中国科学技术大学

博士学位论文

支持快速随机游走的
大规模图分析系统研究

作者姓名： 汪睿

学科专业： 计算机系统架构

导师姓名： 许胤龙教授 李永坤副教授

完成时间： 二〇二一年五月十九日

University of Science and Technology of China

A dissertation for doctor's degree

Research on Large Scale Graph Analytic System for Supporting Fast Random Walks

Author: Rui Wang

Speciality: Computer System Architecture

Supervisors: Prof. Yinlong Xu, Assoc. Prof. Yongkun Li

Finished time: May 19, 2021

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

随着大数据时代的到来，数据规模迅猛增长，数据之间也产生着日益复杂的交互关系。挖掘出数据及其关联关系间蕴藏的丰富价值，可以为人们的日常生活提供非常多的有效信息。图结构能很好地表达真实世界中实体之间的关联关系，图分析可以充分挖掘这些关联关系中的隐藏信息，因此图和图分析被广泛应用于众多服务场景。但随着图数据规模的不断增大，给大图上的图分析计算带来挑战。一方面，大图上的遍历分析因为运算量大而难以高效运行，此时基于随机游走的采样方法可以实现可靠的近似计算。随机游走因其高效的计算效率和坚实的理论保障，经常被应用于一些重要的图分析、排序和嵌入式算法。另一方面，大规模的图数据放不下单机的内存，此时基于磁盘的单机图处理系统通过对图数据的存储和 I/O 管理，可以实现磁盘驻留图上的高效分析计算。

然而现有的图处理系统在支持随机游走类应用时存在一定的局限性：（1）在图数据的存储管理方面，现有的图处理系统往往采用一种基于迭代的模型，顺序迭代地从磁盘加载子图数据进入内存计算，从而缓解对磁盘的大量随机 I/O 造成的性能瓶颈。但这种基于迭代的模型在支持基于随机游走类的图计算时表现出很低的 I/O 效率，从而限制了图上随机游走的效率和扩展性。（2）在随机游走的更新计算方面，现有工作中采用的基于迭代的同步计算模型，为了保证图上所有随机游走之间的同步更新，牺牲了随机游走的计算效率。另外，现有的图处理系统中对随机游走数据的存储管理往往采用以边或顶点为中心的索引机制和基于大量动态数组的存储方案。这样的随机游走的存储管理方式不仅带来非常大的索引开销，而且大量的动态数组也会频繁地重新分配内存，带来内存碎片和额外的时间开销。因此限制了图处理系统可以处理的图数据和并发随机游走的规模。（3）在随机游走算法设计方面，基于随机游走的图采样需要在随机游走收敛之后才能收集样本从而实现无偏估计。而现有的随机游走算法的收敛速度都很慢，往往需要成千上万步随机游走才能收敛，严重影响了采样效率。

本文聚焦于图上的随机游走类应用场景，针对上述的三个层面的问题，分别研究了随机游走友好的图存储管理、随机游走的存储计算框架和快速收敛的随机游走算法优化，最终结合这三个部分的研究内容实现了一个完整的支持快速随机游走的图分析系统。本文的主要研究内容和贡献如下。

（1）随机游走友好的图存储管理系统

为了实现随机游走友好的图存储管理，我们结合随机游走类应用的访存特征，提出一种基于随机游走状态感知的 I/O 模型，并设计实现了面向大规模并发随机游走的高效图数据存储管理机制 SAStore 。SAStore 根据图上随机游走的

数量、步长以及随机游走在图中的分布情况等状态信息，来执行不同的图数据的组织划分、加载和缓存策略，从而实现 I/O 效率的最大化。此外针对动态图处理的场景下更新效率慢的问题，SAStore 进一步提出一种基于分块日志的 CSR 存储实现快速的图更新。在真实世界图数据集上的实验评估表明，SAStore 对比最新的单机随机游走图系统 DrunkardMob，平均可以提升 *2* 倍到 *4* 倍的 I/O 效率。另外在动态图场景下，基于分块日志的 CSR 存储管理方案，对比于基础的 CSR 存储格式，在保证图数据查询效率的同时显著提升了图数据的更新效率。

（2）支持快速随机游走的图计算框架

为了实现支持快速随机游走的图计算框架，我们首先提出一种异步的随机游走更新策略来加速随机游走更新速率。然后提出以子图为中心的随机游走数据的索引策略，减少索引开销；并采用定长缓冲区来存储每个子图包含的随机游走数据，避免大量动态数组带来频繁内存重分配。最后，在前面提出图存储管理系统 SAStore 之上，实现了一个支持快速随机游走的原型系统 GraphWalker，实验结果表明，GraphWalker 可以高效地处理由数十亿个顶点和数千亿条边组成的非常庞大的磁盘驻留图和并发运行的数百亿条、数千步长的随机游走。对比于最新的单机随机游走图系统 DrunkardMob，GraphWalker 的处理速度快一个数量级；对比于最新的分布式随机游走图系统 KnightKing，GraphWalker 在一台机器上的处理速度可以达到其 8 台机器上的速度；对比于支持随机游走性能最好的通用单机图处理系统 Graphene 和 GraFSoft，GraphWalker 的速度提升也非常明显，在最好的情况下，速度也快一个数量级。

（3）快速收敛的随机游走算法优化

随机游走收敛慢的一个很重要的原因是他可能经常被困在某个局部的子图，需要很多步才能走出去，拖慢了随机游走对全局图数据的探索进程和收敛速度。基于上述分析，我们设计了一种非回溯的公共邻居感知的随机游走算法 NB-CNARW，利用随机游走的历史信息以及各个候选顶点的领域信息来实现加权随机游走，从而加速随机游走的收敛，并提高随机游走的采样效率；实验结果表明，NB-CNARW 算法相较于当前最新的随机游走算法，可以显著提升随机游走的收敛速度，收敛所需要的步数最多可以减少 29.2%；在使用 NB-CNARW 算法进行基于图采样的无偏估计时，在给定相同估算精度的要求下，可以降低高达 25.4% 的采样所需的查询成本。

关键词：随机游走；图采样；图处理系统；存储管理；I/O 调度；计算模型

ABSTRACT

With the advent of the era of big data, the scale of data has grown rapidly, and meanwhile the interaction between data has become increasingly complex. Mining the rich value of data and its correlation can provide a lot of valueable information for our daily life. Graph can naturally express the associated relationship between entities in the real world, and graph analysis can fully explore the hidden information in these associated relations. Therefore, graphs and graph analysis are widely used in many applications. However, the increasing scale of the graph data brings challenges to graph analysis. On the one hand, the traversal analysis on the large graph is difficult to run efficiently, so the sampling method based on random walk can achieve reliable approximate calculation. Due to its high computational efficiency and solid theoretical support, random walk is often used in some important graph analysis, sorting and embedded algorithms. On the other hand, large scale graph data can not fit the memory of a single machine, so many disk-based single-machine graph processing systems were proposed to store the large scale graph data in disk, and processed graph analysis via disk I/Os.

However, existing graph processing systems have some limitations when supporting random walk based applications: (1) At the level of graph data storage, existing graph processing systems usually adopt an iterative model, which successively and iteratively loads graph data blocks from disk into memory for computational analysis, thereby alleviating performance bottlenecks caused by heavy random I/O to the disk. However, this iteration-based model shows very low I/O efficiency when supporting random walks. Thus, the efficiency and scalability of the random walks on the graph are limited. (2) At the level of random walk calculation, the iteration-based synchronous compute model used in the existing work, sacrificed the computational efficiency of the random walk in order to ensure the synchronism between all walks. In addition, the storage management of random walk data used in the existing graph processing systems usually adopt an edge/vertex-centric indexing strategy and dynamic arrays based storage method. These methods not only bring a very large indexing overhead, but also frequent memory re-allocation cost, which resulting to massive memory fragmentations and additional time overhead. Therefore, the scale of graphs and concurrent random walks that can be processed by the graph processing systems are both limited. (3) At the level of random walk algorithm, random walk based graph sampling algorithms need to collect samples after the random walk converges to the stationary distribution, in order to

achieve unbiased estimation. However, existing random walk algorithms usually require tens of thousands of walk steps to converge to the stationary distribution, which seriously affects the sampling efficiency.

This dissertation focuses on the random walk based applications, and aims for the three levels of problems stated above, respectively studies the random walk friendly graph storage management, random walk computing framework and random walk algorithm with fast convergence rate. Finally, we implementation a complete graph analysis system that supporting fast random walks. The main research contents and contributions of this dissertation are state as follows.

(1) Random walk friendly graph storage management system.

Combining with the characteristics of random walk based applications, we propose a walk-state-aware I/O model. Based on this I/O model, we design and implement an efficient graph data storage management mechanism SAStore for supporting large scale concurrent random walks. According to the number of random walks, the walk steps and the distribution of the walks in the graph, SAStore adopts different graph partitioning, loading and caching strategies to maximize the I/O efficiency. In addition, to solve the problem of slow update efficiency in the scenario of dynamic graph processing, SAStore further proposes a new blocked log method based on CSR to enable fast graph updates. Experimental evaluations on real world graphs show that, compared with the state-of-the-art random walk graph processing system Drunkardmob, SAStore can improve the I/O efficiency *by 2 times* to *4 times* on average. In the scenario of dynamic graph, blocked CSR with edge log management significantly improve the efficiency of dynamic graph updates compared with the basic CSR storage format.

(2) Fast random walk computing framework.

In order to realize a graph computing framework supporting for fast random walks, we firstly propose an asynchronous random walk updating strategy to speed up the walk updating rate. Then we propose a block-centric walk indexing strategy, and use a fixed-length buffer to store all walks of each subgraph, thus to avoid frequent memory reallocation caused by a large number of dynamic arrays. Finally, we implement a prototype system GraphWalker that supports fast random walks. The experiment results show that GraphWalker can efficiently handle very large graphs with billions of vertices and hundreds of billions of edges, and GraphWalker can also process massive concurrent random walks, i.e., tens of billions of random walks with thousands of steps. GraphWalker can achieve more than an order of magnitude speedup compared with

the random-walkspecific system DrunkardMob, as well as two stateof-the-art single-machine graph systems, Graphene and GraFSoft. Furthermore, GraphWalker is more resource friendly as its performance is even comparable with the state-of-the-art distributed random walk system KnightKing running on a cluster of eight machines.

(3) Random walk algorithm with fast convergence rate.

A very important reason for the the slow convergence rate of random walk algorithm is that random walk may often be stuck in a local subgraph, and may take many steps to walk out, thus retard the random walk to fast explore the whole graph and slow down the convergence rate. Therefore, we design a non-backtracking commom neighbor aware random walk algorithm NB-CNARW, to use the historical information of the random walk and the domain information of each candidate vertex to realize the weighted random walk, thus to accelerate the convergence rate of the random walk and improve the sampling efficiency of the random walk. Experimental results show that the proposed NB-CNARW algorithm is better than the latest random walk algorithm. It can significantly improve the convergence speed of the random walk, and the number of steps required for convergence can be reduced by 29.2% at most. When using NB-CNARW algorithm for unbiased estimation based on graph sampling, given the same estimation accuracy, the query cost of sampling can be reduced by up to 25.4%.

Key Words: Random Walk; Graph Sampling; Graph Process System; Storage Management; I/O Schedule; Compute Model

目　　录

插 图 清 单

表 格 清 单

第 1 章　绪　　论

本章摘要: 大数据时代的到来，使得数据对全球的政治、经济、信息安全和社会的稳定等方面都产生重要的影响。随着互联网、社交网络、物联网、暗网等平台的不断涌现，用户在这些网络平台进行着频繁且错综复杂的交互行为，使得数据之间也产生复杂的关联关系。图结构数据因其可以很好地表达真实世界中实体之间的关联关系，因而被广泛应用于这些网络平台。随着图数据的规模不断增长，很多迭代遍历的图分析算法由于过高的计算复杂度而难以实现。随机游走是图数据分析和机器学习中一个重要的分析工具，可以利用图中顶点之间的集成路径提取信息。随机游走因其高效的计算效率经常被应用于一些重要的图分析、排序和嵌入式算法。本文聚焦于随机游走这一类图分析算法，首先结合随机游走算法的特征，从系统层面考虑现有图分析系统在处理随机游走应用时存在的问题，并针对现有方案存在的不足分别从图数据的存储管理与 I/O 模型、随机游走数据的存储与计算方式两个方面进行系统优化，最终实现了原型系统。然后本文从一个具体的基于随机游走的应用场景（图采样问题）出发，进一步研究相应的算法优化。本章首先介绍了基于随机游走的图分析应用；然后介绍了大规模图分析系统的研究现状以及现有工作在支持随机游走类应用时存在的不足；最后介绍了本文的主要研究内容以及本文的组织结构。

1.1　图分析简介

本节首先介绍什么是图（graph）结构数据、图如何表达数据之间的关联关系，以及图分析能解决的实际问题，然后引出由于数据规模的不断增长带来的大规模图分析的挑战。

1.1.1　图和图表达

信息时代的迅速发展和互联网技术下各类网络平台的普及，既带来了数据规模的迅猛增长，又加快了用户之间错综复杂的数据交互，使得这些庞大的数据之间存在着复杂的关联关系。这些数据不但本身中蕴含着丰富的价值，数据之间的关联关系同样可以挖掘出很多有效信息。**图结构刻画顶点之间通过边的连接关系，可以自然直观地表达真实世界中实体之间的联系。**例如，图 1.1展示了一个电商场景下图结构数据表达的示例，图中的顶点表达各种不同类型的实体，如消费者、商品、卖家店铺等，图中各个顶点之间的连接边表达实体之间的各种关联关系，如浏览、收藏、购买、评论等。相比较于适用于表达结构化数据的关

系型表结构、以及适用于完全非结构化数据的键值存储结构，图结构是描述现实世界中大量存在的半结构化数据的理想的数据结构。因此图在现实生活的许多领域都被广泛应用，比如网页链接[1-3]、社交网络[4-5]、导航系统[6-8]和推荐系统[9-10]等[11-13]。近年来图数据成为学术界和工业界共同关注的研究热点之一。

图 1.1　电商场景下的图表达

1.1.2　图数据的分析计算

随着近年来图数据广泛应用，探索图数据内部关系以及利用图数据的分析计算也受到了越来越多的关注。例如，对社交网络图数据的分析能指导商家更准确地进行一些商业活动，如病毒式营销。通过分析图上的各种图中心性可以获取社交网络中用户的属性，进而用来促进商品营销。可以通过下面具体的两个例子来直观的说明。

- **社交网络平台的投资选择**，根据病毒式营销中的“口碑效应”，一个用户购买商品时可能会受到其朋友的影响而去买同一件商品。所以利用在线社交网络（Online Social Network, OSN）可以很好的进行商品营销。不同的 OSN 会呈现不同的潜力，比如 OSN 中用户的活跃性和影响力。所以一个商家来说，选择哪个网络平台进行投资能吸引到最多的用户购买商品？这个问题可以通过估算 OSN 中所有用户对之间的平均相似性来衡量。
- **商品营销中的捆绑销售策略**，将多个商品在一起打折捆绑销售也是一种常见的营销策略。但是具体选择哪些商品放在一起捆绑销售能带来最大的销售额。这个问题可以通过估算每个商品在用户之间的兴趣分布，捆绑有相似分布的商品来解决。

随着网络图数据的快速发展及普及，越来越多的服务于各个行业的图分析算法及应用也不断涌现，比如，对网页连接图的分析能产生出更佳的网页排序，服务于搜索引擎对用户的网页推荐、对电商网络图的分析能分析消费者的购物偏好，服务于对用户个性化的产品推荐等。

1.1.3　大规模图分析带来的挑战

随着信息技术下移动终端的普及以及人类社会活动的不断发展，各类网络平台相应产生的图数据的规模也不断增大，例如很多网页链接图已经达到数十亿的顶点和数千亿的边[14]。Facebook 在 2021 年 1 月份发布的最新数据显示，其社交网络的月活跃用户截止到 2020 年第四季度已经接近 28 亿。这些用户之间通过好友关系构成的连接边已突破万亿条[15-16]。阿里巴巴电商平台的实际场景中的图数据也通常包含数十亿个顶点和数万亿条边[17]。这样大规模的图数据给图分析计算带来挑战，主要可以包含下面两个方面：

- **大图上的遍历分析难以高效运行**，很多图分析算法往往都需要循环迭代的遍历分析所有的图数据去产生最终的分析结果，比如典型的网页排序算法 PageRank，需要循坏迭代地遍历图中的每个顶点，并根据该顶点的邻居顶点的值来计算更新当前顶点的新 PageRank 值，直至所有顶点的值都趋于稳定。当图数据规模变得非常大时，每一轮迭代需要遍历的顶点/边的数目也大大增加，从而产生的运算开销也急剧增长，因此很多迭代遍历的图算法由于过高的计算复杂度，而在大规模图数据上难以实现。
- **大图数据放不下单机的内存**，大规模的图数据带来的存储开销也很大，比如一个包含数千亿条边的图数据，仅仅是存储点边关系的图结构数据，即使采用最压缩的存储方式存储，最少也需要数百 GB 的存储空间。再加上图分析过程中产生的元数据信息或与点边关联的属性信息，存储开销甚至可能达到 TB 级别。这样大的存储开销远远大于目前单个机器的内存容量。此时，就需要借助分布式集群或者单机的外存来存储这些图数据，然后通过机器间的通信，磁盘到内存的 I/O 调度来实现图数据的分析处理，进一步增加了大规模图数据分析的挑战。

另外，除了图数据规模大，图结构数据本身也在不断动态变化，其中包括顶点和边数据的更新变化，比如社交网络中注册新用户、用户发送博客，用户之间添加新的好友关系、用户在博客之间相互的评论点赞等，这众多场景都会带来图结构数据的更新[18]。动态图数据的处理场景进一步加剧了图分析的挑战。

1.2　基于随机游走的图分析

面对这样大规模的图数据，很多迭代遍历的图算法由于过高的计算复杂度而难以实现，此时通过采样的方法来实现近似计算是一种可行的替代方案，其中基于随机游走的方法能够很好的表达很多算法。随机游走是图数据分析和机器学习中一个重要的分析工具，可以利用图中顶点之间的集成路径提取信息，经常被应用于一些重要的图分析、排序和嵌入式算法中，比如个性化的 PageRank[1,19]、

SimRank[20]、Trustwalker[21]、DeepWalk[22] 和 node2vec[23] 等。这些基于随机游走的算法也被广泛的应用于很多图分析场景中，例如个性化的网页排序[24]、相似性计算[20]、影响力传播[25] 以及机器学习等一些其他应用场景[26-29]。近年来，学术界和工业界也有越来越多的工作关注随机游走，根据微软学术的统计结果，2018 年就有大约 1700 篇学术论文是关于随机游走[30]，工业界的很多大公司比如 Facebook、谷歌、微软、阿里巴巴和腾讯等也都使用了随机游走相关的技术。

1.2.1　随机游走过程

如图 1.2 所示，基于随机游走的算法通常开始时同时启动多个随机游走，每个随机游走具有一定的行走长度，然后利用这些随机游走的访问模式进行图数据分析。一个随机游走首先从一个初始顶点开始，随机选择当前顶点的一个邻居顶点并跳转到该顶点，重复上述过程直至满足预设的终止条件，比如随机游走达到一定的步长或者随机游走在每一步有一定的概率终止。很多基于随机游走的应用程序常常需要同时运行大量的随机游走，从而保证计算的准确度[24-25,31-32]。

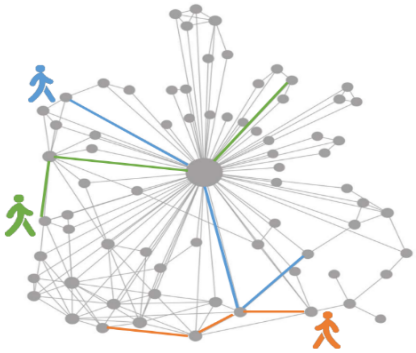


图 1.2　随机游走过程

1.2.2　基于随机游走的算法应用

随机游走已经被证明是分析大规模图数据的一个非常有效的方法，有非常广泛的应用场景[24,33,20,25-27,34,29,28]。其中一些应用场景是通过随机游走来近似计算从而提高计算效率，比如个性化的 PageRank (PersonalizedPageRank，PPR)[24,33]，PPR 从一个初始的源顶点出发数千条随机游走，然后统计这些随机游走经过图上其他所有的顶点的访问频率作为这些顶点 PPR 的近似值。另外也有一些应用场景由于过高的时间复杂度只能使用随机游走来进行模拟计算，比如计算顶点对的相似度的 SimRank(SR)[20]，SR 从要求解的顶点对分别出发数千条随机游走，然后计算这些随机游走的预期的相遇时间；还有随机游走影响域 (Random walk domination, RWD)[25]，RWD 从所有的顶点都出发若干条随机游走，以度量随机游走在整个图上的影响扩散。当需要计算每个顶点作为源顶点的 PPR，或者为图

上每对顶点对计算相似度时，都将带来大量并发随机游走的场景。为了更直观的理解随机游走怎么实现大规模的图分析，接下来我们以个性化的 PageRank 作为应用案例，阐述具体的算法流程。

应用案例——个性化的 PageRank： PageRank[1] 是 Google 公司提出的著名的网页排序算法。如图 1.3所示，PageRank 根据网页链接图中一个网页被其他网页链接的次数以及质量来计算每个网页的 PageRank 值，并根据这些值进行搜索引擎下的网页排序。该算法一个更为复杂的版本是个性化的 PageRank[24]，即根据用户的个性化的偏好，进行个性化的网页排序推荐。通常 PageRank 算法是可以用幂法迭代进行计算，但是个性化的 PageRank，尤其当要对很多用户甚至是全部用户同时进行个性化的 PageRank 算法计算时，用幂法迭代计算往往需要很高的时间和空间开销，对计算（尤其是大规模图数据上的计算）带来困难[19]。

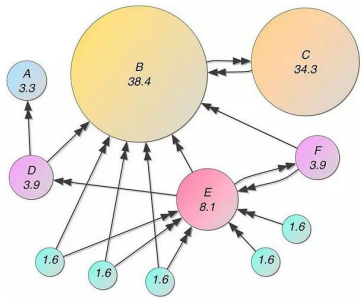


图 1.3　基于 PageRank 的网页排序

此时，基于随机游走的近似算法是个常用的替代算法，通过在图上模拟一定数量的随机游走，并统计图上各个顶点被这些随机游走访问的概率，即可近似得到个性化的 PageRank 值[24]。具体的算法如下：要计算图上一个顶点的个性化的 PageRank 值，可以从该顶点出发，模拟一条很长的基于重启的随机游走（RWR），即随机游走在每一步以一定的概率回到初始顶点。随机游走在走到一定的步数后会到达收敛状态，在其收敛以后，统计该条随机游走访问各个顶点的访问频率，近似计算的得到该顶点的个性化的 PageRank 值。为了达到收敛以保证近似计算的准确度，从该顶点出发的 RWR 的步数通常需要很长，对于一些大规模的图数据，甚至需要成千上万步随机游走，这也给计算带来了很大的时间开销，尤其是针对基于磁盘的大规模图数据的处理场景。所以一种替代的做法是将这一条很长的基于重启的随机游走，在其每次重启的地方切割，转换成很多条很短的随机游走，然后通过并发执行来提升计算效率。具体的算法执行过程可修改如下：想要计算图上一个顶点的个性化的 PageRank 值，可以从该节点同时出发很多条 RWR，并将每条随机游走的重启条件变成终止条件。这样就可以快速得到很多短的随机游走，然后统计这些随机游走对图中各个顶点的访问频率，即可快速近似计算的得到该顶点的个性化的 PageRank 值。理论上可以证明这种基于拆

分拼接的方法不会带来任何计算误差。这种基于拼接的随机游走方法，将一条很长的随机游走的计算转换成了很多条短随机游走的并发计算，大大提升了计算效率。但与此同时，这种方法也给图上带来了大量并发的随机游走，尤其是当需要同时计算多个顶点的个性化的 PageRank，甚至是同时计算所有顶点的个性化的 PageRank 的时候，更是进一步加倍了图上并发的随机游走的个数。

1.2.3　随机游走类应用的特征

基于对现有随机游走算法应用的调研分析，我们总结基于随机游走的图计算具有如下几个方面的特征：

(1) **对图数据访问的随机性更强**：由于图数据之间通过边数据错综复杂的关联关系，图计算场景的数据访问往往具有很强的随机性。而随机游走在每一步的转发过程中随机挑选当前顶点的邻居顶点跳转的方式，更是加剧了其对图数据访问的随机性。

(2) **并发随机游走的数量可能很大**：基于随机游走的采样中有一种独立采样的方式，需要从图中同时出发很多条随机游走，然后在这些随机游走收敛之后分别采样一个样本。这种情况下，为了保障采样估算的精确度，往往需要同时出发大量的随机游走。此外还有一些其他的基于随机游走的图计算场景比如基于随机游走的社交网络影响力最大化算法等也需要同时从图中出发大量的随机游走，带来大规模并发随机游走的应用场景。

(3) **随机游走的步长可能非常长**：基于随机游走的图采样都需要在随机游走达到收敛以后再进行采样，而随机游走的收敛往往也需要非常多的步数，因此带来超长随机游走的应用场景。

如何在大规模的图数据上进行这样大规模的并发随机游走？如何管理这样大规模的图数据和随机游走数据？目前有一些针对通用图算法的基于磁盘的单机图处理系统，但是这些系统在支持随机游走时可能会表现出一些局限性，比如 I/O 效率低、难以扩展到动态图处理场景、随机游走数据管理开销大、随机游走更新速率低等问题。所以需要一个支持快速随机游走的大规模图分析系统来支持这些基于随机游走应用的计算效率。

本章的后面几个小节将首先介绍大规模图处理系统，包括其在图存储管理和更新计算等方面的国内外研究现状以及现有系统在支持随机游走类应用时所表现的局限性；然后阐述本文如何针对这些不足去开展研究内容，设计并实现支持快速随机游走的大规模图分析系统。

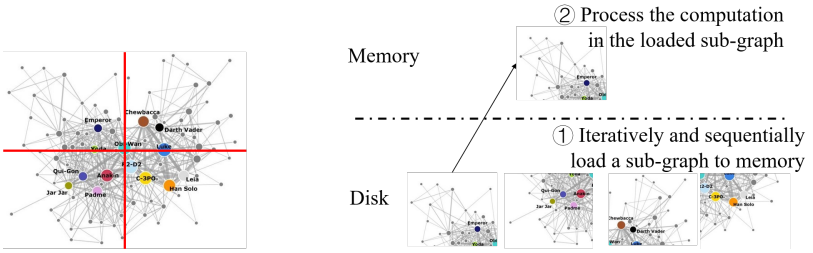
1.3　大规模图处理系统

本章第1.1.3小节已经阐述过，当前的一些大规模的图数据难以放下单机的内存，所以完整的图数据必须存储在磁盘上。图处理时对图数据的随机访问，会带来对磁盘大量的随机 I/O，从而使得这种大规模磁盘驻留图的分析非常耗时。对于某些应用程序，甚至需要运行几个小时或几天才能产生结果。针对大规模图数据的计算处理，一种解决思路是开发分布式图计算框架，通过分布式集群之间的通信和同步来协作完成图计算任务[35-43]。然而分布式图数据系统通常需要高效的图数据划分和机器之间低成本的通信和同步。另一种思路是研究基于磁盘驻留图的单机图计算框架，即将大规模的图数据存放在单个机器的磁盘上，并通过合适的存储模型、I/O 调度策略和计算模型来处理图计算任务。因此，基于单机的磁盘驻留图的图处理系统也备受关注[44-54]。接下来，我们主要关注基于单机的磁盘驻留图的图处理系统，并从图存储与 I/O 优化、图计算模型的优化、基于新硬件的存储和计算优化，动态图场景下图数据更新与查询优化以及针对随机游走类场景的系统优化这多个维度去介绍大规模图处理系统的处理流程和国内外研究现状。

1.3.1　国内外研究现状

（一）图存储与 I/O 优化

为了提高在单机上分析大型图的性能，很多基于磁盘的单机图处理系统被提出，例如 GraphChi[44], X-Stream[45], GridGraph[46], FlashGraph[51], ODS[47], CLIP[49], Mosaic[52], Graphene[48], GraFBoost[53], V-Part[54] and LUMOS[50]。这些系统的其中一个主要的目标就是减少随机磁盘 I/O。通常当一个图太大而不能放入内存时，这些系统会将整个图划分成许多子图，并将每个子图存储在磁盘上，例如 GraphChi 中的 *shard*[44]。为了进行图分析，现有系统通常采用一种基于基于迭代的模型，如图 1.4。在每次迭代中，按照顺序迭代的方式将各个子图加载到内存中，然后执行与加载的子图相关的分析计算，一直循环迭代直到完成所有的图计算任务。通过这种方式，可以将大量的随机 I/O 转换成一系列的顺序 I/O，并且保证了在每次迭代运算中对所有子图进行同步地分析。随后很多工作在此基础上进一步减少磁盘 I/O，例如清华大学 Zhu 等人提出的 GridGraph[46] 首先提出一种二维图划分的方式将图数据划分成网格的形式，然后通过有选择地加载所需的图数据块，从而避免无用的图数据块带来的 I/O 开销。DynamicShards[47] 和 Graphene[48] 通过动态调整图的划分布局，更加细粒度地进行 I/O 选择和减少无用边数据的加载。CLIP[49] 和 Lumos[50] 尝试多次重复利用已经加载的图数据块，从而提高 I/O 效率并减少后续 I/O 的数量。



(a) 图数据划分　　(b) 基于迭代的处理模型

图 1.4　图划分和基于迭代的处理模型

（二）图计算模型的优化

在上述基于迭代的处理模型的模型下，这些图处理系统大多也都采用一种基于块同步并行（Bulk Synchronous Parallel, BSP）计算模型。即按照加载的子图数据块为单位执行计算，并在每一轮迭代结束后进行数据同步。比如 GraphChi[44] 采用以顶点为中心的 BSP 计算模型，即依次遍历加载子图上的所有顶点，为每个顶点 v 执行图计算任务定义的更新函数 $f(v)$，以随机游走为例就是将当前正停留在该顶点上的随机游走转发出去。随后，X-Stream[45] 提出一种以边为中心的 BSP 计算模型，即依次遍历加载子图上的所有边，为每条边 e 执行图计算任务定义的更新函数 $f(e)$。以边为中心的 BSP 计算模型在支持基于迭代类的图算法（例如基于幂法迭代计算的 PageRank 算法）时能表现出更好的性能，因为避免了 CPU 对于内存中大量边数据的随机访问，但是这种计算模型很难支持随机游走类的应用，因为一个顶点的所有边数据并不一定会存在一起，随机游走很难对当前顶点进行随机的邻居选择。以上两种计算模型都是典型的计算模型，在分布式和单机图处理系统中都是非常常用的。

（三）基于新硬件的存储和计算优化

随着近年来很多新型存储硬件的不断涌现，比如 NVRAM 和 NVMe SSD。很多图计算系统也尝试结合新硬件的特性来提升图数据的存储访问效率，比如麻省理工学院的 Jun 等人提出的 GraFBoost[53] 和三星公司的 Elyasi 等人提出的 V-Part[54]，利用 SSD 设备相对较好的随机访问性能来进行相应的存储设计优化。近年来也有一些图计算框架关注于基于 GPU 加速器的优化计算[55-58]，比如北京大学 Ma 等人提出的 Garaph[59] 和华中科技大学 Zheng 等人提出的 Scaph[60]。此外，清华大学 Dai 等人也进一步利用 FPGA 技术来实现节能和高可扩展的图计算，并提出 FPGP[61] 系统。

（四）动态图场景下存储优化

现实中的很多应用场景都是动态图场景，比如网页链接中插入新网页、电商平台用户购买产品、社交媒体中删除推文和社交网络中好友取消关注等，都会带

来顶点和边的插入或删除，造成图结构的改变。而一般动态图的处理场景图大多是一种在线实时的处理场景，往往需要实时得到反馈。但目前来说，大多数系统考虑的是静态图的处理场景，不能很好的支持动态增量图数据的实时插入和更新。一种通用的方式是采用快照的方式单独存储动态图场景下的增量数据，即静态图数据与动态增量图数据分离存储。这种方式虽然能很快的插入图数据的更新信息，但是在进行图分析计算时，需要分别访问静态图数据以及动态增量图数据，造成额外的查询开销，影响分析计算的性能。

（五）针对随机游走场景的存储和计算优化

随着近年来随机游走在机器学习领域得到越来越多的广泛关注，专门针对随机游走优化的单机图处理系统和分布式图处理系统相继被提出，分别是针对随机游走的存储管理优化的单机随机游走系统 DrunkardMob[33] 和针对随机游走计算流程进行优化的分布式随机游走系统 KnightKing[30]。

（1）针对随机游走场景的存储优化

DrunkardMob[33] 发表于 2013 年的 RecSys 会议，是一个基于 GraphChi 的随机游走处理框架，在底层采用 GraphChi 的图数据管理方式存储和加载图数据。DrunkardMob 主要针对此前图处理系统中随机游走数据管理开销大的问题，进行随机游走的存储管理优化。具体来说，DrunkardMob 首先将每条随机游走编码为一个 32 位或 64 位的数据存储，从而减少随机游走数据本身的存储开销。然后 DrunkardMob 将当前正停留在相邻 128 个顶点的随机游走放入同一个随机游走缓冲区并存储在内存中，以减少随机游走数据索引的开销，并避免了随机游走数据的 I/O 开销。

（2）针对随机游走场景的计算优化

最近于 2019 年 SOSP 上提出的 KnightKing[30] 是一个专门的随机游走图系统。KnightKing 主要针对复杂的动态随机游走（即随机游走的每一步的邻居选择的概率是根据随机游走的状态动态变化的），提出一种拒绝采样的策略。拒绝采样开始用于通用的任意概率的采样，将一个一维的采样过程转换成一个二维的采样过程。具体对应到随机游走的边采样过程为：随机地产生一个位置 (x, y)，其中 x 是从当前顶点的出边中均匀随机选取的一条边 e，而 $y \in [0, Q(v)]$，其中 $Q(v)$ 是当前顶点的出边的动态转移概率的最大值。当 $y \leqslant P_d(e)$，则 e 被接收，当前的随机游走通过 e 跳转；否则 e 被拒绝，需要重新采样 (x, y) 并重复上面过程，直至随机游走成功跳转。通过拒绝采样的方式，消除了对当前顶点所有出边的访问，才能获取随机游走转移概率的问题。一般只需要几次尝试就可以成功采样一条边（复杂度为 $O(1)$），大大减少了对高度顶点的采样开销。

1.3.2　现有系统支持随机游走的局限性

虽然基于随机游走的算法从算法层面降低了图分析的时间复杂度，但目前的图分析系统对于大规模图数据上的随机游走的支持还存在一些局限性。目前的图分析系统在支持随机游走时也通常采用一种基于迭代的处理模型，它们会根据顶点或边将整个图划分成许多子图数据块存储在磁盘上；然后按照顺序迭代的顺序，每次加载一个子图进入内存；然后按照一种同步的方式将该子图上的随机游走向前转发一步；循环迭代直至所有的随机游走都完成计算。通过这种方式，基于迭代的处理模型可以将大量的随机访问转化为对磁盘的一系列连续访问从而降低 I/O 开销。但是我们发现这些基于迭代模型的图系统更适用于那些需要整个图数据的算法和应用，而不能有效地支持基于随机游走的图算法，主要有下面四个方面的局限性。

（一）I/O 是性能瓶颈

首先，随机游走过程中的随机的邻居选择进一步加剧了算法对图数据的随机访问，而且这些随机游走会非常不均匀地分散在图的不同部分，即使从一个顶点出发的随机游走在经过几步后也会遍布到每一个子图当中，但分布的不均衡导致有些子图只包含很少的几条步随机游走。在基于迭代的处理模型下，系统并不会关注这些随机游走的分布状态，只是顺序地将所有需要的子图加载到内存中进行分析计算，因此当处理这些只包含很少量随机游走的子图时将导致非常低的 I/O 效率。其次，许多在线图数据查询算法也是基于随机游走实现的，此时随机游走本身就只需要访问整个图数据的一部分，而基于迭代的模型迭代的加载整个图数据信息，导致在执行基于随机游走的算法时 I/O 效率不高。我们定义 I/O 效率为被随机游走使用的边数除以系统一次加载的图数据中的总边数，并通过实验观察最新的单机随机游走图计算框架 DrunkardMob[33] 的 I/O 效率。通过真实世界的数据集上的实验发现，DrunkardMob 在 Friendster 数据集[62] 上运行一亿条 10 步的随机游走，平均 I/O 效率只有 3.2×10^{-4}, 当随机游走数目更少时 I/O 的利用率将更低。这意味着每次系统花费很多时间读入大量的图数据块，但是其中真正被用来计算随机游走的部分却很少。而大量花费在 I/O 上的时间造成了整个计算过程的性能瓶颈。

近年来，为了提高基于迭代的模型的 I/O 效率，人们做出了各种设计努力，例如: DynamicShards[47] 和 Graphene[48] 在每次迭代中动态调整图块布局，减少无用数据的加载。CLIP[49] 提出了*加载子图重新进入的方案*和 Lumos[50] 提出了*交叉迭代值传播技术*，两者的目的都是充分利用已加载的块，避免在以后的迭代中加载相应的图部分。这些系统极大地提高了性能，但它们没有考虑到随机游动的特性。而上述分析的局限性仍然没有完全解决。

（二）动态图场景下响应不及时

最新的一些分布式和单机图处理系统中大多采用一种压缩稀疏行（Compressed Sparse Row, CSR）的方式存储图数据，因为 CSR 可以实现理论上最小的存储开销，并且能支持高效的图查询。但 CSR 存储格式只适用于静态图的存储。动态图场景下顶点或边数据的更新插入需要重构 CSR 存储内容。当有频繁的增量数据插入时重构的开销就会非常大，因此在真实的场景中往往难以实现。一些图处理系统中采用一种基于快照的动态图数据的管理方式，将现有的静态图数据与短时间内动态插入的增量图数据分开存储。基于快照的方式能很快的插入图数据的更新信息，但同时也增加了图数据的查询开销，因为访问图数据时需要分别访问静态图数据和增量图数据，影响图分析计算的性能。工业界的一些图数据库产品比如 Neo4j[63]、ArangoDB[64] 等[65]，虽然能实现高效的图数据的插入和简单查询，但是对于图分析的场景就很难高效支持。

（三）随机游走存储管理开销大

在随机游走数据的存储管理方面，大多数图处理采用以顶点为中心的随机游走数据的索引方式，即为图中每个顶点分配一个数组用于存储当前正停留在该顶点的所有随机游走信息，这种索引设计需要在内存中分配非常多的数组索引信息，导致不包括随机游走数据本身的索引数据就带来非常大的内存开销，比如对于一个中等大小的图数据集 YahooWeb[66]（包含 14 亿个顶点）就需要 5.6GB 来仅仅存储随机游走的数组的索引。DrunkardMob 通过将相邻的 128 顶点的随机游走数据存储到同一个内存中的缓冲区中，将随机游走数据的索引数量减少到原来的 1/128，但依旧带来大量的随机游走数据的索引数目。此外由于随机游走的高度随机性，每个顶点上关联的随机游走的数目也是动态变化的，所以现有的图系统通常使用动态数组来记录每个索引数组，从而带来内存中大量的动态数组。这些动态数据随着随机游走在各个顶点间的游走而带来大量的内存重新分配，不仅带来很多额外的时间开销，也在内存空间中造成很多内存碎片，增大了内存管理的开销。因此，现有图处理系统中的随机游走的存储管理方案限制了系统能够处理的图数据的规模和能并发处理的随机游走的规模，极大地限制了图处理系统的可扩展性。

（四）随机游走更新速率低

在随机游走数据的更新计算方面，基于迭代的图处理系统为了保证图计算任务在图上的同步性，往往在每次迭代中依次加载所有需要的子图，然后让子图上所有随机游走都只移动一步，之后便丢弃最近加载的子图将被删除，即使它们仍然可以更新更多的随机游走的步数。因此现有方案中随机游走的更新效率也受到限制，尤其是对于需要运行很长的随机游走的场景。

1.4　本文的主要研究内容

考虑到随机游走的广泛应用场景，本文聚焦于随机游走这一类图分析算法，研究支持快速随机游走的大规模图分析系统。具体地，针对基于磁盘的单机图处理场景，向下支持大规模的图数据，向上支持基于随机游走的图算法及应用。如图 1.5所示，本文的主要研究内容分三个层次，分别从支持快速随机游走的大规模图数据的存储管理、随机游走存储和计算框架和快速收敛的随机游走算法优化三个方面展开研究，最终结合这三个方面的研究内容，组成一个支持快速随机游走的原型系统。下面分别阐述这三个方面的主要研究内容。

