-4} 以内。另一个现象是在图3-3a中 sssSimRank 的平均误差与 apSimRnak 的差距比图3-3b更小，这是因为我们的算法中使用了概率筛除的缘故。对于同一个概率阈值 δ，图的平均顶点度数越大，小概率游走越多，相应的，被剔除的小概率游走越多，所以对最终精度的影响越来越大。本质上，δ 的作用是牺牲一定的精度来换取计算效率的提高。即便如此，sssSimRank 的相似性误差仍然非常小（$< 10^{-4}$），完全可以满足大部分应用的精度需求。

算法的效率

为了比较算法的运行效率，我们分别基于 Spark 平台实现了分布式的全点对算法 apSimRank，3.1算法 nssSimRank，以及本文提出的算法 sssSimRank。直接比较这三个算法的运行时间非常困难，因为 apSimRank 以及 nssSimRank 实际运行非常耗时，在规模最小的图 p2p-gnutella08 上，apSimRank 需要运行 2.1 个小时才能完成计算，而 nssSimRank 需要计算 1.2 个小时。因为对于基于随机游走模型的算法而言，其运行效率主要取决于所生产的随机游走的数量，进一步地说，取决于需要从多少个领域 *Nei* 来展开生产随机游走。因此，我们比较 sssSimRank 和 nssSimRank 算法中生产 *Nei* 的数量。结果如图 3-4所示。从图中可以看出，sssSimRank 极大地减少了生成领域的数量。当 $\delta = 0$ 时，即算法没有筛除概率极小的游走时，领域数量减少了大约 1500x 倍。从图中还可以观察到领域数量与概率阈值 *delta* 之间的关系。当 δ 越来越大时，意味这概率“筛子”的“孔”变得越来粗，幸存的随机游走会变得越来越少。图3-4还表明，图 eu-2005 和 arabic-2005 减少领域的比例比图 p2p-gnutella08 和 ljournal-2008 要大很多，这一观察同样表明算法中的概率筛子对稠密的图能更好地提高性能。

算法的可扩展性

我们考察分布式环境下算法的可扩展性。首先考察算法运行时间随着输入图的大小的变化关系，结果如图 3-5所示。

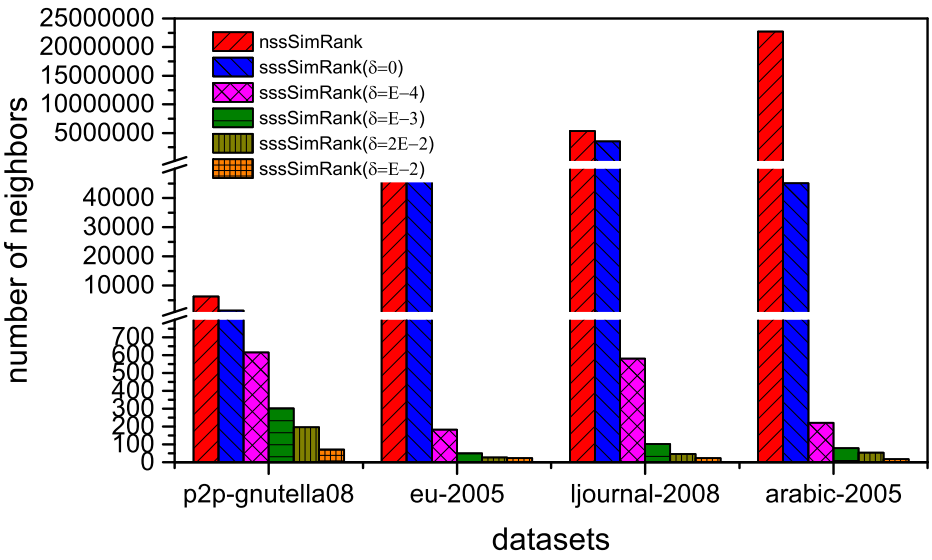


图 3-4: 生成领域数量的比较

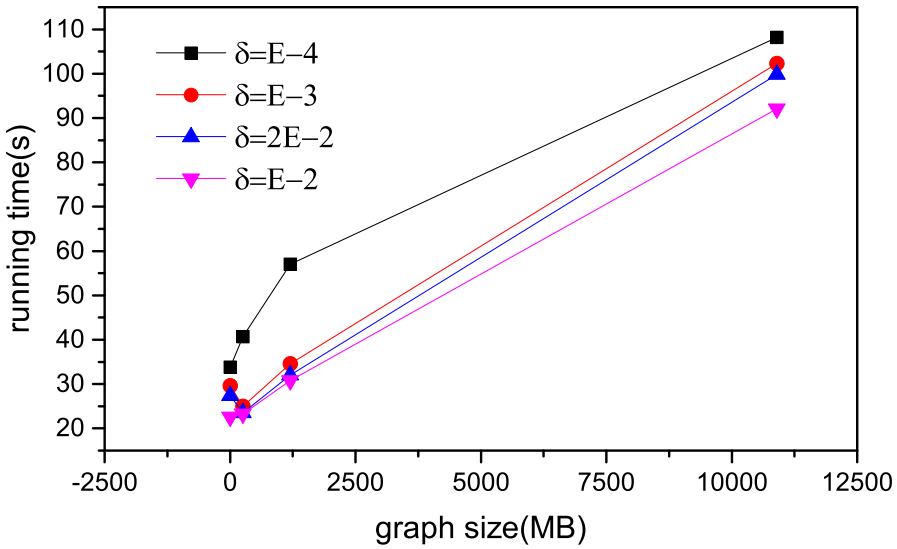


图 3-5: 算法运行时间与输入图大小的变化关系

图中展示了当集群计算节点数目固定时，对于不同的 δ，运行时间随输入图规模大小变化的情况。可以看出，输入图的规模从 215KB 到 10.9GB，而对应的运行时间基本上近似的随着输入规模大小线性的变化，这一结论对不同的 δ 都成立。这展示出 sssSimRank 良好的数据可扩展性。我们还考察了当输入图的规模固定时，算法运行效率随集群中计算节点数量变化的情况。所有的输入

图使用同样的参数配置，$k = 6$，$\delta = 10^{-4}$。计算节点数量从 2 增加到 6。注意到我们把 δ 取得非常小，是因为 *delta* 越小，算法剔除的游走越少，算法的计算量越大。这样更能提现计算量较为饱和的情况下算法的节点可扩展性。实验结果如图3-6所示。从图中可以看到，对不同规模的输入图，算法的运行时间随集群计算节点的增加而近乎线性的减少，这表面算法在分布式环境下具有良好的节点可扩展性。

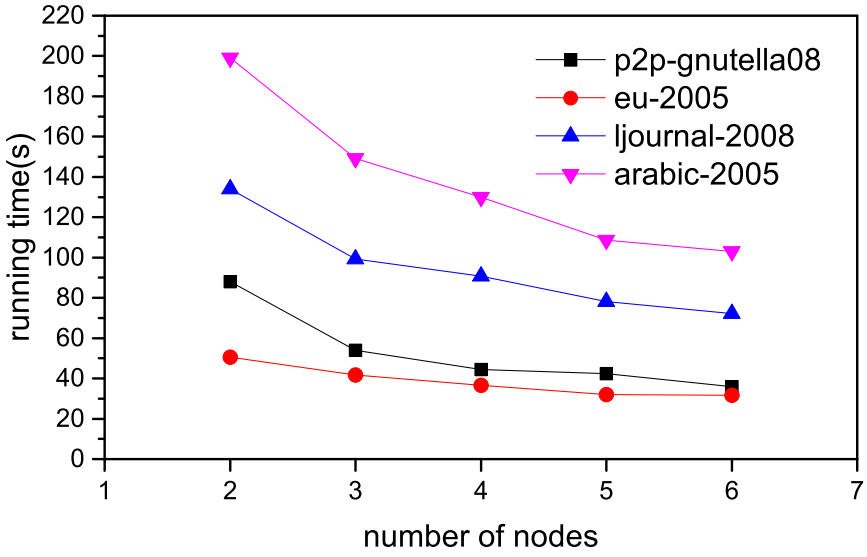


图 3-6: 算法运行时间与集群计算节点数量的变化关系

小结

分布式图分割方法

概述

随着现实生活中数据规模的不断扩大，与之对应的图数据网络结构愈加复杂，典型的大图甚至有数十亿个节点和上万亿条边。普通的单台计算机由于运算内存的限制，远远无法承载大规模图数据的分析任务，这给基于图计算的数据分析任务带来了巨大的挑战。为了解决单计算节点计算问题有限的问题，最理性的方案就是利用分布式计算，即将大规模图数据划分成多个子图装载到不同的分区或者不同计算节点上中，然后进行后续的图分析、图信息挖掘任务。

