分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题　目：基于大规模图的随机游走技术研究

学　　号　M202273732

姓　　名　田志鹏

专　　业　计算机技术

课程指导教师　施展 童薇 胡燏翀

院（系、所）　计算机学院

2023 年 1 月 9 日

目录

基于大规模图的随机游走技术研究 I
Research on random wandering technique based on large-scale graph I
1.引言 5
1.1.1 随机游走描述 5
1.1.2 基于随机游走的算法 5
1.1.3 随机游走中的采样方法 6
1.1.4 国内外研究现状总结 7
2. 相关研究 7
2.1 GraphWalker 7
2.1.1 介绍 7
2.1.2 研究背景 7
2.1.3 GraphWalker 设计 8
2.1.4 实验部分 9
2.1.5 总结 10
2.2 ThunderRW 10
2.2.1 介绍 10
2.2.2 研究背景 10
2.2.3 ThunderRW 设计 11
2.2.4 实验部分 12
2.1.5 总结 13
2.3 KnightKing 13
2.3.1 介绍 13
2.3.2 研究背景 13
2.3.3 KnightKing 设计 14
2.2.4 实验部分 15
2.1.5 总结 15
2.4 FlashMob 16
2.4.1 介绍 16
2.4.2 研究背景 16
2.4.3 FlashMob 设计 17
2.4.4 实验部分 18
2.4.5 总结 19
2.5 GraSorw 19
2.5.1 介绍 19
2.5.2 研究背景 20
2.5.3 GraSorw 设计 20
2.5.4 实验部分 22
2.5.5 总结 23
3. 总结 23
4. 参考文献 23

基于大规模图的随机游走技术研究

田志鹏 [1)]

[1)](华中科技大学计算机学院 湖北 武汉 430074)

摘 要 随机游走是一种基本且应用广泛的图处理任务。它是一种从图实体间的集合路径中提取实体间信息的强大的数学工具，是许多重要的图度量、排序和嵌入算法的基础。针对近年来对大型图上的随机游走引擎的研究列出了五篇相关文献。GraphWalker 是一种用于随机行走的 I/O 高效图形系统，通过部署一种具有异步行走更新的新型状态感知 I/O 模型，可以有效地处理由数千亿条边组成的非常大的圆盘图，而且它还可以扩展到运行数百亿个具有数千步长的随机行走。ThunderRW 是一个通用且高效的内存 RW 框架。通过采用了一个以步长为中心的编程模型，从步行者移动一步的局部视图中抽象出计算，同时还提出了步进交错技术来解决缓存失速问题。KnightKing 是一个通用的分布式图随机漫步引擎，采用了直观的以 walker 为中心的计算模型，围绕创新的基于拒绝的抽样机制，极大地降低了高阶随机漫步算法的成本。FlashMob 通过使内存访问更加有序和规则，提高了缓存和内存带宽的利用率。GraSorw 是一种 I/O 高效的基于磁盘的大型图随机游走系统。通过开发了一个双块执行引擎与基于学习的块加载模型，与现有的基于磁盘的图形系统相比提升不少性能。

关键词 随机游走；图处理系统；I/O 优化；

Research on random wandering technique based on large-scale graph

Zhipeng Tian[1)]

[1)](Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract Random walk is a fundamental and widely used graph processing task. It is a powerful mathematical tool for extracting inter-entity information from set paths between graph entities and is the basis for many important graph metrics, ranking and embedding algorithms. Five related papers are listed for recent research on random walk engines on large graphs.GraphWalker is an I/O efficient graph system for random walks that can efficiently handle very large disc graphs consisting of hundreds of billions of edges by deploying a novel state-aware I/O model with asynchronous walk updates, and it can scale to run tens of billions of random walks with thousands of steps. ThunderRW is a general and efficient in-memory RW framework. By employing a step-centric programming model that abstracts computation from a local view of a walker moving one step, it also proposes step interleaving techniques to solve the cache stall problem. KnightKing is a general-purpose distributed graph random walk engine that employs an intuitive walker-centric computation model, based around an innovative rejection-based sampling mechanism that dramatically reduces the cost of higher-order random walk algorithms. FlashMob improves cache and memory bandwidth utilization by making memory accesses more ordered and regular.GraSorw is an I/O efficient disk-based large graph random walk system. By developing a dual block execution engine with a learning-based block loading model, it improves performance considerably compared to existing disk-based graph systems.