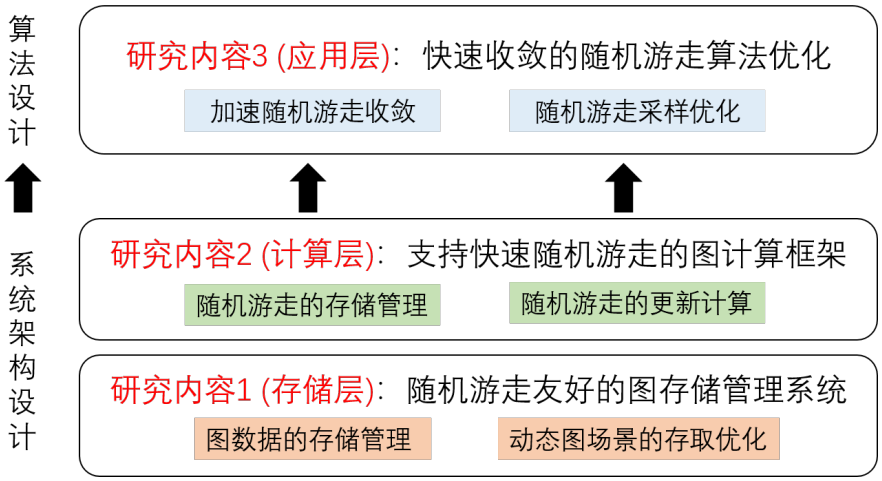


图 1.5　本文的主要研究内容

1.4.1　随机游走友好的图存储管理系统

结合随机游走算法的特征，本文首先从系统层面考虑现有图分析系统在处理随机游走应用时存在的问题实现相应的优化设计。第一个研究内容考虑大规模图数据的存储管理和 I/O 机制。当前的基于单机的图分析系统大多采用一种基于迭代的模型来处理大规模的图数据，即将图数据划分成多个子图数据块存储在磁盘上，并通过顺序迭代的方式依次加载各个子图到内存中进行计算处理。通过这种方式将访问磁盘的大量随机 I/O 转换成一系列的顺序 I/O，缓解 I/O 性能瓶颈的问题。然而这种基于迭代的模型在支持基于随机游走的图计算任务时表现出很低的 I/O 效率。因为随机游走的高度随机性导致大量并发的随机游走往往遍布各个子图且分布非常不均衡，所以很多子图中只包含很少量的几条随机游走，却需要在每一轮迭代中都被加载一遍，造成很低的 I/O 效率，从而限制了图上随机游走的效率和扩展性。

研究内容一首先提出一种基于随机游走状态感知的 I/O 模型，即根据图上随机游走的总数目、各个随机游走已走过的步长以及随机游走在各个子图中的分布情况等状态信息，自适应地执行不同的图数据的组织划分、加载和缓存策略，

从而实现 I/O 效率的最大化。然后进一步针对动态图处理的场景下更新效率慢的问题，提出一种基于分块日志的 CSR 存储实现快速的图更新。基于上述随机游走状态感知的 I/O 模型和分块日志的 CSR 存储方案的设计，实现了面向大规模并发随机游走的高效图数据存储管理机制 SAStore 。实验结果显示，SAStore 对比于最新的单机随机游走图系统 DrunkardMob 可以显著提升 I/O 利用率。在动态图场景下，SAStore 对比于基础的 CSR 存储格式，在保证图数据查询效率的同时，显著提升了动态图数据的更新效率。

1.4.2　支持快速随机游走的图计算框架

在上一个研究内容（图存储管理系统）的基础上，本文继续研究随机游走的存储管理和更新计算方案，使其向下融合随机游走状态感知的大规模图存储管理机制，向上支持各种规模的随机游走算法及应用场景。现有的支持随机游走的图处理系统中对随机游走数据的存储管理往往采用以边或顶点为中心的随机游走状态数据的索引机制和基于大量的动态数组的存储方案。这样的随机游走的存储管理方式不仅带来非常大的索引数据的开销，而且大量的动态数组也会带来频繁的内存空间的重新分配，从而限制了系统可以处理的图数据的规模和能并发运行的随机游走的规模。另一方面，现有的支持随机游走的图处理系统在进行随机游走的更新计算时，也往往采用基于迭代的同步计算模型，以保证图上所有随机游走之间的同步性。但这种同步计算模型也牺牲了随机游走的计算效率，导致随机游走更新速率慢的问题。

针对上面两个方面的问题，研究内容二首先基于随机游走数据的特征，提出一种基于定长缓冲区的以子图为中心的随机游走数据的存储管理方案，减少索引开销的同时也避免了大量动态数组频繁重新分配内存空间。然后提出一种以随机游走为中心的异步更新策略，大大提升了随机游走的更新速率。最后在上一个研究内容提出的 SAStore 上，结合本研究内容中随机游走的存储和更新方案的设计，实现了一个支持快速随机游走的图分析框架 GraphWalker 。实验结果表明：（1）从可扩展性方面，GraphWalker 可以高效地处理包含数十亿个顶点、数千亿条边的大规模图数据，也可以在这样大规模的图上处理包含数百亿条、数千步长的并发随机游走的应用场景。（2）在系统性能方面，GraphWalker 与最新的单机随机游走图处理系统 DrunkardMob 相比，随机游走的运行速度一般来说能快一个数量级；GraphWalker 与最新的通用单机图处理系统 Graphene 和 GraFSoft 相比，支持随机游走任务时的运行速度提升也非常明显；（3）在资源友好性方面，GraphWalker 与最新的分布式随机游走系统 KnightKing 相比，单机上的处理速度可以达到 KnightKing 使用 8 台机器的速度。

1.4.3　快速收敛的随机游走算法优化

最后，本文在前两个研究内容实现的原型系统上进行基于随机游走的应用验证，并进一步从算法层面研究随机游走的算法优化。具体以图采样问题为例研究相应的随机游走的优化算法。随着图数据规模的不断增大和图分析算法的复杂度逐渐升高，图采样技术通过分析采样的一些有代表性的样本避免遍历整个网络图数据，大大减少了计算开销因而被广泛应用于各个行业的图分析场景。其中基于随机游走的图采样算法计算简单、可以方便地在网络环境下通过给定接口访问局部图数据而且随机游走收敛后访问各个顶点的概率趋于稳态分布，从而可以利用采样样本实现无偏估计。因此基于随机游走的图采样成为当前主流的图采样策略。但是现有的随机游走算法的收敛速度非常慢，往往需要成千上万步随机游走才能收敛，收敛之后才能进行具有理论基础的无偏估计。因此，随机游走收敛速度慢的问题严重影响了采样效率。

导致收敛速度慢的一个很重要的原因是随机游走可能经常被困在某个局部的子图当中，需要经过很多步才能有比较大的概率走出去，从而拖慢了随机游走探索到全局图数据的进程，从而造成随机游走收敛速度慢。基于上述分析，研究内容三提出一种基于公共邻居感知的随机游走算法，根据随机游走的历史路径信息，修改随机游走在每一步跳转到当前顶点的各个邻居顶点的概率，从而诱导随机游走尽快跳出当前所在的局部子图，加快随机游走的收敛进程。之后进一步结合 NBRW[28] 中非回溯的思想，设计并实现了非回溯的公共邻居感知的算法 NB-CNARW，加速随机游走的收敛速度的同时也减少了基于随机游走的图采样的查询开销。实验结果表明，NB-CNARW 算法相较于当前最新的随机游走算法，可以显著提升随机游走的收敛速度；在使用 NB-CNARW 算法进行基于图采样的无偏估计时，可以在保证相同估算精度的情况下，可以大幅度减少采样所需要的查询开销。

1.5　本文的组织结构

本文剩余章节的组织结构如下：第 2 章首先进行相关的背景介绍，第 3 章从系统层面图数据的存储管理出发，介绍随机游走友好的图数据的存储格式和 I/O 模型的设计方案；第 4 章在图存储设计的基础上，阐述了随机游走数据的存储管理和随机游走的更新计算方案；第 5 章从一个具体的图分析场景（图采样问题）出发，进一步设计基于随机游走的图算法优化；第 6 章总结全文并展望未来的研究计划。

第 2 章　相关背景介绍

本章摘要: 随机游走因其高效的计算被广泛采用到很多应用场景中。本文聚焦于随机游走这一类图计算场景，研究专门针对随机游走类应用驱动的系统设计和算法优化。随机游走在系统中的处理流程通常首先需要将随机游走需要访问的图数据加载到内存（图数据的存储和 I/O）；然后在加载的子图数据上执行相应的随机游走的计算转发（包括随机游走数据的存储及计算模型）；最后根据具体的应用场景的需求进行相应的随机游走的算法层面的优化设计进一步加速程序执行（随机游走应用算法优化）。本章首先介绍随机游走运算处理的上述三个层次架构，即存储层、计算层和应用层。然后自下而上地依次介绍每个层次的相关背景和相应的研究现状。

2.1　随机游走处理的整体架构

图 2.1展示了处理随机游走类应用时的整体架构图，主要包含底层的存储层、中间的计算层和上层的应用层这三个部分。存储层主要负责底层大规模图数据的存储管理，包括图数据的组织划分和 I/O 调度；计算层主要负责基于存储层图数据存储管理的随机游走的计算框架，包括随机游走数据的组织索引和更新计算；应用层主要负责基于计算层随机游走的计算框架的随机游走类应用算法的优化设计。随机游走处理的这三层架构，自下而上地提供相应的接口支持，共同完成快速的随机游走。接下来依次阐述各个层次的背景介绍和相关工作。

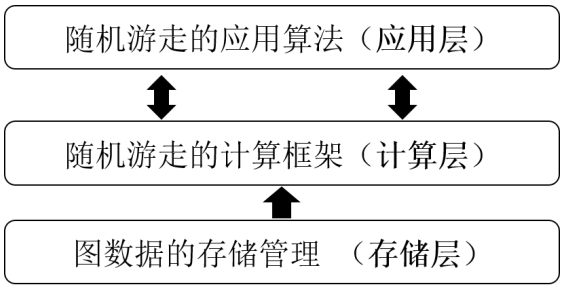


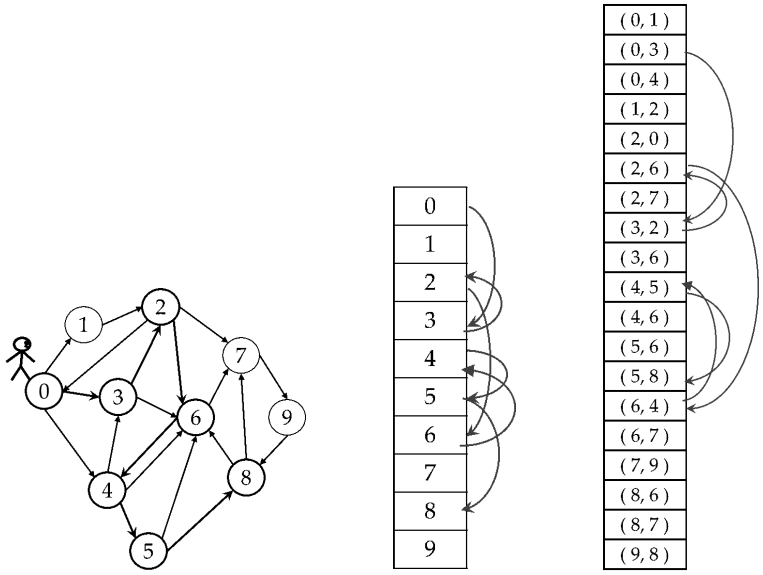
图 2.1　随机游走处理的整体架构图

2.2　支持随机游走的图计算系统（存储层）

本章首先介绍基于单机的磁盘驻留图的图处理会带来的对磁盘图数据大量随机访问的问题；然后阐述现有的基于迭代的模型下图数据的存储管理方案以及如何解决大量随机访问问题。

2.2.1　图数据随机访问问题

这些超大规模的图数据很难放入一台机器的内存中，必须存储在磁盘上。由于图数据中顶点之间连接紧密且复杂，会带来对磁盘的大量的随机 I/O，而随机游走本身的随机邻居选择更增加了对图数据的访问的随机性。图 2.2展示了一个样例图的随机游走对图中顶点和边的随机访问过程。其中图 2.2(a) 表示样例图上的一个 6 步长的随机游走，游走轨迹为 0→3→2→6→4→5→8；图 2.2(b) 和图 2.2(c) 分别表示样例图的顶点列表和边列表，图中的箭头代表访问数据的顺序。从图中可以看出，随机游走对图中顶点和边的访问都会带来大量的随机访问。当这些数据放在磁盘上的时候，就会带来对磁盘的大量随机 I/O。因此分析这些需要驻留在磁盘中的大规模图数据非常耗时，对于某些应用，需要运行几个小时甚至几天的时间才能得到一些计算结果。



(a) 图上的 6 步随机游走　　(b) 顶点列表　　(c) 边列表

图 2.2　随机游走对图数据的随机访问

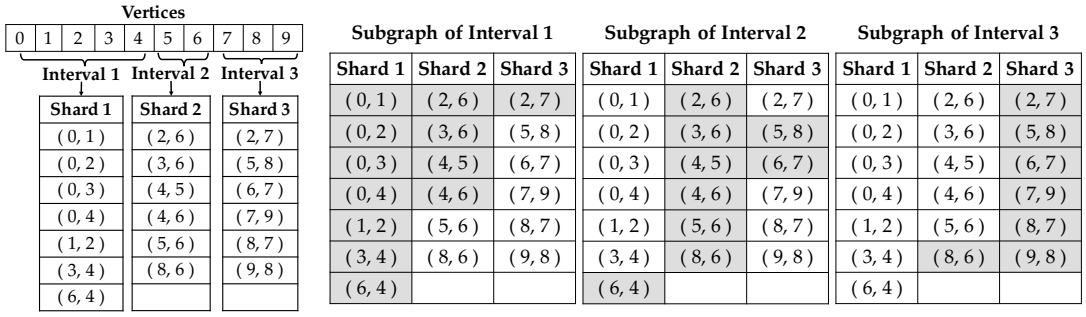
2.2.2　基于迭代的图处理模型

（一）GraphChi 的存储和 I/O 模型

针对图分析算法对磁盘驻留图的大量随机 I/O 问题，很多基于磁盘的单机图处理系统提出 I/O 优化方法。GraphChi[44] 是这类工作的先驱，首先提出基于迭代的模型来将对磁盘驻留图的大量随机 I/O 转换成若干个顺序 I/O 来加速基于磁盘驻留图的图处理。这种基于迭代的模型是非常具有代表性的，它在许多图计算系统中都得到广泛的应用，如[45-46,51,47-48,53,50,54]。接下来，我们以 GraphChi 为例介绍基于迭代的模型下图数据的存储和 I/O 过程。

（1）图的划分与存储

当图数据太大，无法放入内存时，GraphChi 将所有顶点分割成 P 个不相交的区间，并为每个区间与关联一个“shard”，存储目标顶点位于此区间内的所有边。每个 shard 中的边根据它们的源顶点 ID 进行排序。图 2.3(a) 显示了图 2.2(a) 中的示例图的 shard 的数据组织。



(a) shard 中的数据组织　　(b) 子图加载与平行滑动窗口

图 2.3　GraphChi 的存储和 I/O 模型

（2）子图加载

GraphChi 每一次将一个区间顶点所对应的子图加载到内存中进行分析。为了加载子图，它首先从相应的 shard 加载入边，然后从其他 P－1 个 shard 中加载出边，如图 2.3(b) 所示。由于在每个 shard 中都是根据源顶点来对边排序的，所以总共只需要 P 个连续的磁盘读取来加载一个区间对应的子图。通过这种方式，GraphChi 将对磁盘的随机访问转换为一系列连续的访问，极大地提高了磁盘驻留图数据处理的性能。

（3）基于迭代的子图调度

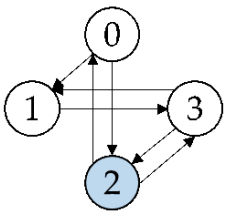
为了同步各个顶点上计算任务的进度，GraphChi 使用基于迭代的模型加载所有的子图，在 GraphChi 中称为并行滑动窗口 (parallel slide window, PSW)，如图 2.3(b) 所示。在每个迭代中，它按顺序循环加载子图，这保证了整个图上所有计算任务之间的同步。通过这种方式，GraphChi 将对磁盘驻留图的大量随机访问转换成一系列的顺序访问，从而极大地提高了磁盘驻留图处理的性能。这种基于迭代的模型在许多图数据系统中得到了广泛的应用。

（二）基于 CSR 的图存储格式优化

随后，有很多研究工作在 GraphChi 的基础上进一步优化基于磁盘的单机图处理的性能。X-Stream[35] 通过将对图数据的存储转化为边数据流的方式，进一步减少了随机 I/O，并且避免了对边数据排序的巨大开销。GridGraph[36] 提出一种二维图划分的方式，并采用双重滑动窗口的方式对边进行流处理。FlashGraph[37] 是一个半外核的图系统，它将顶点数据存储在内存中，将边数据存储在用户空间

的 flash 文件系统中，以提高小型 I/Os 的性能。ODS[38] 提出使用动态分区来调整图数据的布局。CLIP[39] 采用了复用已加载子图的思想，充分利用了每一个 I/O。Graphene[40] 通过采用 I/O 中心计算模型和基于位图的异步 IO 技术优化了 I/O 管理。GraFBoost [41] 和 V-Part [42] 利用高性能的新兴设备进行存储或计算来进一步提升基于磁盘的单机图处理系统的性能。总体来说，这些图系统都是采用的基于迭代的模型来完成图计算任务。

上面介绍的最新的图处理系统中最常用的一种存储图数据的方式为压缩稀疏行，即 CSR（Compressed Sparse Row）。CSR 使用两个序列来存储图数据，CSR 序列用来顺序存储每个顶点的出边邻居的 ID，beg_pos 序列用来存储每个顶点在 CSR 序列中开始位置。图 2.4直观地展示了 CSR 存储格式的数据表示以及数据查询方式。其中图 2.4(a) 为样例图；图 2.4(b) 为它的边列表表示；图 2.4(c) 为它的 CSR 表示，CSR 序列中顺序写入各顶点的出边邻居，分别为 1 和 2、有 3、0 和 3、1 和 2，beg_pos 序列中顺序写入各顶点在 CSR 序列中的开始位置分别为 0、2、3、5，最后再加上一个总边数 7。通过这种方式，CSR 存储格式将一个图数据的存储开销减少到 |E|+|V|+1，相比之下邻接矩阵的存储开销高达 2|E|。同时，CSR 存储格式的查询开销也很低，比如查询顶点 2 的邻居顶点，只需要首先访问 beg_pos 序列中第二个值和第三个值，分别为 3 和 5，就可以得知 CSR 序列中区间 [3,5) 存储的是顶点 2 的邻居，读取得知为顶点 0 和顶点 3。由于其很低的存储开销和查询访问开销，CSR 存储格式被广泛地应用于各个最新图分析系统中，比如 FlashGraph、Graphene 和 GraFBoost。



(a) 样例图

边列表
(0, 1)
(0, 2)
(1, 3)
(2, 0)
(2, 3)
(3, 1)
(3, 2)

(b) 边列表存储

- csr[]
  - ✓ 1 2 3 0 3 1 2
- beg_pos[]
  - ✓ 0 2 3 5 7

(c) CSR 存储

图 2.4　CSR 存储格式

2.3　随机游走的存储和计算框架（计算层）

本节首先介绍当前支持随机游走的图处理系统中随机游走数据的存储管理方式；然后阐述现有工作中随机游走的计算更新策略。

2.3.1　随机游走的存储索引

上一章中提出了随机游走状态感知的图加载模型，为了能对其提供更高效的接口，从而支持大规模图数据上的快速的随机游走。我们首先需要考虑如何存储这些随机游走的状态数据，其中包括每条随机游走出发的源顶点信息、当前停留的顶点信息和已经移动的步长，还有未完成计算的随机游走的总数、这些随机游走在各个子图之间的数量分布、各个子图包含的最短的随机游走的步长等等。因为随机游走的随机性的特征，所以在某个时刻停留在各个顶点上的随机游走的数目也是随机且难以预测的，所以这些随机游走的状态数据往往被存储在大量的内存中的动态数组中。比如，GraphChi[44] 在执行基于随机游走的图算法时，会为图中的每一条边创建一个动态数组，用于存储当前正通过这条边移动的随机游走数据。有些其他的图计算系统采用以顶点为中心的方式来管理这些随机游走数据，即为图中的每一个顶点创建一个动态数组存储当前正停留在该顶点的随机游走数据，比如 FlashGraph[51]、CLIP[49] 和 Graphene[48] 等。

为了支持大规模图数据上的随机游走，近年来也有很多各式各样的优化设计被提出，其中一个典型的随机游走计算框架就是 DrunkardMob[33]。DrunkardMob 提出了几种优化方法来减少随机游走索引的内存使用量，从而支持大量的随机游走。比如为了尽可能地减少随机游走状态数据的存储开销，DrunkardMob 将一条随机游走的状态数据压缩成一个 *32* 位或 *64* 位的编码表示。另外为了减少对这些随机游走数据的索引开销，DrunkardMob 将相邻的 128 个顶点存储到同一个内存中的随机游走缓冲区中，这样减少随机游走索引数据的规模，具体的随机游走编码和索引方式如图 2.5所示。

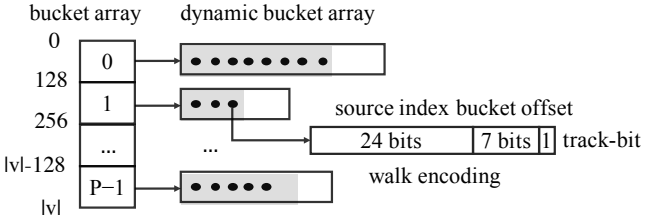
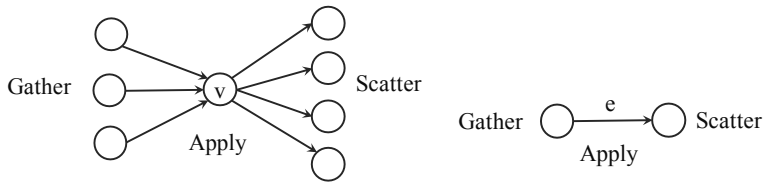


图 2.5　DrunkardMob 的随机游走数据的管理

2.3.2　随机游走的更新计算

结合上一章中提出的状态感知的图加载模型将对应的子图数据加载到内存中后，则对该子图上的随机游走进行更新计算。现有的图处理系统中往往采用遍历顶点/边的方式来进行相应的图计算任务，比如 GraphChi[44] 采用一种以顶点为中心的计算模型，该模型在 Pregel 中首次提出 [38]。以顶点为中心的计算模型依次遍历所加载的子图上的所有顶点，对于每个顶点 v，首先从 v 的入边收集

（Gather）信息，然后执行（Apply）顶点 v 上定义的更新函数 f(v)，最后将更新后的信息通过出边散播（Scatter）出去，这种方法也被称为以顶点为中心的 GAS 计算模型。另一个计算模型是 X-Stream[45] 中提出的以边为中心的模型，它通过遍历子图上的边来工作。图 2.6分别展示了以点为中心的计算模型和以边为中心的计算模型，这是分布式和单机图处理系统中最常用的两种模型。



(a) 以顶点为中心的计算模型　　(b) 以边为中心的计算模型

图 2.6　图计算系统中的计算模型

专门针对随机游走优化的图系统 DrunkardMob[33] 在底层使用 GraphChi 来存储管理图数据，并沿用了使用 GraphChi 的以顶点为中心的计算模型来计算更新随机游走，即依次遍历加载子图上的所有顶点，若当前正有随机游走停留在该顶点上，则将这些随机游走都向前转发一步，即随机选择当前顶点的一个邻居顶点将随机游走转移过去。这种计算方式可以看成是一种同步的随机游走更新策略，其好处是可以很好的保障各个子图上所有随机游走更新的同步性，即在基于迭代模型的每一轮迭代计算中，每一条随机游走都同步地向前更新一步。

2.4　随机游走算法的优化设计（应用层）

图采样技术是图上一类典型的图计算技术，他可以将计算复杂度非常高的图计算问题通过采样得以实现。在众多采样算法中，基于随机游走的图采样算法由于其高效的计算效率和可靠的理论保障而收到广泛的关注和应用。本节首先介绍基于随机游走的图采样技术如何实现对图数据的采样和无偏估计，然后阐述现有的随机游走算法如何优化基于随机游走的图采样过程。

2.4.1　基于随机游走的图采样问题

（一）图采样的重要意义

近年来，随着 Facebook、Twitter、微博和微信等国内外各类网络平台的流行，对社交网络分析能更准确的进行一些商业活动，比如病毒式营销和产品推荐等。通过分析图上的各种图中心性可以获取社交网络中用户的属性，进而用来促进商品营销。这些应用场景往往需要对图上代表不同用户属性的各种中心性指标进行准确的估计。然而想要准确的估计图上的这些中心性不是件简单的事情，主

要有如下挑战：（1）OSN 的图规模通常很大，例如前面已经提到的 Facebook 的用活跃户数已经有约 28 亿。（2）为了保护用户的隐私，很多 OSN 只允许第三方代理通过固定的速度受限的 API ①访问部分的网络数据。为了分析这些大规模图数据，图采样是个常用的技术，通过分析采样的一些有代表性的样本，而避免遍历整个网络数据，这样大大减少了对网络的访问开销。基于图采样的思想，可以在大规模的网络图上采样获得的一个或多个有代表性的采样子图，然后在采样子图上执行相应的分析算法，得出近似的计算结果。也可以先通过在大规模的网络图上采样得到得到一定数量的顶点集合或边集合，然后在样本集合上对感兴趣的目标量（比如图的各种中心性）进行无偏估计。

（二）现有的图采样算法

现有的大规模网络图的采样算法根据采样目标的不同可大概分为两类。

（1）**采样子图**：以获得能够代表原图完整拓扑结构的采样子图为采样目标，这类采样算法中最常见的为基于遍历的采样算法，如广度优先遍历 (BFS) 算法、深度优先遍历 (DFS) 算法以及森林着火 (Forest Fire) 算法[67]，即首先均匀的选取一个种子顶点，然后开始向其邻居顶点“燃烧”，如果一个它的邻居顶点被燃烧了，那么该邻居顶点又有机会去燃烧它自己的邻居顶点，如此迭代燃烧。

（2）**采样点/边样本集合**：以获得已知采样概率的顶点/边的样本集合为采样目标，这类采样算法又包含着两种采样模式。

- **基于网络全局信息已知的模式**，比如随机顶点选择采样和随机边采[68,34,69]，即根据顶点/边的 ID，随机的选取部分采样顶点/边。其中随机顶点选择采样根据图中每个顶点被采样到的概率，又可以细分为：随机顶点采样 RN，即每个顶点被采样到的概率都是 1/N ②；随机度顶点采样 RDN，即根据各个顶点度（Degree） ③ 的大小来决定每个顶点被采样到的概率; 以及随机 PageRank 顶点采样 RPN，即根据各个顶点的 PageRank 值来决定每个顶点被采样到的概率。随机顶点选择采样往往需要知道整个网络图数据的某些全局的统计信息，比如总顶点数、各个顶点的度或 PageRank 值等，因此不适用于上述提到的对完整拓扑结构未知的大规模网络平台。
- **基于爬虫的采样模式**，即首先随机地选取一个顶点然后在这个顶点的邻域附近探索选取其他顶点进行采样的方式。这类采样方式比较典型的采样算法有：均匀顶点邻居采样（每次均匀随机选取一个顶点，然后将它以及它指向的所有的邻居都加入采样样本）、基于遍历的采样算法（BFS、DFS、Forest Fire 等）和随机游走采样等。

①用户程序接口。

②N 为网络图中的总顶点数

③该顶点连接边的数目

在上述提到的两类采样算法中，基于爬虫的采样算法通常只需访问网络图中的局部信息，因此是当前图采样算法中的研究热点。但 BFS、DFS 等算法采样得到的样本往往偏向于度数大的顶点且无法理论分析这种偏差。基于随机游走的采样算法因为其不仅计算高效，而且具有良好的理论基础，所以成为当前主流的图采样算法。

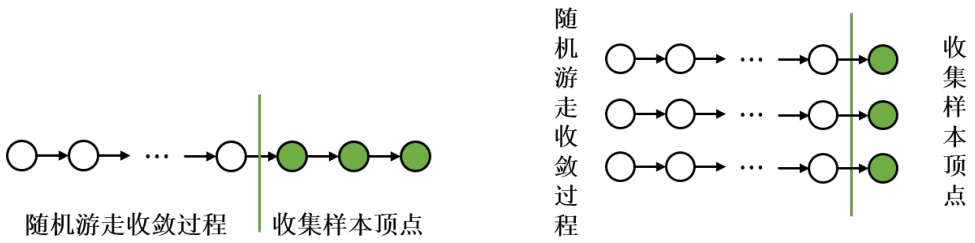
（三）基于随机游走图采样算法

接下来首先介绍基于随机游走的采样算法的计算流程，然后阐述随机游走具有的理论基础如何能实现图上的无偏采样。

（1）基于随机游走的采样过程

基于随机游走的采样过程主要包括通过随机游走算法收集采样样本和对采样样本进行无偏估计这两个步骤。

- **通过随机游走收集样本**，一般有两种方法：（1）连续采样，在图中只开启一条随机游走，收敛以后持续采集样本直到收集到足够的样本顶点。（2）独立采样，在图中同时开启多条随机游走，每条随机游走收敛以后只采集一个样本顶点。这两种方法都需要在随机游走收敛以后才能开始收集样本，然后根据稳态分布对样本进行进行无偏估计。



(a) 连续采样　　(b) 独立采样

图 2.7　通过随机游走收集样本

- **对样本进行无偏估计**，假定图上的一个测量指标可以表示成一个函数 $f: V \to R$，在一个达到稳态分布 π 的随机游走上收集到的足够的样本上应用函数 f，即可得到该测量指标的一个无偏估计值：

$$E_\pi[f] \triangleq \sum_{u \in V} f(u)\pi(u). \tag{2.1}$$

只要采样收集的样本数目足够，则估计值的准确性可由强大数定律（*the Strong Law of Large Numbers, SLLN*）保证。

（2）随机游走采样的理论分析

考虑一个无向连通图 $G(V, E)$，$N(v)$ 表示图中顶点 v 的邻居集合，$deg(v) = |N(v)|$ 表示顶点 v 的度。在该无向图上简单随机游走过程是：首先从图中随机选

取一个顶点，然后重复从当前顶点中随机挑选一个它的邻居顶点进行跳转。这个过程可以看成是一个有限的马尔科夫链，每一步访问的顶点 id 就是该马尔科夫链的状态，每一步的状态转移概率可以表示成一个 $|V| \times |V|$ 的概率转移矩阵 $\boldsymbol{P}$，$\boldsymbol{P}_{uv}$ 为从顶点 u 通过一步随机游走走到顶点 v 的概率。

以**简单随机游走（Simple Random Walk, SRW）**为例来说，SRW 在每一步均匀随机的选取一个邻居顶点，它状态转移矩阵可以表示为：

$$P_{uv} = \begin{cases} 1/deg(u) & \text{if } v \in N(u), \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

很多步以后达到收敛状态，即随机游走访问图中每个顶点的概率呈现稳态分布。SRW 收敛后的稳态分布可以表示成：

$$\pi(u) = \frac{deg(u)}{(2|E|)}. \tag{2.3}$$

基于随机游走的采样算法在随机游走收敛以后开始采样收集样本。根据收集到的样本和收敛后的稳态分布，就可以对感兴趣的一些图测量指标（如各种图的中心性）进行无偏估计。

2.4.2　针对随机游走采样的算法优化

根据上面的分析可以得知，SRW 收敛之后得到的稳态分布并不是均匀分布，图中每个顶点被采样到的概率与该顶点的度成正比。因此 SRW 不适用于某些需要均匀采样顶点的图采样场景。针对这个问题，部分相关工作设计优化的随机游走算法来矫正随机游走的稳态分布，使之成为均匀分布。此外，SRW 的收敛速度往往非常慢，严重影响的图采样的效率。因此部分相关工作则致力于加速随机游走的收敛速度。下面分别介绍这两类针对随机游走采样的优化算法的研究现状。

（一）矫正随机游走的稳态分布

这类优化工作中最典型的两个优化算法就是 Metropolis-Hasting 随机游走 (MHRW)[70] 和最大度数随机游走 (MDRW)[71]。

- **MHRW** 采用一种基于拒绝的采样方式，即在每一步随机游走的邻居选择时，首先随机均匀的选取当前顶点 u 的一个邻居顶点作为候选顶点，然后以概率 $\min\{1, \frac{deg(u)}{deg(v)}\}$ 去接收该候选顶点作为随机游走的下一跳，如果跳转失败则重新选择候选顶点，重复上述步骤直至随机游走成功跳转。可以证明 MHRW 的稳态分布是均匀分布[70]，即 $\pi_u = \frac{1}{|V|}$。根据上述过程，可以得出 MHRW 的状态转移矩阵 p_{uv} 为：

$$P_{uv}=\begin{cases}\frac{1}{deg(u)}\cdot\min\{1,\frac{deg(u)}{deg(v)}\}, & \text{if } v\in N(u),\\ 1-\sum_{w\in N(u)}P_{uw}, & \text{if } v=u,\\ 0, & \text{otherwise.}\end{cases}\tag{2.4}$$

- **MDRW** 根据 SRW 的稳态分布中每个顶点被采样到的概率与该顶点的度成正比这个特点，首先获取图中顶点的最大度数，然后为图中每个顶点增加多条连向自己的边，使得新构造的图中的每个顶点度数都等于原图中的最大度数。因此原图中的 MDRW 就相当于在新图上运行 SRW 的过程。因为在 MDRW 中每个顶点被访问到的概率都是相同的，所以其对应的稳态分布也是均匀分布。MDRW 的状态转移矩阵 p_{uv} 可以表达为：

$$P_{uv}=\begin{cases}1/d_{max} & \text{if } v\in N(u),\\ 0 & \text{otherwise.}\end{cases}\tag{2.5}$$

MHRW 和 MDRW 都可以使得随机游走最终产生均匀的稳态分布，但是 MHRW 和 MDRW 在每一步都会有很大的概率回到自身，所以采样过程中会产生大量的重复样本顶点，从而影响了采样效率。另外，MDRW 还需要预先获得给定图数据集中的最大度数，这对于需要采样的大规模的图数据来说往往也是不切实际的。

（二）加速随机游走的收敛速度

由于随机游走的收敛速度直接关于随机游走采样的效率，所以各种加快随机游走收敛速度的算法被不断提出，其中主要有两个优化方向。因为随机游走的收敛速度与图的导通性成正比，所以一类优化思路就是增加图的导通性[72-75] 另外随机游走收敛慢的一个重要原因就是随机游走容易陷入局部区域，因此另一类优化思路就是修改随机游走的转移概率[28,76]。下面分别介绍这两类优化思路的代表算法。

（1）增加图的导通性

这类优化算法的主要思想是通过对给定的图数据增加不同社区子图之间的边，或者删除同一个社区子图内部的边来增大相应的图数据的导通性，从而提升随机游走的收敛速度。这类算法中有代表性的算法是 Heuristic X-1-C 算法[74] 和 MTO-Sampler 算法[72]。

- **Heuristic X-1-C** 首先利用现有的图分解算法，将给定的图数据分解成多个不相连的社区子图，然后在不同的社区子图之间随机地增加一些连接边，即在任意两个子图中各自任选一个顶点组成一条新的边，从而增加图数据的整体导通性。

- **MTO-Sampler** 从图的导通性定义出发，用同一个社区子图内部的边来替换不同社区子图之间的边，具体来说对于图中任意一对相邻的顶点 u 和 v，若满足 $\lceil \frac{|N(u)\cap N(v)|}{2} \rceil + 1 > \frac{1}{2} max\{deg(u), deg(v)\}$，则判断这两个顶点组成的边 (u, v) 是同一个局部区域子图内部的边，就将其随机替换为属于不同局部区域子图之间的边 (u, w)。

这类优化算法大多是在假定图全局或局部的结构数据信息已知的前提下，通过改变图的拓扑结构来提升随机游走的收敛速度，因此并不适用真实世界场景中的一些图数据。

（2）修改随机游走的转移概率

在这类算法中有代表性的算法是 Non-backtracking Random Walk (NBRW) 算法[28]、Circulated Neighbors Random Walk (CNRW) 算法[76] 和公共邻居感知的随机游走算法 (CNARW)[77-78]。

- **NBRW** 注意到随机游走在每一步都会有一定的概率回到刚刚访问过的顶点，这会有很大的概率带来很多重复的样本，从而影响采样效率。因此 NBRW 在随机游走的每一步只从除了刚刚访问过的顶点之外的邻居顶点中选择随机游走的下一跳。文献[28] 中指出 NBRW 具有时间不可逆 (Non-reversible) 的性质，因此，NBRW 拥有比 SRW 更快的收敛速度的同时，其收敛后的稳态分布也与 SRW 的一致，因此很方便对其进行和 SRW 同样的稳态分析和无偏估计。

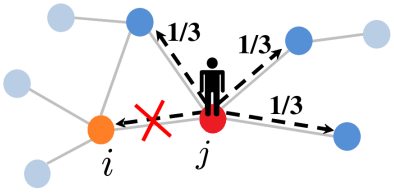


图 2.8　NBRW 算法

- **CNRW** 在随机游走的每一步优先选择未曾被随机游走访问过的邻居顶点，因为当图中的一个顶点曾经被访问过，然后现在又回到当时的状态，则说明从该邻居顶点出去时可能会容易引起随机游走在局部子图的循环。文献[76] 中指出 CNRW 也拥有与 SRW 相同的稳态分布。

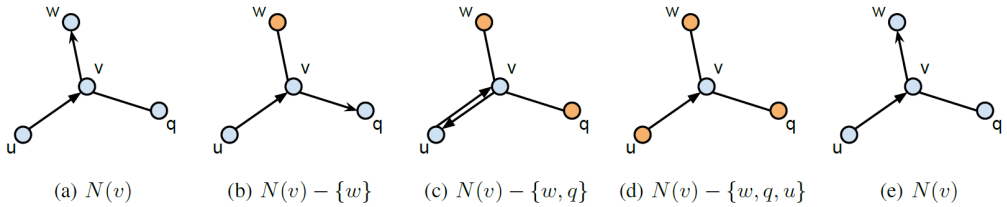
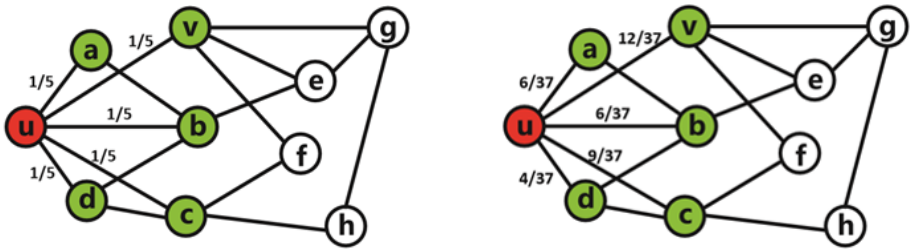


图 2.9　CNRW 算法

- **CNARW** 进一步利用随机游走访问路径上的历史顶点信息加快随机游走的收敛速度，具体来说，CNARW 在选择随机游走的下一跳时会优先选择和当前顶点公共邻居较少且自身邻居数目较大的候选顶点。图 2.10展示了 SRW 算法和 CNARW 算法在一个样例图上进行一步随机游走的对比示例。



(a) SRW（均匀游走）　　(b) CNARW（加权游走）

图 2.10　SRW 和 CNARW 的对比

注：SRW 从当前顶点的邻居中均匀随机地选择下一个顶点，而 CNARW 有较高的概率走到与当前顶点的邻居度较大且邻居较少的邻居。

这些随机游走优化算法利用随机游走的历史路径及顶点邻居信息，不同程度的提升了随机游走的收敛速度，验证了随机游走历史路径信息以及各个候选节点邻域信息的重要作用。如何更加充分地去利用好这些信息来进一步加速随机游走的收敛仍然是一个充满意义和挑战的问题。

第 3 章　随机游走友好的图存储管理系统

本章摘要: 图数据中的顶点通过大量的边相互链接，从而构成错综复杂的关联关系。当大规模的图数据存放在磁盘上的时候，图计算任务对图数据的访问往往会带来对磁盘大量的随机 I/O，从而造成性能瓶颈。传统的图系统通常采用基于迭代的模型，顺序迭代地从磁盘加载图数据块进入内存进行计算分析，从而减少随机 I/O。然而这种基于迭代的模型在支持基于随机游走的图计算任务时表现出很低的 I/O 效率，从而限制了图上随机游走的效率和扩展性。本章结合随机游走应用的特征，提出一种基于随机游走状态感知的 I/O 模型，并基于此模型设计了面向大规模并发随机游走的高效图数据存储管理机制 SAStore 。SAStore 根据图上随机游走的数量、步长以及随机游走在图中的分布情况等状态信息，来执行不同的图数据的组织划分、加载和缓存策略，从而实现 I/O 效率的最大化。此外，SAStore 进一步针对动态图处理的场景下更新效率慢的问题，提出一种基于分块日志的 CSR 存储实现快速的图更新。本章基于上述设计实现了原型系统，并在真实世界的图数据集上进行了性能评估。实验结果显示，SAStore 在支持随机游走类的应用时，对比于最新的单机随机游走图系统 DrunkardMob，I/O 效率平均可以提升 *2* 倍到 *4* 倍。另外，本章进一步针对动态图场景下数据更新效率低的问题，设计了一种基于分块日志的 CSR 存储管理方案，对比于基础的 CSR 存储格式，在保证图数据查询效率的同时，显著提升了动态图数据的更新效率。