为了能够进行分布式图计算任务，必然会涉及到一个预处理步骤：将大规模图数据进行分布式划分。图划分问题可以用数学语言描述如下：

定义 4-1　图划分 (graph partitioning)。给定一个图 $G(V,E)$，设其顶点数量 $|V|=n$, 边的数量 $|E|=m$，图的划分即将 V 划分为 k 个子集 $(C_1,C_2,\ldots,C_k)$，使得对任意 $1\le i,j\le k$，有

$$\begin{aligned} C_i \cap C_j &= \emptyset \\ \bigcup_{i=1}^{k} C_i &= V \end{aligned} \tag{4-1}$$

给定一个划分，顶点在不同子集的边的权重（或数量）称作边割 (edge-cut)。

图划分问题往往还需要满足其他的约束条件，例如每个子集 C_i 的大小需要满足一定限制。边割的大小揭示了划分是否最大程度地保留了原图中的稠密子结构，是衡量最终划分结果质量的重要标准，因此通常图划分的目标函数里需要以某种形式最小化边割的大小。

作为图论领域的经典问题，目前已经有大量的工作对图划分做了各式各样的研究。文献【23】证明，图划分是一个 NP 完全问题，通常无法在有限的时间内找到图划分的最优解。对于大规模图数据而言，将图中的顶点划分为不同的分块并将其存储到集群中不同的计算节点上去，这一过程中每个计算节点为了能够访问到非本地存储的拓扑信息，必然会产生网络开销。因此，采用什么

样的方法对图分割，对划分过程中的网络开销、负载均衡、划分质量有着决定性影响。大规模图数据的划分任务面临着许多新的挑战：

1. 真实的图拓扑结构往往节点分布极不规则，这往往使得并行算法或分布式算法无法拥有很好的并发度或并行度
2. 真实图的节点的度数往往表写出非常倾斜的分布，这使得不同处理器上的负载无法均衡

因而设计一个好的分布式图划分方法具有重要的显示意义。针对以上问题，我们提出一种高效的分布式图划分算法，实验结果表明，该方法具有较高的效率以及较好的可扩展性。

相关工作

目前学术界对图划分问题做了大量的一些研究，概括地说，目前的划分方法主要有以下几种：

单节点串行算法。 文献【23】证明了图划分问题是一个 NP 完全问题。因此，早起很多工作都集中在设计一些近似算法以求得次优解。文献【5】提出了 Kernighan-Lin(KL) 算法，该方法首先将图初始划分为偶数个分块，然后基于一些启发式信息不断交换不同分块中的节点，直到不同划分之间的边割 $(edge-cut)$ 的权重之和达到最小。KL 算法每次迭代的时间复杂度为 $O(n^3)$，它的划分质量较高，但是不能应用在超图上，而且对于规模稍大的输入图就显得无能为力。FM 算法【6】改进了 KL 算法中的顶点交换策略，每次不再交换一堆顶点而是改为基于分割线使得顶点在分割线两侧移动。顶点移动的标准是最大化划分的负载均衡度。同时 FM 算法将算法的运用范围扩展到了超图 (supergraph)，它的每次迭代拥有线性的时间复杂度。还有一些基于模拟退火【24】，遗传算法【25】的解决方案。这些方法的研究对象都是基于内存存储的小图。

为了能够应对更大规模的图，一些多层次划分的方法相继被提出，包括 Metis[8]，Chaco[7] 和 Scotch[10]。这些方法主要由三个步骤组成：图的塌缩 (coarsening phase)、初始划分 (initial partition)、恢复 (uncoarsening phase)。其中，塌缩是指将图中若干顶点变为一个顶点，同时减少边的个数，从而提取出整个图拓扑信息的“骨骼”并基于此构造一个更小的图; 初始划分指的是经过多次塌缩过程后，在小图上运行传统图划分算法得到一个原始划分；最后恢

复过程是按照塌缩步骤的相反顺序逐渐将图恢复成原来的形状，并将原始划分投影成原输入图上的划分。这三个阶段缺一不可，共同完成了从大规模图数据到简单小图再重新恢复的过程。尽管可以处理的图规模有所扩大，它们的研究对象依然都是基于内存存储的小图。

单节点并行算法。 为了进一步充分利用现代 CPU 的多核特性，从以上这些单线程算法中演化出一些并行化算法，典型的有 ParMetis[11]，Pt-Scotch[12] 等。这些算法或多或少地挖掘出图划分问题本身潜在的计算可并行性，所以其能够处理的图的规模也获得了的提升。但总的来说这些方法仍然是基于单计算节点的，其计算能力受到有限的计算资源的限制，对规模更大的（上百万节点）图无能为力。

分布式图划分。 近年来，一些专门用于图计算的分布式平台越来越流行，包括 Pregel[15]，Giraph[X]，Spark GraphX[TODO] 等等。这些平台的一个典型特点是支持以顶点为中心 (vertex-centric) 的编程范式，并且在系统层面拥有原生的顶点间消息通信机制。然而上面几个流行的平台尽管也开放了一些接口支持用户按照需求设计自己的划分方法，但都没有提供一个直接可用的、高效的分布式图划分方法。在读入输入数据时，它们往往使用随机划分的方法加载图数据，其划分质量没有任何保障。

算法的基本框架

我们从以下两方面来考虑图划分算法的优化目标:

1. 划分结果的负载均衡。考虑一个由 k 个计算节点组成的集群，因为存储资源（硬盘）散落在每个计算节点上，所以理想情况下最终划分的结果应当尽可能的均匀分布在集群中。也就是说，每个划分的大小应当在 $\frac{|V|}{k}$ 左右，或者说划分过程中应该避免 $|C| > \frac{|V|}{k}$ 的分块的产生。如果仍然有某个分块分布式地存储在超过一个结算节点上，那所谓的分布式图划分就没有意义。
2. 划分结果的质量。理想情况下图划分算法应该可以把图中稠密的子图归类于一个分块中。但是因为有负载均衡的限制，而图中稠密子图的规模天然是不均匀的，因此我们从整体考虑划分的质量。设 P 是图 $G(V,E)$ 的一个划分，定义其边割为 $W_P = \sum_{e(u,v)\in E, u\in C_i, v\in C_j} w\Big(e(u,v)\Big)$，即所有连接不同分块的边的权重之和。理论上，$W_P$ 应尽可能的小。

我们的算法借鉴多层次划分的思想，即通过不断的 coarsen 步骤使得原图的规

模越来越小，当图的规模小到某个可以接受的阈值时，再使用 KL 或 FM 等单节点算法直接进行划分。最后划分的结果再直接投影到原图中，这样就得到了最终的划分结果。图4-1揭示了多层次划分算法的大致框架。