Key words Random walk; Graph processing systems; I/O optimization;

1.引言

社会和信息技术的迅速发展，不但带来了数据的大规模增长，数据之间还形成了一种错综复杂的关系。海量数据背后蕴含着丰富的价值，并且数据之间的相互依赖关系同样可以挖掘出大量隐含的有效信息。图作为一种重要的数据结构，是用于建模实体之间复杂关联关系的经典工具，被广泛应用于各个领域。在传统的应用中，使用图结构来解决路径预测、地图搜索、最优运输路线等问题；新兴的应用中，在生物医学领域中进行蛋白质分子结构分析，在社交网络中进行社区发现、社交数据分析与挖掘，在知识图谱领域中进行语义网络分析、知识图谱分析，在计算神经科学中分析大脑区域之间的相互作用。随着社会信息化程度的提高和大数据时代的到来，图数据规模和复杂性也在不断增长。2022 年 Google 收录的 Web 图已经超过了 565 亿个页面和万亿条链接 1。根据 Facebook 于 2021 年第三季度发布的报告显示，旗下社交网络 Facebook 月活跃用户数已经超过了 29 亿，用户之间的好友关系链已经突破万亿条。

面对这样大规模的图数据，基于迭代的图遍历算法面临计算复杂度过高和空间开销过大的问题而难以实现，基于随机游走的方法除了具备强大的图算法表达能力，同时为算法的准确性和高效性提供了保证。随机游走作为许多科学应用中的重要组成部分，可以从实体之间的集成路径中提取隐含的重要价值信息，因而被广泛应用于大规模图数据分析，以及各种动态模拟中。近年来，随机游走的发展如火如荼，在学术界和工业界受到越来越多的关注，根据 Google 学术在 2021 年的统计结果，这一年中就有将近 40 万篇研究工作围绕随机游走开展。

总而言之，随机游走是一种提取实体之间关系的有力工具，具有很高的研究价值。

1.1.1 随机游走描述

基于图的随机游走可以如图 1 所示。总的来说基于随机行走的算法将一个图 G 作为输入，同时开始多条 walker。它们从一个特定的顶点开始行走，然后它们独立地在图中漫游。在每一步中，行走器从其当前驻留顶点的外向边中采样一条边，跟踪它到下一挑。当达到预设的路径长度或满足预设的异常条件时，每条 walker 以预设的终止概率退出。输出可以通过随机行走过程中嵌入的计算来生成，也可以通过转储结果的随机行走路径来生成。这个过程可以重复多个回合。经过该过程我们得到了一系列的随机游走序列，该序列可以用于后序下游的工作任务。

从直观上看，随机行走的计算量主要集中在边缘采样过程中，这也体现了各个随机行走算法之间的差异。更具体地说，随机游走算法定义了它自己的边缘转移概率。随着随机行走变得越来越流行，采样逻辑也变得更加复杂，最近的算

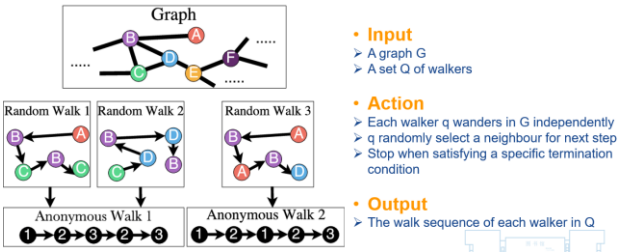


图 1 随机游走算法描述

法执行动态采样，其中边缘转移概率取决于行走者的当前状态，它访问的前一个顶点(或多个顶点)，以及它当前驻留顶点的边的属性。因此，复杂的随机行走算法实现了更灵活的、特定于应用程序的行走，但代价是采样的复杂性。

1.1.2 基于随机游走的算法

总的来说，不同的 RW（随机游走）算法都遵从算法 1 的流程，它们的主要区别在于下一跳邻居节点的选择。我们可以根据转移概率将其分为两种类型，有偏和无偏。对于无偏的算法，采样邻居时选择概率都是相同的。而有偏差 RW 的转移概率是不均匀的，比如可以取决于边权值。我们进一步将有偏差的 RW 分为静态和动态两类。如果在执行前确定了转移概率，则 RW 是静态的。否则是动态的，受 RW 查询状态的影响。下面，我们将介绍四种有代表性的 RW 算法，它们已经在许多应用中得到了应用。