3.1　本章介绍

本章工作聚焦于随机游走友好的图存储管理系统的研究，传统的基于磁盘的单机图处理系统通常采用基于迭代的模型，预先将图数据块划分存放于磁盘上，然后顺序迭代地从磁盘加载图数据块进入内存进行计算分析，从而减少随机 I/O。章节 §2.2中已经详细阐述了基于单机的磁盘驻留图的图处理会带来的对磁盘图数据大量随机访问的问题，以及现有的基于迭代的模型下图数据的存储管理方案以及如何解决大量随机访问问题。

但现有的图处理系统都没有专门针对随机游走类应用的图存储和 I/O 优化，因此在支持随机游走类应用时 I/O 效率依旧不高。本章首先分析这种基于迭代的模型在支持随机游走类应用时表现出的局限性，从而引出本章研究工作的出发点——提升随机游走对图数据访问的 I/O 效率; 其次，介绍本章工作的主旨思想和主要贡献点；然后，详细介绍随机游走状态感知的图存储管理系统 SAStore 的设计与实现；最后对 SAStore I/O 效率进行实验评估。

3.2　研究出发点

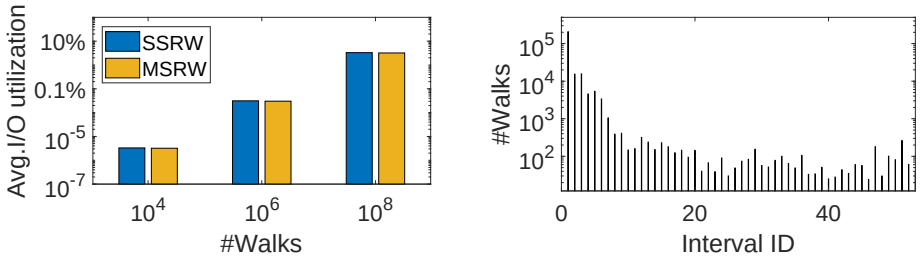
上述介绍图系统都采用基于迭代的 I/O 模型，迭代地加载磁盘上的所有子图图数据，在每轮迭代中，顺序地加载所有需要的子图一次。但是，由于随机游走步骤中随机邻居的选择加剧了对图数据的随机访问，且造成各个子图分区之间的访问非常不均衡，均匀的加载每个子图并不是一个合适的方式；另外许多在线图数据查询算法也是基于随机游走实现的，此时随机游走只需要访问整个图数据的一部分，迭代的加载所有图数据更造成大量的无效 I/O，所以带来*加载子图 I/O 效率低*的问题。另外，上述介绍的图系统大多都没有考虑动态图处理的场景，因此处理图数据更新的效率就会特别慢，带来*图更新效率低*的问题。下面仔细阐述现有图数据存储管理的这两点不足。

3.2.1　加载子图的 I/O 效率低

首先，*基于迭代的模型*在支持基于随机游走的应用时会带来 *I/O 效率低*的问题的原因，这里 *I/O 效率*定义为：通过一次 I/O，也就是加载一个子图后，子图中用于更新随机游走的边数除以加载子图的总边数。从图中出发的很多并发的随机游走，即使这些随机游走从同一个源顶点出发，在经过几步游走之后，这些随机游走可能散布到整个图中。而且这些随机游走在各个子图之间的分布是非常不均衡的，所以有些子图中可能只有很少的随机游走，但也需要在每一轮都被加载到内存中，因此它带来了极低的 I/O 效率。

我们进一步通过实验来观察基于迭代的模型在运行随机游走时的 I / O 的利用率以及随机游走在各个子图之间的分布。我们使用最新的单机随机游走图计算框架 DrunkardMob 在真实世界的数据集 Friendster （包含 6800 万个顶点和 26 亿条边）分别运行 10^4、10^6 和 10^8 条长度为 10 的随机游走，并考虑这些随机游走从单个源顶点出发（Single-Source Random Walk, SSRW）和从多个随机的源顶点出发（Multi-Source Random Walk, MSRW）的场景。具体实验设置请参考 §3.5.1。图 3.1(a) 显示了平均 I/O 效率，SSRW 在随机游走的数量为 10^4、10^6 和 10^8 的情况下的平均 I/O 效率只有 3.1×10^{-6}，3.2×10^{-4} 和 0.032，MSRW 下的运行结果也是类似的。值得注意的是，这里表现出的 I/O 效率非常低，特别是在很少的步进次数的情况下。图 3.1(b) 进一步展示了从单个源顶点开始运行 10^6 条随机游走时，经过 4 次迭代后这些随机游走在各个子图之间的分布。实验结果显示，这些随机游走仅仅经过 4 步就分布在所有的子图上，而且分布是严重倾斜的。例如，一个子图包含的随机游走的数目的最大值。

最近的一些工作，比如 DynamicShards[47] 和 Graphene[48] 采用*按需加载策略*，动态调整图块布局，跳过不包含任何随机游走的子图，减少无用边的加载。但只要在一次迭代中某个子图中只要包含一条边，那么这个子图仍然需要加载到内



(a) I/O 效率　　(b) 随机游走在各个子图间的分布

图 3.1　I/O 效率对比和随机游走分布

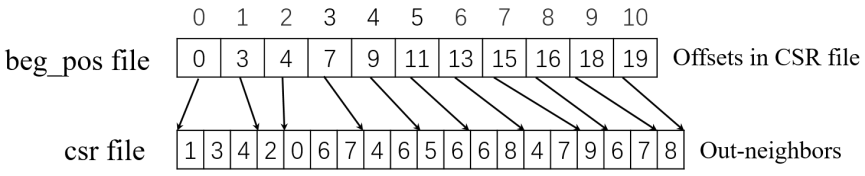
存中进行计算。所以低 I/O 效率的问题仍然没有完全解决。我们也对 Graphene[48] 进行了相同的实验，运行 MSRW 算法，分别设置随机游走的数目为 10^4、10^6 和 10^8 的情况下，平均 I/O 效率分别为 6.1×10^{-3}、3.1×10^{-3} 和 0.032。实验结果显示，Graphene 可以在步行次数较少的情况下大大提高 I/O 效率，但在大量步行的情况下，I/O 效率仍然有限。

3.2.2　动态图场景下图更新和图查询性能难以均衡

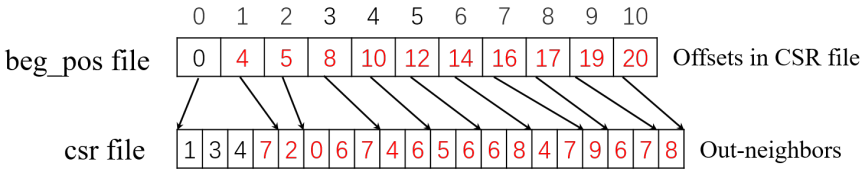
（1）CSR 格式下的图数据的更新与查询

本章第 §2.2.2小节介绍了最新的图处理系统中最常用的图存储格式 CSR，基于 CSR 的图存储格式存储开销非常低，针对图数据的查询访问性能也非常高，因此非常适用于顶点和边都不会发生变化的静态图的存储场景。但是在动态图场景下，当有新的顶点或边数据插入时，想要融入增量图数据，CSR 需要重构 CSR 序列和 beg_pos 序列的存储内容。举例说明，当在磁盘中采用 CSR 格式来存储图 2.2(a) 中的样例图，则会产生两个文件：依次存储各个顶点的出边邻居顶点的 *csr* 文件和存储各个顶点在 CSR 中偏移量的 *beg_pos* 文件，其存储内容如图 3.2(a) 所示，图中的箭头表示各个顶点在 *csr* 文件中的偏移量的逻辑对应关系。当需要在图中插入一条 $0\rightarrow7$ 的边时，则首先需要在 *csr* 文件中的对应位置插入 7（对应到文件操作几乎要重写整个 *csr* 文件），然后要更新 *beg_pos* 文件中顶点 0 后面所有顶点对应的偏移量的值，也需要重写整个 *beg_pos* 文件，重写之后的存储内容如图 3.2(b) 所示，其中红色的内容表示文件中需要更新重写的部分。因此，可以得知基于 CSR 的图存储格式在处理图更新时效率非常低。而当不断有增量数据插入时，CSR 重构的开销就非常大，在真实的场景中往往难以实现。

为了支持动态图场景下的图计算，部分图处理系统中提出采用一种快照的方式，将静态图数据与动态插入的增量图数据分开存储。这种方式虽然能很快的插入图数据的更新信息，但是在进行图分析计算时，需要分别访问静态图数据以及动态增量图数据，造成额外的查询开销，影响分析计算的性能。而且随着增量图数据的不断增加，也越来越难以维护，存储和计算开销都将变得越来越困难。



(a) 图 2.2(a) 的 CSR 存储



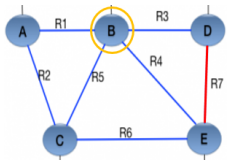
(b) 图 2.2(a) 中插入一条边后的 CSR 存储更新

图 3.2　CSR 中的图数据更新

（2）图数据库中图数据的更新与查询

为了支持动态图场景下图结构数据的快速更新，工业界也有很多图数据库产品应运而生，其中最典型的原生图数据库为 Neo4j[63]，这也是目前使用最多的图数据库，在所有数据库排行榜上排名 18 位，且一直呈上升趋势[65]。接下来介绍 Neo4j 的底层存储结构以及 Neo4j 如何实现图结构数据的快速更新。Neo4j 采用一种双向链表（double-linked list）的结构来存储管理图结构数据，由顶点列表和边列表两个链表构成，其中顶点列表依次存储各个顶点的顶点 ID、顶点是否有效和该顶点的第一条连接边的边 ID，边列表中依次存储各条边的边 ID 以及其源顶点和目的顶点的 ID 以及他们连接的上一条边和下一条边的边 ID。图 3.3中简单举例说明，其中图 3.3(b) 和图 3.3(c) 分别表示图 3.3(a) 中例图的顶点列表存储和边列表存储。在此存储结构下，当要进行图更新，比如新增一条边数据 $R7：D \rightarrow E$，只需要往边列表的尾部追加一条边记录，并修改这条边的源顶点和目的顶点连接的上一条边记录中的 *next* 域的信息，因此图数据的更新开销非常高效。

然而，Neo4j 的这种双向链表图存储结构对于图数据的查询性能不友好，比如当要查询顶点 ***B*** 的所有邻居顶点，首先需要去顶点列表中找到顶点 ***B*** 的第一条边 ***R***1；然后去边列表中访问边 ***R***1，获得顶点 ***B*** 的第一个邻居顶点为 ***A***；然后通过 ***R***1 中顶点 ***B*** 的 *next* 信息得知下条边为 ***R***3，再去访问边 ***R***3，得到顶点 ***B*** 的第二个邻居顶点为 ***D***；然后再按照同样的步骤得到顶点 ***B*** 的第三、四个邻居顶点分别为 ***E*** 和 ***C***，一直到得到顶点 ***B*** 的 *next* 信息为 *null* 时，即获取了顶点 ***B*** 的所有邻居顶点。从上面的过程可以看到，这种双向链表图存储在进行图数据的查询时会来带对边列表数据的大量随机访问，因此图数据的查询效率比较低。尤其是当图数据的规模比较大不得不存放在磁盘时，更是带来对磁盘大量的随



(a) 例图

	inUse	1stEdge
A	1	R1
B	1	R1
C	1	R2
D	1	R3
E	1	R4

(b) 顶点列表

	sId	prep	next	tId	prep	next
R1	A	null	R2	B	null	R3
R2	A	R1	null	C	null	R5
R3	B	R1	R4	D	null	R7
R4	B	R3	R5	E	null	R6
R5	B	R4	null	C	R2	R6
R6	C	R5	null	E	R4	R7
R7	D	R3	null	E	R6	null

(c) 边列表

图 3.3　Neo4j 的双向链表图存储结构中的图更新和图查询

注：红色表示插入一条边 $D \rightarrow E$ 需要修改的边列表的内容（图更新），黄色表示查询顶点 B 的所有边需要访问的顶点列表和边列表中的内容（图查询）。

机 I/O，造成性能瓶颈。因此 Neo4j 一般只适用于内存图数据场景，文献[79] 也指出 Neo4j 在外存场景下访问图数据的效率会很差。

3.3　基本思想和贡献点

（一）本章工作的主旨思想

本章第 §3.1节介绍了现有的图处理系统中采用的基于迭代的模型在支持随机游走类的应用时会带来 I/O 效率低下的问题。为了提升随机游走对图数据访问的 I/O 效率从而支持基于磁盘驻留图的单机图处理系统上的快速随机游走，**本章提出了一种基于随机游走状态感知的图存储管理机制，其主要思想是利用随机游走在图中的状态（比如随机游走当前停留的顶点 ID、随机游走的步长等）来自适应地调整图加载策略和加载子图的选择。简单来说，与基于迭代的模型中盲目地顺序加载子图数据块的策略不同，状态感知的模型优先选择加载包含随机游走数目最多的子图数据块。通过这样的方式，可以在每个 I/O 中都使得尽可能多的随机游走得到更新，因此 I/O 效率可以有效地提高。**

下面通过一个具体的样例来更直观地说明基于随机游走状态感知的图存储模型的工作模式以及其为什么能提升随机游走访问图数据的 I/O 效率。考虑从图 2.2(a) 中的示例图上的顶点 0 开始出发三条随机游走，要求每条随机游走移动 4 步。图 3.4展示了状态感知的模型中图加载和随机游走更新的过程。具体来说，样例图被划分成三个子图 b_0、b_1 和 b_2。在第一个 I/O 中，子图数据块 b_0 被加载到内存因为它包含了所有的三条随机游走，之后通过更新计算，随机游走 w_0 和 w_1 移动两步，w_2 只移动一步因为它要走更多步数所需要的图数据此时并不在内存中。然后其中两条随机游走都进入子图 b_2 时，因此在第二个 I/O 中，子图数据块 b_2 被加载进入内存，然后随机游走 w_0 移动两步完成任务，w_1 移动一步。最

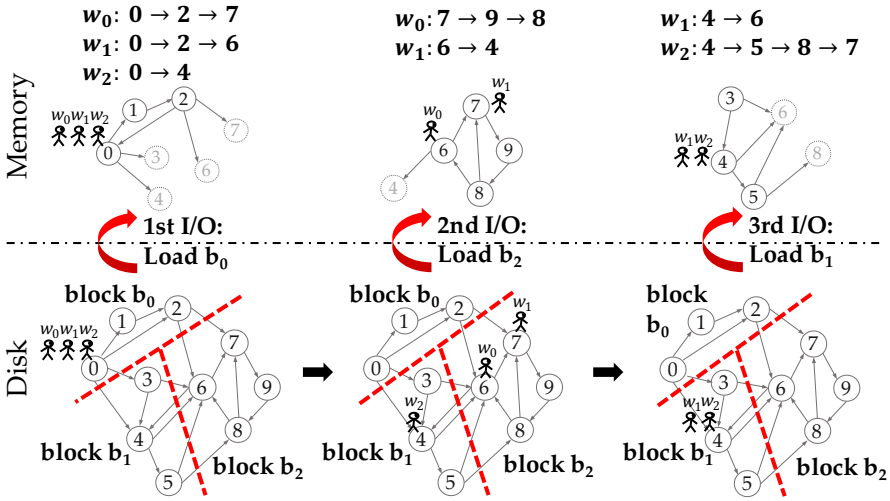


图 3.4　随机游走状态感知的图存储管理

后，剩下的两个随机游走都在子图 b_1 中，所以将 b_1 加载到内存中，最终所有的随机游走都完成了计算任务。因此在本例中状态感知的图加载模型总共只需要 3 次 I/O 就能完成所有计算任务，然而基于迭代的图加载模型可能最多得需要 12 个次 I/O，因为其总共需要进行 4 轮迭代计算来将随机游走移动 4 步，每次迭代运算最多就需要产生 3 次 I/O。

（二）本章工作考虑的主要问题

基于上述设计，本章实现了相应的原型系统 SAStore （State-Aware Graph Storage Management）。SAStore 的设计过程中主要考虑下面几个方面的问题：

- 针对大规模的图数据，采用何种数据组织格式来在磁盘和内存中进行轻量级的图存储，以减少图数据的磁盘和内存的空间占用？
- 在图数据规模过大需要划分子图时，如何设计图数据的划分方案来实现轻量级的图划分，以适应不同应用场景下需要不同大小子图时实现对图数据的快速重新划分？
- 在随机游走的计算过程中，如何决策下一个要进行加载计算的子图，以及如何利用多余的内存空间来更有效的缓存子图数据以减少后续数据 I/O？
- 在动态图场景下，如何设计在现有的图存储格式的基础上添加对增量图数据更新的支持，以及如何实现图更新和图访问之间的性能均衡？

（三）本章工作的主要贡献点

相比现有的基于磁盘的单机图处理系统中图数据的存储格式和 I/O 调度方案和基于对上述问题的考虑，本章工作具有以下贡献点：

(1) **针对随机游走场景的专门优化设计：**本章工作着手于基于随机游走的图计算这一类特殊的应用场景，并结合上一章节中对随机游走类应用场景的探索，针对随机游走的应用特征考虑专门的优化设计，通过感知随机游走计

算的状态信息来自适应地调整更高效的图数据访问策略，为上层随机游走的计算提供高效的图数据管理方案和接口。

(2) **状态感知的子图划分配置：**本章工作在现有 CSR 存储格式的基础上设计了轻量级的子图数据的逻辑划分方案，以适应不同应用场景下不同大小子图的需求，快速重新划分子图；并且从当前图计算任务的随机游走的规模配置出发，设计启发式的子图大小配置；并在随机游走的计算过程中，随时感知随机游走状态的变化，设计状态感知的自适应的子图大小调整策略，从而实现子图划分的最优配置。

(3) **状态感知的图加载策略：**本章工作在划分好子图的基础上，进一步根据随机游走在各个子图之间的分布状态，设计状态感知的图加载策略，使得通过一次 I/O 能让最多的随机游走得到转发，以最大化每次数据加载的 I/O 效率，从而提升整体的 I/O 效率和计算性能。

(4) **基于分块日志的动态图存储方案：**为了进一步支持动态图场景下的图数据的动态更新，本章工作在现有图存储格式 CSR 的基础上，设计基于分块边日志的管理方案用于缓存新增的边数据，在保证 CSR 格式高效的图访问性能的同时，提升图数据的更新效率，从而实现动态图场景下图更新和图查询的性能均衡。

(5) **性能提升：**本章工作基于上述设计，实现了大规模图数据的存储管理机制的原型系统 SAStore，并在真实的图数据集上使用基于随机游走的图算法对其进行性能评估。实验结果表明，SAStore 对比于最新的单机随机游走图系统 DrunkardMob 可以实现平均 *2* 倍到 *4* 倍的 I/O 效率的提升。基于分块日志的 CSR 存储管理方案对比于基础的 CSR 存储格式，在保证图数据查询效率的同时，实现了高达两个数量级的更新效率的提升。

3.4　SAStore 设计与实现

为了能够提高基于磁盘驻留图的大规模图分析系统中图数据的 I/O 效率，从而支持图上高效的随机游走，本章设计并实现了面向大规模并发随机游走的高效图数据存储管理机制 SAStore。SAStore 的主要设计思想是根据大量并发随机游走在整个图数据中分布的状态，包括随机游走的总量、随机游走在各个子图之间的数量分布、随机游走的最短步长等状态信息，来实现图数据的组织划分、加载缓存以及动态更新等，从而实现对图数据 I/O 效率的最大化。本节首先介绍 SAStore 的系统架构，主要包含主要包含图数据的组织和划分方案、图数据的加载和缓存策略以及动态图场景下的图数据更新和查询机制。然后依次详细介绍三个设计模块在系统中的实现细节。

3.4.1　SAStore 系统架构

SAStore 的系统架构图如图 3.5所示，SAStore 的设计主要包含三个部分：图数据的组织和划分、图数据的加载和缓存以及动态图的更新和查询。

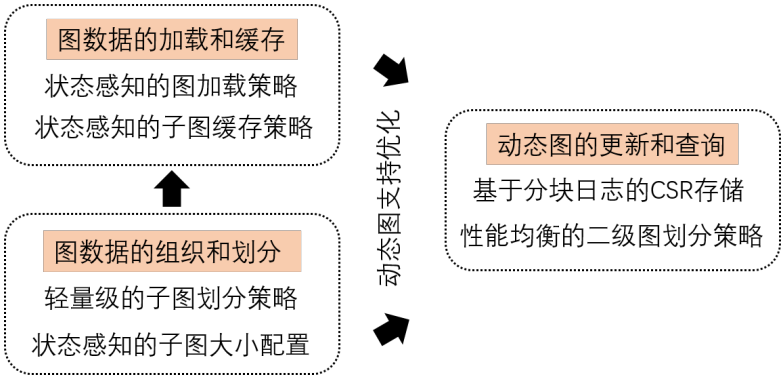


图 3.5　SAStore 的整体架构

- **图数据的组织和划分模块**首先采用基于 CSR 的数据组织方式，设计了一种轻量级的图划分策略来实现高效灵活的子图划分；然后根据随机游走的总量状态设计状态感知的自适应的子图大小配置，实现最优的图数据的组织和划分，为接下来的子图加载和缓存提供基础。
- **图数据的加载和缓存模块**首先根据随机游走在各个子图之间的数量分布，设计了一种基于状态感知的图加载策略，即总是优先选择包含最多随机游走的子图加载到内存进行计算，从而实现通过一次 I/O 使得最多的随机游走能够得到转发，提高 I/O 效率；然后进一步设计了状态感知的图缓存策略，以子图为单位选择性地在内存中缓存随机游走数目最多的子图，从而进一步提升缓存效率。

  动态图的更新和查询模块进一步针对动态图场景下数据更新慢的问题，设计了一种基于分块日志的 CSR 的图存储策略，在保障图查询性能的同时实现快速的图更新。

3.4.2　图数据的组织和划分

（1）轻量级的数据组织和子图划分策略

SAStore 采用广泛使用的压缩稀疏行（Compressed Sparse Row, CSR）的存储格式来组织管理图数据，首先在磁盘的一个 *csr* 文件中按照顶点 ID 的顺序依次存储图中各个顶点的出边邻居的顶点 ID，然后使用一个 *index* 文件来记录每个顶点在 *csr* 文件 中的开始位置的偏移量。这样通过两次索引查找就能准确找到某个顶点所有的出边邻居有哪些。值得注意的是，由于随机游走通常只需要访问顶点的出边而不会访问一个顶点的入边，因此默认情况下只存储顶点的出边

邻居信息。这种轻量级的图数据组织降低了每个子图的存储成本，从而减少了图加载的时间成本。

然后，SAStore 根据顶点的 ID 将完整的图数据划分成多个子图。具体的划分方式如下：按照顶点 ID 的升序顺序依次将一个顶点及其出边邻居添加到当前的子图中，直到当前子图的数据量超过预先定义的子图大小时，就创建一个新的子图。SAStore 中轻量级的子图划分策略只需要简单地扫描一遍 *index* 文件就可以知道每个顶点包含的出边的个数，根据这些顶点出边的个数进行均匀边的子图划分，然后将每个子图的开始顶点记录在一个 *block* 文件中，这样通过简单的 *block* 文件的扫描就能知道每个子图包含的顶点的区间范围。这种轻量级的子图划分策略使得子图的划分非常高效，可以灵活地根据不同的应用场景来重新调整子图的大小的划分，后续高效 I/O 奠定了基础。图 3.6展示了图 2.2(a) 中样例图的数据组织和划分。

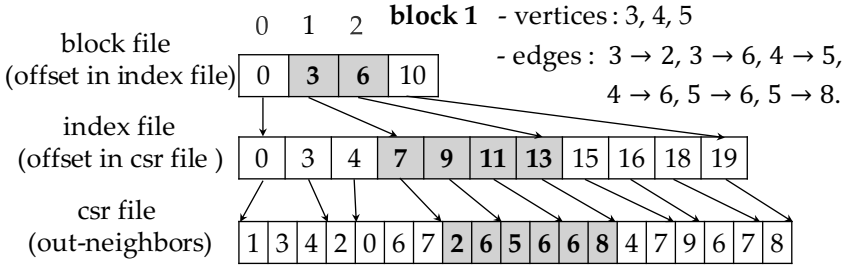


图 3.6　图 2.2(a) 的数据组织和划分

注：图数据以 CSR 的格式存储，记录于 *csr* 文件和 *index* 文件中，划分的子图的顶点区间范围记录于 *block* 文件中。比如子图 1 包含顶点的区间范围是 $[3, 6)$，即包含顶点 $3, 4, 5$，包含的边包括 $3 \to 2, 3 \to 6, 4 \to 5, 4 \to 6, 5 \to 6, 5 \to 8$。

（2）启发式的子图大小配置

上一步确定了图数据的组织方式和子图的划分策略，接下来考虑子图大小的配置。我们发现，子图大小的设置存在一个性能权衡：将每个子图的大小设置的比较小时，可以避免加载更多的无效数据，即更新该子图中的随机游走时并不会访问的子图中的那些边数据；而将每个子图的大小设置的比较大时，每次通过一次 I/O 加载一个子图到内存中计算时，就可以使得更多的随机游走得到更新，每条随机游走也有更大的机会能够在当前子图中更新多步。此外，不同的基于随机游走的分析任务通常需要同时出发不同规模的随机游走，因此这些不同分析任务通常也偏好不同的子图大小的配置。只出发少量随机游走的轻量级的分析任务通常更倾向于设置较小的子图，因为在此设置下，I/O 效率可以得到提高。相反，同时出发大规模随机游走的重量级的分析任务通常更倾向于设置较大的子图，因此在此设置下，可以提高随机游走的更新率。我们通过一个实验上的

实证分析（见章节 §3.5.2）验证了上述理解，并基于上述理解，设计了一个状态感知的启发式的子图大小配置策略：即根据当前分析任务的随机游走的总量，启发式的设置一个默认的子图大小，即设置每个子图的大小为：$2^{(\log_{10} R+2)}$ MB，其中 R 为随机游走的总数目。例如，当一个分析任务需要同时出发十亿（10^9）条随机游走时，默认的子图大小可以设置为 $2^{(9+2)}$ MB，即 2 GB。默认设置的子图的大小通常比普通机器的内存容量小，所以很容易在内存中保存一个完整的子图信息。

（3）自适应的子图大小配置

此外，实验结果表明一个固定的子图大小的设置也不完全适用于同一个基于随机游走的分析任务的全部执行过程，因为当前分析任务的随机游走的总量也会随着程序的执行而逐渐减少。并且这些随机游走的转发会出现*全局游荡者*（*Global Stragglers*）问题，即图中的大部分的随机游走会在加载的子图当中得到很多步的转发，因为这些随机游走转发所需要的图数据正好都在加载的子图当中，然而另外有一些随机游走则会移动的非常慢，因为这些随机游走可能会被困在一个比较冷的子图数据块中，因而很长一段时间内没有被加载到内存中进行处理。因此，大多数的随机游走能够很快计算完成，但最后少量的随机游走（游荡者）却需要花费大量的 I/O 和计算时间。为了更好地理解*全局游荡者问题*，我们在真实世界的图数据 Twitter 数据集上运行 RWD 算法，该算法从每个顶点开始出发一条随机游走，并要求每条随机游走总共转发 6 步。具体的数据集描述、算法介绍以及其他实验设置详情参见章节 §3.5.1。图 3.7中显示了完成每次 I/O 后的计算（即完成每次加载的子图上的随机游走转发）之后剩余的随机游走的数量占总量的比例。通过实验发现，最后少量的随机游走通常需要将近一半的 I/O 和计算时间来将其转发完成。

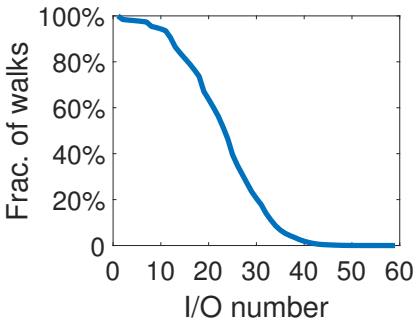


图 3.7　全局游荡者问题

注：大多数的随机游走能够很快计算完成，但最后少量的随机游走（游荡者）却需要花费大量的 I/O 和计算时间，比如最后 20% 的随机游走需要超过一半的 I/O 来将其转发完成。

通过后面章节的优化（参见下一小节 §3.4.3），可以在一定程度上缓解*全局游荡者问题*，即减少最后少量的随机游走（游荡者）需要花费的 I/O 的数量。与此同时，考虑到随着程序的执行，图中随机游走的数量逐渐减少，也可以通过减少每次 I/O 的数据量来进一步减少*全局游荡者问题*带来的时间开销。具体的，本章进一步设计一种*自适应的子图大小配置策略*，在程序的执行过程中，根据随机游走的剩余数量的状态，自适应地调整子图的大小。为了便于子图的管理，首先将完整的图数据划分成很多个细粒度的小子图（比如每个子图的大小设置为 2MB），然后以多个连续的子图为一组组成一个大的子图，以大子图为单位进行图数据的加载和随机游走的转发计算。每次要选择子图数据进行加载和计算时，首先根据当前剩余的随机游走的总量来计算适合的待加载子图的大小，比如在前面的例子中，同时出发十亿条随机游走时，适合的子图大小为 2 GB，那么在程度执行的开始阶段，可以加载连续的 2 GB / 2 MB = 1024 个小子图进入内存进行计算。随着程序的执行和随机游走总量的减少，比如当随机游走只剩下一万条未完成计算时，适合的子图大小为 2^6 = 64 MB，此时可以加载连续的 64 MB / 2 MB = 32 个小子图进入内存进行计算。通过这样自适应的子图大小配置，可以进一步提升 I/O 效率。

3.4.3　图数据的加载和缓存

（1）状态感知的图加载策略

SAStore 在预处理阶段完成上述的图数据的格式转换和子图划分操作。在随机游走的执行阶段，SAStore 每次选择一个子图加载到内存进行计算。那么在众多子图当中，如何选择要加载的子图呢？SAStore 根据随机游走在各个子图中的数量分布状态，设计*状态感知的子图加载策略*。具体来说，SAStore 总是加载包含随机游走数量最多的那个子图的数据块，在完成当前加载的子图数据上的随机游走的计算转发之后，再按照同样的方式选择另一个子图加载计算。

前面章节曾提到（参见上一小节 §3.4.2），*状态感知的子图加载策略*会带来*全局游荡者问题*，即有些随机游走可能移动得非常快，因为它们所需要的图数据因为被加载到内存中所以总是能够得到满足，而有些随机游走可能移动得非常慢，因为它们可能在很长一段时间内都被困在一些没有被加载到内存中的子图数据中。因此，SAStore 可以快速完成大部分随机游走的计算，但是需要花很长时间才能完成剩下的少数随机游走的转发。前面的实验结果显示，最后少量的随机游走（全局游荡者）通常需要将近一半数量的 I/O 来将其计算转发完成。

为了解决*全局游荡者问题*，SAStore 在上述*状态感知的子图加载策略* 的基础上进一步提出一种概率方法，即给这些全局游荡者一定的机会去移动几步从而追赶上大多数随机游走的步伐，从而实现这些随机游走之间的进度同步。具体

来说，每次需要选择一个子图数据块加载时，以一个概率 p 来选择包含进程最慢的那条随机游走（即已经走过的步数最短的随机游走）所在的子图，以概率 $1-p$ 依然选择包含随机游走数量最多的那个子图。结合上述的概率方法，最终本章设计一种*状态感知的子图加载策略*，如图 3.8所示。值得注意的是，随着 p 的增大，*全局游荡者问题*将得到更有效的缓解，但大多数随机游走的计算效率也将降低。因此 p 的设置也关系到性能的权衡。根据实验分析发现 $p = 0.2$ 是一个合适的设置，在某些情况下可以获得 20% 的性能提升。

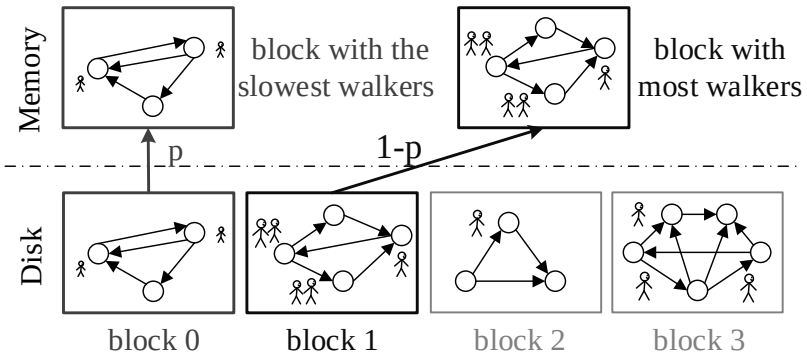


图 3.8　状态感知的子图加载策略

（2）状态感知的子图缓存策略

为了减轻子图大小的设置对性能影响并且提升缓存效率，SAStore 提出一种基于随机游走状态感知的子图缓存策略，允许在内存中同时缓存多个子图数据。考虑到包含更多数目的随机游走的子图数据在不久的将来更容易被访问，因此 SAStore 总是在内存中缓存包含随机游走数目最多的那些子图数据。具体地，状态感知的子图加载和缓存过程如图 3.9所示，首先根据状态感知模型选择一个候选子图数据块，选择完成之后，检查它是否已经缓存在内存中。如果它已经在内存中，那么直接访问内存来执行分析。否则将从磁盘加载该子图数据。如果缓存已满，即内存中已经没有空闲的空间可以用于存储该子图数据，则 SAStore 将从内存的缓存队列中剔除包含随机游走数目最少的子图数据从而空出内存空间。

本章提出的*状态感知的子图缓存策略*与传统的页面缓存（Page Cache）有下面几个方面的不同：首先，*状态感知的子图缓存策略*中不采用预取策略，因为状态感知模型往往并不会顺序访问图数据，所以基于预取的页面缓存在状态感知的 I/O 模型下并不高效。其次，页面缓存策略按照页的粒度管理内存中的数据，而*状态感知的子图缓存策略* 按照子图的粒度管理数据，以适应基于子图的图加载和计算。最后，*状态感知的子图缓存策略*中对缓存队列施行的驱逐政策是根据随机游走的状态决定的，这也不同于页面缓存中使用的*最近最少使用策略*（*LRU*）。章节 §3.4.3中展示了两种缓存策略在不同缓存大小的设置下的性能对比。

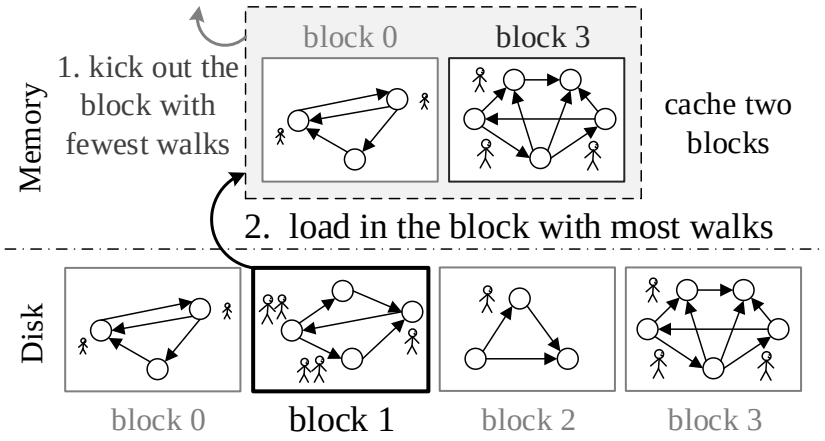


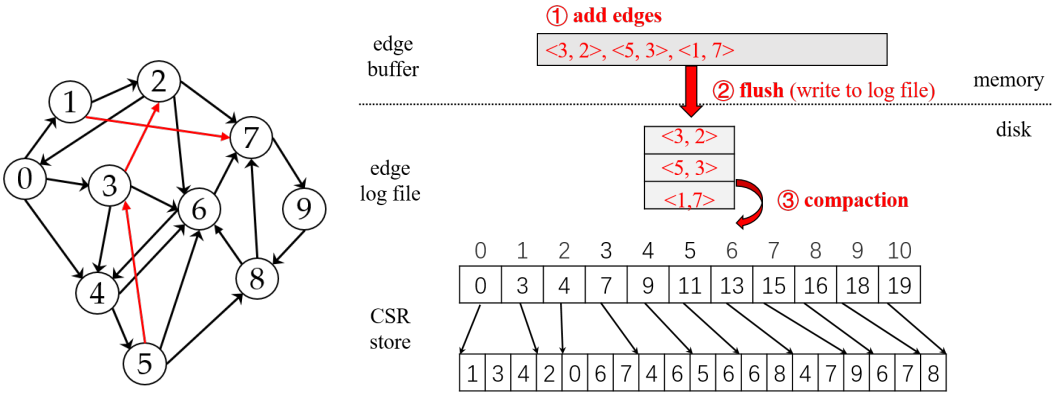
图 3.9　状态感知的子图缓存策略

3.4.4　动态图的更新和查询

（1）基于分块日志的 CSR 存储

为了提升 CSR 中插入增量图数据的性能，一种做法是使用边日志来缓存最近插入的边数据更新，等新增的边数据更新积累到一定数量之后再统一合并到磁盘当中的 CSR 中，这样可以摊销 CSR 的重构开销。考虑到增量数据的持久化需求，本章考虑两层的日志结构，即分别在内存创建一个边缓冲区（*edge buffer*）和在磁盘创建一个边日志文件（*edge log file*）。当新增的边数据到达，首先追加写入内存的边缓冲区；当内存边缓冲区满了，将边缓冲区中的所有的边日志刷盘，追加写入到磁盘的边日志文件中并清空边缓冲区（这个过程称为 *Flush* 操作）；当边日志文件的大小也到达一定的阈值时，再将边日志文件与 CSR 的图数据进行合并，并清空边日志文件（这个过程称为 *Compaction* 操作）。图 3.10(a) 中显示在图 2.2(a) 中新增三条边数据，图 3.10(b) 展示了基于日志的 CSR 存储中的插入这三条边数据的更新过程。

这种基于边日志的 CSR 管理策略可以有效地提升 CSR 中图数据更新的开销，当边日志文件（*edge log file*）的阈值设置的越大，则能缓存的边数据更新就越多，Compaction 操作的频率也就越低，增量图数据的更新性能提升越大。实验表明，随着边日志文件的阈值大小逐渐增大，图数据的更新性能显著提升，但与此同时图数据的查询性能也显著下降，具体的实验设置和实验结果参见本章的第 §3.5.4小节。这是因为在这种基于日志的 CSR 存储管理方案中查询某个顶点关联的边数据时，需要同时到内存的边缓冲区、磁盘的边日志文件和底层的 CSR 存储中去查询，而边数据在边缓冲区和边日志文件中都是无序存放的，因此需要遍历扫描整个边日志文件去查找其中和查询顶点关联的边数据。在系统实现中，使用了位图（*bitmap*）去指示每个顶点在边日志文件中是否有关联的边，以过滤



(a) 图中插入边数据　　(b) 基于日志的 CSR 存储中的更新过程

图 3.10　基于边日志的 CSR 存储管理

掉一些不必要的遍历。但是，只要在边日志文件中有该查询顶点关联的边，就需要遍历查找整个边日志文件去找出它的边。当边日志文件的阈值设置的比较大时就会带来非常大的查询开销，因此就会导致非常差的查询性能。

为了实现动态图场景下图更新性能和图查询性能的均衡，本章进一步提出基于分块日志的 CSR 存储管理策略。如图 3.11所示，将整个大图划分成很多子图，每个子图单独存储成一个 CSR，并且每个子图配一个边日志文件用于暂存该子图中的顶点新增的边数据，后面再批量地合并到对应的 CSR 中去，这样减少图数据更新的开销。具体看一下图更新的过程，首先将新增的边数据写到内存中的边缓冲区中，当边缓冲区满了，通过 Flush 操作将边缓冲区中的边日志追加写到磁盘的边日志文件中，这里 Flush 的过程包括分类边日志（也就是判断每个边日志属于哪个子图）和写边日志（即将每个子图新增的边日志写到该子图对应的边日志文件中）。这里边日志在边缓冲区和边日志文件都是无序存放的，或者说是按照边到达的时间顺序排列的，所以写边日志的过程就是追加写文件。然后，当某个子图的边日志文件增长到一定程度时，就将其通过 Compaction 操作合并到该子图对应的 CSR 当中。通过这种做法，将更新 CSR 的开销拆分并分摊到批量的边日志更新中去。而图查询过程中也只需要查找对应子图的边日志文件而不用遍历所有的边日志，也大大减少了查询开销。因此，这种基于分块边日志的 CSR 存储管理可以在保证图查询性能的同时大幅度提升图数据的更新性能。