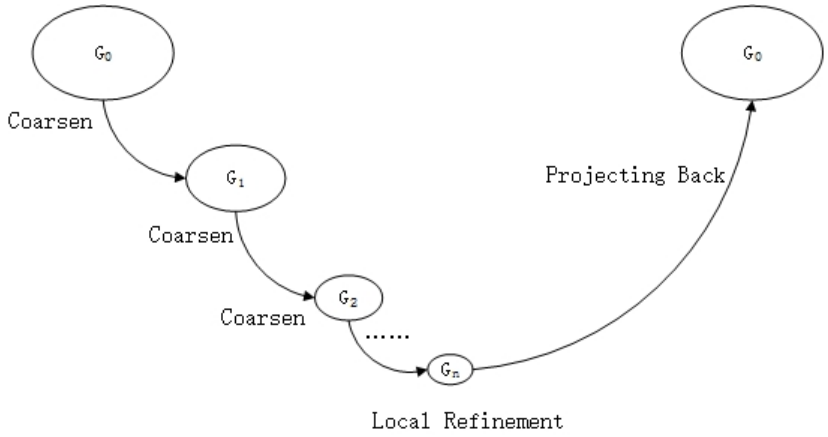


图 4-1: 多层次图划分的算法框架

塌缩步骤

设原始输入图为 $G_0(E_0, V_0)$，并假设迭代塌缩过程中产生的中间图由记号 $G_1, G_2, \ldots, G_t$ 表示。相应的，由中间图产生的划分称作 $P_1, P_2, \ldots, P_t$。注意到迭代过程中图的划分与其输入图下标是对应的，即 $G_{i-1} \to P_i \to G_i$。我们首先给出塌缩的具体过程。

定义 4-2 塌缩图的拓扑结构。迭代塌缩过程中，给定图 $G_i(V_i, E_i)$ 上的划分 $P_i = (C_1, C_2, \ldots, C_n)$，基于 P_i 上构造塌缩图 $G_{i+1}(V_{i+1}, E_{i+1})$ 如下：令 $V_{i+1} = P_i$，即 P_i 中的每一个划分对应 G_{i+1} 中的一个新顶点；边 $(C_i, C_j) \in E_{i+1}$ 当且仅当 $\exists u \in C_i, v \in C_j$ 并且 $(u, v) \in E_i$。

通过这样的塌缩过程，显然有 $|V_{i+1}| < |V_i|$，也就是说图的规模在迭代中不断变小。经过反复塌缩后，图 G_i 的规模会降低到某个阈值以下。

定义 4-3 塌缩图的权重。设初始输入图 $G_0(V_0, E_0)$ 中每个顶点有单位权重 $w_u = 1$，每条边 (u, v) 同样有单位权重 $w_{u,v} = 1$。设迭代过程中基于图 G_i 构造塌缩图 G_{i+1}，并且有 $V_{i+1} = (C_1, C_2, \ldots, C_n)$，那么新的图中顶点 u 的权重定

义为:

$$w(u) = \sum_{v \in C_k} w(v) \tag{4-2}$$

其中，C_k 为图 G_{i+1} 中新顶点 u 在图 G_i 中对应的分块。类似的，边 $e = (C_i, C_j)$ 的新权重定义为:

$$w(C_i, C_j) = \sum_{e(u,v) \in E_i, u \in C_i, v \in C_j} w\Big(e(u,v)\Big) \tag{4-3}$$

即压缩图中的新边的权重等于该边在原图中代表的两个分块间所有相连接的边的权重之和，而新顶点的权重为原图中对应分块中的所有顶点的权重之和。

通过这样的权重赋值方式，保证了塌缩后的图具有如下特性:

1. 边的权重赋值方式使得划分塌缩后的图所的边割代价与划分原图保持一致，这样可以保证图划分质量的一致性。
2. 顶点的权重赋值方式使得划分塌缩后的图的负载均衡状况与划分原图保持一致，这样可以保证划分过程中负载均衡的一致性。

那么，迭代过程中如何基于 G_{i-1} 生成划分 P_i 呢？ParMetis【】基于最大匹配 (maximal match) 来对图进行压缩。匹配指的是图中边集 E 的子集，其中每条边的端点两两不相交。如果一个匹配无法再加入其他边，那么这个匹配就是一个最大匹配。具体进行塌缩时，ParMetis 采取的做法是对匹配中的边，直接将其两个端点融合成一个新的顶点；而不在匹配中的顶点继续保留，这样就可以产生塌缩后的图。选择最大匹配的目的是最大程度地通过塌缩减小图的规模。

文献【todo】给出了一种基于标签传播 (Label Propogation，LP) 的图划分思路。LP 算法被广泛运用于解决社区发现 (Community Detection) 问题。社区，从直观上来看指的是网络中的一些密集群体，每个社区内部的顶点间的联系相对紧密，但是各个社区之间的连接相对来说却比较稀疏。社区发现任务与图划分任务的区别在于，前者不会固定社区的数量，因为具体社区的个数是由输入图的内在结构决定的；而后者往往会设定需要划分的分块个数 k。两者的相似之处在于，社区检测过程中图中局部联系紧密的子结构往往会共享同一个社区 ID，这与图划分任务的内在要求相一致。与其他图塌缩方法（例如最大匹配）相比，基于社区检测的方法能够更好地感知原图中的语义。

LP 算法的思想非常简单：最初的时候我们给图中每个顶点一个单独的社

区 ID，然后我们迭代式地为每个顶点更新社区 ID。每次迭代中，每个顶点将其邻居顶点所属社区 ID 出现次数最多的那个作为自己新的 ID。当 G 中每个顶点的社区 ID 基本上不再变动时，算法结束。图 4-2展示了一个典型的社区发现结果。从图中可以直观的看到算法总共检测到了 4 个社区。如果直接将其当做图的一个 4 划分 $P = (C_1, C_2, C_3, C_4)$，其划分质量也是不错的。

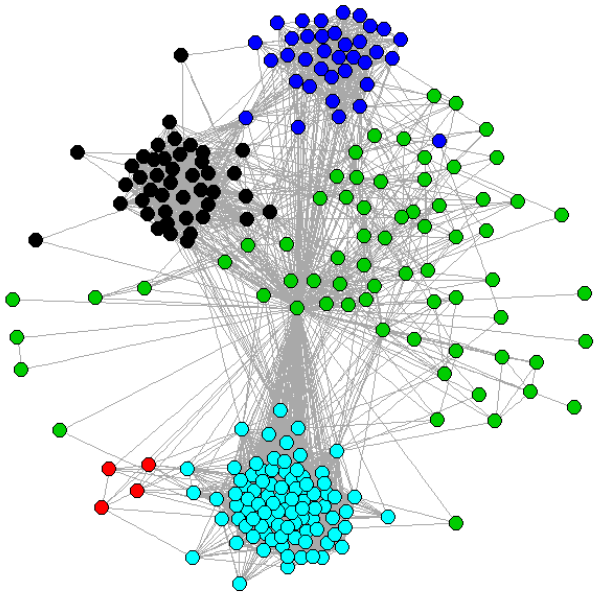


图 4-2: 社区发现算法的结果，不同颜色代表不同的社区 ID

上述社区发现过程中，确定每个顶点所属的社区 ID 并没有使用 G 中任何的先验信息或者是既定的目标函数，而只是利用了图中顶点与其邻居顶点的局部相似性原理。而确定每个顶点所属社区 ID 这一步骤是迭代过程中的计算主体，因此有很搞的提升空间。

本文借鉴了文献【7】中模块度 (modularity) 的思想，试图在迭代过程中有方向性地寻找合适的社区 ID。模块度是常用的一种衡量网络中社区稳定度的方法。简而言之，对于一个已经分配好社区（或划分）的图，分配在同一社区的边占总边数的比例，与对这些边随机分配得到的概率期望，这两者的差就是其模块度。用数学预言描述，设 d_v 表示顶点 v 的度数, C_v 表示节点 v 所在社区的 ID，图的模块度的定义如下:

$$Q = \frac{1}{2W} \sum_{u,v \in V} \left[w_{u,v} - \frac{d_u d_v}{2W} \right] \delta(C_u, C_v), \tag{4-4}$$