Algorithm 1: Execution Paradigm of RW algorithms

Input: a graph G and a set $\mathbb{Q}$ of random walk queries;
Output: the walk sequences of each query in $\mathbb{Q}$;
1 **foreach** $Q \in \mathbb{Q}$ **do**
2 **do**
3 Select a neighbor of the current residing vertex $Q.cur$ at random;
4 Add the selected vertex to Q;
5 **while** *Terminate(Q) is false*;
6 **return** $\mathbb{Q}$;

算法 1 随机游走算法

（1） PPR (Personalized PageRank)

PageRank 算法的一个更复杂的版本是个性化 PageRank (PPR)。与通常使用幂迭代计算的一般 PageRank 问题不同，PPR 需要大量的时间或空间成本才能有效计算，特别是对于大型图而言。这是一种有偏的静态算法，可用来模拟网页浏览中的个人偏好。

在每一步中，当前驻留顶点的外向边被采样的概率与它的权重成正比。更具体地说，$P_s(e) = f(v,x)$，其中 e 连接当前驻留的顶点 v 到顶点 x,也就是对应的边。为了获得更好的性能和并行性，通过让 walker 采用固定的终止概率P_t，将长时间的随机行走分解为许多短时间的行走。

（2） DeepWalk

DeepWalk 是另一个有偏差的静态随机行走算法的例子，这是一种广泛用于机器学习的重要图嵌入技术。它利用语言建模技术进行图分析，使用截断随机游走生成许多游走序列。通过将每个顶点视为一个单词，将每个序列视为一个句子，然后应用 SkipGram 语言模型来学习这些顶点的潜在表示。

这种表示在很多应用中都很有用，比如多标签分类、链路预测和异常检测。虽然最初的 DeepWalk 是无偏的，但后来的工作将其扩展到有偏随机漫步。

与 PPR 类似，转移概率也与边的权值成正比。主要的区别在于终止情况不同，PPR 可以随机“提前终止”，DeepWalk 会一直持续到给定的路径长度才会终止。

（3） Meta-path

Meta-path 是一种基于元路径的算法，该算法旨在捕获顶点和边的异质性背后的语义。在这些算法中，每个 walker 都与指定行走路径中边缘类型模式的元路径方案相关联。例如，在一个出版物图中，为了探测引用关系，我们可以将起始顶点设置为作者，并将元路径方案设置为“isAuthor→citedBy→authorredby^{-1}”。可以通过重复这样的模板来创建更长的行走，例如，在这种情况下，通过交替的边表示“引用”(作者到论文)和“作者”(论文到作者)关系来生成较长的引用链。

实际使用的元路径是特定于应用程序的，通常由领域专家定义。具体来说，元路径执行将从用户提供的 N 个元路径方案中随机为每个 walker 分配一个。对于给定元路径方案 S 的 walker，在第 k 步的转移概率如下公式所示：

$$P_d(e) = \begin{cases} 1, if\ type(e) = S_{k\ mod|S|} \\ 0, otherwise \end{cases}$$

这给出了一个动态的一阶随机游走算法的例子。在同一顶点，对于不同方案的 walker 或不同步数的 walker，其边缘转移概率分布是不同的，不能在执行开始时预先计算。同时，下一条边的选择仍然是一阶的，因为它只涉及分配方案上的当前位置，而不考虑之前访问过哪些顶点。

（4） Node2vec

Node2vec 是一个高阶算法，这样的算法非常强大，因为步行者会在选择下一站时记录他们最近的步行历史，这反映了许多使用场景中的现实。到目前为止，我们在实际应用中看到的绝大多数高阶算法都是二阶的。Node2vec 应用与 DeepWalk 类似，但更灵活和表达能力更强。

在给定的无向图上，对于连接 v 和顶点 x 的边 e, 对于一跳游走 w，它能记得它的上一跳顶点 t。在其当前驻留顶点 v 处具有以下动态边缘转移概率:

$$P_d(e,v,w) = \begin{cases} \frac{1}{p}, if\ d_{tx} = 0 \\ 1, if\ d_{tx} = 1 \\ \frac{1}{q}, if\ d_{tx} = 2 \end{cases}$$

这里 p 和 q 是用户配置的超参数，d_{tx}是 t 和 x 之间的距离。d_{tx}= 0 表示 t 和 x 是同一个顶点。因此，p 被称为返回参数，给出在行走中立即重新访问节点的可能性。d_{tx}=1 表示 x 与 t 相邻否则为d_{tx}= 2。其中 q 被称为输入-输出参数，其中较高的设置生成的行走倾向于获得关于开始顶点和近似 BFS 行为的底层图的“局部视图”。另一方面，较低的设置更倾向于探索“更远”的节点，类似于 DFS 行为。