（2）性能均衡的二级图划分

为了更好的实现上面提出的这种基于分块边日志的 CSR 存储管理策略，一个关键的问题就是如何划分子图？一个最直观的划分策略就是均匀顶点的图划分方式，即固定每个划分子图中顶点的数目，当当前子图中顶点的数目达到预设的阈值时，则创建一个新子图去存放后续新增的顶点。这种均匀顶点划分带来的好处是，在 Flush 操作的分类边日志阶段，通过对某个边日志的源顶点 ID 进行

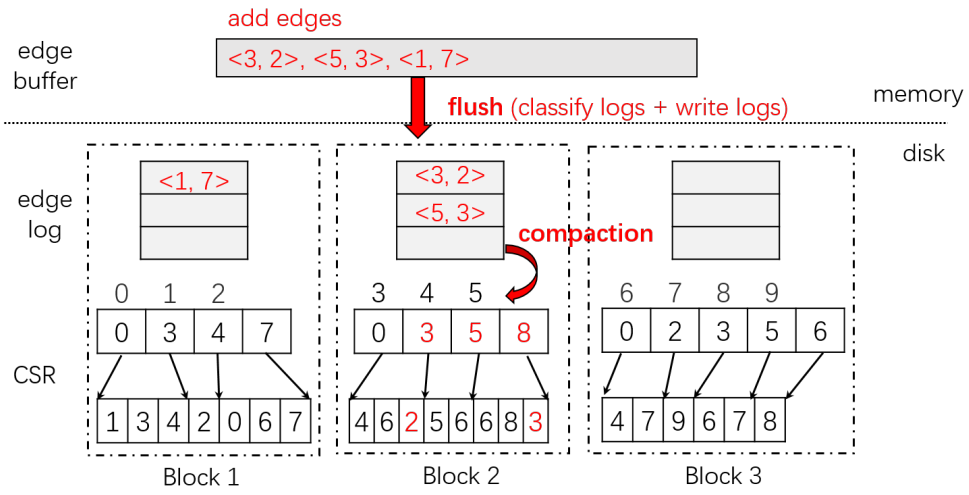


图 3.11　基于分块边日志的 CSR 存储管理

简单运算就可以得知该边日志属于哪个子图，分类边日志非常高效。但这种均匀顶点的划分方式划分的子图大小会非常不均衡，因为真实世界的图数据中各个顶点连接边的数目往往是非常不均衡的，且顶点的度分布往往呈现幂律分布，即 $p(d) \propto d^{-\gamma}$，其中 $p(d)$ 表示度数为 d 的顶点的个数或概率密度，γ 是某个大于 1 的常数。这种非常不均匀的子图划分会产生一些非常大的子图，这些大子图上又会产生频繁的数据更新，于是造成该子图频繁地执行 Compaction 操作来合并这些更新的边日志数据，造成更严重的读写放大问题。

下面通过一个简单的例子来对比分析均匀的子图划分和不均匀的子图划分的写放大问题。假设一个图数据总共包含 *8M* 条边，每条边日志大小是 *8 Byte*，则该图数据存储成边日志文件总共需要 *64MB*。假设要按照上面提出的基于分块边日志的存储管理方式，将该子图划分成两个子图，并设置每个子图都有一个边日志文件来缓存更新的边日志信息，其阈值大小为 *2MB*。也就是说，每个子图的边日志文件满 *2MB* 后就需要与底层 CSR 通过 Compaction 操作进行合并，合并的过程需要先从磁盘读取该子图的 CSR 数据，然后在内存中通过计算将边日志信息插入到 CSR 数据中，最终再将新生成的 CSR 数据写回到磁盘。通过计算得到在不同划分方式下总体的读写数据的大小和读写放大倍数，如图 3.12所示。图中第一列表示划分的两个子图各自的大小，第二列和第三列分别表示这两个子图通过若干次 Compaction 操作总共需要从磁盘读取的图数据的大小、第四列和第五列则分别表示这两个子图总共需要写入磁盘的图数据的大小、第六列和第七列则分别表示读写放大的倍数。可以看到，随着图数据划分的越来越不均衡，造成的读写放大的倍数也在不断增大，最差情况下的读写放大倍数约为均匀图划分时的 *2* 倍。严重的读写放大问题限制了基于分块边日志的 CSR 存储管理策略能够带来的图数据更新的性能提升幅度。

Graph partition	Total read (MB)		Total written (MB)		RA	WA
32MB+32MB	240	240	272	272	7.5	8.5
30MB+34MB	210	272	240	306	7.53125	8.53125
28MB+36MB	182	306	210	342	7.625	8.625
26MB+38MB	156	342	182	380	7.78125	8.78125
24MB+40MB	132	380	156	420	8	9
22MB+42MB	110	420	132	462	8.28125	9.28125
20MB+44MB	90	462	110	506	8.625	9.625
18MB+46MB	72	506	90	552	9.03125	10.03125
16MB+48MB	56	552	72	600	9.5	10.5
14MB+50MB	42	600	56	650	10.03125	11.03125
12MB+52MB	30	650	42	702	10.625	11.625
10MB+54MB	20	702	30	756	11.28125	12.28125
8MB+56MB	12	756	20	812	12	13
6MB+58MB	6	812	12	870	12.78125	13.78125
4MB+60MB	2	870	6	930	13.625	14.625
2MB+62MB	0	930	2	993	14.53125	15.546875

图 3.12　Compaction 操作中的读写放大问题

注：不均匀的子图划分加剧了读写放大问题。

为了缓解读写放大问题和保证图数据更新的性能，需要设计一个子图大小均衡的图数据划分策略。一种直观的做法是采用均匀边的图划分方式，即限制每个划分子图中包含的边的个数，也即给每个子图大小设置一个阈值。当一个子图中边的数目达到预设的阈值时，则将该子图分裂成两个更小的子图，之后的 Compaction 操作就以更小的子图为单位进行。通过这种方式，可以通过设置子图大小阈值来限制每次 Compaction 的读写放大倍数，从而减小整体图数据更新的开销。实验表明，均匀边的图划分方式确实能显著缓解读写放大问题，降低图数据更新的 Compaction 开销。但是却增加了 Flush 操作的时间开销。因为在这种划分方式下各个子图中包含的顶点的个数就不均衡了，在 FLush 操作的分类边日志阶段就不能通过简单的计算直接得到每个边日志所属的子图 ID 了，在该情况下最优的方式是使用二分查找来检索各条边属于的子图，因此大大增大了分类边日志开销，图数据更新的总体时间开销反而增大了。

因此，均匀顶点的图划分方式和均匀边的图划分方式之间存在的性能权衡，为了实现性能最优，本章提出一种性能均衡的二级图划分方式，如图 3.13所示。首先按照均匀顶点的方式将图数据划分成多个子图块（*block*），然后再按照均匀边的方式将各个子图块再划分成内部的多个子图段（*segment*）。具体的实现方式是使每个子图块包含相同数目的顶点，每个子图块默认包含一个子图段，在 Compaction 的过程中当某个子图段的大小超过一定阈值时，将该子图段分裂成两个更小的子图段，为每个子图段分配一个边日志文件，之后的 Compaction 操作就以更小的子图段为单位进行。这样在进行 FLush 操作的分类边日志时，通过计算得知各个边日志属于的子图块的 ID，若该子图块包含多个子图段时，通过子图块内部的二分查找也很快就能找到该边日志属于的子图段 ID，因此也避免

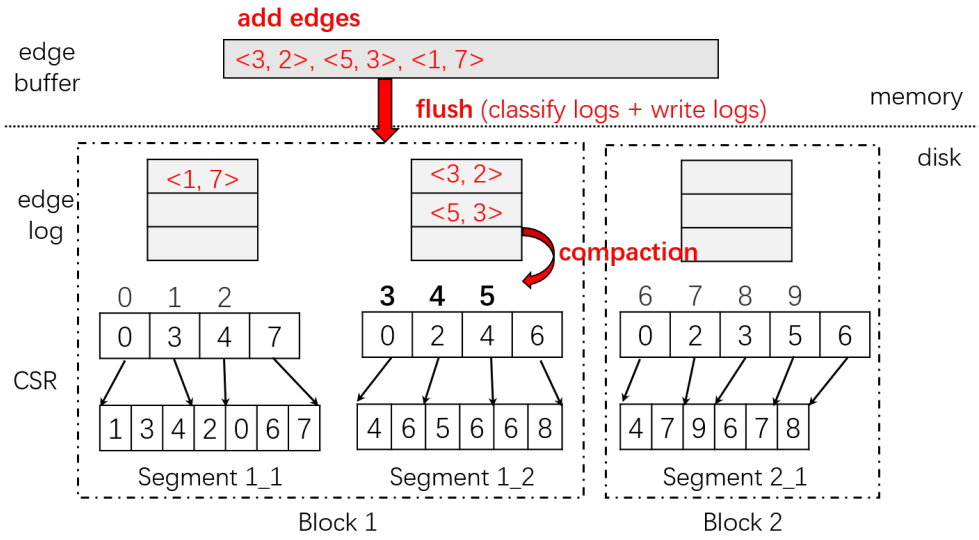


图 3.13　性能均衡的二级图划分方式

了均匀边的图划分方式带来的额外的分类边日志开销。因此，性能均衡的二级图划分方式能很好地实现动态图的更新与查询间的性能均衡，

3.4.5　参数配置和用户接口

本小节介绍 SAStore 原型系统的用户接口，包括主要的一些系统参数配置和系统函数接口，以增强对原型系统的表述。

（1）系统参数配置

- **子图划分大小配置：** SAStore 在章节 §3.4.2中提出一种启发式的子图大小配置方案来自动配置划分子图的大小为：$2^{(\log_{10} R+2)}$ MB，其中 R 为随机游走的总数目。用户也可以根据具体的应用场景的需求通过参数 *blocksize*，以 KB 为单位来自定义的设置划分子图的大小。
- **子图缓存容量配置：** SAStore 在章节 §3.4.3中提出一种随机游走状态感知的缓存策略来实现以子图为单位的高效图缓存，实验 §3.5.3中也展示了该缓存策略对比于系统页面缓存的性能提升。因此系统默认会充分利用内存空间来缓存尽可能多的子图，即根据系统内存容量和子图大小来计算最多能缓存的子图个数。用户也可以根据具体的应用场景的需求通过参数 *nmblocks* 来自定义的设置最多能缓存的子图的个数。
- **子图选择概率配置：** SAStore 在章节 §3.4.3中提出一种概率方法来缓解全局游荡者问题，每次需要选择一个子图加载时，以概率 p 来选择包含进程最慢的那条随机游走所在的子图，以概率 $1-p$ 选择包含随机游走数量最多的子图。根据实验结果，系统默认设置 $p=0.2$，用户也可以根据具体的应用场景的需求通过参数 *prob* 来自定义的设置 p 的值。

（2）用户函数接口

接下来介绍 SAStore 系统中主要的几个用户函数接口，通过这些接口可以向上支持随机游走类应用的分析计算以及动态图场景下的数据更新和查询。

- **格式转换和子图划分：*convert_to_csr(filename, blocksize)*，** SAStore 在底层采用 CSR 格式来存储图数据，而一般获取到的图数据集是使用边列表形式存储的，因此我们首先将其转换成 CSR 格式，其中 *filename* 是包含完整路径的文件名。转换完成之后，根据系统设定的 *blocksize*，进行逻辑子图划分，产生各个子图的划分区间。
- **子图选择：*chooseBlock(prob)*，**SAStore 在处理随机游走类应用时，每次选择一个子图加载进入内存进行计算，子图的选择根据系统预设的概率 *prob*，根据产生的随机数和概率的关系，选择包含最慢的随机游走所在的子图，或者选择包含随机游走数目最多的子图。
- **子图缓存查找：*findSubGraph(p, beg_pos, csr, nverts, nedges)*** SAStore 在选择好一个子图之后，首先去子图缓存队列中查找是否已经缓存该子图，如果该子图已经存在在缓存队列中，则直接将指针指向对应的缓存区域。其中 *p* 为子图 ID，*beg_pos* 和 *csr* 是存储 CSR 格式的两个数组，*nverts* 和 *nedges* 分别是该子图的顶点数目和边数目。
- **子图加载：*loadSubGraph(p, beg_pos, csr, nverts, nedges)*，**若 SAStore 选择的子图不在子图缓存队列中，则根据子图 ID 将该子图的边数据从磁盘加载到内存中，
- **更新状态信息：*updateWalkNum(exec_block)*，**在完成一个子图中随机游走的更新计算之后，SAStore 首先清空掉当前子图 *exec_block* 的随机游走数据信息，并根据随机游走在各个子图之间的转发情况，更新随机游走在各个子图之间的统计信息，包括数量分布和各个子图的最短随机游走的步长。
- **访问单个顶点：*getNeighbors(v)*，**在程序执行的后期，当随机游走的数目在图中非常稀疏时，以顶点为粒度去访问图数据能显著提升 I/O 效率。即以顶点为单位去访问其的所有邻居信息，其中 *v* 是节点 ID。
- **增加顶点：*addVertex()*，**动态图场景下新增一个顶点，令该顶点 ID 为当前图的顶点个数，然后将顶点个数加一。
- **增加边：*addEdge(s, t)*，**动态图场景下新增一条边，其中 *s* 和 *t* 分别为该边的源顶点和目的顶点。

3.5　实验评估

上一小节详细介绍了 SAStore 的设计和实现细节，本节通过在真实世界的图数据集上运行基于随机游走的图算法来实验评估 SAStore 的 I/O 效率，并对 SAStore 的三个功能模块分别进行了详细的评估。本章的实验评估主要回答以下几个问题：

- 不同的子图划分方案对图数据的划分结果有什么区别？以及子图大小的配置对整体的性能有什么影响？
- SAStore 中状态感知的 I/O 模型对比于基于迭代的 I/O 模型，是否能提高随机游走访问图数据的 I/O 效率？以及状态感知的缓存策略是否能进一步提高系统的缓存效率？
- 动态图处理的场景下，基于分块日志的 CSR 存储管理方式是否能达到图更新和图查询之间的性能均衡？对比于基础的 CSR 存储是否能提升图数据的更新效率，是否能保证图数据的查询效率？

3.5.1　实验设置

（一）系统环境

本章的所有实验都是在一台戴尔服务器上运行的，服务器的型号为 Dell Power Edge R730，内存容量为 64 GB，共配备了 24 个 Intel(R) Xeon(R) CPU E52650 v4@ 2.20GHz 处理器。该服务器的其他硬件配置和软件环境如表 3.1所示。

表 3.1　服务器的硬件配置和软件环境

配置环境	详细信息
服务器型号	Dell Power Edge R730
CPU	Intel(R) Xeon(R) CPU E5-2650 v4 @2.20GHz processor ×24
内存	64GB 2133MHz DDR4
外存	500GB SamSung 860 SSD ×7 （组成 RAID-0 阵列）
操作系统	Ubuntu 18.04.5
内核版本	Linux version 5.0.0-37-generic

（二）实验数据集

本章的实验所使用的六个图数据集信息如表 3.2所示。其中，Twitter (TT)[80]、Friendster (FS)[62]、YahooWeb (YW)[66] 和 CrawlWeb (CW)[14] 是真实世界产生的社交网络或网页链接的图数据集，Kron30 (K30) 和 Kron31 (K31) 是使用 Graph500 kronecker 图生成器[81] 生成的两个合成图，这些图数据集都被广泛的用于各类单机或分布式的图处理系统的实验评估中。表 3.2中的 CSR Size 表示以 CSR 格式

存储图数据的最小存储开销，Text Size 是以边列表格式存储图数据的文本大小。需要指出的是，Kron30、Kron31 和 CrawlWeb 这三个数据集都是超大规模的图数据集，即使采用存储开销最小的 CSR 格式存储，也都不能完全放入实验测试服务器的内存中，其中 CrawlWeb 也是目前公共可用的真实世界最大的图数据集。

表 3.2　图数据集信息

Dataset	$\|V\|$	$\|E\|$	CSR Size	Text Size
Twitter (TT)	61.6M	1.5B	6.2GB	26.2GB
Friendster (FS)	68.3M	2.6B	10.7GB	47.3GB
YahooWeb (YW)	1.4B	6.6B	37.6GB	108.5GB
Kron30 (K30)	1B	32B	136GB	638GB
Kron31 (K31)	2B	64B	272GB	1.4TB
CrawlWeb (CW)	3.5B	128B	540GB	2.6TB

（三）图算法

本章的实验中，考虑了下面四个典型的基于随机游走的图计算算法来进行性能评估。其中，RWD 和 Graphlet 是利用整个图数据进行计算的图分析算法，PPR 和 SR 是只需要访问部分图数据的的图查询算法。

- **随机游走影响域（Random Walk Domination，RWD）**[25]，RWD 的目的是找到图上的一个顶点集合，使得从该顶点集合出发的长度为 L 的随机游走可以达到图中的其他顶点的数目最大化。该算法的计算过程就是从图中每个顶点出发长度为 L 的随机游走，然后取这些随机游走访问最多的顶点的集合作为目标顶点集合。本章的实验中，设置 $L = 6$，并从图中每个顶点出发一条随机游走然后统计结果。
- **图元统计（Graphlet Concentration，Graphlet）**[82-83]，Graphlet 的目标是统计图数据中各种图元结构所占的比例，本章使用一种特殊的图元——三角形作为研究案例。该算法的计算过程就是从图中随机出发 10 万条长度为 4 的随机游走，然后统计这些随机游走中恰好能构成一个三角形的比例。
- **个性化的 PageRank（Personalized PageRank，PPR）**[24]，我们采用前面提到的基于随机游走的方法来计算 PPR（具体的计算原理和算法流程参见章节 §1.2.2），我们为每个要求解 PPR 的源顶点出发 2000 条长度为 10 的随机游走，文献[24] 中指出这个规模的随机游走足以保证估算 PPR 的准确度。
- **顶点对相似性（SimRank，SR）**[20]，SR 的目的是估计图中一对顶点之间的相似度，该算法从查询的顶点对分别出发若干随机游走，然后计算从顶点对出发的随机游走的期望相遇时间。参考文献[20] 中的设置，我们从查询的顶点对的两个顶点分别出发 2000 条长度为 11 的随机游走。

（四）对比系统

本章的实验对比的系统是最新的单机随机游走图系统 DrunkardMob[33]。DrunkardMob 发表于 2013 年的 RecSys 会议，其底层采用 GraphChi[44] 存储图数据，并在上层设计基于随机游走的编码和索引优化，具体设计优化参见章节 §2.3.1。为了保证实验的准确性，我们将每个实验运行十次，计算十次运行结果的平均完成时间。并且在每次执行之前，清除系统的页面缓存，以避免系统缓存对性能的影响。

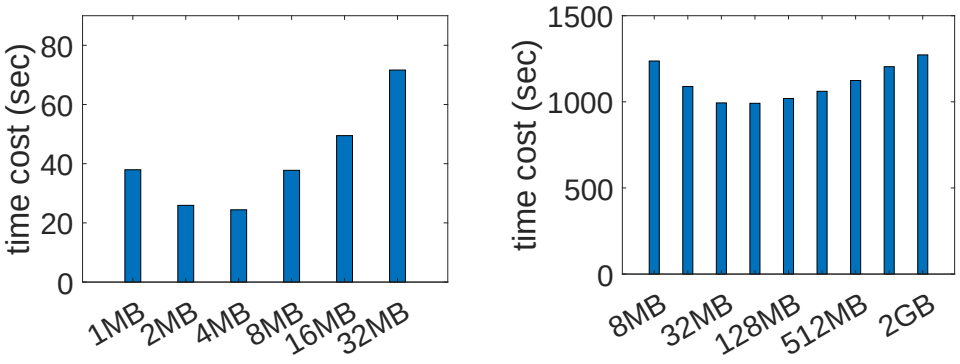
3.5.2　图组织和划分模块性能评估

（一）不同子图划分方案下的划分结果

根据章节 §3.4.2提出的启发式子图大小配置策略，对应不同随机游走规模的应用场景，SAStore 会默认设置不同大小的子图配置。本小节首先对比了 SAStore 和 DrunkardMob 在上述的图分析算法（RWD 和 Graphlet）和图查询算法（PPR 和 SR）的场景下对六个图数据集的不同划分结果。表 3.3中展示了 DrunkardMob 和 SAStore 设置的划分子图的大小以及对六个图数据集进行划分后子图的个数。DrunkardMob 的划分策略是根据内存容量，尽量设置最大的子图大小，考虑到图数据的存储以及随机游走数据的存储（DrunkardMob 将所有的随机游走数据全部存放于内存），因此 DrunkardMob 对所有应用场景设置子图大小为 1GB。SAStore 根据章节 §3.4.2提出启发式的计算公式，对 RWD、Graphlet 和 PPR/SR 这三种规模的随机游走计算场景，分别设置子图大小为 512MB、128MB 和 32MB。所以对比于 DrunkardMob，SAStore 的划分获得的更多细粒度的子图。基于这种划分策略，并配合状态感知的图加载策略，下一小节将进一步展示 SAStore 的 I/O 数量和 I/O 效率。

表 3.3　设置的子图大小和划分产生的子图的个数

#Blocks	DrunkardMob	SAStore		
		RWD	Graphlet	PPR/SR
Block size	1GB	512MB	128MB	32MB
TT	5	11	44	177
FS	10	20	78	309
YW	25	13	198	792
K30	125	65	1027	4120
K31	-	129	2054	8251
CW	-	238	3797	15186



(a) 为 10^3 个源顶点计算 PPR　　(b) 为 10^6 个源顶点计算 PPR

图 3.14　划分子图的大小对性能的影响

（二）划分子图大小对性能的影响

在 SAStore 中，划分子图的大小对性能有很大的影响。具体来说，较小的子图大小的配置可以避免更多无效图数据的加载，从而提高每次子图加载的 I/O 效率，而较大的子图大小的配置能够使得加载子图中的随机游走有更大的概率在当前子图中更新多步。为了研究子图大小的配置对整体性能的影响程度，我们每次在内存中只保存一个子图，然后通过实验来观察不同子图大小的配置下总体的运行时间的对比。这里，我们在 CrawlWeb 数据集上运行 PPR 算法作为研究案例。具体考虑了两种 PPR 算法的配置，即分别从 1000 和 1000000 个源顶点出发随机游走，并计算关于这些源顶点的 PPR，因为这两种情况配置可以代表两种典型的图计算访问场景，即只访问图中部分数据的图计算场景和需要访问全局图数据的图计算场景。

图 3.14展示了实验结果。实验表明由于上述的性能的权衡，过大或过小的划分子图大小的配置都会带来比较长的运行时间，所以往往需要一个适中的子图大小的配置才能实现最佳性能。另外不同的应用场景下，最适中的子图大小的配置也有所不同。轻量级的随机游走图计算任务，即随机游走的数量较少时，较小的子图大小的配置更适合，因为这种配置下可以有效地提高每次子图加载的 I/O 效率。相反，重量级的随机游走图计算任务，即随机游走的数量较大时，则较大的子图大小的配置更适合，因为一个较大的子图有更大的概率能够使得当前加载子图中的随机游走在当前子图中更新转发多步。基于这组实验的观察结果，本章提出了一种启发式的子图大小的配置策略，即根据图计算任务的随机游走的规模来启发式地设置一个默认的子图大小，具体的配置策略参见章节 §3.4.3。

3.5.3　图加载和缓存模块性能评估

本小节首先展示 SAStore 支持随机游类应用时在 I/O 总量和 I/O 效率方面的提升，然后评估子图缓存策略对性能的影响。

表 3.4　平均需要加载子图的次数

		RWD	Graphlet	PPR	SR
TT	DrunkardMob	30	20	50	55
	SAStore	11	44	177	177
FS	DrunkardMob	60	40	100	110
	SAStore	20	78	115	130
YW	DrunkardMob	150	100	250	275
	SAStore	46	198	23	34
K30	DrunkardMob	750	500	1250	1375
	SAStore	405	2254	2402	5825
K31	DrunkardMob	-	-	-	-
	SAStore	1466	6810	243	287
CW	DrunkardMob	-	-	-	-
	SAStore	783	5329	3325	8425

（一）I/O 效率的对比

（1）减少 I/O 总量

基于上一小节和表 3.3中展示的 DrunkardMob 和 SAStore 对各个数据集的划分结果，本小节进一步展示 DrunkardMob 和 SAStore 完成四个基于随机游走的图计算算法的计算平均所需要的加载子图的次数，如表 3.4所示。值得注意的是，DrunkardMob 采用 GraphChi 的平行滑动窗口的加载策略，需要多次 I/O 从不同的图分片（shards）中读取一个子图所包含的所有出边，而 SAStore 总是只需要一次 I/O 就可以从磁盘读取当前子图所的所有出边。根据实验结果发现，对于需要访问全图信息的图分析算法 RWD 和 Graphlet，SAStore 对比于 DrunkardMob 总是需要加载更少的子图来完成所有计算。而对于只需要访问局部图信息的图查询算法 PPR 和 SR，SAStore 对比于 DrunkardMob 需要加载数目相对多一点的子图来完成计算，但是需要注意的是，SAStore 中每个子图的大小要比 DrunkardMob 中的子图小很多（1/32），因此 SAStore 加载的整体的数据量还是比 DrunkardMob 要小很多。因此得出结论，SAStore 对比于 DrunkardMob 可以避免很多无效 I/O 的加载。

（2）提升 I/O 效率

I/O 效率定义为通过一次 I/O 加载的一个子图中用于计算随机游走的边的使用次数，除以加载子图中的总边数。注意，因为同一条边可能会被多个随机游走重复使用，所以我们统计所有边的使用次数的总和。因此，当一条边被多条随机游走重复使用时，这里定义的 I/O 效率可能超过 100%。我们分别统计使用

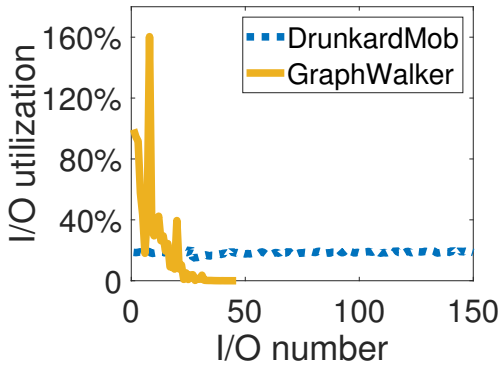


图 3.15　I/O 效率的提升

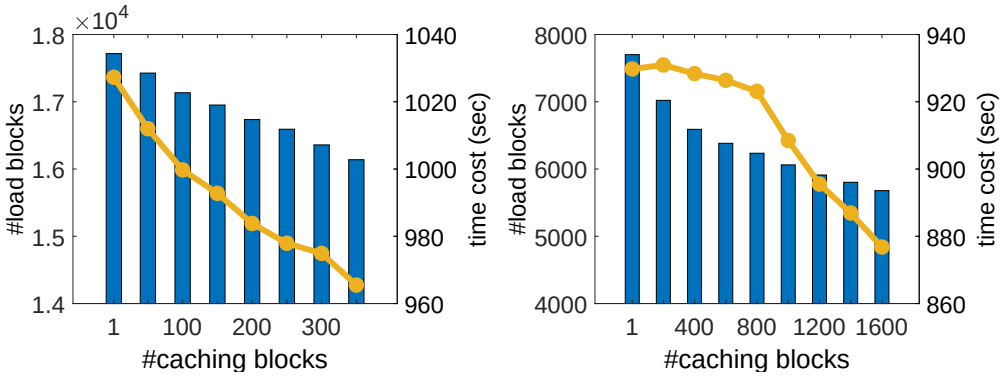
DrunkardMob 总共需要 150 次 I/Os，而 SAStore 只需要 46 次 I/Os。

DrunkardMob 和使用 SAStore 在 YahooWeb 数据集上运行 RWD 算法时每次 I/O 的利用率，实验结果如图 Figure 3.15所示。

DrunkardMob 采用 GraphChi 的图数据组织和划分策略，将 YahooWeb 图数据划分成 25 个图分片（shards），RWD 算法中要求每条随机游走移动 6 步，即 DrunkardMob 总共需要 6 轮迭代计算，因此 DrunkardMob 总共需要的 I/O 次数为 $25 \times 6 = 150$。从图中可以看到，DrunkardMob 每次 I/O 的 I/O 效率都比较均衡，大约都在 20% 左右。相较之下，SAStore 采用状态感知的图加载策略，总共只需要 46 次 I/Os 就可以完成所有随机游走的转发计算，即 SAStore 显著减少了所需要的 I/O 次数，因此，SAStore 的平均 I/O 效率也比 DrunkardMob 的平均 I/O 效率高很多。具体来说，SAStore 中前几次 I/O 的利用率甚至高达 80%-160%，这是因为状态感知的图加载策略总是加载包含随机游走数目最多的子图进入内存进行计算，并将每条随机游走都尽可能向前转发多步。刚开始随机游走总量多，因此加载的子图包含的随机游走数量也很大，I/O 效率就会非常高。随着程序的执行，大多数随机游走都完成了计算任务，只剩下少量的随机游走还有几步没有转发完成，因此加载子图包含的随机游走的数量也在逐渐减少，导致 I/O 效率逐渐降低直至趋近于 0。但是即使对于大多数 I/O，SAStore 的 I/O 效率也在 40% 到 80% 之间，是 DrunkardMob 的 2× 到 4× 倍。

（二）子图缓存策略对性能的影响

SAStore 提出一个*随机游走状态感知的子图缓存方案*，用来替代 Linux 系统的*页面缓存*（*Page Cache*）策略。这一小节通过实验研究 SAStore 的状态感知的子图缓存方案对比页面缓存策略的缓存效率，从而验证本章设计方案的有效性。具体通过改变由 *nmblocks* 表示的缓存子图的数量来改变*状态感知的子图缓存方案* 和 Linux 系统的*页面缓存策略*的缓存比例，因为当 *nmblocks* 更大时就意味着用于页面缓存的内存空间就更少，进而比较这两种缓存方案在不同缓存比重下



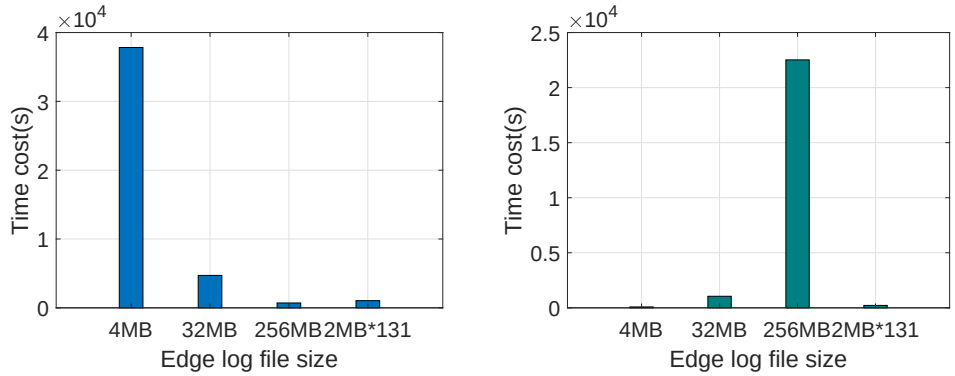
(a) 10^5 randomly started walks　　(b) PPR with 1000 sources

图 3.16　子图缓存策略对性能的影响

的缓存效率。具体考虑两种应用场景：1）从图中任意顶点随机出发的 10^5 条长度为 10 的随机游走；2）为连续的 1000 个源顶点计算 PPR。本组实验使用最大的数据集 CrawlWeb (CW)。图 3.16展示了实验结果，其中横坐标表示通过状态感知的子图缓存方案缓存的子图的数目，即 *nmblocks*；蓝色柱子记录了从磁盘加载子图的数量，即没有命中缓存的实际 I/O 的数目，刻度由左边的纵坐标所示；黄色的折线表示程序运行的时间开销，刻度由右边的纵坐标所示。实验表明随着 *nmblocks* 的逐渐增大，即通过状态感知的子图缓存方案缓存的子图的数目增加，通过 Linux 系统的页面缓存策略缓存的数据量减少时，实际 I/O 的数量和程序总体的时间开销总是在减少。这个结果表示，状态感知的子图缓存方案总是比 Linux 系统的页面缓存策略更加高效。

3.5.4　动态图更新和查询模块性能评估

我们通过一组实验观察在不同边日志文件大小配置下基于边日志的 CSR 存储管理的性能，首先按照上述图更新的过程导入 Friendster 数据集中的所有边，统计图更新的性能，然后再在导入完成的图上随机查询 500 万个顶点的邻居顶



(a) 图更新性能　　(b) 图查询性能

图 3.17　不同边日志文件大小配置的图更新与图查询性能

点，统计图查询性能，实验结果如图 3.17所示。可以看到，随着边日志文件的阈值大小逐渐增大，图数据的更新性能显著提升，当将边日志文件的阈值设置为 *256MB* 时，导入 Friendster 数据集中的所有边数据只需要 *707* 秒，约为设置为 *4MB* 时的 *1/55*。但与此同时，随着边日志文件的阈值大小的设置逐渐增大，图数据的查询性能也显著下降，这是因为在这种基于日志的 CSR 存储管理方案中查询某个顶点的边数据时，需要同时到内存的边缓冲区、磁盘的边日志文件和 CSR 存储中去查询，而边数据在边缓冲区和边日志文件中都是无序存放的，因此需要遍历扫描整个边日志文件去查找其中和查询顶点关联的边数据。实现中使用了位图（*bitmap*）去指示每个顶点在边日志文件中是否有关联的边，以过滤掉一些不必要的遍历，但是如果在边日志文件中有该查询顶点关联的边，就需要遍历查找整个边日志文件去找出它的边。当文件大小比较大时就会带来比较大的查询开销，面对大量查询的场景时就会导致非常差的性能。例如边日志文件大小为 *256MB* 时，查询 500 万个顶点的邻居就需要约 *6* 小时。

相比之下，采用基于分块日志的 CSR 存储管理策略将 Friendster 数据集划分成多个子图，设置每个子图最多包含 512K 个顶点，则总共得到 131 个子图。我们将每个子图单独存储 CSR 文件，并为每个子图分配一个 *2MB* 的边日志文件缓存新增的边数据。注意，这里边日志文件的总大小为 $2 \times 131 = 262MB$，和不分块时将边日志文件 的阈值设置为 *256MB* 的总量是差不多的。从图中结果可以看到，基于分块日志的 CSR 存储管理策略能够实现和不分块的 *256MB* 的边日志文件 实现相当的图更新性能，同时又能实现和不分块的 *4MB* 的边日志文件相当的图查询性能，显著地实现了图更新性能和图查询性能之间的性能均衡。

3.6　本章小结

本章基于随机游走类应用对图数据的访问特征，提出一套基于随机游走状态感知的图加载方案并实现了原型系统 SAStore 。SAStore 首先采用存储开销最低的图存储格式 CSR 将图数据存放于磁盘，然后设计了一种逻辑划分方式，采用非常低的时间开销将图数据在逻辑上划分成多个子图，并提出一种随机游走状态感知的子图大小配置适应不同随机游走规模的应用场景和随机游走计算的不同阶段；同时，SAStore 提出一种随机游走状态感知的 I/O 策略，选择包含随机游走数目最多的子图加载到内存当中以最大化每次子图加载能够更新的随机游走的步数，并设计相应的子图缓存策略来进一步提高缓存效率；最后，SAStore 针对动态图场景下图更新性能和图访问性能难以均衡的问题，在 CSR 格式的基础上设计基于分块日志的图数据管理方式。实验结果表明 SAStore 可以有效地提高随机游走访问图数据的 I/O 效率和增量图数据的更新效率。

第 4 章　支持快速随机游走的图计算框架

本章摘要: 现有的支持随机游走的图处理系统中往往采用以边或顶点为中心的随机游走状态数据的索引机制，并使用大量的动态数组存储这些随机游走的状态信息。这样的存储索引机制会带来比较大的索引数据的开销和动态数组频繁重新分配内存的开销，从而限制了系统可以处理的图数据集的规模和能并发运行的随机游走的规模。另一个方面，现有工作中采用的基于迭代的同步计算模型，为了保证图上所有随机游走之间的同步更新而牺牲了随机游走的计算效率。本章首先基于随机游走数据的特征，提出以子图为中心的随机游走数据索引，并采用定长缓冲区来存储每个子图的所有随机游走数据，从而避免大量动态数组带来频繁内存重分配。然后，本章继续针对现有随机游走图处理系统随机游走更新速率慢的问题，提出异步的随机游走更新策略。最后，在上一章提出的状态感知的图存储管理机制 SAStore 之上，我们结合我们的随机游走的存储管理和更新计算策略，提出一个支持快速随机游走的图分析框架 GraphWalker，该系统可以高效地处理由数十亿个顶点和数千亿条边组成的非常庞大的磁盘驻留图，而且还可以并发地运行数百亿条、数千步长的随机游走。实验结果表明，在运行大量随机行走时，与原有最好的单机随机游走图处理系统 DrunkardMob 相比，GraphWalker 的处理速度快一个数量级；与最新的分布式随机游走图处理系统 KnightKing 相比，GraphWalker 在一台机器上的处理速度可以达到其 8 台机器上的速度；与性能最好的通用单机图处理系统 Graphene 和 GraFSoft 相比，GraphWalker 在支持随机游走任务时，速度提升也非常明显，在最好的情况下，速度也快一个数量级。

4.1　本章介绍

在上一章我们提出基于随机游走状态感知的图数据的 I/O 策略，那么如何去存储管理这些随机游走的状态数据，又如何计算这些随机游走的移动转发。在章节 §2.3中我们已经详细阐述了现有随机游走数据存储与计算的相关优化工作。但当前支持随机游走的图处理系统中随机游走数据的存储管理方式和随机游走的计算更新策略都不够高效。本章我们首先分析现有工作在支持随机游走的存储和计算方面还存在的局限性；其次，我们介绍本章工作的主旨思想和主要贡献点；然后我们详细介绍我们提出的支持快速随机游走的图计算框架 GraphWalker 的设计与实现；最后对 GraphWalker 进行实验评估。

4.2　研究出发点

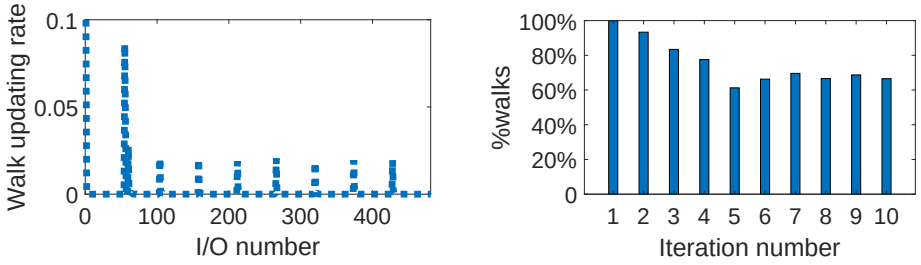
4.2.1　随机游走的存储开销大

上一小节提到现有工作中的随机游走状态数据往往会被存储在内存中的大量动态数组中，并采用以边/顶点为中心的方式来管理这些随机游走数据。这样的设计因为会在内存中创建非常多的动态数组，所以带来非常大的内存开销，比如对于一个中等大小的图数据集 YahooWeb[66] 来说，它总共包含 14 亿个顶点和 66 亿条边，那么 GraphChi[44] 中的以边为中心的索引方式仅仅是存储随机游走的这些动态数组的索引数据就至少需要 26.4 GB 的内存空间，这还不包括具体的随机游走的状态数据。由于图中顶点的数目往往要远小于边的数目，所以以顶点为中心的索引方式相对减少了存储随机游走的动态数据的数目，但是管理随机游走数据的内存索引开销依然很大，比如对于 YahooWeb 图数据集来说，也需要 5.6 GB 来仅仅存储随机游走的动态数组的索引。