其中 $\delta(C_u, C_v)$ 表示顶点 u, v 是否处于一个社区，即如果 $C_v = C_u$，那么 $\delta(c_u, c_v)$ 等于 1；否则为 0。$w_{u,v}$ 表示边 (u, v) 是否存在，即如果 $(u, v) \in E$, 那么 $w_{u,v} = 1$，否则为 0。W 表示图中所有边的权重之和，即 $W = \sum_{(u,v)\in E} w_{uv}$。

按照上面的定义，我们有如下的定理：

定理 4-1 根据上面的定义，我们实际上可以得到模块度 Q 定义的简单形式：

$$Q = \sum_{c\in P}\left[\frac{I_c}{2W} - \left(\frac{S_c}{2W}\right)^2\right] \tag{4-5}$$

其中，I_c 表示 ID 为 c 的社区中两个端点都在 c 中的边的权重之和，而 S_c 表示社区 c 中所有顶点的度数之和，即 $S_c = \sum_{v\in V} d_v$

证明: 根据的定义，如果顶点 u, v 属于不同社区的话，那么显然有 $\delta(C_u, C_v) = 0$，因此只要考虑属于同一社区的点对，即我们可以把公式 4-4改写为：

$$\begin{aligned} Q &= \frac{1}{2W}\sum_{u,v\in V}\left[w_{uv} - \frac{d_u d_v}{2W}\right]\delta(C_u, C_v) \\ &= \frac{1}{2W}\sum_{c\in P}\sum_{u,v\in c}\left[w_{uv} - \frac{d_u d_v}{2W}\right] \\ &= \frac{1}{2W}\sum_{c\in P}\left[\sum_{u,v\in c} w_{uv} - \frac{\sum_{u,v\in c} d_u d_v}{2W}\right] \end{aligned} \tag{4-6}$$

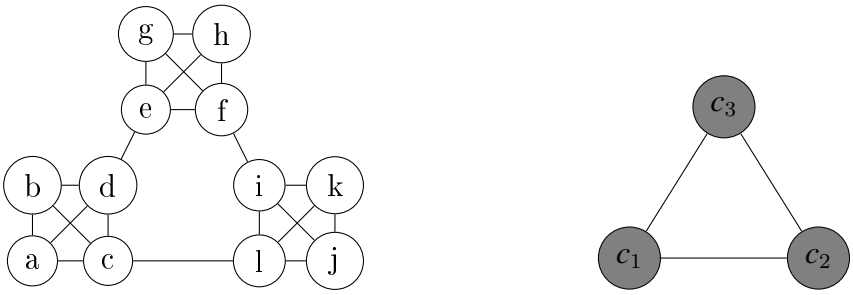
注意到有 $I_c = \sum_{u,v\in c} w_{uv}$, 并且有 $\sum_{u,v\in c} d_u d_v = \sum_{u\in c} d_u \sum_{v\in c} d_v = S_c S_c = S_c^2$。因此，我们有如下的定理：

$$\begin{aligned} Q &= \frac{1}{2m}\sum_{c\in C}\left[I_c - \frac{S_c^2}{2m}\right] \\ &= \sum_{c\in P}\left[\frac{I_c}{2m} - \left(\frac{S_c}{2m}\right)^2\right] \end{aligned} \tag{4-7}$$

□

我们下面给出基于模块度优化的图划分方法。迭代初始化时每个顶点拥有一个单独的 ID，考虑把某个顶点 u 重新放入到某个新的分块 C 后，整个图 G 的模量会发生变化，我们用符号 $\Delta Q_{u,C}$ 表示。根据模量的定义，我们有：

$$\Delta Q_{u,C} = \left[\frac{I_c + w_{u,C}}{2W} - \left(\frac{S_C + d_u}{2W}\right)^2\right] - \left[\frac{I_C}{2W} - \left(\frac{S_C}{2W}\right)^2\right] - \left(\frac{d_u}{2W}\right)^2 \tag{4-8}$$



(a) G (b) 经过塌缩后的图 $G\prime$

图 4-3: (a) 为一个简单的示例图，(b) 为改图经过塌缩 (coarsen) 后的图 $G\prime$, 每个顶点代表 G 中的一个划分

其中，$w_{u,C}$ 指的是顶点 u 指向分块 C 中的所有的边的权重之和。经过若干步骤的化简，最终我们可以得到模块度变化的简单形式:

$$\Delta Q_{u,C} = \frac{w_{u,C}}{2W} - \frac{S_C d_u}{2W^2} \tag{4-9}$$

上式告诉我们，将某个顶点从之前的分块移除并重新放入新的分块之中时，只需考虑图中的局部信息就可以计算出模块度的变化量，而不需要按照式 4-4那样暴力遍历所有边。

因为 Spark 平台内部 RDD 的计算无法按照异步 (asynchronize) 方式生成，即在第 t 轮迭代中计算维护图中顶点的相关信息只能基于第 $t-1$ 轮迭代各顶点已有的信息，每次迭代时所有的顶点一并计算更新。Spark 需要严格同步（synchronize）本轮迭代中所有线程的计算结果，才会进入下一步的迭代。基于这个特性，我们对图进行基于模块度优化的塌缩算法如 4.2所示。算法包含三个参数，α,β,k。其中，α 指的是反复塌缩过程的目标划分数目，如果迭代过程中当前的划分数目大于 α，就继续迭代下去。α 的取值与第二步中单节点图划分算法可以处理的输入图规模有关。β 指的是我们基于模量的塌缩方法的迭代次数。参数 k 控制了图划分过程中每个分块大小的上限 $|V|/k$。如果 k 设置得较大，那么最终每个分块规模会比较小，而分块数目较多；如果 k 设置得较小，那么每个分块规模可能较大，而分块数目较少。首先根据参数 k 求出分块大小的阈值 bs（第 2 行），并且初始时赋予图中每个顶点一个不同的 ID（第 3 行）。算法会一直迭代直到当前的分块数目小于 α（第 4 行），每次迭代过程中都会更新划分 P 并生成新的塌缩图（第 18-19 行）。每次的迭代内部包含一个迭代 β 次的社区发现过程（第 6-17 行）。如果模块度有增益，那么一个顶点会移入到其邻居顶点所属的社区中（第 8-17 行）。注意到如果待加入的那

个社区中包含的顶点权重之和（根据公式4-2，权重之和表示该社区在原输入图 G_0 对应的顶点数目）已经超出了分块阈值 bs，那么该社区不会接受新的顶点加入（第 10 行）。这保证了最终分块的负载均衡，即每一个分块的顶点数目都不会超过 bs。

Algorithm 4.1 Modularity-based Graph Coarsening

```
1: procedure GraphCoarsening(G, α, β, k)
2:     Set block size threshold bs ← |V|/k;
3:     Initialize partition P, which assigns each vertex a unique partition ID;
4:     while |V| ≥ α do                                  ▷ #Paritions greater than α
5:         W ← Σ_{u,v∈V} w_{u,v};
6:         for i = 1 to β do
7:             for u ∈ V do
8:                 maxGain ← −1;
9:                 for v ∈ N_v do
10:                    if |C_v| ≥ bs then          ▷ Partitions with size smaller than bs
11:                        ΔQ_{u,C_v} ← w_{u,C_v}/2W − S_{C_v} d_u/2W²;
12:                        if ΔQ_{u,C_v} > maxGain then
13:                            maxGain ← ΔQ_{u,C_v}
14:                            newPartition ← v
15:                if maxGain > 0 then
16:                    C_newPartition ← C_newPartition ∪ {u};
17:                    C_u ← C_u − {u};
18:        update P
19:        G ← the coarse-grained graph constructed from P
20:    return G.
```