1.1.3 随机游走中的采样方法

随机游走中的采样方法指的是根据当前的概率分布去选择下一跳顶点。在这里重点介绍了四种抽样技术，包括朴素抽样、逆变换抽样、偏置采样和拒绝采样。

（1） 朴素采样

朴素重采样，对数据没有任何假设，也没有使用启发式方法。所以，易于实现且执行速度快，这对于非常大和复杂的数据集来说是高效的。

（2） 逆变换采样

假设 X 为一个连续随机变量，其累积分布函数为 Fx。此时，随机变量服从区间[0,1]上的均匀分布。逆变换采样即是将该过程反过来进行：首先对于随机变量 Y，我们从 0 至 1 中随机均匀抽取一个数 u。之后，由于随机变量与 X 有着相同的分布，即可看作是从分布中生成的随机样本。

（3） 偏置采样

等概率抽样和二项抽样（对二项分布的事件进行抽样）的时间复杂度均为 O(1)。Alias 算法即将等概率抽样和二项抽样结合，实现不等概率事件下 O(1)时间复杂度的抽样。

Alias 算法的流程如下。首先对每个事件发生的概率乘以事件的数目。接着将所有事件拼成等概率的分布，每一列的最大值为 1，而且每一列最多有两个事件。整体来看，即等概率分布与二项分布的组合：列与列之间的等概率分布，而列内为二项分布。最后依据等概率采样随机采取其中一列，同时依据该列所在的二项分布采样事件。

（4） 拒绝采样

蒙特·卡罗方法(Monte Carlo method)也称统计模拟方法，通过重复随机采样模拟对象的概率与统计的问题，在物理、化学、经济学和信息技术领域均具有广泛应用。拒绝采样(reject sampling)就是针对复杂问题的一种随机采样方法。

首先举一个简单的例子介绍 Monte Carlo 方法的思想。假设要估计圆周率 π 的值，选取一个边长为 1 的正方形，在

正方形内作一个内切圆，那么我们可以计算得出，圆的面积与正方形面积之比为π/4。

简单分布的采样，如均匀分布、高斯分布、Gamma 分布等，在计算机中都已经实现，但是对于复杂问题的采样，就需要采取一些策略，拒绝采样就是一种基本的采样策略，其采样过程如下。

给定一个概率分布已知的概率分布 p(z)。要对该分布进行拒绝采样，首先借用一个简单的参考分布(proposal distribution)，记为 q(x)，该分布的采样易于实现，如均匀分布、高斯分布。然后引入常数 k，使得对所有的的 z，满足 kq(z)≥p~(z)。在每次采样中，首先从 q(z)采样一个数值 z0，然后在区间 [0,kq(z0)]进行均匀采样，得到 u0。如果 u0<p~(z0)，则保留该采样值，否则舍弃该采样值。最后得到的数据就是对该分布的一个近似采样。

1.1.4 国内外研究现状总结

随着随机游走技术在学术界和工业界得到越来越广泛地关注，针对随机游走不同层面优化的专有系统相继被提出。现有随机游走算法大多基于一阶马尔可夫模型，当前状态以特定的概率转移到另外一个状态，这一过程中状态的存储以及状态的转移成为关键。常见的研究思路会从数据管理、数据访问以及更新模型等层面切入。

现有游走系统在 I/O 方面做了不同层面的优化工作降低 I/O 开销，FlashMob 在随机游走中探索访存的时间局部性和空间局部性，通过分区、重安排和批处理的操作，使内存的访问更加具有顺序性；DrunkardMob 中的基于迭代处理模型，将随机游走过程中的随机访问转化为对外存的顺序访问；C-SAW 采用负载感知的分区调度，优先处理包含负载更多的分区；SOOP 采用缓存高度顶点来减少邻居数据通信开销。除此之外还有诸多研究，将在如下第二节介绍。

2. 相关研究

本节主要是对五篇关于图随机游走引擎相关文献进行简单讲述与介绍，并列出文献中作者观测得到的结论以及实验后的研究成果。

2.1 GraphWalker

2.1.1 介绍

本论文出自 ATC 20[1]，作者提出了 GraphWalker，一种 I/O 高效且资源友好的快速可扩展随机行走图分析系统。

传统的图系统主要采用基于迭代的模型，迭代地将图块加载到内存中进行分析，以减少随机 I/O。然而，这种基于迭代的模型限制了运行随机游走的效率和可伸缩性，而随机游走是分析大型图的基本技术。在本文中，作者提出了 GraphWalker，这是一种用于随机行走的 I/O 高效图形系统，通过部署一种具有异步行走更新的新型状态感知 I/O 模型。GraphWalker 可以有效地处理由数千亿条边组成的非常大的圆盘图，而且它还可以扩展到运行数百亿个具有数千步长的随机行走。