DrunkardMob 通过将相邻的 128 个顶点的随机游走数据存储到同一个内存中的缓冲区中，将随机游走数据的索引数量减少到原来的 1/128，即对于 YahooWeb 图数据集来说，只需要 44.8 MB 来存储随机游走数据的数组索引了。然而，DrunkardMob 中的每个随机游走的缓冲区还是以动态数组的形式来管理，这种管理策略还是限制了系统的可扩展性：（1）首先，DrunkardMob 依然需要创建大量的动态数组，比如需要为 YahooWeb 图数据集创建 1120 万个动态数组，大量的动态数组会带来频繁的内存空间重新分配的问题，这不仅会在内存空间中带来很多内存碎片，而且也为带来大量额外的时间开销。因此，DrunkardMob 能处理的图数据集的规模受限于内存的大小，即 *DrunkardMob 难以支持超大规模图数据上的随机游走*，比如它不能在 CrawlWeb[14] 图数据集上运行随机游走类应用。（2）其次，DrunkardMob 将所有的随机游走的状态数据都存放在内存中，所以 *DrunkardMob 能处理的随机游走的规模也受限于内存容量*。比如，当我们需要并发地计算 100 亿条随机游走时，按照 *32* 位的压缩编码方式存储，每条随机游走的状态数据需要 4 B，则总共至少需要 40 GB 的内存空间来存储所有的随机游走的状态数据。由于大量的动态数组，DrunkardMob 也很难将这些随机游走数据刷盘存储到外存中，因为会造成*打开太多文件的问题*（*Too many open files error*）。（3）最后，即使是针对中等规模的图数据集上运行中等规模的随机游走的图处理场景，DrunkardMob 的执行效率也是有限的，比如我们的实验表明它需要花费约 2.3 个小时在 YahooWeb[66] 图数据集上运行十亿条长度为 10 的随机游走，具体的实验设置及实验结果参考章节 §4.5.2和图 4.7(c)。

4.2.2　随机游走的更新速率慢

我们将随机游走的更新速率定义为当前加载子图中更新的所有随机游走的步数的总和除以所有随机游走所要求更新的步数的总和。现有的基于迭代的计算模型中，随机游走的更新速率会非常慢，因为为了保证随机游走之间的同步性，基于迭代的计算模型在每一轮迭代计算中，让每条随机游走只更新一步。这严重浪费了加载到内存中的图数据，因为很多随机游走仍然可以再加载的子图上继续一段转发，甚至有些还可以移动多步。为了验证上述论证，我们使用最新的基于单机外存的随机游走图计算系统 DrunkardMob 在真实世界的社交网络图 Friendster 数据集上运行 10^6 条从单个源顶点出发的随机游走。图 4.1(a) 展示了随机游走的更新速率，我们发现所有的随机游走通过一次子图加载平均会更新一千步，所以随机游走的更新速率低到 10^{-6}。我们进一步统计了在每一轮迭代中，加载第一个子图并完成该子图上的随机游走计算之后，依然停留在该子图上的随机游走的比例，如图 4.1(b) 所示。我们发现，平均 75.3% 的随机游走在更新完一步之后依然停留在第一个子图中，这些随机游走本可以在这一轮计算中继续更新移动一步甚至多步，因此造成了随机游走更新速率慢的问题。



(a) Walk updating rate　　(b) Walks in the 1^{st} interval

图 4.1　随机游走的更新速率及分析

注：随机游走的更新速率以及在每一轮迭代计算子图 1 结束后依然停留在子图 1 的随机游走的比例。

最近，CLIP[49] 提出了一种*加载子图重新进入*的方法、Lumos[50] 提出了*交叉值传播技术* 来重复利用已经加载到内存的子图数据以提高加载子图的 I/O 利用率和随机游走的计算效率。但是当他们往往需要多次遍历访问整个加载的子图，也会带来额外的开销，比如在进行随机游走的计算时，*加载子图重新进入*的方法需要去重复遍历加载子图中的所有顶点去查看每个顶点上是否还存在随机游走数据，带来额外的访问开销。而且由于子图中的各个随机游走也会有不同的步伐，所以也很难确定*加载子图重新进入*的次数。最新的分布式随机游走系统 KnightKing 主要是针对复杂随机游走中随机邻居选择的优化，没有考虑太多系统层面的存储管理、I/O 调度以及计算框架等方面的优化。存储管理方面，

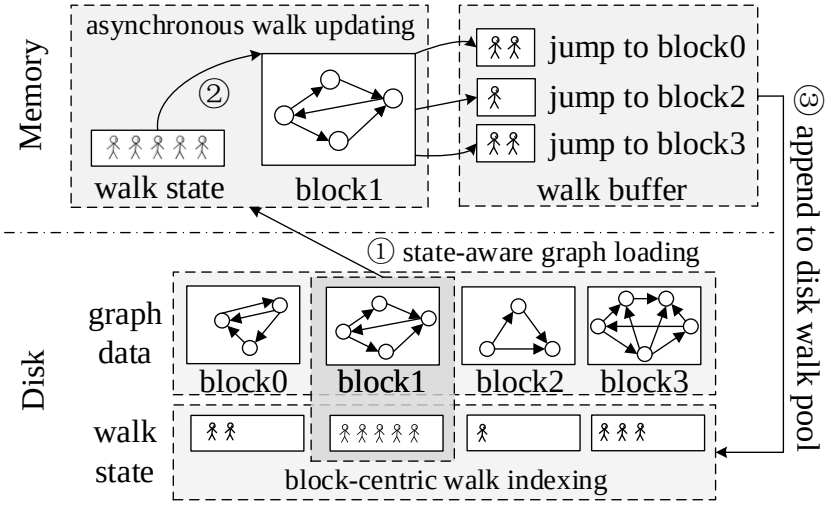


图 4.2　GraphWalker 中大规模随机游走的并发计算流程

KnightKing 也没有专门针对随机游走数据管理的优化，随机游走数据的索引开销依旧很大；I/O 调度方面，KnightKing 底层还是沿用现有系统中的基于迭代的计算模型，导致支持随机游走的应用时 I/O 的效率还是不高。

4.3　基本思想和贡献点

（一）本章工作的主旨思想

本章第 §4.1节介绍了当前的图计算系统中随机游走的状态数据存储索引开销大和随机游走的转发更新速率慢的问题。为了能实现基于磁盘的单机图处理系统上支持大规模图数据集上的大规模并发随机游走，我们一方面需要降低管理随机游走状态数据的内存开销，另一方面需要加速随机游走的更新速率。针对这两个方面的目标，本章分别提出了*以子图为中心的随机游走状态数据的存储和索引机制* 和*异步的随机游走更新策略*，其基本思想是首先通过以子图为中心的索引方式大幅度降低随机游走索引的数量，然后通过内存中定长缓存区的数组管理方式来限制每个子图的随机游走数据的内存占用，从而从整体上显著减少随机游走状态数据的存储管理开销。此外在每个子图内部采用以随机游走为中心的异步计算方式，允许每条随机游走持续更新到走出当前子图的边界为止，大幅度提高随机游走的更新速率。

基于上述设计，本章在上一章的 SAStore 的基础上进一步实现了支持快速随机游走的图计算框架 GraphWalker 。图 4.2显示了 GraphWalker 中运行大规模并发随机游走的处理流程。GraphWalker 先通过一个预处理过程将所有的随机游走数据按照子图分类，并和图数据一起存储在磁盘中。在随机游走的执行过程中，（1）首先通过状态感知的图加载策略（参考上一章第 §3.4节）选取一个子

图加载到存储，同时加载该子图对应的随机游走数据；（2）然后在内存中将该子图上对应的随机游走数据通过访问该子图数据进行异步更新并转发出去；（3）最后再将跳转到其他子图上的随机游走数据持久化到磁盘文件。循环执行上述流程直至所有的随机游走都完成计算。

（二）本章工作考虑的主要问题

基于上述随机游走的处理流程，我们归纳 GraphWalker 系统在设计过程中主要考虑的几个方面的问题如下：

- 如何存储一条随机游走所包含的状态信息？包括其出发的源顶点、当前正经过的顶点以及已经走过的步长的等信息等。
- 针对图中大规模的并发随机游走数据，如何建立索引机制来实现对随机游走数据的高效访问并同时减少这些索引数据带来的额外开销？
- 如何实现随机游走状态数据在内存和磁盘的存储方式以及将他们从磁盘加载到内存和从内存持久化到磁盘的 I/O 过程？
- 在随机游走的转发过程中，如何实现随机游走的异步更新方式？
- 如何实现对大量并发随机游走的并行计算？

（三）本章工作的主要贡献点

结合现有随机游走图处理系统中存在的局限性和对上述问题的思考，本章工作实现以下贡献点：

(1) **以子图为中心的随机游走数据的索引机制：**本章工作首先设计随机游走的编码存储将一条随机游走的多个状态数据压缩存储在一个 *64* 位的 *long* 类型数据中，减少随机游走状态数据的内存占用；并且将当前正停留在同一个子图上的所有随机游走数据存放在同一个随机游走池中，大大减少了索引数据的数量和占用的内存空间。

(2) **基于定长缓冲区的随机游走数据存储方式：**本章工作在内存中采用定长缓冲区的方式来属于一个子图的随机游走池，从而避免了大量动态数组带来的频繁的内存重新分配的开销，并将超出的随机游走信息持久化到磁盘从而支持超大规模随机游走的图计算场景。

(3) **异步随机游走更新策略：**为了加速随机游走在加载子图中的更新速率，本章工作设计了一种异步随机游走更新策略将子图中的随机游走移动尽可能多的步数，并采用多线程优化来同步更新并发的随机游走，显著加快了图计算任务的完成时间。

(4) **针对 RWR 的计算优化：**针对基于重启或跳跃的随机游走 RWR 和 RWJ，本章工作进一步设计了一种基于拼接的优化方案，使用大量并发计算的短随机游走来拼接得到一条很长的随机游走来加速计算效率。

(5) **性能提升**：本章工作基于上述设计，在 SAStore 的基础上实现了支持快速随机游走的图计算系统 GraphWalker，并在真实的图数据集上对其进行性能评估。评估的实验结果显示，GraphWalker 对比于 DrunkardMob 平均能实现超过一个数量级的性能提升；GraphWalker 对比于最新的分布式随机游走图系统 KnightKing，能实现其在 8 台相同机器上相当的性能；GraphWalker 对比于性能最好的单机图处理系统 Graphene 和 GraFSoft 也能实现高达一个数量级的速度提升。

4.4　GraphWalker 的设计与实现

4.4.1　GraphWalker 系统架构

GraphWalker 的系统架构图如图 4.3所示，GraphWalker 的设计主要包含两个主要模块：随机游走的存储索引和随机游走的更新计算。

- **随机游走的存储索引模块**首先采用以子图为中心的索引策略来实现轻量级的高效随机游走索引；然后对每个子图的随机游走数据，采用内存中定长缓冲区的存储方式，限制随机游走数据对内存的占用，使得系统能处理的随机游走的规模不再受限于内存容量，提高系统的可扩展性。
- **随机游走的更新计算模块**首先在上一章中提出的大规模图存储管理系统 SAStore 和本章的随机游走的存储索引模块的基础上，提出加载子图上随机游走的异步更新策略，进一步提升随机游走的更新速率；然后针对 RWR 和 RWJ，提出一种基于拼接的优化计算方法，进一步加速基于 RWR 和 RWJ 的计算任务的计算性能。

4.4.2　随机游走数据的存储和索引

我们采用三个变量 source, current and step，分别表示一条随机游走的三个状态数据随机游走的起始顶点、随机游走当前停留顶点在当前子图的 ID 的偏移量以及该条随机游走已经移动的步数。我们将每条随机游走的三个状态数据压

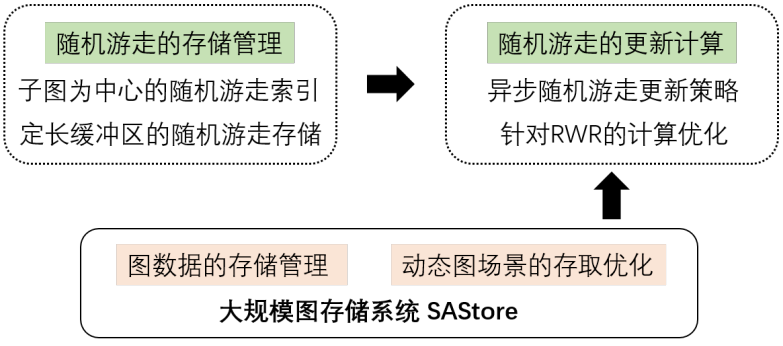


图 4.3　GraphWalker 的整体架构

缩成一个 *64* 位的压缩表示，为每个状态分配的位数如图 4.4所示。这个数据结构可以支持同时从 2^{24} 个源顶点出发的随机游走，每条随机游走最多可以移动 2^{14} 步。注意，随机游走的总数目没有限制，因为我们可以在每个顶点同时出发很多条随机游走。

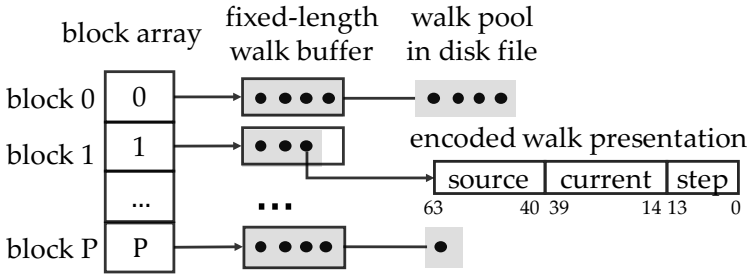


图 4.4　以子图为中心的随机游走数据管理

为了减少管理所有随机游走数据的内存开销，我们提出一种以子图为中心的索引策略。对于每个子图，我们使用一个随机游走数据池来记录当前正停留在该子图的所有随机游走数据，索引结构如图 4.4所示。我们将每个随机游走数据池在内存中表示成一个固定长度的缓冲区，默认情况下，该缓冲区最多存储 1024 条随机游走，从而避免动态内存分配的空间和时间开销。当一个子图中有超过 1024 条随机游走时，我们就将它们都刷新到磁盘，并为每个子图在磁盘上创建一个随机游走数据池文件。注意，我们用 64 位的 *long* 数据类型对没条随机游走进行编码，因此每个随机游走数据池在内存中只花费 8 KB。通过这种方式，管理随机游走状态数据的内存成本非常低，比如对于之前提到的案例，在拥有 14 亿个顶点的 YahooWeb 上运行 10 亿条随机游走，如果划分成 100 个子图，GraphWalker 只需要 800 KB。然而，DrunkardMob 记录 10 亿条随机游走的内存开销超过 4 GB，因为每次行走至少需要 4 个字节, 另外，DrunkardMob 还需要 44.8 MB 的内存空间用于存储动态数据的索引数据（参见章节 §4.2）。除此之外，这些随机游走会在 1120 万个动态数组之间跳跃，导致频繁的内存重新分配，带来额外的时间成本。

要在内存随机游走数据池和磁盘的随机游走数据池文件之间进行同步，当我们将一个子图的数据块加载到内存中时，我们也将该子图对应的随机游走数据池文件 加载到内存中，并将其中的随机游走状态数据合并到到内存中的随机游走数据池中。然后，我们为当前子图中的随机游走执行计算，将他们转发。在更新过程中，当某个子图的内存中随机游走数据池已满时，我们将该子图的随机游走数据池中的所有随机游走状态数据追加写到该子图在磁盘对应的随机游走数据池文件的尾部，并清除对应子图内存中随机游走数据池的数据。完成当前加载子图中所有随机游走的计算后，我们清除当前子图的随机游走数据池的数据，

并且对每个子图内存中随机游走数据池和磁盘中随机游走数据池文件中随机游走进行汇总，以更新随机游走在各个子图之前的分布状态。

通过这种轻量级的随机游走管理方案，我们节省了存储随机游走状态数据的大量内存开销，从而能够支持大量并发随机游走的应用场景。此外，定长随机游走缓冲区策略将计算完成后对各个子图中随机游走刷盘的小 I/O 转换成批量的大 I/O，极大地降低了将随机游走状态数据持久化存储的 I/O 成本。

4.4.3　异步的随机游走更新策略

为了提高图计算系统的 I/O 效率，最近的一些工作提出了一种加载子图重进（*loaded data re-entry*）的方式[49] 来允许随机游走重新利用当前加载的图数据进行更多步数的转发。这种方式工作的原理就是当完成当前子图的一次遍历计算之后，再次遍历该子图使得还停留在上面的随机游走再移动一步，并且，我们可以通过多次加载子图的重新遍历来将该子图上的所有随机游走都转发出去直到所有随机游走都不在当前子图了。

然而，这种在基于迭代的计算模型中采用加载子图重进的方式来计算基于随机游走的图算法会带来局部游荡者问题（*local straggler problem*），即当一个子图刚被加载到内存中的时候，该子图上的随机游走通过遍历一遍子图全部都可以移动一步，然后有一部分会跳出当前子图，剩下的一部分还在当前子图，可以通过加载子图重进方式再一次地遍历当前子图将剩下的随机游走再移动一步。随着加载子图重进的次数的增加，大部分随机游走都已经跳出当前子图的边界，只剩下很少的一部分随机游走还滞留在当前子图中，而这很小一部分的随机游走可能会需要花费很多次的加载子图重进才将他们转发完成，而每次加载子图重进 又会带来额外的子图顶点的遍历计算开销。

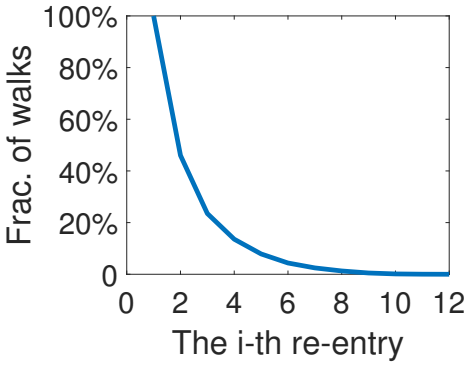


图 4.5　局部游荡者问题

注：最后 20% 的随机游走需要额外 8 次（66.7%）加载子图重进才将他们转发完成，即需要额外遍历 8 次子图完成他们的转发计算。

为了更好的理解*局部游荡者*问题的影响，我们使用 CLIP 中*加载子图重进*的方法在真实世界图数据集 Twitter 上运行 RDW 算法，具体的数据集和算法介绍参见章节 §4.5.1。我们统计了在一个加载的子图当中，每一次通过*加载子图重进*的方式遍历子图后，可以更新的随机游走的比率，实验结果如图 4.5所示。我们发现总共需要额外的 11 次的*加载子图重进*才能将当前子图中的所有随机游走都转发跳出当前子图，但实际上在 4 次*加载子图重进*之后，超过 80% 的随机游走都已经走出当前子图了，也就是说剩下的 7 次*加载子图重进*只用于更新小于 20% 的随机游走，但每次*加载子图重进*都需要去遍历整个子图去完成计算，带来较大的计算开销。在我们的实验中，几乎所有的加载子图的计算都表现出类似的趋势。所以在基于迭代的计算模型下，即使采用*加载子图重进*的策略，随机游走的计算效率仍然很低。

我们发现，简单地在完成几次*加载子图重进*的遍历计算之后停止随机游走的计算，并不能完全解决*局部游荡者*问题，也并不能减少程序执行的整体完成时间。主要原因是，即使我们在几次*加载子图重进*后停止随机游走的计算，例如上面的示例中 4 次*加载子图重进*之后，仍然还有 20% 的随机游走还停留在当前子图的一些顶点上，这就意味着当前子图在下一轮的迭代计算中仍然需要通过 I/O 加载到内存，然后通过遍历子图完成计算。

为了提高随机游走的更新速率，GraphWalker 采用了一种异步的随机游走更新策略，即以随机游走为中心进行计算，允许每条随机游走持续更新，一直到走出当前子图的边界。当完成一条随机游走的计算之后，我们选择另一条随机游走开始计算，一直到当前加载子图中的所有随机游走都被转发完成。然后，我们根据上一章提出了状态感知的图加载策略加载另一个子图数据，然后重复上面的异步的随机游走更新过程。图 4.6中展示了在同一个子图内处理两条随机游走的示例。为了加速计算过程，我们还使用多线程并行地更新这些随机游走。我们强调，通过我们的异步随机游走更新策略，我们完全避免了无用的顶点访问，并自然地消除了*局部游荡者*问题。通过对加载子图数据块的充分利用，GraphWalker 显著提高了每次子图加载的 I/O 利用率和随机游走的更新速率。

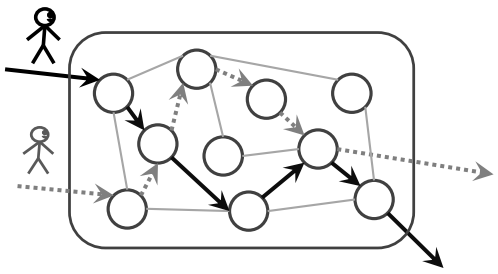


图 4.6　并行的异步随机游走更新

4.4.4　针对 RWR 的计算优化

重启随机游走（*RWR*）[84] 是简单随机游走的一种变种，其跟简单随机游走唯一的区别就是在随机游走的每一步选择的时候，RWR 可以以很小的概率 c 跳回源顶点，即以概率 c 重启该条随机游走，然后以概率 $1-c$，它就像简单的随机游走一样，随机转发到当前顶点的一个邻居顶点。RWR 在许多应用程序中被广泛使用，比如个性化的 PageRank[85] 和图像自动字幕[86] 等等。与之相似的还要*跳跃随机游走*（*RWJ*），在随机游走的每一步选择的时候，RWJ 以概率 c 随机跳跃到图上的任意顶点上。对于有向图上的随机游走，其中一些顶点可能没有出边，这类顶点一般被称为汇聚顶点。若简单随机游走移动到汇聚顶点上就无法再跳出去。因此为了避免被卡在汇聚顶点中以及保证收敛性，RWR 和 RWJ 往往更加常用。

我们发现，运行少量较长的随机游走比运行大量较短的随机游走要花费更长的时间，因为少量较长的随机游走很难并行计算。因此，为了进一步提高 RWR 的性能，我们在 GraphWalker 中采用*拼接短随机游走*的思想。拼接多个短随机游走用于计算一条长随机游走的思想首次在文献[24] 中被提出，并被证明是正确的，具体来说，我们从同一个源顶点开始很多条随机游走，在每一步，如果原始 RWR 以概率 c 重启随机游走，那么我们就以概率 c 停止随机游走，我们称之为*停止随机游走*（*RWS*）。对于概率 $1-c$，我们遵循与简单随机游走相同的过程。这样，我们可以生成很多不同长度的随机游走序列，但它们都从同一个源顶点开始，然后我们可以将这些随机游走拼接起来，形成 RWR 生成的一条长的随机游走序列。下面我们通过一个具体的例子来进一步阐述上述基于拼接的方法。

假设我们从顶点 0 出发三条 RWS，并生成以下三段较短的随机游走序列。

$$
\begin{aligned}
w_0 &= 0 \to 1 \to 2 \to 3 \to 4 \\
w_1 &= 0 \to 4 \to 7 \to 3 \\
w_2 &= 0 \to 7 \to 4 \to 8 \to 2 \to 5 \to 4 \to 7 \to 2 \to 1 \to 6
\end{aligned}
$$

然后我们可以将上面三段短的随机游走序列 w_0、w_1 和 w_2 拼接成一条长的随机游走序列 W，只要 RWS 中的停止概率与 RWR 中的重启概率相等，则 W 可以作为 RWR 生成的一个随机游走序列。

$$
\begin{aligned}
W = &\underbrace{0 \to 1 \to 2 \to 3 \to 4}_{w_0} \to \underbrace{0 \to 4 \to 7 \to 3}_{w_1} \to \\
&\underbrace{0 \to 7 \to 4 \to 8 \to 2 \to 5 \to 4 \to 7 \to 2 \to 1 \to 6}_{w_2}
\end{aligned}
$$

我们指出，在现有的基于迭代的计算模型中，很难去支持这种基于拼接的方式来计算随机游走，主要是因为通过 RWS 生成的随机游走的长度是各不相同的，很难进行高效的同步计算。GraphWalker 中提出的状态感知的图加载策略和异步随机游走更新策略则可以非常简单高效地去实现这种针对 RWR 的基于拼接的优化策略。该优化策略也同样适用于 RWJ，即从图中随机选取的一些起始顶点出发很多条随机游走，然后遵循 RWS 的过程生成多条较短的随机游走序列，最后将这些短随机游走序列拼接起来形成一条较长的随机游走来模拟 RWJ。

4.4.5　实现优化

我们用 C++ 实现了我们的原型系统 GraphWalker 。除了上面介绍的设计之外，我们还在实现过程中进行了一些优化。

避免数据竞争的随机函数：

为了快速处理一个子图中的多条随机游走，我们使用多线程并行计算，计算的过程中，我们需要生成一个随机数来进行随机邻居选择。但是如果多线程同时调用库函数 *rand()*，它们实际上使用相同的种子进行计算，这将带来数据竞争从而导致死锁问题。因此，我们使用函数 *rand_r(&seed)* 来指定种子值，避免数据竞争，并使用每个随机游走特有的状态总值作为种子值，以保证随机性。

基于多线程的随机游走数据池管理：在并行随机游走的计算过程中，多个随机游走也很有可能并行跳转到同一个子图，在随机游走状态数据的插入过程中也会带来对该子图的*随机游走数据池*的同时访问和修改。为了避免对内存中相同数组的数据访问竞争，我们为每个*随机游走数据池*的每个线程分配一个数组，用于存储通过该线程转发到该子图的随机游走数据，然后在每次计算完成之后，再将属于同一个子图的随机游走数据统一汇总到同一个*随机游走数据池*。

4.5　实验评估

GraphWalker 的目标是提供快速和可扩展的随机游走，所以我们使用最新的单机随机游走图计算系统 DrunkardMob[33] 作为性能比较的基准对比系统。此外，还有一些通用的单机图处理系统，这些系统从不同的方面进一步优化了系统的性能，比如细粒度子图分区策略，支持 I/O 与计算之间管道传输的异步 I/O 策略，减少 TLB 失误率的大页支持等。但这些优化并不是特地针对随机游走的优化策略，它们中的许多也正交于 GraphWalker 中的优化策略。为了完整性，我们还将 GraphWalker 与两个最新的开源单机图处理系统 Graphene[48] 和 GraFSoft[53] 进行比较。为了进一步验证 GraphWalker 的可扩展性，我们还将 GraphWalker 与最新的分布式随机游走图计算系统 KnightKing[30] 进行了比较。

4.5.1　实验设置

（一）系统环境

本章所有实验都是在一台戴尔服务器上进行的，服务器的型号为 *Dell Power Edge R730*，内存容量为 *64 GB*，共配备了 24 个 *Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz* 处理器。如果没有特别说明，实验中处理的图数据集都存储在一个 *32 TB* 的 RAID-0 阵列中，该 RAID-0 阵列由 7 个 *500GB SamSung 860 SSDs* 组成，另外我们也研究了 HDD 上的性能评估。分布式随机游走图系统 KnightKing[30] 运行在一个由 8 个机器节点组成的分布式集群上，由 *10Gbps* 的以太网互连，其中每个设备节点配备两个 *8-core Intel Xeon E5-2620 v4* 处理器、*20 MB* 的 L3 缓存和 *64 GB* 的内存。

（二）实验数据集

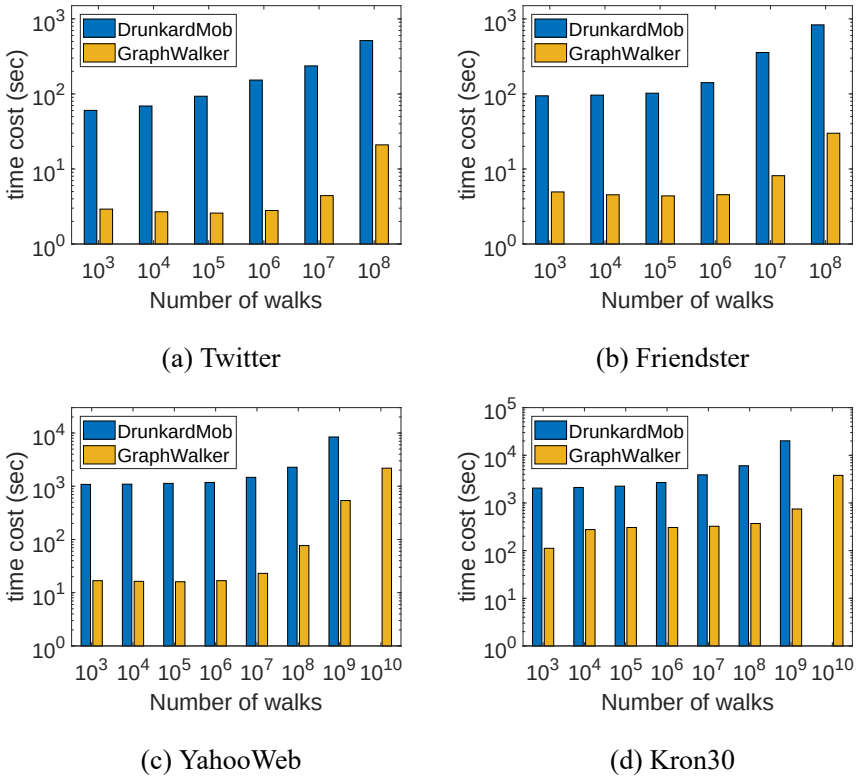
本章的实验所使用的数据集和上一章相同，总共包含 4 个真实世界的数据集 Twitter (TT)[80]、Friendster (FS)[62]、YahooWeb (YW)[66] 和 CrawlWeb (CW)[14] 以及 2 个使用 Graph500 kronecker 图生成器[81] 生成的合成图 Kron30 (K30) 和 Kron31 (K31)。具体的数据集介绍参考上一章的第 §3.5.1 小节和表 3.2。我们再次强调，其中 CrawlWeb 是目前公共可获得的最大的图数据集，它和 Kron30、Kron31 都是不能完全放入我们实验中测试服务器的内存的超大规模的图数据集。

（三）图算法

本章实验首先使用各种不同规模（不同随机游走的数量和长度）配置下的随机游走算法来进行 GraphWalker 在所有设计空间下的性能评估。此外，本章的实验也使用了上一章节介绍的四个经典的基于随机游走的图算法，即 RWD、Graphlet、PPR 和 SR，来对 GraphWalker 进行进一步的性能评估。具体的算法介绍和他们中随机游走的规模配置参考上一章的第 §3.5.1 小节。

4.5.2　GraphWalker 与随机游走系统的对比

我们通过比较 GraphWalker 与 DrunkardMob（最新的专为优化随机游走的单机图处理系统）来验证 GraphWalker 的效率。注意，基于随机游走的图计算算法通常需要同时出发特定数量及特定长度的随机游走，所以我们考虑不同的随机游走配置下的性能评估，从而分析 GraphWalker 与 DrunkardMob 在支持随机游走的整个设计空间下的性能表现，并展示了 GraphWalker 在支持大规模随机游走以及超长随机游走时的可扩展性。此外，我们还展示了运行四种基于随机游走的图计算算法时的性能对比。最后，我们通过一些微基准测试验证分析了 GraphWalker 带来性能提升的原因。



(a) Twitter　(b) Friendster

(c) YahooWeb　(d) Kron30

图 4.7　不同随机游走数量下的性能对比

注：固定每条随机游走的长度为 10

（一）整个设计空间下的性能研究

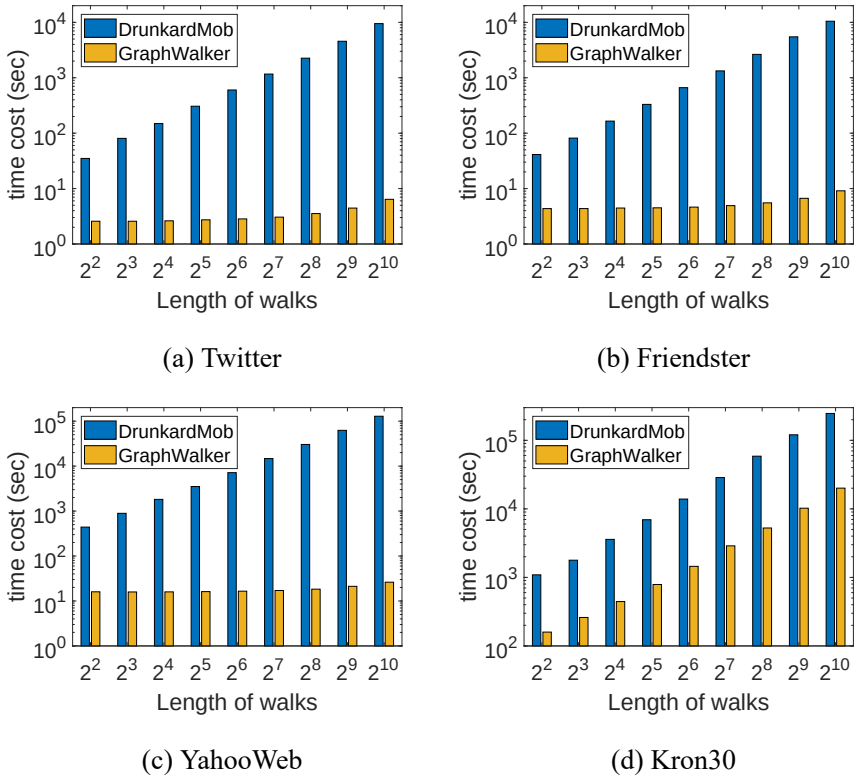
（1）不同随机游走数量下的性能对比

首先，我们固定随机游走的长度为 10，在图 4.7中展示不同随机游走规模下，GraphWalker 与 DrunkardMob 的总时间开销对比，这里我们取随机游走的数量为 10^3 到 10^{10}。在每个子图中，*x* 轴表示实验配置的随机游走的数量，*y* 轴表示完成所有这些随机游走的转发所需的时间。首先我们可以看到，在不同的随机游走规模和不同的图数据集的所有设置下，GraphWalker 的性能始终优于 DrunkardMob，即时间开销更低。具体举例来说，在 YahooWeb 数据集上运行 10^6 条长度为 10 的随机游走，DrunkardMob 花费近 *20* 分钟，而 GraphWalker 只需要 *17.8* 秒，也就是说，GraphWalker 实现了 *70* 倍的性能提升。一般来说，在我们实验所有配置的情况中，GraphWalker 可以实现 *16* 倍到 *70* 倍的性能提升。此外我们从图中观察到在随机游走的数量不多的情况下，随着随机游走数目的增加，GraphWalker 的运行时间几乎是个常量，这是因为随机游走的数量不是很大时，I/O 时间开销是主导因素，相对而言随机游走的计算开销的增加对整体时间开销的影响不大。然而，随着随机游走数量的不断增加，随机游走的计算成本的占比也越来越大，所以当我们运行更多的随机游走时，总时间成本也呈线性增加。

这里，我们希望特别强调的一点是 GraphWalker 的可扩展性。即使在超大规模的图数据集上运行数百亿次的随机游走，GraphWalker 仍然可以在一合理的时间开销内完成，即取决于不同的图规模，时间开销大约在几十秒到一小时之间。相比之下，DrunkardMob 甚至不能在较大规模图数据集上（比如 YahooWeb 和 Kron30）同时运行 10^{10} 条随机游走，因为 DrunkardMob 将所有随机游走存放在内存中会导致内存不足的问题，所以很难支持大规模的随机游走（详情参见章节 §4.2）。所以我们没有展示 DrunkardMob 在随机游走数量设置超过 10^{10} 是的运行结果。更重要的是，当图规模变得非常大时，DrunkardMob 也无法正常运行。例如，对于 Kron31 和 CrawlWeb，DrunkardMob 也会遭遇内存不足的错误。因为 DrunkardMob 对每 128 个顶点使用一个动态数组来索引随机游走的状态数据，所以当图变得非常大的时候这些索引数据就会产生很大的内存开销，而且 DrunkardMob 也很难将这些随机游走信息写到磁盘，因为关联的索引数量太多需要打开太多的文件（详情参见章节 §4.2）。因此，我们也没有展示这两个大规模数据集下的性能对比。相比之下，GraphWalker 由于采用了以子图为中心的随机游走索引机制，大大减少了索引数量和索引数据的内存开销，并且 GraphWalker 可以很轻易地将随机游走的状态数据存储到磁盘中，因此，GraphWalker 能够支持像 Kron31 和 CrawlWeb 这样超大规模的图数据，甚至可以在他们之上同时运行数百亿条随机游走，例如我们的实验结果显示，GraphWalker 在最大的图数据集 CrawlWeb 上同时运行 10^{10} 条长度为 10 的随机游走只需要大约一个小时的时间开销。

（2）不同随机游走长度下的性能对比

接下来，我们固定随机游走的数量为 10^5，在图 4.8中对比运行不同长度的随机游走时，GraphWalker 与 DrunkardMob 的时间开销，这里我们取随机游走的长度为 2^2 到 2^{10}。首先我们可以看到，GraphWalker 的时间开销总是比 DrunkardMob 要小得多，在最好的情况下它甚至可以达到超过三个数量级的性能提升。特别指出，当图数据集的规模不是特别大的时候，例如对于 Twitter、Friendster 和 YahooWeb 来说，随着随机游走长度的增加，DrunkardMob 的时间开销呈现线性增长，而 GraphWalker 的时间开销却几乎是恒定的。这是因为得益于 GraphWalker 中轻量级的子图存储和优化的缓存策略，对于中等大小的图，GraphWalker 几乎可以将整个图数据都缓存到内存中，因此只产生非常低的 I/O 开销。对于不能完全存储在内存中的非常大的图数据集（例如 Kron30），DrunkardMob 和 GraphWalker 的时间开销都随着随机游走步数的增加而增加，因为在这种情况下，GraphWalker 和 DrunkardMob 一样都需要在内存和磁盘之间对各个子图数据进行频繁换入换成。然而我们指出，GraphWalker 的总体运行时间对比于 DrunkardMob 要快得多，即使对于 Kron30 数据集，GraphWalker 也能实



(a) Twitter　(b) Friendster

(c) YahooWeb　(d) Kron30

图 4.8　不同随机游走长度下的性能对比

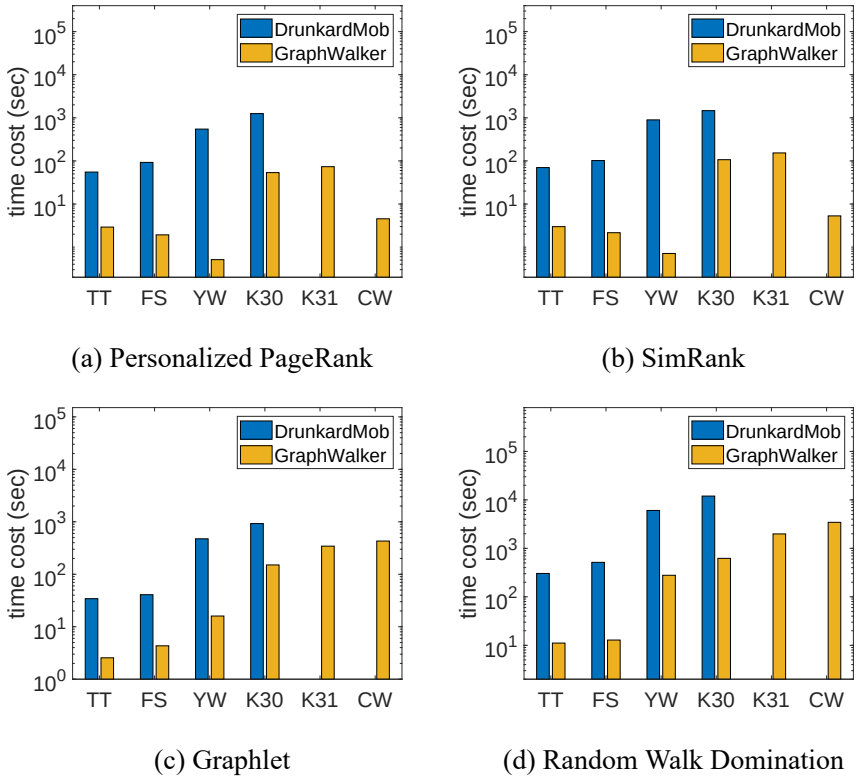
注：固定随机游走的的数量为 10^5

现 *7* 倍到 *10* 倍的性能提升。另外，这个实验还展示了 GraphWalker 在支持数千步的长随机游走时表现出的可扩展性。

（二）基于随机游走的图算法性能研究

本小节中，我们评估四种常见的基于随机游走的图计算算法（包括图分析算法和图查询算法，具体的算法描述见章节 §3.5.1）的性能。从图 4.9中我们可以看到，一般而言 GraphWalker 对比于 DrunkardMob 能实现 *9* 倍到 *48* 倍的性能提升。在一些特殊情况下，例如在 YahooWeb 上运行 PPR 和 SR 算法时，GraphWalker 甚至可以在 DrunkardMob 的基础上实现超过三个数量级的性能提升。这是因为 YahooWeb 在 PPR 和 SR 算法的查询顶点邻域的子图具有非常好的局域性，所以 GraphWalker 只需要加载较少几个对应的子图就完成所有随机游走的计算转发。然而 DrunkardMob 却需要迭代地扫描整个图数据，并且以同步的方式更新随机游走，所以它的 I/O 利用率很低，耗时很长。

我们想要指出的是，由于在本章节 §4.5.2第一组实验中已经解释过的相同原因，DrunkardMob 无法处理两个最大的图数据集，所以我们跳过这些情况下的运行结果。注意，这个实验同时也证明了 GraphWalker 在支持大量随机游走和大规模图数据集这两个方面的可扩展性。