初始划分与恢复步骤

经过反复塌缩后，原图 $G_0(V_0, E_0)$ 变成了规模足够小的图 $G_t(V_t, E_t)$，其中 $|V_t| < \alpha$。此时 G_t 中每个顶点 u 都对应原图 G_0 中的若干紧密连接的顶点集合 C_u，u 的权重对应 $|C_u|$；G_t 中的每条边 (u, v) 都对应 G_0 中顶点集合 C_u 与 C_v 之间的边割，(u, v) 的权重对应边割集中边的数目。因此，对 G_t 的划分 P_t 可以一一对应到对 G_0 的划分 $P^\star$。并且，顶点 $u \in V_t$ 是 P_t 的最小划分单元，而对应到 G_0 中，集合 C_u 才是 $P^\star$ 的最小划分单位。C_u 中的顶点是不可重新进行划分的，因为它们的分块在之前的塌缩过程中就已经由基于模块度的社区发现逻辑决定了。

我们直接使用 METIS 软件包提供的单节点算法求解 G_t 的划分 P_t。具体

求解时，仍然按照之前的目标：即划分的负载均衡以及边割权重的最小化。因为 METIS 只能处理规模较小的图，所以 α 的值应当设置在合理的范围内。

恢复过程比较简单，对 P_t 中的每个顶点 u，直接把其在 G_0 中一一对应的顶点集合 C_u 中的所有顶点的分块号，设置为 u 在 P_t 中的分块号即可。

Algorithm 4.2 Multi-Level Graph Partitioning

```
1: procedure MultiLevelGraphPartition(G, α, β, k)
2:     while #partitions ≥ α do
3:         P ← results of modularity-based community detection for β iterations;
4:         Construct G′ from P;                                  ▷ Coarsen.
5:     P_1 ← Run local graph partitioning on G′.                 ▷ Final refinement.
6:     restore partition P′ from P_1                             ▷ Project back to the orignal graph.
```

基于 Spark 平台的算法的实现

我们主要基于 Spark 提供的分布式图计算框架 GraphX 来实现算法的逻辑。因为图中顶点在计算过程中需要与相邻顶点通过网络交换信息，我们首先给出顶点的定义： 可以看到，Node 这个结构记录了顶点的 ID、所属分块

```
class Node extends Serializable{
    val vID:Long
    val pID:Long
    var weight:Double = 1.0,
    var pWeightSum:Double = 0.0,
    var nbrPWeightSum:Double = 0.0
    var modularity:Double
    ...
}
```

Listing 4.1: My Caption

图 4-4: 简化的 Node 结构体

ID、权重、顶点到所属分区的连接数、顶点到邻居分区的连接数等信息。这些信息在计算过程中会不断的在顶点之间传输、交换、维护。

GraphX 具体的编程范式比较复杂，但是总的来说，算法工作的流程是在 Map 阶段所有顶点向其邻居顶点发送自己包含本身信息的消息，这些消息经过本身包含 (srcID, dstID)，发送至同一目的顶点的若干消息经过 Spark 内部的 Partitioner 之后组装成消息列表准确发送至目的地。目标顶点收到这些消息

后，在本地进行必要的计算就获得了周围顶点的必要信息，然后就可以根据公式4-9计算出 ΔQ 的大小。 其中，*generateEdges* 是塌缩过后重新生成规模更

```
var graph = Graph(vertexRDD, edgeRDD)
while(run) {
    var newVertexRDD = graph.aggregateMessages()
    var tmpGraph = Graph(newVertexRDD, graph.edges)
    var edges = generateEdges(tmpGraph)
    graph = Graph(newVertexRDD, edges)
    num = graph.vertices.filter().count()
    run = if {num == 0} True else False
    ...
}
```

图 4-5: 程序运行逻辑

小的图的过程，其具体细节如下所示： 我们首先统计有哪些边在当前图中横

```
graph.triplets.map{
  t => Edge(t.srcAttr.pID, t.dstAttr.pID, t.attr)
   .flatMap {
       e => {
          ...
          List()
       }
   }.reduceByKey()
    .map{
       e => Edge(new Node(), new Node()) //construct new edge
       }
}
```

图 4-6: 塌缩过程中生成新的边

跨了不同的分块，然后使用 reduceByKey 操作把这些边变成变成一条新的边，这个过程中变得数量大大地减少了，并且新边的权重，以及新边两边端点的权重，都同时被更新了。

实验评估

本章小结

分布式全点对相似度计算方法算法

概述

上一章节描述了计算单源点 SimRank 相似度的计算方法，如果进一步地考虑计算图 G 中全点对的相似度，那么不管是直接根据 SimRank 的定义采用矩阵迭代计算的方法，还是将全点对计算问题简单看做 $|V|$ 个单源点相似度计算问题，整个计算的开销无疑是不可承受的。考虑一个现实生活中的拥有数百万节点的大规模图，其邻接矩阵的元素规模达到万亿级别 ($|V|^2$)，涉及到这么大规模的矩阵乘法需要做复杂的算法设计和程序优化，甚至需要有专门的高性能超算平台来配合程序的运行。如果简单地将全点对相似度计算看做 $|V|$ 个单源点计算问题，那显然没能够充分挖掘出问题内在的可并行性。因此，需要重新设计设计高效的算法来计算图 G 中全点对的相似度。本文总体的算法思路是“分而治之”的思想，试图将复杂的计算任务分解为如果如果更小、更容易处理的子任务。而本章将重点叙述其中的“分解”步骤：图划分，作为一个重要的中间步骤，分割的质量、效率直接关系到最终全点对相似度的计算效果。

相关工作

算法框架

基于章节 4中对图做出的划分，我们现在给出一个计算全点对相似度的近似算法。文献【11】描述了一种基于图中分块结构（Block Structure）的若干性质来加速计算 PageRank 的方法，我们的近似算法直接受到了该算法的启发。对于现实生活中的图数据集，图中天然可以划分为着若干稠密的分块，这些稠密的分块包含了更多节点之间的连接，因此往往也蕴含着最多的人们感兴趣的信息。本文正是充分利用了这一性质，提出了一种计算全点对相似度计算的近似算法。相比原来的 SimRank 的计算方法，该算法不仅极大地提高了计算效率，而且保留了原来的计算精度。

在计算节点对之间的相似性时，节点对可以分为两种：

1. 位于同一稠密分块之中的顶点对。在上一章节中，我们已经详细介绍了将原图 G 划分为若干稠密分块的方法，每一个划分好的分块都被存放在集群中的某一个计算节点中。这种情况下可以2-2中的串行计算方法直接计算出该点对的相似性。
2. 位于不同分块之中的顶点对。这种情况下，注意到因为两个顶点分别处于不同的稠密分块之中，这意味这两个顶点的距离会以较大概率地比较远，也即意味着这两个顶点的相似性会较大概率地在数值上表现得非常之小。这个时候我们通过估算近似地给出最终的相似度。

与上一章节类似，假设图中的每一个分块是一个顶点，那么图的划分就形成了一个塌缩图。基于这个塌缩图，我们首先定义图中边的权重：

定义 5-1 塌缩图中边的权重。假设对图 G 有划分 $P = (C_1, C_2, \ldots, C_n)$，设 $L(C_i)$ 表示分块 C_i 中所有边的数量，包括该分块内部的边以及连接到其他分块的边; $L(C_i, C_j)$ 表示分块 P_i 与分块 P_j 中相连接的边的数量。设图 G' 为划分 P 形成的塌缩图，$v_i \in G', v_j \in G'$, 且 v_i 为分块 C_i 对应的顶点，v_j 为分块 C_j 对应的顶点。那么定义边 (v_i, v_j) 的权重为：

$$w(v_i, v_j) = \frac{L(C_i, C_j)}{L(C_i)} \tag{5-1}$$

v_i 自己到自己的权重定义为

$$w(v_i, v_i) = 1 - \sum_{v_j \in O(v_i)} w(v_i, v_j) \tag{5-2}$$

显然，一般情况下 $w(v_i, v_j)$ 与 $w(v_j, v_i)$ 是不同的。

这样经过塌缩后的图 G' 变成了一个加权有向图。边 (v_i, v_j) 的权重可以粗略理解为某个随机游走者从点 v_i “逃逸” 到点 v_j 的概率。$w(v_i, v_j)$ 越大，说明在原图 G 中分块 C_i 与 C_j 的交叉边越多，联系越紧密。