作者开发了一种新的状态感知 I/O 模型 GraphWalker，该模型利用每个随机游走的状态，优先将游走最多的图块从磁盘加载到内存中，从而提高 I/O 利用率。作者还提出了一种步行意识的缓存方案来提高缓存效率。

作者采用了基于重入方法的异步行走更新方案，使得每次行走都可以移动尽可能多的步长，从而充分利用加载的子图，大大加快了随机行走的进度。为了解决异步更新引起的离散问题，作者还采用了一种概率方法来平衡每次行走的进度。作者提出了一种轻量级的以块为中心的索引方案来管理步行状态，并采用固定长度的步行缓冲策略来减少记录步行状态的内存成本。

为了比较该引擎的性能效果，作者开发了了一个原型，并进行了大量的实验来证明其效率。结果表明，与随机行走特定系统 DrunkardMob 以及两个最先进的单机图形系统 Graphene 和 GraFSoft 相比，GraphWalker 可以实现超过一个数量级的加速。此外，GraphWalker 更加资源友好，因为它的性能甚至可以与运行在机器集群上的最先进的分布式随机漫步系统 KnightKing 相媲美。

2.1.2 研究背景

目前基于迭代模型的图系统不能有效地支持随机行走。主要的限制有三个方面。首先，由于高度的随机性，很多行走不均匀地分散在图的不同部分，所以有些子图可能只包含很少的行走。然而，基于迭代的模型不知道这些行走状态，只是按顺序将所有需要的子图加载到内存中进行分析，因此它导致 I/O 利用率非常低。其次，由于基于迭代的模型确保了同步分析，所有的 walker 在每次迭代中都只移动一步。因此，遍历更新效率也受到限制，从而进一步加剧了 I/O 效率，对于需要长时间行走的应用程序尤其如此。最后，由于行走的随机性，每个顶点的行走次数是动态变化的，因此现有的图系统通常使用海量动态数组来记录当前通过图中每条边或每个顶点的行走次数。然而，这种索引设计需要很大的内存空间，因此限制了处理非常大的图形的可伸缩性。

（1）限制 1：低 I/O 利用率

基于迭代的模型导致随机行走的 I/O 利用率较低，它的定义是用于更新行走的边数除以在一个 I/O 中加载的边数。主要原因是即使从同一个源顶点开始，在走了几步之后，

行走可能会不均匀地分散在整个图上。因此，即使在一些块中只有很少的行走，它们仍然需要被加载到内存中，因此它带来了极低的 I/O 利用率。

为了解决这个问题，DynamicShards 和 Graphene，采用了按需 I/O 策略，动态调整图块布局，跳过不包含任何 walks 的加载块，以减少无用边的负载，但 I/O 利用率低的问题仍然没有完全解决。只要在一个块中有一次行走，那么这个块仍然必须加载到内存中进行计算。

（2）限制 2：低 walk 更新率

基于迭代的模型也导致了较低的 walk 更新率，它的定义是加载子图中所有步行的步行步数之和除以需要步行的总步数。这是因为在基于迭代的模型中，每次行走只能以同步的速度在每次迭代中移动一步，这严重浪费了内存中的数据，因为许多行走仍然可以在加载的子图上进行更多的移动。

最近，CLIP 提出了一种重入方法，Lumos 提出了交叉值传播技术，重用加载的数据，提高了 I/O 和计算效率，但由于多次访问整个子图，带来了额外的成本

（3）限制 3：管理 walk 数据的内存成本高

由于每个顶点上的行走次数是动态的且不可预测的，因此行走次数通常存储在大量的动态数组中，例如，GraphChi 将每条边与一个动态数组关联起来，以存储当前通过边缘的行走次数。这种设计产生了很高的内存成本，例如，对于像 YahooWeb 这样的中等规模图，它有 14 亿个顶点和 66 亿个边，仅存储行走数组索引，不包括行走状态信息，至少需要 26.4 GB 的空间。一些系统使用以顶点为中心的方式来管理行走，但这也会导致较高的内存成本，例如，为 YahooWeb 存储行走数组索引需要 5.6 GB。并且，将索引遍历到磁盘也会导致较高的开销，因为索引涉及很多文件。