(a) Personalized PageRank　(b) SimRank

(c) Graphlet　(d) Random Walk Domination

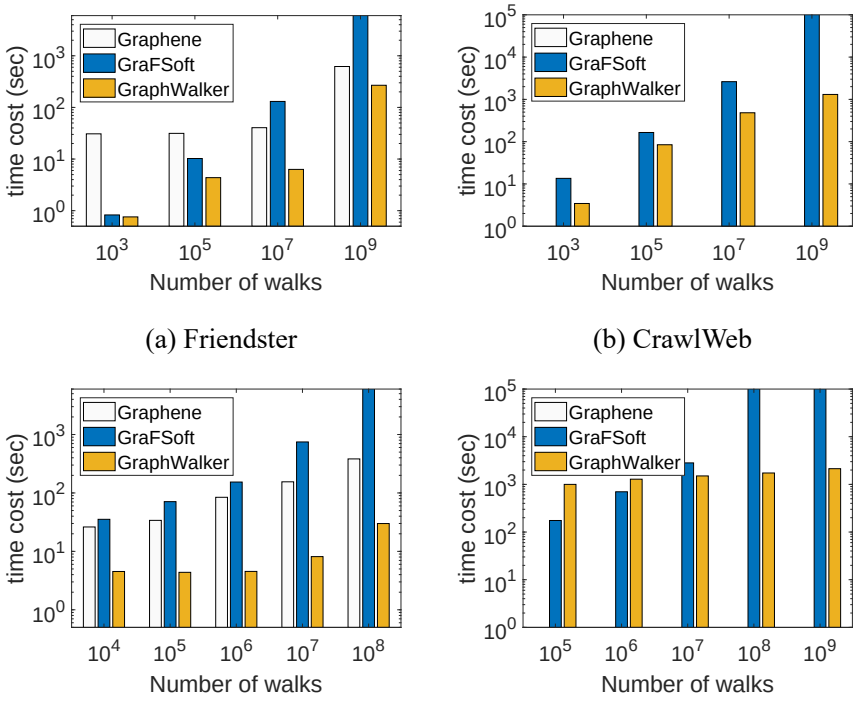
图 4.9　基于随机游走的图算法下的性能对比

4.5.3　GraphWalker 与最新图分析系统的对比

（一）对比最新的单机图处理系统

近年来，最新发表的一些单机图处理系统中提出了很多针对图算法的优化设计，比如细粒度的子图划分策略、支持 I/O 和计算之间通过管道运输的异步 I/O 策略以及可以减少 TLB 失误率的大页支持等。然而这些优化策略并不是专门针对随机游走这一类算法提出的，所以很多这些优化策略也是正交于 GraphWalker 的优化设计的。因此，为了进一步展示 GraphWalker 的效率，我们还将其与两个最新开源的单机图处理系统 Graphene[48] 和 GraFBoost[53] 进行了比较请注意，GraFBoost 中除了软件设计以外，同时也使用了硬件来加速计算，为了公平比较，我们只关注 GraFBoost 的纯软件实现的版本，称为 GraFSoft。注意，GraphWalker 中并不包括上面提到的设计优化。

这个实验中我们主要关注两种随机游走场景，即从单个源顶点出发所有的随机游走（Single-Source Random Walk，SSRW）和从多个源顶点分别出发随机游走（Multi-Source Random Walk，MSRW）。我们将随机游走的长度固定为 10，改变随机游走的数量。请注意，Graphene 是一个半外核的图处理系统，即它将图数据存储在磁盘上，但将所有元数据信息，即随机游走的状态数据保存在内存中。因此由于高昂的内存成本，Graphene 无法处理大规模的随机游走场景，例如



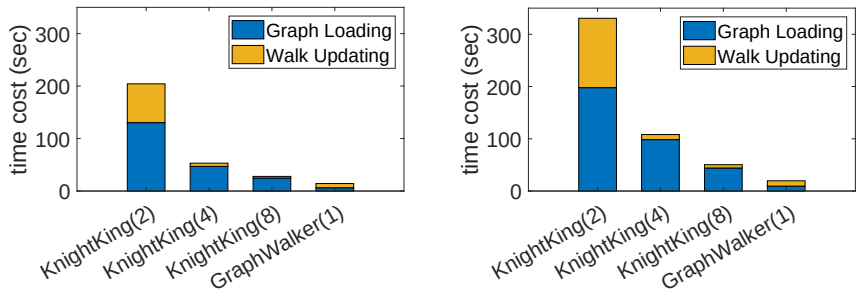
(a) Friendster　(b) CrawlWeb

(c) MSRW on Friendster　(d) MSRW on CrawlWeb

图 4.10　对比 Graphene 和 GraFSoft

大于 10^9 条随机游走的情况，Graphene 也无法处理大规模图数据集的场景，例如比 Friendster 大的图数据集。因此，我们只展示了 Friendster（Graphene 可以处理的最大的图数据集）和 CrawlWeb（公众可获得的最大规模的图数据集）下的对比结果，我们在其他图数据集下也观察到了类似的结果。

实验结果如图 4.10所示，我们可以从中得出多个结论：（1）首先，我们观察 GraphWalker 对比于 Graphene 的性能差异，尽管 Graphene 是一个半外核的图处理系统，即不需要通过 I/O 从磁盘读写随机游走的状态数据，但 GraphWalker 的运行时间仍然始终快于 Graphene，而且最高可以带来 *19* 倍的加速。更重要的是，GraphWalker 具有非常好的可扩展性，可以支持数百亿条的大规模随机游走的并发执行，GraphWalker 也可以处理包含数千亿条边的大规模图数据集，比如 CrawlWeb。然而 Graphene 由于其半外核的设计造成的高内存开销而无法在这些情况下运行。（2）其次，我们观察 GraphWalker 对比于 GraFSoft 的性能差异，当随机游走的数目较少时，GraphWalker 能带来的性能提升幅度是有限的，因为在总随机游走数目较少的情况下，每个子图都只有很少量的随机游走，此时状态感知的 I/O 模型并不能带来太多的收益。然而，随着随机游走数量的增加，GraphWalker 的性能改进幅度也会增加。例如，在 CrawlWeb 上并发运行 10 亿条随机游走时，GraphWalker 处理 SSRW 和 MSRW 任务分别只花费了 *21.8* 分钟和 *35.6* 分钟，而 GraFSoft 甚至无法在 *24* 小时内完成任务。也就是说，GraphWalker 在 SSRW 和 MSRW 两种计算任务的情况下对比于 GraFSoft



(a) Twitter　　(b) Friendster

图 4.11　对比分布式随机游走系统 KnightKing

都达到了至少 *40* 倍的性能提升。更重要的是，当我们增加随机游走的数量时，GraphWalker 的时间开销呈次线性增长，比 GraFSoft 的增长幅度要小很多，这进一步证明了 GraphWalker 在支持大量随机游走方面的可扩展性。

（二）对比最新的分布式随机游走图系统

为了进一步验证 GraphWalker 的可扩展性及其资源友好的特性，我们还将其与最新的分布式随机游走图系统 KnightKing[30] 进行了比较。值得注意的是，KnightKing 主要关注随机游走的更新效率，而 GraphWalker 主要考虑的是单机场景下图数据从磁盘传输到内存的 I/O 效率。在本小节实验中，我们着重研究了 KnightKing 中使用的基于终止随机游走的 PPR 算法，即从图中每个顶点开始一条随机游走，每条随机游走在每一步以概率 t 终止。这里我们设置 $t = 0.15$，这是许多场景下的常见设置[19,24]。需要注意的是，t 设置的越小，那么随机游走的平均步数就越大，需要的计算量也就越多，所以 KnightKing 就使用一个较小的 t 设置来证明它的计算效率。由于 KnightKing 在其论文中使用了 8 台机器的集群进行实验评估，为了实现交叉验证，我们也最多使用 8 台机器，并主要关注 KnightKing 使用 8 台机器可以处理的最大的两个图数据集（即 Twitter 和 Friendster）上的性能对比，并按照 KnightKing 论文中的描述，将这两个图数据集转换成无向图之后进行实验，转换成无向图之后，他们分别包含 24 亿条边和 36 亿条边。

Figure 4.11显示了实验结果，每个系统名后面的数字表示使用的机器数量。我们可以看到，对于 KnightKing 来说，随着集群大小的增加，计算时间（也就是更新随机游走的时间）大大减少，但是 I/O 时间（也就是加载图数据块）仍然需要花费大量的时间。这是因为 KnightKing 主要专注于优化随机游走的计算效率，而不是磁盘到内存的 I/O 效率。注意，本实验中计算时间的结果与 KnightKing 论文的结果是一致的，而 KnightKing 论文中展示的结果并没有包含加载图数据块这部分的时间开销。相反，GraphWalker 主要针对的就是 I/O 效率问题，并根据其 I/O 模型相应地调整了随机游走的更新过程，从而可以在基于单机的磁盘驻留

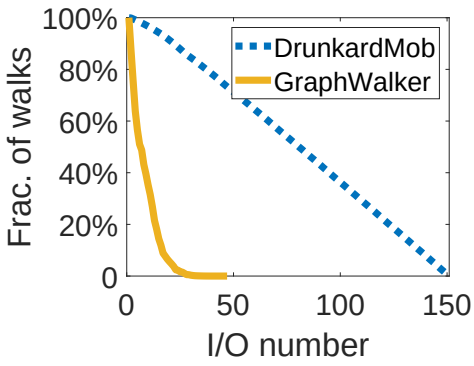


图 4.12　随机游走更新速率的对比

图上实现非常快速的随机游走更新。这里，随机游走的更新时间还包括了随机游走状态数据持久化到磁盘的时间。我们也看到 GraphWalker 的性能甚至可以与在 8 台机器上运行的 KnightKing 的性能相当。更重要的是，对于最大的图数据集 CrawlWeb，KnightKing 在 8 台机器上运行也会遇到内存不足的问题，根据其在处理其他较小图数据集时所使用的资源评估，KnightKing 可能需要一个更大的集群来运行。因此，我们可以得出结论，GraphWalker 对比于 KnightKing 是一种更有利于资源利用的选择，即 GraphWalker 是一个资源友好的图处理系统。

4.5.4　GraphWalker 性能分析

（一）随机游走的更新速率

本小节我们实验评估 GraphWalker 对比于 DrunkardMob 能带来的随机游走的更新速率的提升，我们分别统计使用 DrunkardMob 和使用 GraphWalker 在 YahooWeb 数据集上运行 RWD 算法时每次 I/O 的计算之后剩余待更新的随机游走的步数，实验结果如图 4.12所示。我们在章节 §3.5.3中已经分析过，DrunkardMob 总共需要 150 次 I/O 来完成所有计算任务，每次 I/O 可以更新的随机游走的步数也比较均匀，平均每次 I/O 对图中所有的随机游走总共更新 5000 步。相比之下，GraphWalker 显著提高了图中随机游走的更新速率，平均每次 I/O 可以更新的随机游走的步数高达 1.85 亿步，相比于 DrunkardMob 高出 3.7 倍，最终 GraphWalker 总共只需要 46 次 I/O 就可以完成所有随机游走的更新计算。主要原因是 DrunkardMob 采用了基于迭代的模型，在每一轮迭代计算中，加载一个子图进入内存后，只为子图上的随机游走更新一步，所以随机游走的更新速率非常慢。相反，GraphWalker 设计了一种异步的随机游走更新方法来充分利用内存中加载的图数据（参见章节 §4.4.3），因此每次加载一个子图进入内存后，子图上的随机游走可以在子图上移动多步。因此，GraphWalker 对比于 DrunkardMob 显著提升了随机游走的更新速率，并节省了大量的重复 I/O。

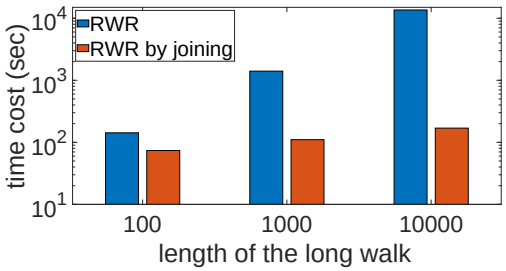


图 4.13　拼接短随机游走实现的性能提升

（二）针对 RWR 的优化提升

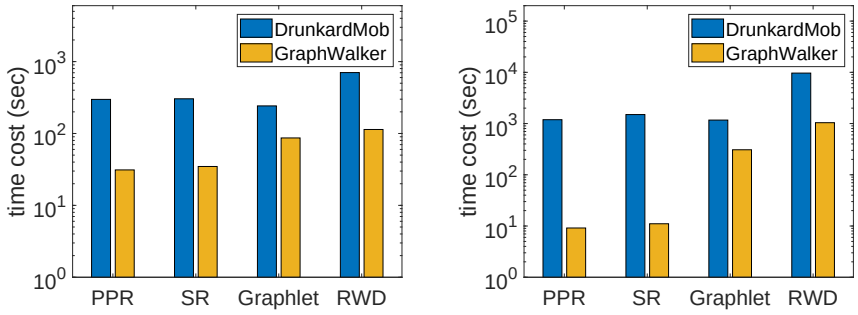
在本小节中，我们评估采用拼接的方式来优化重启随机游走（RWR）的策略带来的性能提升。我们首先通过 RWR 算法（设置随机游走在每一步的重启概率为 $c = 0.15$）运行长度为 L 的一条较长的随机游走序列。作为对比，我们再通过 RWS 算法（设置随机游走在每一步的停止概率为 $c = 0.15$）运行多条较短的随机游走序列，这些较短的随机游走长度不一，平均长度被证明为 $1/c$[24]，也就是说我们总共可以运行 $L \times c$ 条短随机游走即可通过拼接得到一条长度约为 L 的长随机游走。具体的执行和拼接流程参加章节 §4.4.4。在我们的评估实验中，我们将 L 分别设置为 100、1000、10000，即通过 RWS 运行的短随机游走的数目分别为 15、150、1500，实验结果如图 Figure 4.13所示。我们可以看到，基于拼接的方法可以显著减少运行随机游走的计算时间。更重要的是，随着要运行的随机游走的长度的增加，通过拼接可以获得的收益也越来越大。这是因为 RWR 的计算量随步数的增加呈线性增加，而使用拼接方法，我们可以将随机游走的长步数转换成随机游走的大数量，而这些大量的随机游走的步数都可以保持很短。大规模数量的随机游走可以通过并行计算的方式快速完成计算。

（三）时间开销分解

为了更好地理解 GraphWalker 中的设计优化效果，我们还在表 4.1 中展示了 DrunkardMob 和 GraphWalker 的时间成本的分解细分。注意，在随机游走计算的整个执行过程中，有三个关键操作:（1）图数据加载（Graph Loading），通过磁盘 I/O 将子图数据块加载到内存中；（2）随机游走更新（Walk Updating），更新转发当前加载子图上随机游走；（3）随机游走持久化（Walk Persisting），包括

表 4.1　时间开销分解

Time cost (s)	DrunkardMob	GraphWalker	Speedup
Graph Loading	1005	47	21×
Walk Updating	3029	214	14×
Walk Persisting	1056	16	66×
Total Runtime	5110	278	18×



(a) Friendster　　(b) YahooWeb

图 4.14　HDD 上的性能评估

从磁盘读取随机游走状态数据并在更新完成后将更新完成的随机游走状态数据写回磁盘用于持久化。这三个关键操作是交错轮询进行的，所以我们将每一个操作执行的总时间相加。GraphWalker 的时间开销主要就是由这三个关键操作组成，但 DrunkardMob 还需要花费大量的时间来为图中的每条边创建随机游走数据的索引并为这些索引数据分配内存。为了进行一对一的对比，我们在表 4.1中只展示了三个关键操作的时间开销。从结果中，我们可以看到 GraphWalker 的表现在各个步骤上都优于 DrunkardMob。具体来说，（1）图数据加载的性能提升主要来源于轻量级数据组织方式和状态感知的子图加载、缓存策略，因为通过这些策略可以大大减少 I/O 的数目；（2）而随机游走更新效率的提升主要来自状态感知的图加载和异步随机游走更新策略，增加了随机游走可以在加载子图中更新的步数。（3）最后，以子图为中心的随机游走索引机制和定长的随机游走缓冲策略将每个子图处理后的随机游走数据持久化的很多小 I/O 转换成仅在相应的缓冲区已满时才执行的几个大 I/O，从而降低了随即游走状态数据的 I/O 时间开销。值得注意的是，我们很难确定 GraphWalker 中的哪一种设计贡献最大，因为这些优化设计策略都是关联的，他们都为了一个共同的目标：就是尽快完成计算任务所要求的随机游走的运行，所以所有这些优化设计都有助于提高 GraphWalker 的运行效率。

（四）不同存储设备下的性能评估

我们还通过在机械硬盘（HDD）上运行实验来研究存储设备对 GraphWalker 和 DrunkardMob 的性能的影响，图 4.14展示了执行上述介绍的四个基于随机游走的图算法的时间开销，这里我们只展示了数据集 Friendster 和 YahooWeb 上的实验结果。由于 HDD 的随机 I/O 性能比 SSD 低很多，因此 DrunkardMob 和 GraphWalker 的时间开销都增加了。当比较 GraphWalker 和 DrunkardMob 时，我们观察到类似于之前研究的 SSD 下的结果。具体来说，在不同的设置下 GraphWalker 可以实现 *3* 倍到 *135* 倍的性能提升。

4.6　本章小结

本章在上一章节实现的基于随机游走状态感知的图存储管理系统 SAStore 的基础上进一步设计实现了支持快速随机游走的图计算系统 GraphWalker 。GraphWalker 首先采用压缩编码的存储格式来存储每条随机游走的所有状态信息，然后提出一种以子图为中心的索引方式来分类索引这些随机游走数据，大幅度减少了索引数据额外带来的存储开销和计算开销；然后 GraphWalker 采用一种定长缓冲区的方式来存储管理每个子图的随机游走池的数据，通过定长缓冲区大小的设置限制了其对内存的占用开销；同时，GraphWalker 提出一种异步随机游走更新策略，采用以随机游走为中心的计算方式更新每条随机游走直至他们走出当前子图，显著提升了随机游走整体的更新速率；此外，GraphWalker 还针对 RWR 和 RWJ 这两类特殊的随机游走，提出一种基于拼接的优化计算方式来进一步加速基于 RWR/RWJ 的图计算任务的计算效率。我们真实世界数据集上的实验结果表明 GraphWalker 可以有效地加速各类随机游走场景的更新速率，对比于最新的单机随机游走图系统、最新分布式随机游走图系统和最新单机通用图计算系统都有明显的性能优势。

第 5 章　快速收敛的随机游走算法优化

本章摘要: 随机游走因其高效的计算被广泛采用到很多应用场景中。本章从一个典型的基于随机游走的问题场景——图采样问题出发，探索随机游走如何解决实际的图计算问题。基于随机游走的图采样因为计算简单、只需访问局部数据以及具有理论基础，成为当前主流的图采样策略。基于随机游走的采样需要在随机游走达到收敛状态之后才能进行采样从而根据其稳态分布进行无偏估计。但现有的随机游走算法的收敛速度都很慢，往往需要成千上万步随机游走才能收敛，严重影响了采样效率。我们分析随机游走收敛慢的一个很重要的原因是他可能经常被困在某个局部的子图，需要很多步才能走出去，从而阻碍了对全局图数据的尽快探索，拖慢了收敛进程。针对上述分析，我们设计了相应的优化算法：非回溯的公共邻居感知的随机游走算法 NB-CNARW，来加速随机游走的收敛从而提高随机游走采样的效率；实验结果表明，我们提出的 NB-CNARW 算法相较于当前最新的随机游走算法，可以显著提升随机游走的收敛速度，收敛所需要的步数最多可以减少 29.2%; 在使用 NB-CNARW 算法进行基于图采样的无偏估计时，可以在保证相同估算精度的情况下，可以大幅度减少采样所需要的查询开销，最好的情况下可以减少 25.4% 的查询成本。最后，基于对随机游走应用场景的探索，我们总结了基于随机游走的算法对图数据的访问特征以及随机游走的计算方式，为后面章节设计实现支持随机游走的图分析系统提供了理论指导。

5.1　本章介绍

图采样技术是图上一类典型的图计算技术，他可以将计算复杂度非常高的图计算问题通过采样得以实现。在众多采样算法中，基于随机游走的图采样算法由于其高效的计算效率和可靠的理论保障而收到广泛的关注和应用。在章节 §2.4中我们已经详细阐述了基于随机游走的图采样技术如何实现对图数据的采样和对图上某测量指标的无偏估计，也讨论分析了基于随机游走的图采样算法优化设计的国内外研究现状。

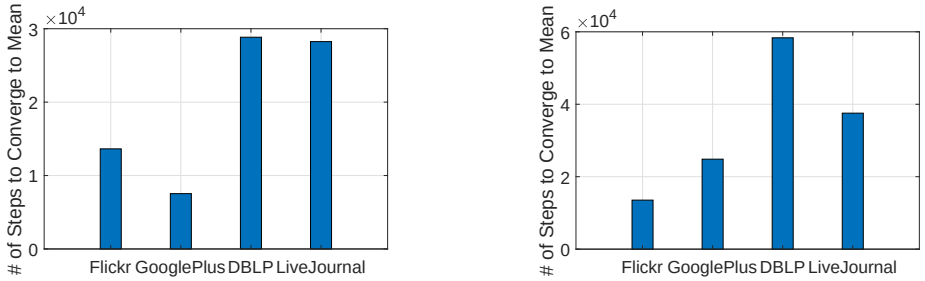
本章工作在已有工作的基础上，进一步关注随机游走的收敛速度和基于随机游走的图采样效率。本章工作首先阐述现有的基于随机游走的图采样算法还存在的局限性，从而引出本章研究工作的出发点——加速随机游走的收敛，从而减少基于随机游走图采样的开销；其次，我们介绍本章工作的主旨思想和主要贡献点；然后我们详细介绍我们提出的 NB-CNARW 算法的设计与理论分析；最后对 NB-CNARW 算法和基于 NB-CNARW 的图采样效率进行实验评估。

5.2　研究出发点

这一小节我们首先介绍随机游走收敛速度慢的严重程度以及收敛速度对基于随机游走的图采样效率的影响。然后介绍最新的加速随机游走收敛速度的优化算法 CNARW 的设计和实现。最后阐述虽然 CNARW 已经大幅度提升了随机游走的收敛速度，但如何充分利用已有的顶点信息来进一步提升基于随机游走的采样效率仍然是一个重要的问题。

5.2.1　随机游走收敛速度慢

从随机游走开始运行到随机游走在图中各个顶点的分布概率到达稳态分布的这段持续时间称为“收敛时间”，在真实的大规模网络图中，这个阶段通常需要一个很大的计算开销，因为通常需要很多步以后才能达到收敛。我们通过实验来观察使用简单随机游走来进行图采样并对一些测量指标进行无偏估计时，随机游走达到收敛状态所需要的步数。我们在四个真实世界的图数据集上使用运行 SRW，并在 SRW 收敛之后再开始进行采样，具体我们以估算顶点平均度和估算顶点对的相似性来作为测量指标，分别对应顶点采样和边采样的场景。详细的实验环境、数据集介绍、随机游走达到收敛状态的判断条件请参考本章的第 §5.5.1小节。图 5.1展示了实验结果，我们首先可以看到不同图数据集以及不同策略指标的情况下，随机游走收敛所需要的步数都是不同的且差异非常大。[①] 另外我们也发现，在我们验证实验的各种场景下随机游走收敛都是非常慢的，其中最优的情况也需要七点多步达到收敛，而最差的情况下甚至需要接近六万步来使随机游走达到收敛，平均来说也都需要数万步。而当图数据的规模变得更大时，随机游走的收敛速度也往往会更慢。



(a) 估算顶点平均度（顶点采样）　　(b) 估算顶点对相似性（边采样）

图 5.1　简单随机游走在各个图数据上的收敛步数

[①]这里的收敛步数是指针对具体的测量指标收敛的均值所需要的随机游走的步数，因为在实际的实验中我们很难评估随机游走是否已经收敛到稳态分布，所以现有的相关工作中大多采用评估是否收敛的均值的方式来判断随机游走的收敛，详细的阐述请参考本章的第 §5.5.1小节。

由于基于随机游走的图采样需要在随机游走达到收敛之后才能获取有效的采样样本，因此在给定一个采样预算（总共访问的顶点数）的情况下，除去随机游走收敛开销，我们只能采样少量的有代表性的样本，因此基于采样分析的准确性就会受到严重的影响。比如我们的实验表明在 GooglePlus 数据集上通过 SRW 图采样估算顶点平均度，在保障估算的相对误差小于 7% 的情况下，大约需要一万个顶点的查询成本，即整个采样过程总共需要访问约一万个顶点，而这其中随机游走收敛过程就需要走八千多步，即使除去这八千多步访问过的重复顶点，这样的收敛开销的影响也是非常大的。

5.2.2　加速收敛的优化算法 CNARW

因此基于随机游走的图采样场景下的一个重要的问题就是：如何加速随机游走在大规模图上的收敛，从而提升基于随机游走的图采样的采样效率？在章节 §2.4中我们已经阐述了当前最常见的两类加速随机游走收敛速度的优化方向：（1）增加图的导通性，这类方法通常需要获取甚至改变全局或局部的图拓扑信息，所以在现实场景中通常不可行。（2）修改每一步随机游走的转移概率，这类方法通常是利用随机游走的历史信息，并且只需要访问少量的局部图信息。其中的关键问题是需要什么样的局部信息以及如何利用这些信息来加快随机游走的收敛。我们课题组最新提出的 CNARW 算法提出一种公共邻居感知的加权游走的策略，使得随机游能更容易走出当前的子图从而更快地探索到全局的图信息，最终加速随机游走的收敛过程。下面我们详细介绍 CNARW 算法的设计与实现。

（一）CNARW 算法设计

CNARW 首先引入集合导通性（Set Conductance）来刻画随机游走被困在某个顶点集的可能性；然后根据最大化集合导通性的原则来公式化随机游走下一跳的顶点选择；最后完成随机游走的状态转移矩阵的设计。下面我们通过一些公式化的表示和理论支持来介绍 CNARW 的具体算法设计及其形成过程。

（1）引入集合导通性

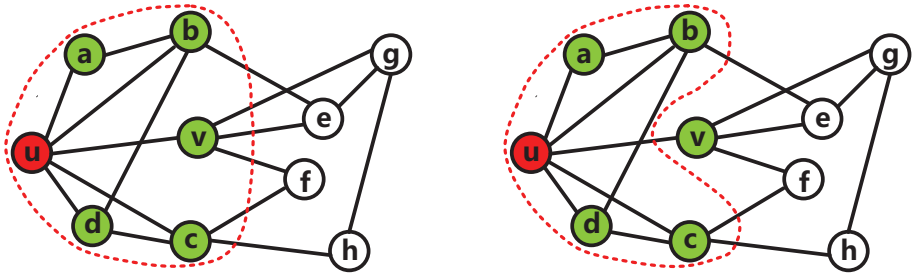
定义：集合导通性（set conductance），用 $G = (V, E)$ 表示一个无向图，$C \in V$ 是图中的一个顶点集，集合 C 的导通性 $\phi(C)$ 定义为

$$\phi(C) = \phi(C, V - C) = |E_{C,V-C}|/Vol(C). \tag{5.1}$$

其中 $E_{C,V-C} = (u, v)|u \in C, v \in V - C$，$Vol(C) = \sum_{u \in C} deg(u)$。集合导通性可以看成是顶点集合 C 和它的差集之间的边数除以 C 内部的边数。一个顶点集与图中其他顶点的连接数越多，顶点集内部的连接数越少，则该顶点集的导通性越大，随机游走被困在该顶点集中可能性也就越小。

（2）通过集合导通性公式化下一跳的顶点选择

假设随机游走当前停留在顶点 u，定义 $S = N(u) \cup \{u\}$ 为包含当前顶点和它的邻居顶点的边界顶点集（*frontier nodes*），如图 5.2(a) 所示。则我们可以用 $\phi(S)$ 来表示随机游走可能被困在 S 的概率程度。而其中一个候选顶点 v 对 $\phi(S)$ 的贡献度可以表示为 $\Delta\phi_v = \phi(S) - \phi(S_{-v})$，其中 $S_{-v} = S \setminus v$，如图 5.2(b) 所示。



(a) 边界顶点集 $S = N(u) \cup \{u\}$　　(b) $S_{-v} = S \setminus v$

图 5.2　边界集合 S 和 S_{-V}

通过定义，我们可以得到 $\Delta\phi_v$ 的表达式为

$$\Delta\phi_v = \frac{(1 - \phi(S)) - 2(C_{uv} + 1)/deg(v)}{(\sum_{i \in S} deg(i))/deg(v) - 1}. \tag{5.2}$$

其中 $deg(v)$ 表示顶点 v 的度，C_{uv} 表示顶点 u 和顶点 v 的公共邻居数。在 $deg(v)$ 不变的情况下，C_{uv} 越大，则 $\Delta\phi_v$ 越小。而在固定 C_{uv} 不变的情况下，$deg(v)$ 越大，则 $\Delta\phi_v$ 也越大。对于每个候选顶点 $v \in N(u)$，$\Delta\phi_v$ 可以作为下一跳顶点选择合适程度的测量指标。$\Delta\phi_v$ 越大，即度数高且与当前顶点公共邻居数少的顶点，我们给予顶点 v 在下一跳邻居选择中更高的权重。

（3）设计状态转移矩阵

直觉来说，可以将顶点 u 到顶点 v 的转移概率 P_{uv} 设置为一个正比于 $\Delta\phi_v$ 的值，比如为了避免 $\Delta\phi_v$ 的复杂计算，转移转移概率可以简单地设置为 $P_{uv} = 1 - \frac{C_{uv}}{deg(v)}$。但是为了保证随机游走的时间可逆性（*time reversible*），从而能够简单地获得它的稳态分布。因此设计转移概率时需要保证对称性（*symmetric*），即 $P_{uv} = P_{vu}$，具体 CNARW 将随机游走的转移转移概率设置为

$$P_{uv} \propto 1 - \frac{C_{uv}}{min(deg(u), deg(v))}. \tag{5.3}$$

（二）CNARW 算法实现

为了实现 CNARW 算法，我们需要在随机游走的每一步的下一跳邻居选择按照公式 (5.3) 中的比例实现加权的随机游走策略，即我们需要知道当前顶点的所有邻居顶点的度数和他们跟当前顶点的公共邻居数，即 $N(u)$ 中每个 v 的 C_{uv}

和 $deg(v)$。在网络环境中这是一个很大的计算成本，尤其是对于度数很大的顶点来说，因为我们需要获取当前顶点的所有邻居顶点的邻域信息。为了实现满足上述转移概率的随机游走，且限制计算开销，我们采用了一种带拒绝的随机游走策略 来确定下一跳顶点。具体的，在每一步随机游走时：

（1）我们首先从当前顶点 u 的邻居顶点 $N(u)$ 中随机均匀的选取一个候选顶点 v；

（2）然后计算顶点 v 的接收概率，表示为 $q_{uv} = 1 - \frac{C_{uv}}{min(deg(u),deg(v))}$；

（3）我们以 q_{uv} 的概率接收然后将随机游走跳转到顶点 v，以 $1 - q_{uv}$ 拒绝，并重新回到步骤（1）选取一个候选顶点；

（4）重复上述过程直至随机游走成功转发；

图 5.3展示了基于拒绝的随机游走转发策略的一个样例，其中随机游走在当前节点 u 被拒绝两次之后，转发到第三次选择的候选顶点 v。

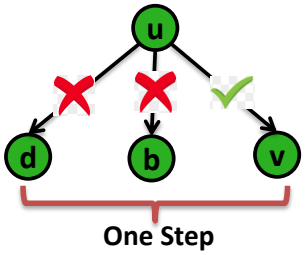


图 5.3　基于拒绝的随机游走转发策略

注：拒绝两次之后，将随机游走转发到第三次选择的候选顶点。

这种带拒绝的随机游走策略的好处是我们只需要访问被接收的顶点 v 及其之前访问过的顶点信息，而不需要访问顶点 u 的所有邻居，从而减少了网络环境中的查询开销。

在这种带拒绝的随机游走策略中，随机游走的每一次尝试可以以概率 $\tilde{p}_{uv} = \frac{1}{deg(u)} \times (1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}})$，从顶点 u 移动到它的邻居顶点 v。值得注意的是，在随机游走的一次尝试中，也有可能被选择的顶点被拒绝，使得该随机游走保留在当前访问的顶点 u 中，被拒绝的概率为 $\tilde{p}_{uu} = 1 - \sum_{v \in N(u)} \frac{1}{deg(u)} \times (1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}}) > 0$。因此，如果选择的顶点被拒绝，该随机游走需要重新从当前顶点的邻居顶点 $N(u)$ 中随机均匀的选取一个候选顶点，并循环这个过程，直到某个选择的顶点被接受。我们把上述过程称为 CNARW 的一步随机游走。根据上述过程，CNARW 的状态转移矩阵 $P = [P_{uv}]_{u,v \in V}$ 可以被重写为：

$$P_{uv} = \begin{cases} \tilde{p}_{uv}/(1 - \tilde{p}_{uu}), & \text{if } v \in N(u), \\ 0, & \text{otherwise}, \end{cases} \tag{5.4}$$

其中，$\tilde{p}_{uv}$ 被定义为：

$$\tilde{p}_{uv}=\begin{cases}\frac{1}{deg(u)}\times(1-\frac{C_{uv}}{\min\{deg(u),deg(v)\}}), & \text{if } v\in N(u),\\ 1-\sum_{k\in N(u)}\tilde{p}_{uk}, & \text{if } v=u,\\ 0, & \text{otherwise.}\end{cases} \tag{5.5}$$

证明: 根据带拒绝的随机游走和循环直至接收的策略，我们记录从顶点 u 到它的一个邻居顶点 v 的转移概率为：

$$\begin{aligned}P_{uv} &= \frac{q_{uv}}{deg(u)}+\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)}\times\frac{q_{uv}}{deg(u)}\\ &+ [\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)}]^2\times\frac{q_{uv}}{deg(u)}+...\\ &= \tilde{p}_{uv}\times[1+\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)}\\ &+ (\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)})^2+...]\\ &= \tilde{p}_{uv}\times\frac{1-(\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)})^n}{1-\frac{\sum_{k\in N(u)}(1-q_{uk})}{deg(u)}}\\ &\overset{n\to\infty}{=} \tilde{p}_{uv}\times\frac{1}{1-\tilde{p}_{uu}}\end{aligned}$$

根据上述步骤实现的 CNARW 算法的一步随机游走过程如算法 5.1所示。

算法 5.1　CNARW 的一步随机游走

Input: current node u

Output: next-hop node v

do

　Select v uniformly at random from u's neighbors;

　Generate a random number $q \in [0, 1]$;

　Compute $q_{uv} = 1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}}$;

while *($q > q_{uv}$)*;

Return v;

5.2.3　基于随机游走历史路径的优化前景

CNARW 算法对比于经典的随机游走算法（SRW）和当时最新的加速随机游走的优化算法（NBRW 和 CNRW）都有明显的性能提升，即 CNARW 的利用随

机游走的历史信息和候选顶点的邻居信息进行加权游走的策略能显著加速随机游走的收敛速度，并提升随机游走的采样效率。在此基础上，我们发现这些历史信息和邻居信息可以进一步挖掘，从而进一步优化随机游走的收敛过程。而且，CNARW 算法的优化与此前的优化算法的优化上正交的，如 NBRW 算法，因此，我们可以将他们融合从而达到更优的结果。

5.3　基本思想和贡献点

（一）本章工作的主旨思想

为了进一步提升随机游走的收敛速度和采样效率，本章在 CNARW 工作的基础上，提出一种非回溯的公共邻居感知的随机游走算法。**我们在设计背后的直觉只是“已访问顶点和候选顶点的邻居信息很重要”，因此我们综合考虑随机游走前一步访问过的顶点及其邻居信息、以及候选顶点的邻居信息来优化随机游走下一步跳转的顶点选择。**

具体来说，我们考虑到简单随机游走收敛慢的主要原因是：一般的社交网络都有很高的聚类特性，即图中形成很多的社区结构，社区内部的顶点连接紧密，社区之间连接稀疏。所以简单随机游走的过程中，随机游走均匀随机地选取当前顶点的一个邻居跳转，大概率会选到社区内部的顶点，而且很容易陷入当前子图，即随机游走只在社区内部反复游走在已经访问过的顶点，只有很小的概率能走出当前的子图从而去探索到全局的图信息。这大大减慢了简单随机游走的收敛速度。所以为了加速随机游走的收敛，我们需要尽量减少对已经访问过的顶点的频繁的再次访问。

本章提出的非回溯的公共邻居感知的随机游走算法在每一步随机游走选择的时候，不同于简单随机游走的均匀随机选择，我们给访问过的顶点小一点的访问概率，而给有更大机会能探索更多未访问的顶点的那些顶点大一点的访问概率。具体来说，我们根据下一跳候选顶点的信息以及一些历史访问信息，综合考虑下面三个方面的结果，重新设置每一步随机游走的转移概率。

(1) 若一个候选顶点的度（即邻居顶点数）越大，则它可能访问到更多未访问的顶点的概率也就越大。
(2) 若一个候选顶点与当前顶点的公共邻居数越少，则随机游走通过该候选顶点再次回到当前顶点的概率也就越小。
(3) 若一个候选顶点正好是随机游走上一步访问过的顶点，则随机游走重复之前的游走路径的概率也就越大。

所以在随机游走的各个候选顶点中，我们给度数高且与当前顶点公共邻居数少的顶点更大的转移概率，并且尽量避免将随机游走回溯到上一步刚刚访问

过的顶点，通过这种加权游走的策略实现更快的随机游走收敛。图 5.4展示了一个例子，其中随机游走上一步从顶点 a 移动到当前顶点 u，我们避免随机游走在下一步再次回到顶点 a，因此我们将顶点 u 剩余的四个邻居顶点作为候选顶点，然后根据这些候选节点的度以及他们与当前顶点的公共邻居数，综合计算随机游走在下一步转移到各个候选顶点的概率，最终计算得到的概率分布如图 5.4所示，其中顶点 v 因为度数高且与当前顶点公共邻居数最少，因此转移概率最大。

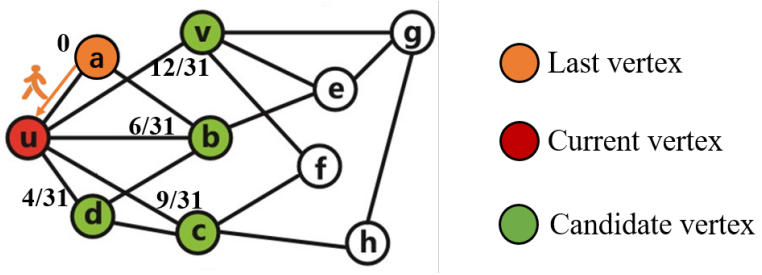


图 5.4　非回溯的公共邻居感知的随机游走

（二）本章工作考虑的主要问题

基于上面阐述的基本思想，本章实现了相应的随机游走算法 NB-CNARW（Non-Backtracking Common Neighbor Aware Random Walk）。NB-CNARW 算法在设计过程中主要考虑了下面几个方面的问题：

- 如何将 NBRW 算法中非回溯的思想融合到 CNARW 算法中，这样是否进一步加速随机游走的收敛过程？
- 如何设计随机游走的在每一步的状态转移概率？以及如何高效地实现基于该转移概率的加权随机游走的过程？
- NB-CNARW 算法的设计是否具有理论保障，包括随机游走算法随机性的证明、随机游走算法的稳态分布的分析等？

（三）本章工作的主要贡献点

基于对上述问题的思考，本章工作主要有以下几个贡献点：

(1) **非回溯的公共邻居感知的加权随机游走**：为了研究 CNARW 算法与现有随机游走算法中设计的融合性，本章工作以 NBRW 为例，在公共邻居感知的随机游走算法的基础上进一步合并 NBRW 中非回溯的随机游走的思想，提出 NB-CNARW 算法进一步加速 CNARW 的收敛速度。

(2) **多种形式的状态转移矩阵的设计**：为了研究如何利用随机游走的历史信息和候选顶点的邻居信息，我们在 NB-CNARW 的算法设计中进一步考虑利用这两点重要信息的其他转移矩阵的设计形式，具体包括如何设计候选顶点和当前顶点的度的使用形式？以及如何设置选顶点和当前顶点的公共邻居数的影响权重？

(2) **理论分析**：针对我们提出 NB-CNARW 算法和基于 NB-CNARW 的图采样，本章工作都进行的相应的理论分析，包括 NB-CNARW 作为随机游走算法的随机性的证明、NB-CNARW 收敛之后的稳态分布的分析以及基于 NB-CNARW 的图采样的采样效率的分析。