基于以上的定义，我们给出图 G 中各个分块之间的相似性定义：

定义 5-2 原图 G 有划分 $P = (C_1, C_2, \ldots, C_n)$，定义分块 (C_i, C_j) 之间的相似性为塌缩图 G' 中顶点 v_i, v_j 之间的相似性。

我们下面给出加权有向图的顶点相似性计算方法。为了将边的权重引入到SimRank 模型中，我们使用如下的公式迭代公式计算相似性：

$$s_{k+1}(u,v) = c \cdot evidence(u,v) \sum_{u' \in I(u), v' \in I(v)} s_k(u',v') W(u',u) W(v',v) \tag{5-3}$$

其中，c 是衰减系数，并且

$$evidence(u,v) = \sum_{i=1}^{|I(u) \cap I(v)|} 2^{-i} \tag{5-4}$$

$$W(u',u) = w(u',u) \cdot \exp(-Var(\{w(u',u) \mid u \in O(u')\}) \tag{5-5}$$

如果写成矩阵的形式，设 V 为 evidence 矩阵，满足 $V(i,j) = evidence(i,j)$，那么整个加权有向图的 SimRank 相似性计算如下：

Algorithm 5.1 Weighted All-pair SimRank:wapSimRank

```
1: procedure WeightedSimRank(G', u, c, k)
2:     N ← |V|;
3:     S ← identity matrix I_N;
4:     for i = 1 to k do
5:         T ← c · W^T · S · W;
6:         S ← T + I_N − D(diag(T))          ▷ D is the diagnal matrix.;
7:     S ← element-wise multiplication of V and S
8:     return S.
```

定义 5-3　不同分块间点对的相似性。如果点对 (u,v) 的两个顶点分别属于图的不同分块 C_i 和 C_j 之中，并且有分块 C_i 与 C_j 的相似性为 $s_{block}(C_i, C_j)$，我们使用下式来估计其相似性：

$$s(u,v) = s(C_i,u) \cdot s_{block}(C_i,C_j) \cdot s(C_j,v) \tag{5-6}$$

其中，$s(C_i,u)$ 可以看做顶点 u 与其分块的中心点间的相似度，由下式近似给出：

$$s(C_i,u) = \frac{1}{|C_i|} \sum_{q \in C_i} s(u,q) \tag{5-7}$$

即我们使用 u 与所以与其属于同一个分块的顶点对额相似性的均值来估计 u 到与分块中心的相似度。

显然，如果由算法5.1计算出的分块间的相似度是随着迭代次数收敛的，并且分块内点对的相似度计算过程也是收敛的，那么由以上式子定义出的分块间点对的相似度也是随着迭代次数收敛的。因此可以分开计算分块之间的相似度与分块内点对的相似度，最后再基于它们计算分块间点对的相似度。综上，我们的算法步骤主要如下:

1. 首先根据上一章节描述的方法，对原图 G 做划分，得到若干个分块 $P = (C_1, C_2, \ldots, C_n)$;
2. 将各个分块当做一个顶点进行塌缩得到一个加权有向图 G'，在 G' 运行适用于加权图的 SimRank 算法，得到分块之间的相似度;
3. 对任一分块 C_i 直接运行串行的全点对 SimRank 算法，得到分块内部的节点对相似度;
4. 使用公式5-6估计属于不同分块顶点之间的相似度;
5. 最后返回全局相似度。

我们综合分析算法的复杂度。simRank 以及其适用于加权图的变种的时间复杂度都为 $O(kd^2n^2)$。假设图 G 已经被划分为 m 个分块，则每个分块的大小在 $O(n/m)$ 左右。若塌缩图 G' 的顶点平均度数为 d_1，那么第 2 步的时间复杂度为 $O(kd_1{}^2m^2)$。第 3 步的时间复杂度为 $O(mkd_2{}^2(n/m)^2)$，即 $O(kd_2{}^2n^2/m)$，其中 d_2 为每个分块中节点的平均度数。第 4 步计算全局相似度时，因为要考虑所有点对，而根据5-6，平摊之后每对顶点对只需常数级别的查询计算时间，因此总的时间复杂度为 $O(n^2)$。如果我们考虑主要的计算过程，即第 2 步和第 3 步，所需的时间为 $O(kd_1{}^2m^2 + kd_2{}^2n^2/m)$。若假设有 $d_1 = d_2 = d$，则时间复杂度为 $(kd^2(m^2 + n^2/m))$。将该复杂度看做变量 m 的函数，则函数在 $m = (2n)^{2/3}/2$ 处有极值，这对我们选择合适的 m 有指导意义。进一步的，考虑分布式环境下第 3 步主要由多个计算节点的多个线程同时完成，不妨设总线程数为 T，则该函数变为 $f(m) = kd^2(m^2 + n^2/(Tm)$，其在 $m = (n^2/(2T))^{1/3}$ 处有极值。对于一个节点数在 $1e6$ 级别的图来说，如果 T 的规模在 $1e2$ 量级，则计算下来的极值点量级在 $1e3$ 左右。当然，不同的数块数 m 会影响计算结果的精度，因为分块数越多，意味着由估算得到的相似度越多，结果的精度越差。实际的 m 取值，需要综合考虑各种因素再加以决定。

基于 Spark 平台的算法实现

首先假设输入图已经由上一章节描述的图分割算法处理过，每个节点 u 都附带一个其属于的分块 ID 号 C_u。即每个节点的实际使用一个二元组 $(nodeID, PartitionID)$ 表示。当输入数据从分布式文件系统读入时，我们以 ACM Computing Classification System(CCS) 中的数据集为例。该数据集是一个揭示计算机科学方向科研论文分类系统。在 CCS 数据集第 F 大类中，总共有 6 个类别。我们把每一篇论文当做一个顶点，论文之间的引用当做边，那么整个数据集就可以抽象为一个图。

定义 5-4　对图 G 中的两个划分 P_1, P_2，

实验评估

本章小结

总结与展望

致　　谢

白驹过隙间，我的三年研究生就要结束，我在南京大学七年的学习生活就要画上句号。在这即将离别校园之际，回想三年的校园生活，很多情景都历历在目。研究生阶段，我的科研能力得到了提升，生活上也变得更加独立。在这里我衷心感谢我的导师、师兄弟、同学、朋友们，你们平时对我的教导、支持、帮助让我愉快地走完了这平凡的三年，感谢你们!

首先感谢我的导师唐杰。唐老师为人谦和，工作负责，在学术上对我们都尽心尽力地指导，在生活上对我们也多有照顾。无论是本课题的研究及论文的撰写，还是平时对我科研能力的塑造，唐老师都以身作则，对我提供了很多帮助。唐老师宽厚谦和的生活作风也潜移默化地影响着我，指导着我做一个脚踏实地的南大人。

我还要感谢武钢山老师。武老师是多媒体教研室主任，他作风严瑾，要求严格，把握着最新的科研动向，同时管理着整个大组的研究学习工作。在此特别感谢武老师对我学习和生活上的种种帮助。

还要感谢实验室的师兄师弟以及师妹们。三年以来你们给了我很多的关心和支持，很高兴能和你们同门，你们让我留下了很多美好的回忆。祝愿两位老师身体健康，万事如意！也祝各位同学学业有成，科研顺利！最后感谢我的家人，他们永远是我最大的支柱和依靠!