2.1.3 GraphWalker 设计

（1）基本思想

主要思想是采用一个状态感知模型，该模型利用每个行走的状态，例如，行走所停留的当前顶点。状态感知模型选择加载包含最多行走次数的图块，并使每个行走在每个 I/O 内尽可能多地移动步数，直到到达所加载子图的边界。因此，可以有效地解决低 I/O 利用率和低 walk 更新速率问题。

为了进一步说明上述思想并分析其好处，图 2 显示了使用状态感知模型加载图和行走更新的过程。在第一个 I/O 中，图块 b0 被加载到内存中，因为它包含了所有的三次行走。对于加载的图形块 b0，遍历 w0 和 w1 移动两步，w2 只移动一步，因为它需要其他不在内存中的图形块来遍历更多的步骤。当两个 walk 进入区块 b2 时，在第二个 I/O 中，区块 b2 被加载到内存中，walk w0 结束，w1 可以移动一步。

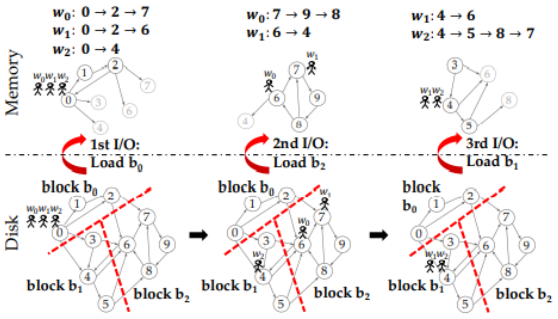


图 2 状态感知模型

最后，剩下的两次行走都在区块 b1 中，所以我们将 b1 加载到内存中，所有的行走都可以完成。本例中只需要 3 个 I/O。但是，对于基于迭代的模型，它可能需要 12 个 I/O，因为它使用了 4 个迭代，并且在每个迭代中产生 3 个 I/O。

基于上述思想，作者开发了一个 I/ O 高效的图形系统 GraphWalker，它支持快速和可扩展的随机行走。GraphWalker 主要由三部分组成:(1)状态感知的图形加载，(2)异步行走更新，(3)以块为中心的行走管理。GraphWalker 的总体设计如图 3 所示。

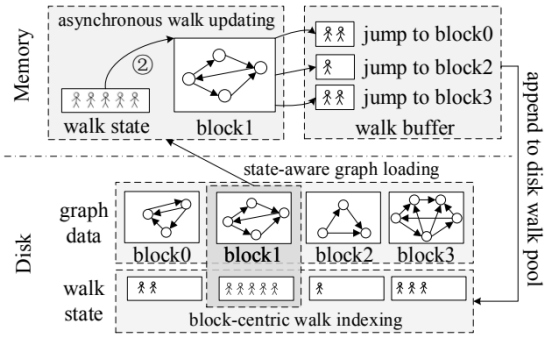


图 3 Graphwalker 设计

（2）状态感知图加载

GraphWalker 使用广泛使用的压缩稀疏行(CSR)格式管理图数据，该格式将顶点的外部邻居按顺序存储为磁盘上的 CSR 文件，并使用索引文件记录 CSR 文件中每个顶点的起始位置。如图 4 所示。

作者使用实证分析将默认块大小设置为 $2(\log 10^{R+2})$ MB，其中 R 是随机游走的总数，这样能取得较好的效果。

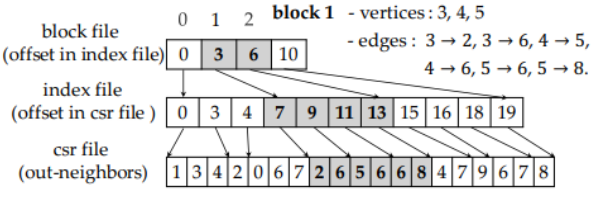


图 4 csr 文件格式

为了减轻块大小的影响并提高缓存效率，GraphWalker 还通过开发一种步行缓存方案来支持块缓存，以在内存中保留多个块。其基本原理是在不久的将来，有更多步行的街区更有可能再次被需要。因此，使用块缓存的图加载过程如下图所示所示。