(4) **性能提升**：本章工作基于上述设计，使用 C++ 实现了我们的 NB-CNARW 算法，并在真实世界的数据集上与现有的几个随机游走算法进行了对比性能评估。实验结果表明，NB-CNARW 对比最新的一些随机游走算法 SRW、NBRW、CNRW 以及 CNARW 都有明显的性能提升，具体来说，NB-CNARW 可以减少高达 71.9% 的收敛步数；使用随机游走算法进行采样评估时，NB-CNARW 在保证相同估算精度的情况下可以减少高达 35.7% 的查询成本。

5.4　NB-CNARW 的设计与分析

为了加速随机游走算法的收敛速度，从而减少随机游走采样过程中收集样本阶段的时间开销。我们在 CNARW 算法的基础上设计并实现了一个非回溯的公共邻居感知的随机游走算法 NB-CNARW ，通过利用随机游走前一步访问过的顶点以及该顶点的邻域信息来优化下一步跳转的邻居选择。NB-CNARW 的算法设计主要包含两个方面：公共邻居的感知策略和访问顶点不回溯策略。我们先介绍 NB-CNARW 的算法设计及其理论分析，然后介绍如何使用 NB-CNARW 实现对大规模图数据的无偏采样。

5.4.1　NB-CNARW 的设计与实现

（一）基础算法的设计与实现

NBRW[28] 是第一个利用随机游走的路径历史，来避免回溯到随机游走上一步经过的顶点来加速收敛的经典随机游走算法，具体的算法介绍请参见章节 §2.4.2。文献[28] 中表明，NBRW 在理论上保证了能够比 SRW 更有效地实现无偏图采样。值得注意的是，公共邻居感知的 CNARW 算法是不同于 NBRW 算法的，且与 NBRW 算法是正交的关系。为了进一步加速随机游走的收敛，我们考虑在 CNARW 算法中加入 NBRW 的不回溯的思想，并提出一个新的随机游走优化算法 NB-CNARW 。具体来说，NB-CNARW 在随机游走的过程中首先避免回溯到其上一步经过的顶点，并在剩余的候选顶点中采用公共邻居感知的加权游走的策略来选择随机游走的下一跳顶点。因此，NB-CNARW 中随机游走的转移转移概率可以设置为

$$P_{uv} \propto \frac{1}{deg(u)-1} \times (1 - \frac{C_{uv}}{min(deg(u), deg(v))}). \tag{5.6}$$

在 NB-CNARW 算法的实现中，我们依然采取基于拒绝的随机游走的转发策略，首先以一定的选择概率随机选择一个候选顶点，然后再以一定的接收概率接收该候选顶点并将随机游走转发到该顶点。我们将随机游走上一步刚刚访问过的顶点记录为 x，则 NB-CNARW 算法中的选择概率 b'_{uv} 表达如下：

$$b'_{uv}=\begin{cases}\frac{1}{deg(u)-1}, & \text{if } v\in N(u), v\neq x,\\ 0, & \text{otherwise,}\end{cases} \tag{5.7}$$

然后我们以概率 q_{uv} 接受上一步选择的顶点，并将随机游走移动到该顶点；以概率 $1-q_{uv}$ 拒绝选择的顶点并以概率 b'_{uv} 重新选择一个顶点，循环这个过程直到随机游走成功移动。NB-CNARW 算法中的接收概率 q'_{uv} 表达如下：

$$q'_{uv}=\begin{cases}1-\frac{C_{uv}}{min(deg(u),deg(v))}, & \text{if } v\in N(u), v\neq x,\\ 0, & \text{otherwise,}\end{cases} \tag{5.8}$$

根据上述过程，NB-CNARW 算法相对应的状态转移矩阵 $\boldsymbol{P}'=[\boldsymbol{P}'_{uv}]_{u,v\in v}$ 表达如下：

$$\boldsymbol{P}'_{uv}=\begin{cases}\tilde{p}'_{uv}/(1-\tilde{p}'_{uu}), & \text{if } v\in N(u), v\neq x,\\ 0, & \text{otherwise,}\end{cases} \tag{5.9}$$

其中，$\tilde{p}'_{uv}$ 定义为：

$$\tilde{p}'_{uv}=\begin{cases}\frac{1}{deg(u)-1}\times(1-\frac{C_{uv}}{\min\{deg(u),deg(v)\}}), & \text{if } v\in N(u), v\neq x\\ 1-\sum_{k\in N(u)}\tilde{p}'_{uk}, & \text{if } v=u,\\ 0, & \text{otherwise.}\end{cases} \tag{5.10}$$

最终我们完成 NB-CNARW 算法的设计与实现，NB-CNARW 算法下一步随机游走的转发流程如算法 5.2所示。

算法 5.2　NB-CNARW 的一步随机游走

Input: current node u, last node x

Output: next-hop node v

do

　Select v uniformly at random from u's neighbors except x;

　Generate a random number $q\in[0,1]$;

　Compute $q_{uv}=1-\frac{C_{uv}}{\min\{deg(u),deg(v)\}}$;

while *($q>q_{uv}$)*;

Return v;

（二）转移矩阵的其他设计

在上述 NB-CNARW 算法的转移矩阵的设计中，我们充分利用了随机游走的历史信息和候选顶点的邻居信息。在这一小节中，我们进一步考虑利用这两点重要信息的其他转移矩阵的设计形式，具体包括如何设计候选顶点和当前顶点的度的使用形式？以及如何设置选顶点和当前顶点的公共邻居数的影响权重？下面我们分别介绍这两种不同的转移矩阵的设计方式。

（1）度的使用形式

在 NB-CNARW 算法的接受概率的设置中，为了保证最终得到的状态转移概率满足对称性从而实现随机游走稳态分布的分析，我们在式 (5.8) 中使用函数 $min(deg(u), deg(v))$ 来利用候选顶点和当前顶点的度的信息。我们也可以考虑其他的使用形式，比如也可以将 min 函数替换成 sum 函数或 max 函数，即 $deg(u) + deg(v)$ 或者 $max(deg(u), deg(v))$，这样的设计也能保证最终得到的状态转移概率的对称性。我们在本章的第 §5.5.3小节中通过实验评估了对候选顶点和当前顶点的度的不同使用形式对性能的影响。

（2）公共邻居数的影响权重

在 NB-CNARW 算法的状态转移矩阵的设置中，我们考虑当前顶点和候选顶点公共邻居数与候选顶点度数的比值，并将公共邻居数的影响权值设为 1。实际上，我们也可以考虑公共邻居数与候选顶点度数的不同权值设置下的比值的影响，我们将公共邻居数的权值记为 α，则相应的状态转移矩阵的表达式可以修改为：

$$P_{uv} \propto \frac{1}{deg(u)-1} \times (1 - \frac{\alpha \times C_{uv}}{\min\{deg(u), deg(v)\}}) \tag{5.11}$$

其中 $0 \leqslant \alpha \leqslant 1$ 从而保证 $0 \leqslant P_{uv} \leqslant 1$。当 α 减小时，则当前顶点与候选顶点公共邻居数的权重也随之减小。特别的，设置 $\alpha = 0$ 和 $\alpha = 1$ 分别代表 NBRW 和 NB-CNARW 。我们还通过实验研究了公共邻居权重的影响，具体的实验设计和结果请参考本章的第 §5.5.3小节。

5.4.2　NB-CNARW 理论分析

为了保证 NB-CNARW 算法的正确性和有效性，我们首先证明 NB-CNARW 算法的随机性，然后提供 NB-CNARW 算法收敛之后稳态分布的理论分析。

（一）NB-CNARW 的随机性证明

NB-CNARW 算法随机性（即某个顶点上的一步随机游走移动到到其各个邻居顶点的概率之和为 1）的证明保证了 NB-CNARW 作为一个随机游走算法的正确性，是后面随机游走稳态分布的理论分析的基础。

定理 5.1　给定一个无向连通图 $G(V,E)$，G 上的 *NB-CNARW* 是具有随机性的随机游走算法，即 $\sum_{k\in N(u)} P'_{uk} = 1$.

证明: 基于公式 (5.9) 和公式 (5.10)，有

$$\begin{aligned}\sum_{k\in N(u)} P'_{uk} &= \sum_{k\in N(u)} \tilde{p}'_{uv} \times \frac{1}{1-\tilde{p}'_{uu}} \\ &= \frac{1}{1-\tilde{p}'_{uu}} \times \sum_{k\in N(u)} \tilde{p}'_{uv} \\ &= \frac{1}{1-\tilde{p}'_{uu}} \times (1-\tilde{p}'_{uu}) = 1.\end{aligned}$$

因此，NB-CNARW 算法的随机性得证。

（二）NB-CNARW 的稳态分析

在 NB-CNARW 算法是一个正确的随机游走算法的基础上，我们可以利用随机游走的马尔可夫性质分析 NB-CNARW 算法的稳态分布，包括 NB-CNARW 算法稳态分布的唯一存在性、以及达到稳态分布之后 NB-CNARW 访问每个顶点和每条边的概率分布。

（1）稳态分布的唯一存在性

定理 5.2　给定一个无向连通图 $G(V,E)$，在 G 上运行 *NB-CNARW* 存在唯一的一个稳态分布。

证明: 对于 G 上的任意一对顶点 $u \in V$ 和 $v \in N(u)$，其对应的接收概率 $q_{uv} = 1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}}$ 大于 0，因此相应的状态转移概率 P_{uv} 也大于 0。又因为 G 是一个无向连通图，所以 G 上的任意一对顶点 $u \in V$ 和 $v \in V$ 之间通过有限的 NB-CNARW 步之后总是相互可达的，也就是说 NB-CNARW 构造的马尔可夫链是不可约的。文献[87] 中指出，在无向连通图上的任何不可约马尔可夫链都具有唯一的稳态分布，所以 NB-CNARW 也具有唯一的稳态分布。因此，NB-CNARW 作为随机游走算法的稳态分布的唯一存在性得证。

（2）稳态分布下访问各个顶点的概率

定理 5.3　给定一个无向连通图 $G(V,E)$，在 G 上运行的 *NB-CNARW* 在收敛以后达到稳态分布，其稳态分布下访问各个顶点的概率 π 满足：

$$\frac{\pi(u)}{\pi(v)} = \frac{(deg(u)-1)(1-p'_{uu})}{(deg(v)-1)(1-p'_{vv})} \tag{5.12}$$

因此，我们可以得出

$$\pi(u) = Z \times (deg(u)-1) \times (1-p'_{uu}) \tag{5.13}$$

其中 Z 是一个归一化的常数。

证明: 我们首先证明，通过上述提到的 NB-CNARW 算法中转移矩阵的对称性的设计，NB-CNARW 构造的马尔可夫链具有时间可逆性。根据文献[88] 中的命题 1.1，我们只需要证明：对于 G 上的任意一对顶点 $u \in V$ 和 $v \in V$，以下方程都有一个唯一解。则证明 NB-CNARW 构造的是一个时间可逆的马尔可夫链。

$$\pi(u) \times P_{uv} = \pi(v) \times P_{vu}. \tag{5.14}$$

基于公式 (5.9) 和公式 (5.10)，公式 (5.14) 可以写成

$$\begin{aligned}
\frac{\pi(u)}{\pi(v)} &= \frac{P'_{vu}}{P'_{uv}} \\
&= \frac{\tilde{p'}_{vu}/(1-\tilde{p'}_{vv})}{\tilde{p'}_{uv}/(1-\tilde{p'}_{uu})} \\
&= \frac{(\frac{1}{deg(v)-1} \times (1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}}))/(1-\tilde{p'}_{vv})}{(\frac{1}{deg(u)-1} \times (1 - \frac{C_{uv}}{\min\{deg(u),deg(v)\}}))/(1-\tilde{p'}_{uu})} \\
&= \frac{(deg(u)-1)(1-\tilde{p}_{uu})}{(deg(v)-1)(1-\tilde{p'}_{vv})}.
\end{aligned}$$

在给定的无向连通图 G 上，上式中的 $1-\tilde{p'}_{vv}$ 和 $1-\tilde{p'}_{uu}$ 都是固定不变的，因此上式存在一个唯一解，NB-CNARW 构造的马尔可夫链具有的时间可逆性得证。根据上式，我们也可以推导出 NB-CNARW 的稳态分布中顶点 u 对应的静态概率是：

$$\pi(u) = Z \times (deg(u)-1) \times (1-\tilde{p'}_{uu}),$$

其中，$Z = [\sum_{u \in V}((deg(u)-1) \times (1-\tilde{p}_{uv}))]^{-1}$ 是标准化之后得到的一个常数。

（3）稳态分布下访问各个边的概率

定理 5.4　给定一个无向连通图 $G(V,E)$，在 G 上运行的 *NB-CNARW* 在收敛以后达到稳态分布，其稳态分布下访问各个边的概率 $\pi(e_{uv})$ 满足：

$$\pi(e_{uv}) = Z \times (1 - \frac{C_{uv}}{\min\{deg(u), deg(v)\}}) \tag{5.15}$$

证明: 基于公式 (5.9) 和公式 (5.13)，有

$$\begin{aligned}
\pi(e_{uv}) &= \pi(u) \times P'_{uv} \\
&= Z \times (deg(u)-1) \times (1-\tilde{p'}_{uu}) \times \frac{\tilde{p'}_{uv}}{1-\tilde{p'}_{uu}} \\
&= Z \times (deg(u)-1) \times \tilde{p'}_{uv} \\
&= Z \times (deg(u)-1) \times \frac{1}{deg(u)-1} \times (1 - \frac{C_{uv}}{\min\{deg(u), deg(v)\}}) \\
&= Z \times (1 - \frac{C_{uv}}{\min\{deg(u), deg(v)\}})
\end{aligned}$$

5.4.3　基于 NB-CNARW 采样的无偏估计

接下来我们介绍如何使用 NB-CNARW 算法来进行无偏图采样，其中包含无偏点采样和无偏边采样。

（1）无偏点采样

通过上面的理论分析可知，NB-CNARW 算法在收敛之后访问各个顶点的概率是已知的，但稳态概率分布并不是均匀的。所以，为了实现图上的基于 NB-CNARW 采样的关于某个顶点指标的无偏估计，我们首先需要根据每个顶点被采样到的概率进行偏差校正。我们设置顶点 u 上的权重 $w(u)$ 为：

$$w(u) = \frac{Z}{\pi(u)} = \frac{1}{(deg(u)-1)\times(1-\tilde{p}'_{uu})}. \tag{5.16}$$

然后基于上述权重设置，我们推导出基于 NB-CNARW 采样的关于某个顶点指标的无偏估计：

定理 5.5　给定一个无向连通图 $G(V,E)$ 以及图上关于顶点的一个指标函数 $f(v)$，通过 G 上的 *NB-CNARW* 采样收集顶点样本集合 R，当样本数量足够大时，比如 $|R|\to\infty$，关于函数 $f(v)$ 的无偏估计 $\mu(f)$ 可以通过下面公式实现：

$$\mu(f) = \frac{\sum_{u\in R} w(u)f(u)}{\sum_{u\in R} w(u)} = \frac{\sum_{u\in R}\frac{U(u)}{\pi(u)}f(u)}{\sum_{u\in R}\frac{U(u)}{\pi(u)}} \to E_U(f),\ a.s. \tag{5.17}$$

其中，$U(i) = 1/n$ 为均匀分布。

证明：基于公式 (5.16) 和强大数定理（SLLN）有：

$$\begin{aligned}\mu(f) &= \frac{\sum_{u\in R} w(u)f(u)}{\sum_{u\in R} w(u)}\\ &= \frac{\frac{1}{n}\sum_{u\in R}\frac{Z}{\pi(u)}f(u)}{\frac{1}{n}\sum_{u\in R}\frac{Z}{\pi(u)}}\\ &= \frac{\sum_{u\in R}\frac{U(u)}{\pi(u)}f(u)}{\sum_{u\in R}\frac{U(u)}{\pi(u)}} \to E_U(f),\ a.s.\end{aligned}$$

（2）无偏边采样

现实的图计算应用中，除了对图中顶点的相关属性进行无偏估计的场景外，对图中边上关联的属性（比如顶点对的平均相似度，顶点对的平均最短距离）无偏估计的场景也很常见。相似的，为了实现图上的基于 NB-CNARW 的图采样的关于某个边指标的无偏估计，我们也首先需要根据每个边被采样到的概率进行偏差校正。我们设置边 e_{uv} 上的权重 $w(e_{uv})$ 为：

$$w(e_{uv}) = \frac{Z}{\pi(e_{uv})} = \frac{1}{1-\frac{C_{uv}}{\min\{deg(u),deg(v)\}}}. \tag{5.18}$$

基于 NB-CNARW 采样的关于某条边指标的无偏估计为：

定理 5.6　给定一个无向连通图 $G(V,E)$ 以及图上关于边的一个指标函数 $f(e)$，通过 G 上的 *NB-CNARW* 采样收集边样本集合 R，当样本数量足够大时，比如 $|R|\to\infty$，关于函数 $f(e)$ 的无偏估计 $\mu(f)$ 可以通过下面公式实现：

$$\mu(f)=\frac{\sum_{e_{uv}\in R} w(e_{uv})f(e_{uv})}{\sum_{e_{uv}\in R} w(e_{uv})}\to E_U(f),\ a.s. \tag{5.19}$$

证明：基于公式 (5.18)、$U(e_{uv})=\frac{1}{2|E|}$ 以及 SLLN 有：

$$\begin{aligned}\mu(f)&=\frac{\sum_{e_{uv}\in R} w(e_{uv})f(e_{uv})}{\sum_{e_{uv}\in R} w(e_{uv})}\\&=\frac{\sum_{e_{uv}\in R}\frac{Z}{\pi(e_{uv})}\times f(e_{uv})}{\sum_{e_{uv}\in R}\frac{Z}{\pi(e_{uv})}}\\&=\frac{\sum_{e_{uv}\in R}\frac{U(e_{uv})}{\pi(e_{uv})}\times f(e_{uv})}{\sum_{e_{uv}\in R}\frac{U(e_{uv})}{\pi(e_{uv})}}\\&\to\frac{E_\pi(\frac{U(X)}{\pi(X)}f(X))}{E_\pi(\frac{U(X)}{\pi(X)})}=E_U(f),\quad a.s.\end{aligned}$$

（3）基于 NB-CNARW 的采样效率

根据文献[28]，在一个随机游走算法中，通过避免回溯到该随机游走上一步经过的顶点从而产生的新的随机游走算法仍然具有一个唯一的稳态分布，因此依然能够用于实现无偏图采样。并且，当我们使用新的随机游走进行无偏图采样时，我们可以得到一个比使用原随机游走更小的渐近方差。也就是说，使用新的避免回溯的随机游走算法，可以在较少样本的情况下实现相同的采样精度。我们重申他的发现如下：

定理 5.7　[28][Theorem3] 假设 $\{X_t\}$ 是状态空间 N 上的一个不可约的马尔可夫链，其状态转移矩阵为 $P=\{P(u,v)\}$，其收敛之后的稳态分布为 π。构建状态空间 Ω 上的一个新的马尔可夫链 $\{Z'_t\}$，其状态转移矩阵为 $P'=\{P'(e_{uv},e_{xy})\}$，若使得其状态转移矩阵 $P'(e_{uv},e_{xy})$ 满足下面两个条件：

对于所有的 $e_{uv},e_{vu},e_{vy},e_{yv}\in\Omega$，其中 $u\neq y$ 有：

$$P(v,u)P'(e_{uv},e_{vy})=P(v,y)P'(e_{yv},e_{vu}), \tag{5.20}$$

$$P'(e_{uv},e_{vy})\geqslant P(v,y). \tag{5.21}$$

则新的马尔可夫链 $\{Z'_t\}$ 不可约、不可逆且具有一个唯一的稳态分布 π'，且 π' 满足 $\pi'(e_{uv})=\pi(u)P(u,v)$，其中 $e_{uv}\in\Omega$。并且对于任意的函数 f，通过新的

马尔可夫链的稳态分布采样计算得到的渐进方差 $\tilde{\mu}_t'(f)$ 总是不大于用原本的马尔可夫链的稳态分布采样计算得到的渐进方差 $\tilde{\mu}_t(f)$, 即 $\sigma'^2(f) \leqslant \sigma^2(f)$。

根据定理 5.7，如果我们可以证明 NB-CNARW 算法满足等式 (5.20) 和等式 (5.21) 中的两个条件，则我们就可以得到 NB-CNARW 算法的稳态分布 π' 和渐进方差 $\tilde{\mu}_t'(f)$ 的理论分析。我们在定理 5.8中进行了阐述并给出了证明。

定理 5.8　*在无向连通图 $G = (V, E)$ 上运行 NB-CNARW 算法，等到 NB-CNARW 收敛之后，G 上的任意一条边 $e_{uv} \in E$ 被访问的静态分布的概率 $\pi'(e_{uv})$ 满足 $\pi'(e_{uv}) = \pi(u) \times P(u, v) = \pi(e_{uv})$，即等于 e_{uv} 在 CNARW 算法中被访问的静态分布的概率。另外，对于任意函数 f，NB-CNARW 算法产生的渐近方差 $\tilde{\mu}_t'(f)$ 不大于 CNARW 算法产生的渐近方差 $\tilde{\mu}_t(f)$，即 $\sigma'^2(f) \leqslant \sigma^2(f)$。*

证明：另 $X_t \in V$ $(t = 0, 1, 2, ...)$ 表示 CNARW 算法的随机游走经过的顶点位置。我们基于定理 5.7构建一个扩展的不回溯的马尔可夫链，然后分别证明扩展的马尔可夫链满足等式 (5.20) 和等式 (5.21) 中的条件。

第一步：证明 $P(v, u)P'(e_{uv}, e_{vy}) = P(v, y)P'(e_{yv}, e_{vu})$：

$$\begin{aligned} & P(v, u)P'(e_{uv}, e_{vy}) \\ =& \frac{1}{deg(v)} \times \frac{1}{deg(v) - 1} \\ =& P(v, y)P'(e_{yv}, e_{vu}) \end{aligned}$$

第二步：证明 $P'(e_{uv}, e_{vy}) \geqslant P(v, y)$：

$$\begin{aligned} P'(e_{uv}, e_{vy}) &= \frac{1}{deg(v) - 1} \\ &\geqslant \frac{1}{deg(v)} = P(v, y) \end{aligned}$$

综合上面两个步骤，我们完成了定理 5.8的证明。

根据定理 5.8，我们可以推导出当我们使用 NB-CNARW 算法进行无偏图采样时，相比较于使用 CNARW 算法进行无偏图采样，我们可以得到一个更小的渐近方差。即使用 NB-CNARW 算法进行无偏图采样可以在使用更少采样样本的情况下实现相同的计算精度。也就是说，通过引入 NBRW 中不回溯的思想，我们可以进一步降低通过 CNARW 算法进行无偏采样的采样成本。

5.5　实验评估

在上一小节中我们详细介绍了非回溯的公共邻居感知的随机游走算法设计并提供了理论分析。本小节我们使用 C++ 实现了 NB-CNARW ，并在真实网络数据集上进行了广泛的实验，以评估 NB-CNARW 的效率和准确度。本章的实验评估主要回答下面几个问题：

- 对比于最新的随机游走算法，NB-CNARW 是否能加速随机游走的收敛？
- 不同状态转移矩阵设计对 NB-CNARW 算法的收敛速度有什么影响？
- 使用基于随机游走的图采样技术实现对某策略指标的无偏估计时，NB-CNARW 是否能在实现相同估计精度的情况下有效减少查询开销？

5.5.1　实验设置

（一）实验设备

我们在一台装有 32 个 Intel Xeon E5-2650 2.60GHz CPU 处理器和 64GB 内存空间的的戴尔服务器上进行我们的实验评估，具体的软硬件配置如表 5.1所示。

表 5.1　服务器的硬件配置和软件环境

配置环境	详细信息
服务器型号	Dell Power Edge R720
CPU	Intel(R) Xeon(R) CPU E5-2650 v2 @2.60GHz processor ×32
内存	64GB 2133MHz DDR4
外存	500GB SamSung 860 SSD
操作系统	Ubuntu 16.04.12
内核版本	Linux version 4.15.0-112-generic

（二）图数据集

本实验所使用的数据集主要来源于斯坦福大学的一个网络分析的项目 SNAP[89]（Stanford Network Analysis Project），和用于交互式图分析和可视化的网络数据存储库[90]，这些数据集都是现有社交网络的真实数据，其简单统计信息如表 5.2所示。为了保证随机游走能够收敛，我们所有的实验都在无向图上进行，对于有向图的数据集，我们去掉图中只出现在一个方向上的边，选择最大的连通分量将其转换为无向图。这种有向图转无向图的方法在以前的相关工作中已经被广泛使用过 [91-93,72,76]。我们将上述数据集分为两组：（1）大规模图数据集，包含 GooglePlus、Flickr、DBLP 和 LiveJournal，用于实验评估随机游走算法的收敛速度和查询成本；（2）小规模图，包含 Facebook、Ca-GaQc 和 Phy1，用于理论计算随机游走算法的 SLEM，指示相应的收敛速度，因为对于大规模图数据集来说，SLEM 的计算成本太高。

（三）对比算法

我们将我们的 NB-CNARW 算法与四种典型的随机游走采样算法进行了比较：(1) 简单随机游走 (**SRW**)[69]，作为我们的比较基线；(2) 非回溯随机游走 (**NBRW**)[28]，首次利用随机游走的上一步的历史信息加速收敛；(3) 循环邻居随

表 5.2　本章实验使用的图数据集信息

Dataset Name	# of Nodes	# of Edges	Avg. Clustering Coefficient
Facebook	775	28012	0.4714
Ca-GaQc	2879	18474	0.4416
Phy1	4158	26844	0.5486
Google Plus	64517	2867802	0.3428
Flickr	80513	11799764	0.1652
DBLP	226413	1432920	0.6353
LiveJournal	1500000	29425194	0.2552

机游走 (**CNRW**)[76]，利用随机游走的所有历史路径信息来加速收敛。(4) 公共邻居感知随机游走 (**CNARW**)[76]，这是目前最新的随机游走历史路径感知采样算法为了进行公平的对比评估，我们将所有算法都用 C++ 实现。

（四）性能指标

（1）转移矩阵的第二大特征值模 SLEM

我们前面提到，基于随机游走的采样性能严重依赖于随机游走的收敛速度。具体来说，如果随机游走采样算法收敛速度快，那么在相同的查询预算下，它可以通过采样得到更多有代表性的样本，从而实现较小的估计误差。因此，我们首先使用混合率的概念来测量随机游走采样算法收敛到其平稳分布的速度有多快，其中关键指标就是转移矩阵的第二大特征值模（SLEM）[94]。

（2）收敛到均值的随机游走步长

转移矩阵的第二大特征值模的计算复杂度非常高，因此只能在小规模图上计算评估。为了进一步对比评估了在大规模图上的收敛速度，我们采用另一个概念：*收敛到均值*。因为我们很难衡量随机游走是否已经收敛到稳态分布，因此我们使用*收敛到均值*来评估图统计量收敛到均值（例如平均顶点度）所需要的随机游走的步长。注意，*收敛到均值*的概念只适用于估计一些定义在抽样变量上的函数的平均值的抽样任务，例如平均顶点度和平均局部聚类系数等，这个度量取决于抽样任务。为了量化*收敛到均值*的速度，我们定义

$$T_{\mathrm{cm}} \triangleq \mathbb{E}[\text{min. \# of steps needed to converge to the mean}].$$

计算 T_{cm} 的精确值的开销也很大，因此我们通过模拟 RWSA 来估计 T_{cm}，并使用 Geweke 收敛监视器来检测 RWSA 是否已经收敛到均值。Geweke 收敛监视器的方法计算效率高且可扩展到大规模网络，因此在之前的著作中已经被广泛使用[95-96]，我们设置其关键参数，即阈值默认为 $Z \leqslant 0.1$。为了评估*收敛到均值*的速度，即 T_{cm}，我们重复模拟 n 次，得到 n 样本 $T_{\mathrm{cm}}^1, \cdots, T_{\mathrm{cm}}^n$，并研究他们的均值

和标准差。在数学上，我们通过以下指标来评估平均收敛速度:

$$\bar{T}_{\mathrm{cm}} \triangleq \sum_{i=1}^{n} T_{\mathrm{cm}}^{i}/n.$$

我们使用标准差（SD）来量化收敛速度的变化，并使用以下估计器来估算 SD：

$$\sigma(T_{\mathrm{cm}}) \triangleq \sqrt{\sum_{i=1}^{n}(T_{\mathrm{cm}}^{i} - \bar{T}_{\mathrm{cm}})^2/(n-1)}.$$

（3）估算误差和查询开销

采样算法的一个基本的性能权衡就是 em 估计误差和 em 查询代价。一般而言，随着查询预算（或查询成本）的增加，采样算法的估计误差也会减小。本章我们采用相对误差来量化估计精度：

$$\text{相对误差} \triangleq |\hat{X} - X|/X,$$

其中 $\hat{X}$ 和 X 表示关于某个测量指标，比如平均度的估计值和真实值。

我们将查询开销定义为采样中访问的去重后独立采样样本的数目，包括收敛阶段访问的样本以及收敛之后的采样阶段访问的样本：

$$\text{查询开销} \triangleq \#\{\text{采样任务中访问的独立样本的数目}\}.$$

假设一个采样任务在随机游走过程中访问的一个顶点序列为 (a,b,c,d,a,c,d)，则查询开销为 4。这是一个合理的开销度量方式，也被广泛用于评估采样算法的效率[76,72,91]。我们只考虑唯一的独立样本的原因是，一旦一个顶点被访问过，我们就可以在本地存储其关联的信息，因此当它再次访问时，我们不需要再次查询图。注意，在 NB-CNARW 中，每一步的随机游走选择都可能导致额外的样本查询，以便找到更好的下一跳，我们在评估 NB-CNARW 的查询开销时也包含了这部分的开销。

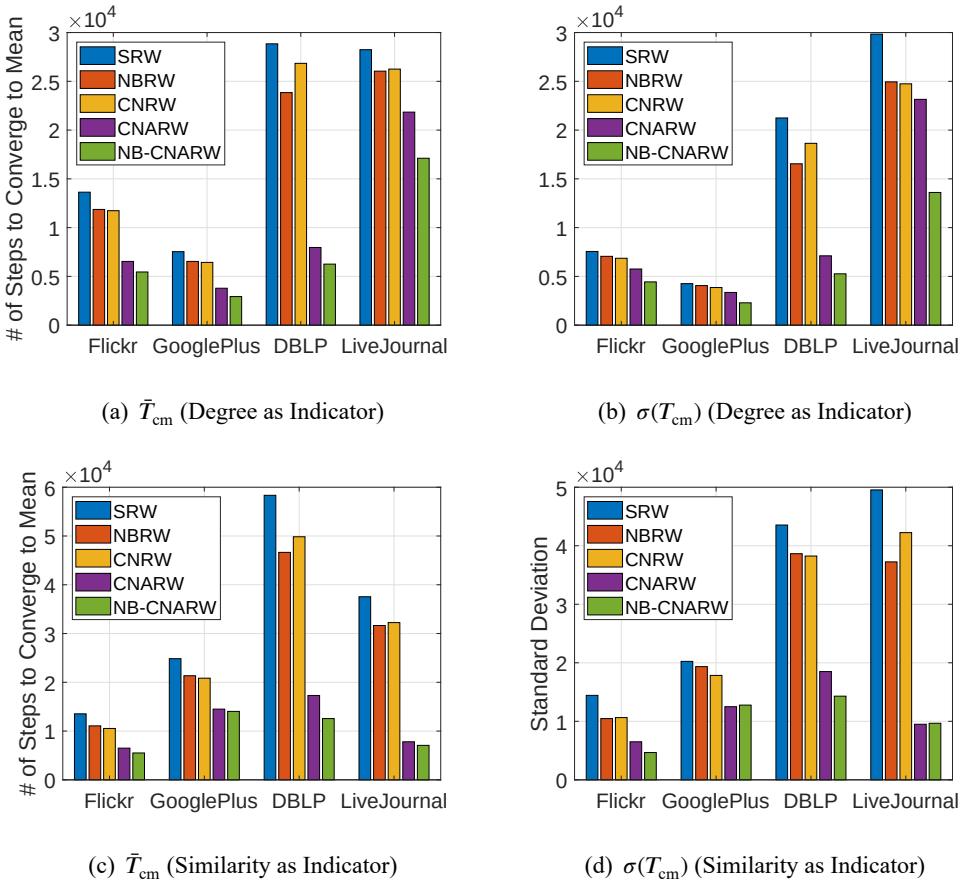
5.5.2　NB-CNARW 的收敛速度评估

（一）对比转移矩阵的第二大特征值模 SLEM

我们首先通过计算随机游走算法的转移矩阵的第二大特征值模（SLEM）来评估随机游走的收敛速度。由于 SLEM 的计算开销非常大，因此这里我们只考虑表 5.2中列出的三个小规模社交网络图。另外，由于 NBRW 算法和 CNRW 算

表 5.3　SRW 和 NB-CNARW 的 SLEM 值

Algorithms	Facebook	Ca-GaQc	Phy1
SRW	0.9923	0.9981	0.9981
NB-CNARW	0.9841	0.9818	0.9843



(a) $\bar{T}_{\mathrm{cm}}$ (Degree as Indicator)　(b) $\sigma(T_{\mathrm{cm}})$ (Degree as Indicator)

(c) $\bar{T}_{\mathrm{cm}}$ (Similarity as Indicator)　(d) $\sigma(T_{\mathrm{cm}})$ (Similarity as Indicator)

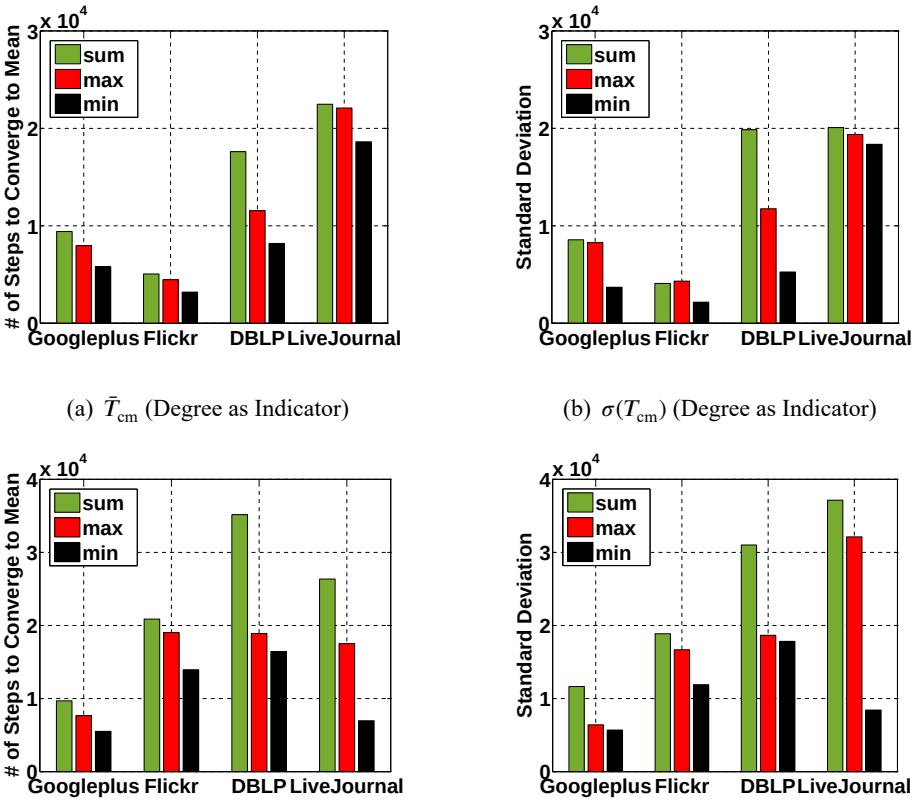
图 5.5　不同随机游走算法的收敛速度对比

注：这里的收敛速度通过多次统计的收敛到均值所需要的步数的平均值（$\bar{T}_{\mathrm{cm}}$）和方差（$\sigma(T_{\mathrm{cm}})$）来衡量。

法的封闭形式的转移矩阵很难推导，所以我们只比较了我们的 NB-CNARW 和 SRW。表 5.3中给出了 SRW 和 NB-CNARW 的 SLEM 的计算结果。我们看到我们的 NB-CNARW 对比 SRW 具有一个较小的 SLEM，这意味着 NB-CNARW 比 SRW 更快地收敛到平稳分布。

（二）对比评估收敛到稳态分布的随机游走步长

为了进一步评估 NB-CNARW 在大规模图上带来的的收敛速度的提升，我们通过统计某个特定估算指标的收敛到均值所需要的随机游走步长来衡量各个随机游走算法的收敛速度。具体来说，我们以平均顶点度和平均顶点对相似度作为估算指标。图 5.5(a) 和图 5.5(b) 分别表示以顶点度为估算指标时收敛到均值所需的最小步数的平均值 $\bar{T}_{\mathrm{cm}}$ 及其对应的标准差 $\sigma(T_{\mathrm{cm}})$。图 5.5(c) 和图 5.5(d) 分别展示了以顶点对相似性为估算指标时收敛到均值平均所需的最小步数 $\bar{T}_{\mathrm{cm}}$ 以及相应的标准差 $\sigma(T_{\mathrm{cm}})$。收敛到均值的评判收敛的方式是使用的 Jaccard 计算方法[97]。$\bar{T}_{\mathrm{cm}}$ 和 $\sigma(T_{\mathrm{cm}})$ 的每个值都是通过运行 300 次算法估算得到的。



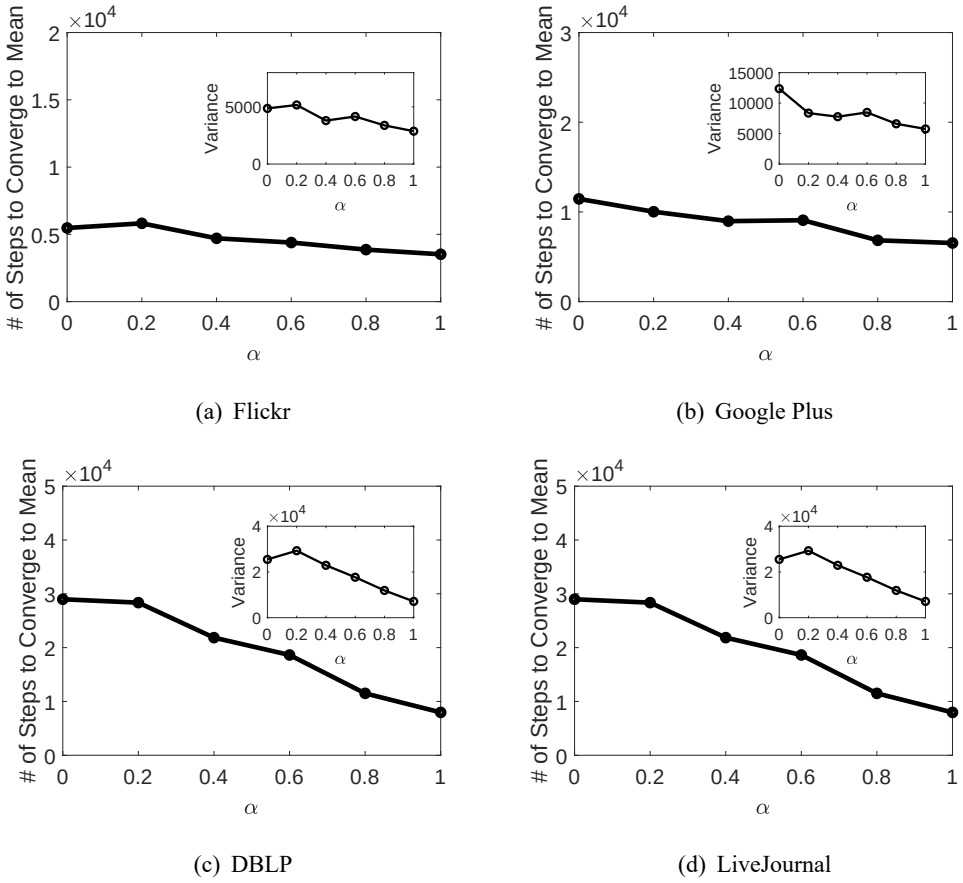
(a) $\bar{T}_{\mathrm{cm}}$ (Degree as Indicator)　(b) $\sigma(T_{\mathrm{cm}})$ (Degree as Indicator)

(c) $\bar{T}_{\mathrm{cm}}$ (Similarity as Indicator)　(d) $\sigma(T_{\mathrm{cm}})$ (Similarity as Indicator)

图 5.6　不同的状态转移矩阵设计下收敛速度

注：*sum*、*max* 和 *min* 分别表示在状态转移矩阵的设计公式 (5.6) 中使用 $deg(u)+deg(v)$、$max\{deg(u),deg(v)\}$ 和 $min\{deg(u),deg(v)\}$ 的情况。