参考文献

[1] NEWMAN M, BARABÁSI A-L, WATTS D J. The structure and dynamics of networks[M]. Princeton : Princeton University Press, 2006.

[2] NEWMAN M E, STROGATZ S H, WATTS D J. Random graphs with arbitrary degree distributions and their applications[J]. Physical Review E, 2001, 64(2) : 026118.

[3] AIELLO W, CHUNG F, LU L. A random graph model for massive graphs[C] // Proceedings of the thirty-second annual ACM symposium on Theory of computing. New York : ACM, 2000 : 171 – 180.

[4] BOLLOBÁS B. Random graphs : Vol 73[M]. [S.l.] : Cambridge university press, 2001.

[5] BARABÁSI A-L, ALBERT R. Emergence of scaling in random networks[J]. science, 1999, 286(5439) : 509 – 512.

[6] ERDŐS P, RENYI A. On the strength of connectedness of a random graph[J]. Acta Mathematica Hungarica, 1961, 12(1) : 261 – 267.

[7] NVIDIA Corporation. Nvidia GRID[EB/OL]. 2013 [2013-03-10]. http://www.nvidia.com/object/cloud-gaming.html.

[8] MARCONI G. Wireless telegraphic communication[G] // Nobel Prize address. 1909.

[9] BARABÁSI A-L. Linked: How Everything is Connected to Everything Else and What It Means for Business, Science, and Everyday Life[M]. New York : Plume, 2003.

[10] de Sola Pool I, KOCHEN M. Contacts and influence[J]. Social Networks, 1978, 1 : 5 – 51.

[11] LUCE R D, PERRY A D. A method of matrix analysis of group structure[J/OL]. Psychometrika, 1949, 14(2): 95 – 116.
http://dx.doi.org/10.1007/BF02289146.

[12] WATTS D, STROGATZ S. Collective dynamics of "small-world" networks[J/OL]. Nature, 1998, 393: 440 – 442.
http://dx.doi.org/10.1038/30918.

[13] GYARMATI L, Trinh T. Scafida: A scale-free network inspired data center architecture[J]. ACM SIGCOMM Computer Communication Review, 2010, 40(5): 4 – 12.

[14] MOLLOY M, REED B. A critical point for random graphs with a given degree sequence[J]. Random structures & algorithms, 1995, 6(2-3): 161 – 180.

[15] FRIEZE A M, ŁUCZAK T. On the independence and chromatic numbers of random regular graphs[J]. Journal of Combinatorial Theory, Series B, 1992, 54(1): 123 – 132.

[16] BENDER E A, CANFIELD E R. The asymptotic number of labeled graphs with given degree sequences[J]. Journal of Combinatorial Theory, Series A, 1978, 24(3): 296 – 307.

[17] ERDÖS P, RÉNYI A. Additive properties of random sequences of positive integers[J]. Acta Arithmetica, 1960, 6(1): 83 – 110.

[18] VÁZQUEZ A, FLAMMINI A, MARITAN A, et al. Modeling of protein interaction networks[J]. Complexus, 2002, 1(1): 38 – 44.

[19] SOLÉ R V, PASTOR-SATORRAS R, SMITH E, et al. A model of large-scale proteome evolution[J]. Advances in Complex Systems, 2002, 5(01): 43 – 54.

[20] GOH K-I, OH E, JEONG H, et al. Classification of scale-free networks[J]. Proceedings of the National Academy of Sciences, 2002, 99(20): 12583 – 12588.

[21] ANTHONISSE J M. The rush in a graph[J]. Amsterdam: University of Amsterdam Mathematical Centre, 1971.

[22] FREEMAN L C. A set of measures of centrality based on betweenness[J]. Sociometry, 1977 : 35 – 41.

[23] GOH K-I, KAHNG B, KIM D. Universal behavior of load distribution in scale-free networks[J]. Physical Review Letters, 2001, 87(27) : 278701.

[24] ADAMIC L A, HUBERMAN B A. Power-law distribution of the world wide web[J]. Science, 2000, 287(5461) : 2115 – 2115.

[25] BIANCONI G, BARABÁSI A-L. Competition and multiscaling in evolving networks[J]. EPL (Europhysics Letters), 2001, 54(4) : 436.

[26] KRAPIVSKY P L, REDNER S, LEYVRAZ F. Connectivity of growing random networks[J]. Physical review letters, 2000, 85(21) : 4629.

[27] DOROGOVTSEV S N, MENDES J F F, SAMUKHIN A N. Structure of growing networks with preferential linking[J]. Physical Review Letters, 2000, 85(21) : 4633.

[28] BARABÁSI A-L, ALBERT R, JEONG H. Scale-free characteristics of random networks: the topology of the world-wide web[J]. Physica A: Statistical Mechanics and its Applications, 2000, 281(1) : 69 – 77.

[29] PRICE D. Statistical studies of networks of scientific papers[C] // Statistical Association Methods for Mechanized Documentation: Symposium Proceedings : Vol 269. 1965 : 187.

[30] REDNER S. How popular is your paper? An empirical study of the citation distribution[J]. The European Physical Journal B-Condensed Matter and Complex Systems, 1998, 4(2) : 131 – 134.

[31] MERTON R K. The Matthew effect in science[J]. Science, 1968, 159(3810) : 56 – 63.

[32] GILBERT E N. Random Graphs[J/OL]. Annals of Mathematical Statistics, 1959, 30(4): 1141 – 1144.
http://dx.doi.org/10.1214/aoms/1177706098.

[33] ERDŐS P, RÉNYI A. On Random Graphs. I[J/OL]. Publicationes Mathematicae, 1959, 6: 290 – 297.
http://www.renyi.hu/~p_erdos/1959-11.pdf.

[34] LI L, ALDERSON D, DOYLE J C, et al. Towards a theory of scale-free graphs: Definition, properties, and implications[J]. Internet Mathematics, 2005, 2(4): 431 – 523.

[35] SOLOMONOFF R, RAPOPORT A. Connectivity of random nets[J/OL]. The bulletin of mathematical biophysics, 1951, 13(2): 107 – 117.
http://dx.doi.org/10.1007/BF02478357.

[36] ALBERT R, JEONG H, BARABÁSI A-L. The diameter of the world wide web[J/OL], 1999, 401: 130 – 131.
http://dx.doi.org/10.1038/43601.

[37] WANG X F, CHEN G. Complex networks: small-world, scale-free and beyond[J/OL]. Circuits and Systems Magazine, IEEE, 2003, 3(1): 6 – 20.
http://dx.doi.org/10.1109/MCAS.2003.1228503.

[38] LESKOVEC J, HORVITZ E. Planetary-Scale Views on a Large Instant-Messaging Network[C] //Proceedings of the 17th international conference on World Wide Web. New York: ACM, 2008.

[39] LESKOVEC J, HORVITZ E. Worldwide Buzz: Planetary-Scale Views on an Instant-Messaging Network[R]. [S.l.]: Microsoft Research, 2007.

[40] FASS C, TURTLE B, GINELLI M. Six Degrees of Kevin Bacon[M]. New York City: Plume, 1996.

[41] REYNOLDS P. The Oracle of Bacon[EB/OL]. [2011-07-12].
http://oracleofbacon.org/cgi-bin/center-cgi?who=Kevin+Bacon.

[42] NEWMAN M. The structure of scientific collaboration networks[J/OL]. Proceedings of the National Academy of Sciences, 2001, 98(2): 404 – 409. http://dx.doi.org/10.1073/pnas.98.2.404.

[43] GROSSMAN J. Publications of Paul Erdős[EB/OL]. [2011-02-01]. http://www.oakland.edu/enp/pubinfo/.