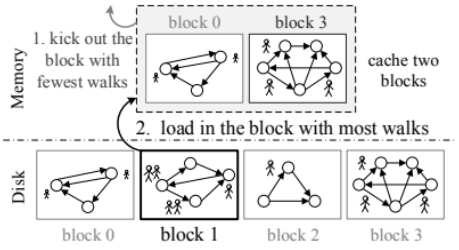


图 5 使用块 cache 的状态感知图加载

如图 5 所示，首先根据状态感知模型选择一个候选块，为了加载这个块，我们检查它是否缓存在内存中。如果它已经在内存中，那么我们直接访问内存来执行分析。否则，我们从磁盘加载它，如果缓存满了，也会从内存中取出包含最少路径的块。

（3）异步行走更新

在基于迭代的系统中，加载一个图块后，加载的子图中的每次行走都只走一步，这导致了非常低的行走更新速率。事实上，在走了一步之后，许多 walk 仍然停留在当前子图中的顶点上，因此可以使用更多的步骤进一步更新它们。

为了进一步提高 I/O 利用率和行走更新速率，GraphWalker 采用了异步行走更新策略，允许每次行走都不断更新，直到到达加载的图块边界。在完成一个 walk 之后，我们选择另一个 walk 来处理，直到处理完当前图块中的所有 walk。然后我们根据上面描述的状态感知模型加载另一个图块。图 6 显示了在同一个图形块中处理两次行走的示例。为了加速计算，还使用多线程并行地更新 walk。

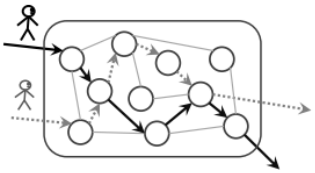


图 6 异步行走更新

然而，状态感知模型可能会导致全局离散问题。也就是说，一些 walks 可能移动得非常慢，因为它们可能被困在一些 coldblock 中，很长一段时间都没有加载到内存中。 为了解决全局离散问题，作在 GraphWalker 的状态感知图加载过程中引入了一种概率方法。每次选择要加载的图形块时，分配一个概率 p 来选择包含步行进度最慢的块，即步行步数最小的块。随着 p 的增加，全局离散问题将更有效地缓解，但大多数行走的效率将降低。所以设置 p 是有取舍的。们从磁盘加载它，如果缓存满了，也会从内存中取出包含最少路径的块。

（4）以块为中心的步行管理

我们用三个变量记录每一次行走的 source,current 和 step 分别表示开始顶点、当前顶点在块中的偏移量和移动的步数。我们用 64 位记录每一次行走。为每个变量分配的比特数如图 7 所示。该数据结构可以同时支持在 2^{24} 个源顶点开始随机行走，并且允许每次行走最多移动 2^{14} 步。

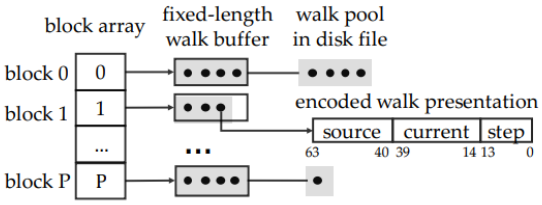


图 7 以块为中心的步行管理

为了减少管理所有 walk 状态的内存开销，作者提出了一个以块为中心的方案。每个步行池实现为一个固定长度的缓冲区，默认最多存储 1024 个步行，以避免动态内存分配成本。当一个块中有超过 1024 个步数时，将它们刷新到磁盘，并将它们存储为一个称为步数池文件的文件。

当将一个图形块加载到内存中时，也将它的行走池文件加载到内存中，并将其行走与存储在内存中行走池中的行走合并。然后在当前行走池中执行随机行走和更新行走。在更新过程中，当行走池已满时，通过将行走池中的所有行走附加到相应的行走池文件并清除缓冲区，将它们刷新到磁盘。

通过这种轻量级的步行管理，我们节省了大量用于存储步行状态的内存成本，从而能够支持大规模的并发步行。

2.1.4 实验部分

在本部分作者通过设置对比实验来验证了 GraphWalker 的优越性。通过实验数据证明在有的数据集下，相比于 DrunkardMob 实现了 9 到 46 倍的加速比，特别是在一些特殊的情况下，例如在 YahooWeb 上运行 PPR 和 SR，GraphWalker 甚至实现了超过三个数量级的加速，这是因为 YahooWeb 在查询顶点有非常好的局地性，所以 GraphWalker 只需要加载几个相应的子图就可以运行随机行走。然而，DrunkardMob 需要迭代扫描整个图，并以同步方式更新行走，因此它的 I/O 利用率非常低，需要很长时间。具体实验结果