从图中展示的实验结果我们观察到，对比于 SRW、NBRW、CNRW 和 CNARW，我们的 NB-CNARW 始终有更小的 $\bar{T}_{\mathrm{cm}}$ 值，也就是说 NB-CNARW 始终有更快的收敛速度。具体来说，NB-CNARW 需要更少的步骤来收敛到均值，例如，CNARW 最多可以减少高达 71.9% 的步数来达到收敛，而 NB-CNARW 可以在 CNARW 的基础上进一步减少高达 29.2% 的收敛步数。我们也注意到 $\bar{T}_{\mathrm{cm}}$ 因数据集而异，这意味着图结构对收敛速度有显著影响。例如，在 Flickr 数据集上收敛到均值所需的步数显著多于 LiveJournal 数据集上的结果。因此我们推导，当我们对一个规模更大图数据进行随机游走采样时，则需要更多的随机游走的步数来收敛到均值。这也意味着加速图采样中的随机游走的收敛过程是非常有意义的。另外，图 5.5也表明了，我们 NB-CNARW 在 $\bar{T}_{\mathrm{cm}}$ 上的标准差 $\sigma(T_{\mathrm{cm}})$ 也小于 SRW、NBRW、CNRW 和 CNARW。这就意味着使用我们的 NB-CNARW 带来的收敛速度更稳定。这一特性在实际系统中也非常重要，例如它可以使我们的 NB-CNARW 更适合于独立采样场景下的并行采样，因为多个随机游走的收敛时间差距小，木桶短板效应也就越小。



(a) Flickr　(b) Google Plus

(c) DBLP　(d) LiveJournal

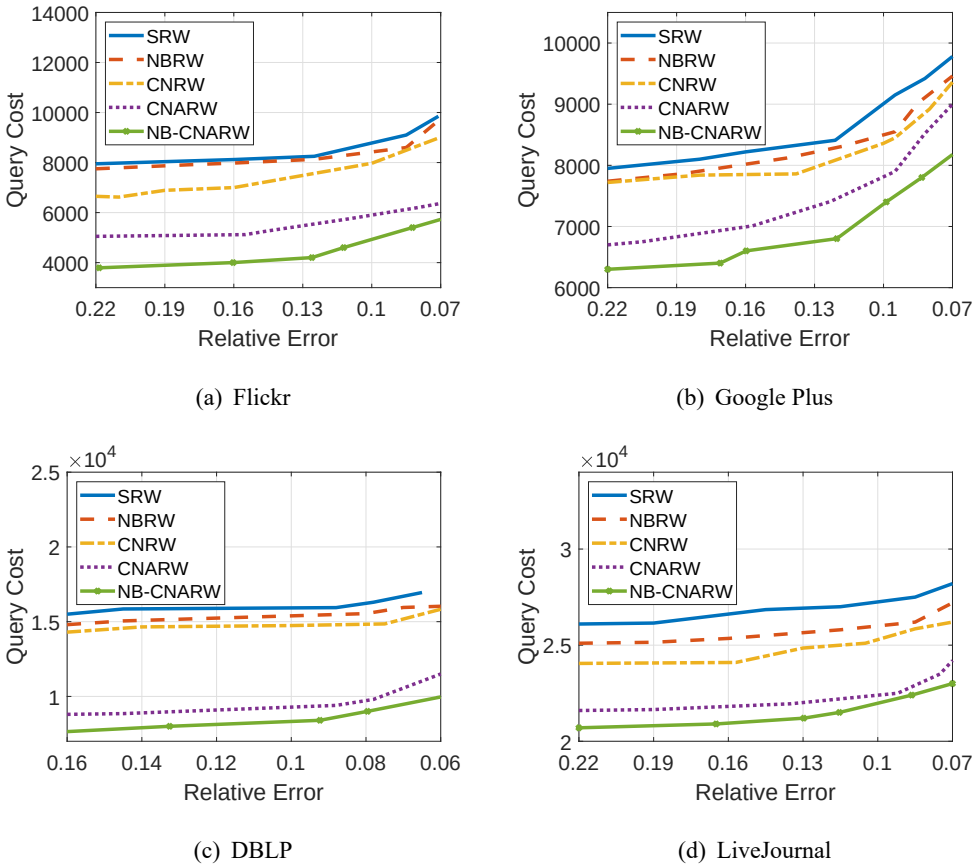
图 5.7　不同公共邻居权值 α 设置下的收敛速度

5.5.3　不同状态转移矩阵设计的影响

根据本章第 5.2.2节的分析，NB-CNARW 中设计转移概率的基本原理是试图使得图中的集合导通性最大化，因此我们设计随机游走的状态转移矩阵 $\boldsymbol{P}_{uv}$ 与 $1-\frac{C_{uv}}{min\{deg(u),deg(v)\}}$ 成正比。NB-CNARW 中状态转移矩阵 $\boldsymbol{P}_{uv}$ 的设计如式 (5.6) 所示。我们当然也可以灵活地考虑其他状态转移矩阵的设计，本小节中，我们评估了使用其他函数形式的状态转移矩阵的设计的性能，以证明 NB-CNARW 的效率。需要注意的是，所有这些转移矩阵的设计都必须满足对称特性，这样它们才能适用于相同的分析框架。

（一）度的使用形式的影响

我们首先研究使用 $deg(u)+deg(v)$ 和 $\max\{deg(u),deg(v)\}$ 来代替 NB-CNARW 的状态转移矩阵中的 $\min\{deg(u),deg(v)\}$。图 5.6显示了不同转移矩阵设计下的收敛速度。这里的收敛速度由收敛到均值的平均值和标准差（即 $\bar{\boldsymbol{T}}_{\mathrm{cm}}$ 和 $\sigma(\boldsymbol{T}_{\mathrm{cm}})$）来衡量。我们同样考虑使用顶点度和顶点对相似度作为策略指标。从结果中我们可以看到，在 NB-CNARW 算法中使用函数 min 总是有最好的性能。例如当使用顶点对相似性作为指标时，使用 *sum* 函数平均需要多 50% – 2.8× 的步



(a) Flickr　(b) Google Plus

(c) DBLP　(d) LiveJournal

图 5.8　不同随机游走采样在估算误差和查询开销之间的性能均衡

数来使得随机游走达到收敛。使用 min 函数性能更好的原因是它可以消除 $deg(u)$ 很大时的主导效应。也就是说，当 $deg(u)$ 非常大时，使用 min 函数仍然可以很好地区分 u 的不同邻居，因此，NB-CNARW 总是可以选择一个更好的顶点来作为随机游走的下一跳。

（二）公共邻居权重的影响

我们进一步评估在状态转移矩阵的设计中考虑公共邻居的权值的影响，即 $P_{uv} \propto 1 - \frac{alphaC_{uv}}{\min\{deg(u), deg(v)\}}$，其中 $0 \leqslant \alpha \leqslant 1$。当 α 减少时，共同邻居的权重也减少。特别的，$\alpha = 0$ 和 $\alpha = 1$ 分别表示 SRW 和 NB-CNARW 。图 5.7显示了各种公共邻居权值的设置下的收敛速度，结果表明公共邻居权值越大性能越好，即收敛到均值所需要的随机游走的步数越少。这进一步验证了在转换矩阵设计中引入共同邻居感知的积极影响。

5.5.4　基于 NB-CNARW 的图采样性能评估

接下来，我们研究基于不同的随机游走算法的图采样在估计误差和查询成本之间的权衡。我们展示了一个典型的采样任务（估算平均度）的结果。我们运行五种采样算法（包括 SRW、NBRW、CNRW、CNARW 以及我们的 NB-CNARW ）。

我们将每个算法重复 200 次来估算平均顶点度，然后取这 200 次结果的平均值来衡量查询成本。图 5.8显示了估计误差和查询开销之间的权衡，其中横坐标表示估算误差，纵轴表示相应的查询开销。从图中我们首先可以观察到，所有采样算法下的查询开销都随着所要求的相对误差的减少而增加，这意味着我们总是需要更多的查询成本来提高估算精度。此外我们也注意到，相比之下 CNARW 和 NB-CNARW 总是需要最小的查询成本来实现相同的估计精度，也可以说 CNARW 和 NB-CNARW 用相同的查询成本可以产生更准确的估计。具体来说，在相同的估算精度的要求下，CNARW 最多可以减少 35.7% 的查询开销，并且在 CNARW 的基础上，NB-CNARW 可以进一步降低查询成本（高达 25.4%）。

5.6　本章小结

本章工作首先通过基于随机游走的图采样问题的研究来探索随机游走如何解决实际的图计算问题。具体来说，我们针对基于随机游走的图采样的收敛速度慢严重影响采样效率的问题，提出一种非回溯的公共邻居感知的随机游走算法 NB-CNARW 。NB-CNARW 首先考虑到随机游走容易被困在某些子图中需要很多步才跳出来，从而拖慢随机游走访问整个全局图数据的进程，因而提出一种公共邻居感知的加权游走的方式来加速随机游走的收敛；此外，NB-CNARW 进一步结合 NBRW 中非回溯的思想，避免回到随机游走上一步刚刚经过的顶点，进一步加速随机游走的收敛过程。我们根据上述思想完成了 NB-CNARW 算法的设计，并提供了其随机性和稳态分布的分析。然后我们采用了一种基于拒绝采样的方式来实现 NB-CNARW 算法中随机游走每一步的加权游走，避免了网络环境下访问当前顶点的所有邻居顶点的网络查询开销。最后我们提出了基于 NB-CNARW 算法的随机游走采样的实现和分析，包括顶点采样和边采样。实验结果表明，NB-CNARW 算法对比于最新的随机游走算法可以显著加速随机游走的收敛；在利用随机游走算法实现图采样并进行某个测量指标的估算时，在保证相同估算精度的情况下，NB-CNARW 可以大大减少采样的查询成本。

第 6 章　总结与展望

6.1　本文的主要工作与成果

随着大数据时代数据规模的迅猛增长和数据之间产生日益复杂的交互关系，挖掘出数据和数据关联关系间蕴藏的丰富价值可以为我们的日常生活提供非常多的有效信息。图和图分析技术因能很好地表达真实世界中实体之间的关联关系和挖掘这些关联关系中的隐藏信息，而被广泛应用于众多服务场景。随机游走是一个重要的图分析工具，图上的随机游走利用其在图中各顶点间的集成路径提取信息。随机游走因其高效的计算效率和坚实的理论保障而经常被应用于一些重要的图分析、排序和嵌入式算法。本文聚焦于图上的随机游走类应用场景，从随机游走友好的图存储管理系统、随机游走的存储计算框架和快速收敛的随机游走算法优化这三个方面，对现有的随机游走算法的计算效率和支持随机游走类应用的图处理系统的存储管理和计算模型进行分析和相应的优化，从而提升大规模图分析系统在支持随机游走类应用时的性能。本文的主要研究内容和贡献如下。

（1）随机游走友好的图存储管理系统：现有的图处理系统往往采用一种基于迭代的模型，顺序迭代地从磁盘加载图数据块进入内存进行计算分析，从而缓解对磁盘的大量随机 I/O 造成的性能瓶颈。然而这种基于迭代的模型在支持基于随机游走类的图计算时表现出很低的 I/O 效率，从而限制了图上随机游走的效率和扩展性。本章结合随机游走应用的特征，提出一种基于随机游走状态感知的 I/O 模型，并设计实现了面向大规模并发随机游走的高效图数据存储管理机制 SAStore 。SAStore 根据图上随机游走的数量、步长以及随机游走在图中的分布情况等状态信息，来执行不同的图数据的组织划分、加载和缓存策略，从而实现 I/O 效率的最大化。此外针对动态图处理的场景下更新效率慢的问题，SAStore 进一步提出一种基于分块日志的 CSR 存储实现快速的图更新。基于上述设计实现的原型系统在真实世界的图数据集上的实验评估表明，SAStore 对比最新的单机随机游走图系统 DrunkardMob，平均可以提升 *2* 倍到 *4* 倍的 I/O 效率。另外在动态图场景下，我们的基于分块日志的 CSR 存储管理方案对比于基础的 CSR 格式，在保证图数据查询效率的同时，显著提升了动态图数据的更新效率。

（2）支持快速随机游走的图计算框架：现有的支持随机游走的图处理系统中对随机游走数据的存储管理往往采用以边或顶点为中心的随机游走状态数据的索引机制和基于大量的动态数组的存储方案。这样的随机游走的存储管理方式不仅带来非常大的索引数据的开销，而且大量的动态数组也会频繁地重新内存空间，带来内存碎片和额外的时间开销。因此限制了图处理系统可以处理的

图数据和并发随机游走的规模。另一个方面，现有工作中采用的基于迭代的同步计算模型，为了保证图上所有随机游走之间的同步更新而牺牲了随机游走的计算效率。针对上面两个方面的问题，我们首先提出以子图为中心的随机游走数据索引，并采用定长缓冲区来存储每个子图的所有随机游走数据，避免大量动态数组带来频繁内存重分配。然后我们提出异步的随机游走更新策略来加速随机游走更新速率。最后，在前面提出图存储管理系统 SAStore 之上，我们实现了上述的随机游走的存储管理和更新计算策略，最终实现一个支持快速随机游走的原型系统 GraphWalker，实验结果表明，GraphWalker 可以高效地处理由数十亿个顶点和数千亿条边组成的非常庞大的磁盘驻留图和并发运行的数百亿条、数千步长的随机游走。对比于最新的单机随机游走图系统 DrunkardMob，GraphWalker 的处理速度快一个数量级；对比于最新的分布式随机游走图系统 KnightKing，GraphWalker 在一台机器上的处理速度可以达到其 8 台机器上的速度；对比于支持随机游走性能最好的通用单机图处理系统 Graphene 和 GraFSoft，GraphWalker 的速度提升也非常明显，在最好的情况下，速度也快一个数量级。

（3）快速收敛的随机游走算法优化：我们研究一个典型的基于随机游走的应用场景，即图采样，探索随机游走如何解决实际的图计算问题以及随机游走类应用场景的数据访存特征。随机游走采样是当前主流的图采样方式，因为其计算高效、探索局部数据方便而且易于对采样结果进行理论分析。但基于随机游走的图采样需要在随机游走收敛到达稳态分布之后才能收集样本从而实现无偏估计。现有的随机游走算法的收敛速度都很慢，往往需要成千上万步随机游走才能收敛，严重影响了采样效率。而随机游走收敛慢的一个很重要的原因是他可能经常被困在某个局部的子图，需要很多步才能走出去，拖慢了随机游走对全局图数据的探索进程和收敛速度。基于上述分析，我们设计了相应的优化算法：基于公共邻居感知的随机游走算法 NB-CNARW，来加速随机游走的收敛从而提高随机游走采样的效率；实验结果表明，我们提出的 NB-CNARW 算法相较于当前最新的随机游走算法，可以显著提升随机游走的收敛速度，收敛所需要的步数最多可以减少 29.2%; 在使用 NB-CNARW 算法进行基于图采样的无偏估计时，在给定相同的采样预算的情况下，可以大幅度提升估算精度。

6.2　未来研究计划

本文基于随机游走在在图数据分析方面的广泛应用，详细剖析了现有随机游走算法的不足以及现有图处理系统在支持随机游走类应用时存在的局限性，并对分析结果进行了详尽的实验验证。本文在随机游走友好的图数据管理机制、随机游走数据的存储和计算方案和随机游走算法收敛加速这三个方面开展了研究

工作。虽然本文的研究工作已经实现了一个支持快速随机游走的大规模图分析系统，在一定程度上支持单机环境下超大规模图数据上的大量随机游走的并发执行。但是本文的研究内容依然存在一定的不足，可以进一步的扩展完善。本文的未来研究计划主要分为下面三个方面的内容。

（1）面向新型存储设备的存储管理优化拓展：近年来新兴的存储设备，包括当前流行的 NVMe SSD、NVRAM 等，提供了相当的顺序和随机访问，在访问延迟、寻址单位等访存性能特征上也完全不同于旧存储设备，这些差异使得之前的图存储系统中的一些优化策略不再高效，比如之前的工作主要关注如何使用大量计算资源将随机访问转换成顺序访问来减少 I/O 访问的延迟，这些优化策略在 NVMe SSD、NVRAM 等新型存储设备下可以获得的收益就大大减少。因此我们可以进一步研究面向新型存储设备的图数据和随机游走数据的存储优化，结合图数据和随机游走数据的访存特征和新型存储设备的性能特征，研究更适用的存储方案及访问模型，实现高效的数据存取。

（2）基于 GPU 的随机游走加速计算拓展：虽然本文提出的异步的随机游走的计算方式可以显著加速随机游走的更新效率，但是在巨量随机游走并发运行的场景下，比如在 CrawlWeb 数据集（35 亿顶点）上运行数百亿甚至数千亿条并发的随机游走时（比如执行 RWD 算法或为图中每个顶点执行 PPR 算法），随机游走的计算时间已然替代 I/O 开销成为整个程序运行的性能瓶颈。此时 CPU 的算力已经不太足够去支持这样巨量随机游走的高效并发执行，因此我们可以考虑引入 GPU 来完成部分的随机游走计算。考虑到 GPU 受限的存储空间以及主机内存-设备内存间受限的传输带宽，我们可以考虑针对不同应用程序以及程序的不同运行阶段的随机游走规模的差异，执行差异化的随机游走的计算调度。具体来说，我们可以考虑将重负载的计算任务传输到 GPU 上计算，轻负载的计算任务直接在 CPU 进行计算，通过 CPU 和 GPU 资源之间的自适应的动态配置来实现性能的最大化。

（3）多任务并发场景下的随机游走调度优化拓展：很多基于随机游走的图分析算法可能并发地从不同方面去分析同一个底层图数据，带来“基于随机游走的并发图分析”。根据一个真实大型社交网络平台的追踪结果，平均就有大约 10 个并发图处理任务提交到这一个公共数据平台，并发地分析同一个图数据，而在峰值的时候更是有超过 20 个并发图分析任务。在并发图分析场景下，各个并发图分析任务之间竞争 CPU、I/O 及内存等资源，并且相互干扰缓存，所以各并发图分析任务的性能都会因为相互影响而下降。我们的初步实验结果也展示了，将四个基于随机游走的图分析算法并发执行时，每个图分析任务的时间开销都会显著增加。文献 CGraph[98] 中指出，随着并发图分析任务的数量的进一步增加，各个并发的图分析任务的平均执行时间进一步显著延长，而他们延长的时间开

销主要来源于更高的数据访问成本。因此我们可以考虑多个基于随机游走的应用并发处理的场景，优化多任务并发图处理的性能。具体来说，我们考虑到多个并发的图分析任务中可能存在重复的随机游走任务，我们可以设计一种随机游走共享的机制，即对于同一类型的随机游走任务，检测它们之间随机游走的重复部分，设计随机游走任务的去重派发方案，实现同一类型的随机游走的复用，减少随机游走的存储和计算开销。

参 考 文 献

[1] PAGE L. The Pagerank Citation Ranking: Bring Order to the Web[J]. Technical report, 1998.

[2] JEH G, WIDOM J. Scaling Personalized Web Search[C]//WWW. ACM, 2003.

[3] BAR-YOSSEF Z, BRODER A Z, KUMAR R, et al. Sic Transit Gloria Telae: Towards an Understanding of The Web's Decay[C]//ACM WWW. 2004.

[4] CHEN W, WANG Y, YANG S. Efficient Influence Maximization in Social Networks[C]// ACM SIGKDD. 2009.

[5] KEMPE D, KLEINBERG J, TARDOS É. Maximizing the Spread of Influence Through a Social Network[C]//ACM SIGKDD. 2003.

[6] DEBNATH S, GANGULY N, MITRA P. Feature Weighting in Content Based Recommendation System Using Social Network Analysis[C]//ACM WWW. 2008.

[7] ANDERSEN R, BORGS C, CHAYES J, et al. Trust-based Recommendation Systems: an Axiomatic Approach[C]//ACM WWW. 2008.

[8] WEI H, YU J X, LU C, et al. Reachability Querying: An Independent Permutation Labeling Approach[J]. Proceedings of the VLDB Endowment, 2014, 7(12): 1191-1202.

[9] LOCHERT C, HARTENSTEIN H, TIAN J, et al. A Routing Strategy for Vehicular Ad Hoc Networks in City Environments[C]//Proceedings of the Intelligent vehicles symposium. IEEE, 2003.

[10] GONZALEZ H, HAN J, LI X, et al. Adaptive Fastest Path Computation on a Road Network: a Traffic Mining Approach[C]//VLDB. 2007.

[11] HONG S, CHAFI H, SEDLAR E, et al. Green-Marl: a DSL for Easy and Efficient Graph Analysis[C]//Proceedings of the ACM SIGARCH Computer Architecture News. 2012.

[12] NGUYEN D, LENHARTH A, PINGALI K. A Lightweight Infrastructure for Graph Analytics [C]//SOSP. ACM, 2013.

[13] HOTHO A, JÄSCHKE R, SCHMITZ C, et al. Folkrank: A Ranking Algorithm for Folksonomies[C]//Lwa. 2006.

[14] Common Crawl Graph[EB/OL]. 2012. http://webdatacommons.org.

[15] H. Tankovska. Number of monthly active facebook users worldwide as of 4th quarter 2020 [R]. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/, 2021.

[16] 廖小飞. 图计算的回顾与展望[J]. 中国计算机学会通讯, 2018, 14(7): 8-9.

[17] 开 源! 一 文 了 解 阿 里 一 站 式 图 计 算 平 台 GraphScope[R]. https://mp.weixin.qq.com/s/JvLQI0asXhjEfUJ4ls6fMg, 2020.

[18] MONDAL J, DESHPANDE A. Managing Large Dynamic Graphs Efficiently[C]//SIGMOD International Conference on Management of Data. ACM, 2012: 145-156.

[19] LANGVILLE A N M C D. Deeper inside Pagerank[C]//Internet Mathematics. 2004.

[20] JEH G, WIDOM J. SimRank: a Measure of Structural-context Similarity[C]//ACM SIGKDD. 2002.

[21] JAMALI M, ESTER M. Trustwalker: a Random Walk Model for Combining Trust-Based and Ttem-Based Recommendation[C]//ACM SIGKDD. 2009.

[22] PEROZZI B, AL-RFOU R, SKIENA S. Deepwalk:Online Learning of Social Representations [C]//SIGKDD international conference on Knowledge discovery and data mining. ACM, 2014: 701-710.

[23] GROVER A, LESKOVEC J. Node2vec: Scalable Feature Learning for Networks[C]// SIGKDD international conference on Knowledge discovery and data mining. ACM, 2016: 855‑-864.

[24] FOGARAS D, RÁCZ B, CSALOGÁNY K, et al. Towards Scaling Fully Personalized Pagerank: Algorithms, Lower Bounds, and Experiments[J]. Internet Mathematics, 2005, 2(3): 333-358.

[25] LI R H, YU J X, HUANG X, et al. Random-walk Domination in Large Graphs[C]// International Conference on Data Engineering (ICDE). IEEE, 2014.

[26] BAR-YOSSEF Z, BERG A, CHIEN S, et al. Approximating Aggregate Queries about Web Pages via Random Walks[C]//VLDB. 2000.

[27] HENZINGER M R, HEYDON A, MITZENMACHER M, et al. Measuring Index Quality using Random Walks on the Web[J]. Computer Networks, 1999, 31(11): 1291-1303.

[28] LEE C H, XU X, EUN D Y. Beyond Random Walk and Metropolis-hastings Samplers: Why You Should Not Backtrack for Unbiased Graph Sampling[C]//SIGMETRICS. 2012.

[29] RIBEIRO B, TOWSLEY D. Estimating and Sampling Graphs with Multidimensional Random Walks[C]//SIGCOMM. 2010.

[30] YANG K, ZHANG M, CHEN K, et al. KnightKing: A Fast Distributed Graph Random Walk Engine[C]//SOSP. ACM, 2019.

[31] ZHAO P, LI Y, XIE H, et al. Measuring and Maximizing Influence via Random Walk in Social Activity Networks[C]//International Conference on Database Systems for Advanced Applications. Springer, 2017: 323-338.

[32] WANG R, LV M, WU Z, et al. Fast Graph Centrality Computation via Sampling: a Case Study of Influence Maximisation over OSNs[J]. International Journal of High Performance Computing and Networking, 2019, 14(1): 92-101.

[33] KYROLA A. Drunkardmob: Billions of Random Walks on Just a PC[C]//ACM RecSys. 2013.

[34] RUSMEVICHIENTONG P, PENNOCK D M, LAWRENCE S, et al. Methods for Sampling Pages Uniformly from the World Wide Web[C]//Proceedings of the AAAI Fall Symposium on Using Uncertainty Within Computation. 2001.

[35] MALEWICZ G, AUSTERN M H, BIK A J, et al. Pregel: A System for Large-scale Graph Processing[C]//SIGMOD. ACM, 2010.

[36] LOW Y, BICKSON D, GONZALEZ J, et al. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud[J]. VLDB, 2012.

[37] GONZALEZ J E, LOW Y, GU H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.[C]//OSDI. USENIX, 2012.

[38] GONZALEZ J E, XIN R S, DAVE A, et al. GraphX: Graph Processing in a Distributed Dataflow Framework.[C]//OSDI. USENIX, 2014.

[39] CHEN R, SHI J, CHEN Y, et al. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs[C]//EuroSys. ACM, 2015.

[40] TEIXEIRA C H, FONSECA A J, SERAFINI M, et al. Arabesque: A System for Distributed Graph Mining[C]//SOSP. ACM, 2015.

[41] ZHU X, CHEN W, ZHENG W, et al. Gemini: A Computation-Centric Distributed Graph Processing System.[C]//OSDI. USENIX, 2016.

[42] CHEN H, LIU M, ZHAO Y, et al. G-Miner: an Efficient Task-Oriented Graph Mining System [C]//ACN EuroSys. 2018.

[43] KHAN A, SEGOVIA G, KOSSMANN D. On Smart Query Routing: for Distributed Graph Querying with Decoupled Storage[C]//ATC. USENIX, 2018.

[44] KYROLA A, BLELLOCH G E, GUESTRIN C. Graphchi: Large-scale Graph Computation on Just a PC[C]//OSDI. 2012.

[45] ROY A, MIHAILOVIC I, ZWAENEPOEL W. X-stream: Edge-centric Graph Processing Using Streaming Partitions[C]//SOSP. ACM, 2013.

[46] ZHU X, HAN W, CHEN W. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.[C]//USENIX ATC. 2015.

[47] VORA K, XU G H, GUPTA R. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing.[C]//USENIX ATC. 2016.

[48] LIU H, HUANG H H. Graphene: Fine-Grained IO Management for Graph Computing.[C]// FAST. USENIX, 2017.

[49] AI Z, ZHANG M, WU Y, et al. Squeezing Out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O[C]//USENIX ATC. 2017.

[50] VORA K. LUMOS: Dependency-Driven Disk-based Graph Processing[C]//ATC. USENIX, 2019.

[51] DA ZHENG D M, BURNS R, VOGELSTEIN J, et al. FlashGraph: Processing Billion-node Graphs on an Array of Commodity SSDs[C]//USENIX FAST. 2015.

[52] MAASS S, MIN C, KASHYAP S, et al. Mosaic: Processing a Trillion-edge Graph on a Single Machine[C]//EuroSys. ACM, 2017.

[53] JUN S W, WRIGHT A, ZHANG S, et al. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics[C]//ISCA. IEEE, 2018.

[54] ELYASI N, CHOI C, SIVASUBRAMANIAM A. Large-Scale Graph Processing on Emerging Storage Devices[C]//FAST. USENIX, 2019.

[55] SHUN J, BLELLOCH G E. Ligra: a Lightweight Graph Processing Framework for Shared Memory[C]//ACM SIGPLAN. 2013.

[56] KHORASANI F, VORA K, GUPTA R, et al. CuSha: vertex-centric graph processing on GPUs[C]//Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. ACM, 2014.

[57] LIU H, HUANG H H. Enterprise: Breadth-first graph traversal on GPUs[C]//Proceedings of the SC-International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2015.

[58] WANG Y, DAVIDSON A, PAN Y, et al. Gunrock: A high-performance graph processing library on the GPU[C]//ACM SIGPLAN. 2016.

[59] Ma L, Yang Z, Chen H, et al. Garaph: Efficient GPU-Accelerated Graph Processing on a Single Machine with Balanced Replication[C]//Conference on Usenix Annual Technical Conference. USENIX, 2017: 195-207.

[60] Zheng L, Li X, Zheng Y, et al. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling[C]//Annual Technical Conference. USENIX, 2020: 573-588.

[61] Dai G, Chi Y, Wang Y, et al. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search[C]//International Symposium on Field-Programmable Gate Arrays. 2016: 105-110.

[62] Friendster Dataset[EB/OL]. http://konect.uni-koblenz.de/networks/friendster.

[63] Neo4j graph database[R]. https://neo4j.com/, 2021.

[64] Arangodb graph database[R]. https://www.arangodb.com/, 2021.

[65] Db-engines ranking[R]. https://db-engines.com/en/ranking, 2021.

[66] Yahoo Webscope Program[EB/OL]. http://webscope.sandbox.yahoo.com.

[67] J L, J K, C F. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations[C]//Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. 2005: 177-187.

[68] HENZINGER M R, HEYDON A, MITZENMACHER M, et al. On Near-uniform URL Sam-

pling[J]. Computer Networks, 2000, 33(1): 295-308.

[69] LESKOVEC J, FALOUTSOS C. Sampling from large graphs[C]//SIGKDD, 2006. 2006.

[70] HASTINGS W K. Monte carlo sampling methods using markov chains and their applications [J]. Biometrika, 1970, 57(1): 97-109.

[71] LI R H, YU J X, QIN L, et al. On random walk based graph sampling[C]//International Conference on Data Engineering (ICDE). IEEE, 2015.

[72] ZHOU Z, ZHANG N, GONG Z, et al. Faster random walks by rewiring online social networks on-the-fly[C]//International Conference on Data Engineering (ICDE). IEEE, 2013.

[73] PAPAGELIS M. Refining social graph connectivity via shortcut edge addition[J]. TKDD, 2015, 10(2): 12.

[74] MOHAISEN A, HOLLENBECK S. Improving Social Network-based Sybil Defenses by Rewiring and Augmenting Social Graphs[C]//International Workshop on Information Security Applications. Springer, 2013: 65-80.

[75] ZHAO J, LUI J, TOWSLEY D, et al. A tale of three graphs: Sampling design on hybrid social-affiliation networks[C]//IEEE International Conference on Data Engineering (ICDE). 2015.

[76] ZHOU Z, ZHANG N, DAS G. Leveraging history for faster sampling of online social networks [J]. VLDB, 2015, 8(10): 1034-1045.

[77] LI Y, WU Z, LIN S, et al. Walking with Perception: Efficient Random Walk Sampling via Common Neighbor Awareness.[C]//35th IEEE International Conference on Data Engineering (ICDE). IEEE, 2019.

[78] 吴志勇. 基于随机游走的在线社交网络采样加速研究[M]. 2019.

[79] Zhu X, Feng G, Serafini M, et al. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans[C]//Proceedings of the VLDB Endowment: volume 13. 2020: 1020-1034.

[80] Twitter Dataset[EB/OL]. 2010. http://an.kaist.ac.kr/traces/WWW2010.html.

[81] Graph500[EB/OL]. https://graph500.org/.

[82] PRŽULJ N, CORNEIL D G, JURISICA I. Modeling Interactome: Scale-Free or Geometric? [J]. Bioinformatics, 2004, 20(18): 3508-3515.

[83] PRŽULJ N. Biological Network Comparison Using Graphlet Degree Distribution[J]. Bioinformatics, 2007, 23(2): e177-e183.

[84] TONG H, FALOUTSOS C, PAN J Y. Fast Random Walk with Restart and Its Applications [C]//ICDM. IEEE, 2006.

[85] HAVELIWALA T H. Topic-Sensitive Pagerank[C]//WWW. ACM, 2002.

[86] PAN J Y, YANG H J, FALOUTSOS C, et al. Automatic Multimedia Cross-modal Correlation

Discovery[C]//ACM SIGKDD. 2004.

[87] HÄGGSTRÖM O. Finite markov chains and algorithmic applications: volume 52[M]. Cambridge University Press, 2002.

[88] SIGMAN K. Time-reversible Markov Chains[EB/OL]. 2009. http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-Time-Reversibility.pdf.

[89] LESKOVEC J, KREVL A. SNAP Datasets: Stanford large network dataset collection [EB/OL]. 2014. http://snap.stanford.edu/data.

[90] ROSSI R A, AHMED N K. The network data repository with interactive graph analytics and visualization[C]//AAAI. 2015.

[91] NAZI A, ZHOU Z, THIRUMURUGANATHAN S, et al. Walk, not wait: Faster sampling over online social networks[J]. VLDB, 2015.

[92] CHEN X, LI Y, WANG P, et al. A general framework for estimating graphlet statistics via random walk[J]. VLDB, 2016, 10(3): 253-264.

[93] MOHAISEN A, LUO P, LI Y, et al. Measuring bias in the mixing time of social graphs due to graph sampling[C]//MILCOM, 2012. 2012.

[94] LOVáSZ L. Random Walks on Graphs: A Survey[M]//Combinatorics. 1993.

[95] COWLES M K, CARLIN B P. Markov chain monte carlo convergence diagnostics: a comparative review[J]. JASA, 1996, 91(434): 883-904.

[96] GEWEKE J, et al. Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments: volume 196[M]. Federal Reserve Bank of Minneapolis, Research Department Minneapolis, 1991.

[97] CHOWDHURY G G. Introduction to modern information retrieval[M]. Facet publishing, 2010.

[98] ZHANG Y, LIAO X, JIN H, et al. CGraph: A Correlations-Aware Approach for Efficient Concurrent Iterative Graph Processing[C]//ATC. USENIX, 2018.

致　谢

仿佛只是转眼一瞬，在科大五年的博士研究生的求学时光就已经这样悄然流逝，读博的这五年可以说是我人生中最埋头奋进的一段时光。在感念韶光易逝的同时，我想在此毕业论文即将完成之际，向在此期间陪伴着我并给予我帮助的老师、同学和家人朋友们表示我最诚挚的感谢。

首先我要感谢我的导师许胤龙教授，每当我在科研上遇到阻碍或困惑时，许老师都会跟我探讨我遇到的问题，并悉心地帮我分析、耐心地给我指导；每当我要准备论文投稿时，许老师会详细地跟我探讨文章的组织思路，并一次次帮我修改论文，甚至经常占用自己的休息时间帮我改论文改到深夜；每当我要做学术报告时，许老师也会很认真地指导我如何准备学术报告的幻灯片、如何注意作报告时的措辞语气、如何做一个让别人容易听懂的学术汇报等等。许老师自己在教学和科研方面深厚的学术功底、严谨的治学态度和敏锐的学术洞察力也一直感染着我，是我一生受用的学习榜样。许老师的言传身教指引着我在科研的道路上一步步的前进，非常感谢许老师！

其次我要感谢的是李永坤老师，李老师是我科研路上的引路人。从我本科阶段的毕业设计开始，李老师就一步步悉心引导着我，从如何慢慢熟悉图计算和系统这些研究方向开始；到如何完成从本科生阶段的接收知识到研究生阶段探索发现知识的这种思维方式的转变；以及如何从一个“high level”层面去看待和表达自己的研究工作；再到如何深入去理解系统中各个细节的原理和优化设计。从我博士毕业论文课题的确定、研究工作的展开、实验的运行、进展的汇报、论文的投稿一直到最终毕业论文的撰写，李老师一直在给予我紧密的指导、帮助和鼓励，这五年来的受李老师指导的科研经历，带给我的不仅仅是论文的成果，更是丰富了我的知识储备、扎实了我的系统功底，也培养了我的科研和工作能力，真心的感谢李老师！

然后，我也要感谢我们 ADSL 课题组的吕自成老师、李诚老师、吕敏老师和吴思老师，各位老师也都一直会在适当的时候给予我教导、关怀和帮助。吕自成老师总是会在我科研遇到困惑时与我深入探讨我的研究课题，并给予我相当宝贵的意见；李诚老师总是积极向上、充满朝气，感染着我对科研和生活的追求与热爱，另外李老师对国内外最新科研成果理解的广度和深度也都深深折服着我。吕敏老师也时常跟我讨论学术问题，打开我看待问题的新思路，给我带来一些新的启发。吴思老师是我进入我们课题组后认识的第一个师兄，亦师亦友的思哥在工作之余也经常跟我讨论我们的学习生活以及未来发展问题，为我的未来方向提供了很好的借鉴和指引。很庆幸能够进入 ADSL 课题组度过我这五年的博士

求学生涯，课题组的各位老师在看待问题时从不同角度的理解和剖析，也让我时常有一些新的思路和启发。非常感谢课题组的六位老师在科研和生活上给予我的指导和关怀，谢谢各位老师！

同时我也要感谢我们 ADSL 这个大家庭一起科研学习的各位小伙伴。感谢赵鹏鹏师兄和吴志勇师兄对我科研上的引导和帮助，帮我快速熟悉和了解图计算这个研究方向，耐心地为我答疑解惑。感谢李志鹏、张伟韬、陈友旭和郭帆师兄们对我科研生活的指导和关心。感谢与我同期入学的几位小伙伴白有辉、田成锦、陈浩、陈吉强、刘彩银和刘军明，我们一起经过保研入学的期待憧憬、一起经受科研投稿的锤炼敲打、一起经历毕业求职的投递抉择，我们相互鼓励，一同奋进，感谢我们无悔的科大梦！感谢林帅在科研中给予我的帮助，我们一起探讨的问题、一起改过的论文都是我们一起努力的印证。最后感谢实验室中一起科研学习的同伴们，他们是巨高莹、孙东东、张月明、赵倩倩、杨陈、张强、徐亮亮、邵新洋、苏景波、李佳伟、吴加禹、光煦灿、姚路路、朱文喆、左泽、王一多、许冠斌、周泉、游翎璟、余东波、刘朕、董健、王千里、李启亮、汪威、杨振宇、王孟、龚正、金泽文、阮超逸、李嘉豪、火净泽、邓龙、王霄阳、卢明祥、来逸瑞。谢谢大家在科研路上的陪伴，祝愿你们接下来能收获好的成果！

另外我也要特别感谢在此期间陪伴鼓励我的家人和朋友。感谢爸爸妈妈对我近二十年求学生涯的付出、支持和鼓励，没有你们的抚育和培养就没有如今独立坚强的我，家人永远是我最温暖的港湾。感谢我的男朋友白有辉，我们一起入学、一起科研、然后一起毕业，一路走来的欢喜陪伴、支持鼓励和包容理解是我们对爱情最美好的诠释，我们一起加油，期待与你一起走下去的余生。还要感谢我的室友李亚同学，虽然我们研究的专业方向都不同，但相同的是我们的漫漫读博路，我们一起努力奋进、相互鼓励；一起笑笑闹闹、畅谈到凌晨；一起探寻美食，然后减肥健身；谢谢你这几年来对我的倾听、开解和陪伴！

最后，感谢在百忙中参与的我论文评审和论文答辩的专家与评委们，谢谢你们用专业的知识给予论文宝贵的修改意见，谢谢你们！再次由衷地感谢所有帮助关心过我的老师、同学、家人和朋友们！

汪睿

二 O 二一年四月

在读期间发表的学术论文与取得的研究成果

参与的科研项目

1. 国家自然科学基金面上项目“键值存储系统架构设计与性能优化研究”，项目号：61772484。
2. 科技部重点研发计划“互联网基础行为巨量测量数据高效管理和共享系统研究”，项目号：2018YFB1800203。
3. 中国科学技术大青年创新重点项目“高性能数据存储与资源管理系统研究”，项目号：YD2150002003。

已发表论文

1. **Rui Wang,** Yongkun Li*, Shuai Lin, Hong Xie, Yinlong Xu, John C.S. Lui.”On Modeling Influence Maximization in Social Activity Networks under General Settings.”Accepted for publication in ACM Transactions on Knowledge Discovery from Data (ACM TKDD)，Accepted
2. **Rui Wang,** Yongkun Li*, Hong Xie, Yinlong Xu, John C.S. Lui.”GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks.”Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 2020), Boston, MA, USA, July 2020.(AR: 65/348 = 18.6%)
3. **Rui Wang,** Min Lv*, Zhiyong Wu, Yongkun Li, Yinlong Xu. ”Fast Graph Centrality Computation via Sampling:A Case Study of Influence Maximisation over OSNs.” IJHPCN, 2019 Vol.14 No.1, pp.92 –101
4. **Rui Wang,** Zhiyong Wu, Yongkun Li, and Yinlong Xu”Fast Graph Centrality Computation via Sampling: A Case Study of Influence Maximization over OSNs.”CCF BigData, Lanzhou, China, October 11-13, 2016.

待发表论文

1. **Rui Wang,** Yongkun Li*, Shuai Lin, Weijie Wu, Hong Xie, Yinlong Xu, John C.S. Lui.”Your Neighbors Matter: Efficient Random Walk Sampling with Common Neighbor Awareness”, Submitted to IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), Under Review.

2. **Rui Wang,** Yongkun Li*, Hong Xie, Yinlong Xu, John C.S. Lui.”State-Aware Data Management to Support Massive Random Walks on Large-Scale Graphs”, To be submitted.