[44] UNIVERSITY O. The Erdős Number Project Data Files[EB/OL]. 2009 [2010-05-29]. http://www.oakland.edu/enp/thedata/.

[45] GOFFMAN C. And what is your Erdős number?[J]. American Mathematical Monthly, 1969, 76(7): 791.

[46] MILGRAM S. The Small World Problem[J]. Psychology Today, 1967, 2: 60 – 67.

[47] TRAVERS J, MILGRAM S. An Experimental Study of the Small World Problem[J]. Sociometry, 1969, 32(4): 425 – 443.

[48] GUREVICH M. The Social Structure of Acquaintanceship Networks[D]. Cambridge, MA: Massachusetts Institute of Technology, 1961.

[49] HERRERA C, ZUFIRIA P J. Generating scale-free networks with adjustable clustering coefficient via random walks[C] //Network Science Workshop (NSW), 2011 IEEE. 2011: 167 – 172.

[50] GREENBERG A, LAHIRI P, MALTZ D A, et al. Towards a next generation data center architecture: scalability and commoditization[C] //Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow. 2008: 57 – 62.

[51] LEIGHTON F T. Introduction to parallel algorithms and architectures[M]. [S.l.]: Morgan Kaufmann San Francisco, 1992.

[52] GUO C, LU G, LI D, et al. BCube: a high performance, server-centric network architecture for modular data centers[J]. ACM SIGCOMM Computer Communication Review, 2009, 39(4): 63 – 74.

[53] GUO C, WU H, TAN K, et al. Dcell: a scalable and fault-tolerant network structure for data centers[C] //ACM SIGCOMM Computer Communication Review: Vol 38. 2008: 75–86.

[54] LOCKWOOD J W, MCKEOWN N, WATSON G, et al. NetFPGA–an open platform for gigabit-rate network switching and routing[C] //Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on. 2007: 160–161.

[55] SourceForge. VDE: Virtual Distributed Ethernet[CP/OL]. 2013 [2013-03-09]. https://vde.sourceforge.net.

[56] Quagga Routing Suite[EB/OL]. [2013-03-09]. http://www.nongnu.org/quagga/.

[57] KOHLER E, MORRIS R, CHEN B, et al. The Click modular router[J]. ACM Transactions on Computer Systems (TOCS), 2000, 18(3): 263–297.

[58] HAN S, JANG K, PARK K, et al. PacketShader: a GPU-accelerated software router[J]. ACM SIGCOMM Computer Communication Review, 2010, 40(4): 195–206.

[59] DOBRESCU M, EGI N, ARGYRAKI K, et al. RouteBricks: exploiting parallelism to scale software routers[C] //Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 15–28.

[60] HU Fei, QIU Meikang, LI Jiayin, et al. A Review on cloud computing: Design challenges in Architecture and Security[J]. Journal of Computing and Information Technology, 2011, 19(1): 25–55.

[61] VARIA J. Architecting for the cloud: Best practices[R]. [S.l.]: Amazon Web Services, 2010.

[62] GREENBERG A, HAMILTON J R, JAIN N, et al. VL2: a scalable and flexible data center network[C] //ACM SIGCOMM Computer Communication Review: Vol 39. 2009: 51–62.

[63] AL-FARES M, LOUKISSAS A, VAHDAT A. A scalable, commodity data center network architecture[C] //ACM SIGCOMM Computer Communication Review: Vol 38. 2008: 63–74.

[64] PALLIS G, VAKALI A. Insight and perspectives for content delivery networks[J]. Communications of the ACM, 2006, 49(1): 101–106.

[65] DEAN J. Designs, Lessons, and Advice from Building Large Distributed Systems.[J]. Keynote from LADIS 2009, 2009.

[66] BRODER A, KUMAR R, MAGHOUL F, et al. Graph structure in the web[J]. Computer networks, 2000, 33(1): 309–320.

[67] KLEINBERG J M. Authoritative sources in a hyperlinked environment[J]. Journal of the ACM (JACM), 1999, 46(5): 604–632.

[68] KLEINBERG J. The Small-World Phenomenon: An Algorithmic Perspective[C] //in Proceedings of the 32nd ACM Symposium on Theory of Computing. 2000: 163–170.

[69] MILGRAM S. The Small World Problem[J]. Psychology Today, 1967, 2: 60–67.

[70] Wikipedia contributors. Moore's law[EB/OL]. Wikipedia, The Free Encyclopedia, 2015 (2015/06/14) [2015/06/15]. https://en.wikipedia.org/wiki/Moore%27s_law.

[71] DUBASH M. Moore's Law is dead, says Gordon Moore[EB/OL]. Techworld, 2010 (2010/4/13) [2015/6/16]. http://www.techworld.com/news/operating-systems/moores-law-is-dead-says-gordon-moore-3576581/.

[72] KANELLOS M. Intel scientists find wall for Moore's Law[EB/OL]. CNET, 2003 (2003/12/1) [2015/6/16]. http://news.cnet.com/2100-1008-5112061.html.

[73] Intel Corporation. Intel discloses newest microarchitecture and 14 nanometer manufacturing process technical details[EB/OL]. Intel Corporation, 2014

(2014/8/11) [2015/6/16].
http://newsroom.intel.com/community/intel_newsroom/blog/2014/08/11/intel-discloses-newest-microarchitecture-and-14-nanometer-manufacturing-process-technical-details.

[74] MOAMMER K. TSMC Launching 10nm FinFET Process In 2016, 7nm In 2017[EB/OL]. WCCF Tech, 2015 (2015/4/19) [2015/6/16].
http://wccftech.com/tsmc-promises-10nm-production-2016-7nm-2017/.

[75] SHROUT R. Intel Xeon E5-2600 v3 processor overview: Haswell-EP up to 18 cores[EB/OL]. 2014 (2014/9/8) [2015/6/16].
http://www.pcper.com/reviews/Processors/Intel-Xeon-E5-2600-v3-Processor-Overview-Haswell-EP-18-Cores.

[76] Intel Corporation. Intel chips timeline[EB/OL]. Intel Corporation, 2012 (2012/7/13) [2015/6/16].
http://www.intel.co.uk/content/www/uk/en/history/history-intel-chips-timeline-poster.html.

[77] Wikipedia contributors. Transistor count[EB/OL]. Wikipedia, The Free Encyclopedia, 2015 (2015/6/10) [2015/6/16].
https://en.wikipedia.org/wiki/Transistor_count.

简历与科研成果

基本信息

韦小宝，男，汉族，1985 年 11 月出生，江苏省扬州人。

教育背景

2007 年 9 月 — 2010 年 6 月 南京大学计算机科学与技术系 硕士
2003 年 9 月 — 2007 年 6 月 南京大学计算机科学与技术系 本科

攻读硕士学位期间完成的学术成果

1. Xiaobao Wei, Jinnan Chen, "Voting-on-Grid Clustering for Secure Localization in Wireless Sensor Networks," in Proc. IEEE International Conference on Communications (ICC) 2010, May. 2010.
2. Xiaobao Wei, Shiba Mao, Jinnan Chen, "Protecting Source Location Privacy in Wireless Sensor Networks with Data Aggregation," in Proc. 6th International Conference on Ubiquitous Intelligence and Computing (UIC) 2009, Oct. 2009.

攻读硕士学位期间参与的科研课题

1. 国家自然科学基金面上项目 ‘‘无线传感器网络在知识获取过程中的若干安全问题研究” （课题年限 2010 年 1 月 — 2012 年 12 月），负责位置相关安全问题的研究。
2. 江苏省知识创新工程重要方向项目下属课题 ‘‘下一代移动通信安全机制研究” （课题年限 2010 年 1 月 — 2010 年 12 月），负责 LTE/SAE 认证相关的安全问题研究。