如下图所示。

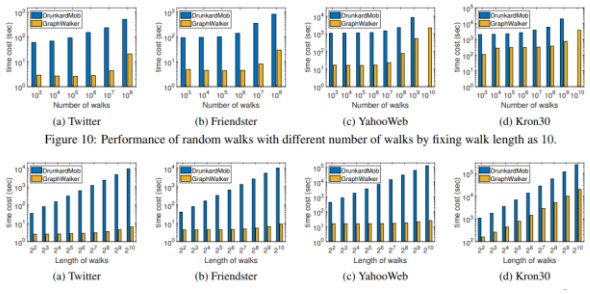


Figure 10: Performance of random walks with different number of walks by fixing walk length as 10.

图 8 实验结果

2.1.5 总结

在本文中，作者提出了 GraphWalker，这是一个 I/O 高效的系统，用于支持在单机上对大型图进行快速和可扩展的随机游走。GraphWalker 仔细管理图数据和遍历索引，并通过使用状态感知的图加载和异步遍历更新优化 I/O 效率。在我们的原型上的实验结果表明，GraphWalker 优于最先进的单机系统，并且它也达到了与运行在集群机器上的分布式图形系统相当的性能。在未来的工作中，我们将考虑将 GraphWalker 中的状态感知设计思想扩展到分布式集群，以并行处理大量的分析任务。

2.2 ThunderRW

2.2.1 介绍

本论文出自 VLDB 21[2]，由于随机行走在许多图处理、挖掘和学习应用中是一种强大的工具，本文提出了一种高效的内存随机行走引擎 ThunderRW。与现有的并行系统相比，ThunderRW 在提高单图操作性能方面支持大规模并行随机行走。ThunderRW 的核心设计是由作者的分析结果驱动的：常见的 RW 算法由于不规则的内存访问而有高达 73.1%的 CPU 流水线停滞，这比传统的图工作负载(如 BFS 和 SSSP)遭受更多的内存停滞。为了提高内存效率，作者首先设计了一个通用的以步长为中心的编程模型，称为 Gather-Move-Update，以抽象不同的 RW 算法。在编程模型的基础上，作者还开发了步进交错技术，通过切换不同随机游走查询的执行来隐藏内存访问延迟。

在本文中，作者提出了 ThunderRW，一个通用且高效的内存 RW 框架。通过采用了一个以步长为中心的编程模型，从步行者移动一步的局部视图中抽象出计算。用户通过在用户定义函数(UDF)中“像步行者一样思考”来实现 RW 算法。该框架将 UDF 应用于每个查询，并通过将查询的一个步骤作为一个任务单元来并行执行。此外，ThunderRW 还提供了多种采样方法，用户可以根据工作负载的特点选择合适的采样方法。

此外，在以步进为中心的编程模型的基础上，还提出了步进交错技术，通过软件预取来解决不规则内存访问导致的缓存失速问题。由于现代 CPU 可以同时处理多个内存访问请求，步骤交错的核心思想是通过发出多个未完成的内存访问来隐藏内存访问延迟，这就利用了不同 RW 查询之间的内存级并行性。

作者在实验部分通过展示 PPR、DeepWalk、Node2Vec 和 MetaPath 四种代表性算法，展示了 ThunderRW 的通用性和编程灵活性。用 12 个真实世界的图表进行了广泛的实验。结果表明:(1)ThunderRW 的运行速度比流行开源包中的初始实现快 8.6-3333.1 倍;(2)与 GraphWalker 和 KnightKing 等在同一台机器上运行的最先进的框架相比，ThunderRW 提供了 1.7-14.6 倍的加速;(3)步进交错技术将内存停滞从 73.1%显著降低到 15.0%。

2.2.2 研究背景

作者使用不同的采样方法执行 RW 查询，并使用自顶向下的微架构分析方法(TMAM)检查硬件利用率，以此来评估当前 RW 算法中存在的缺陷与问题。

TMAM 是一种用于识别无序 CPU 中的性能瓶颈的简化且直观的模型。它使用流水线表示处理微操作(uOps)所需的硬件资源。在循环中，流水线要么是空的(停止)，要么被 uOp 填充。执行停滞是由流水线的前端或后端部分引起的。由于缺乏所需的资源，后端无法接受新的操作。

在 livejournal 上的实验结果如表 1 所示。RW 查询随机访问图上的节点，导致大量的随机内存访问。因此，PPR 和 DeepWalk 高达 73.1%的流水线因内存访问而停滞。相比之下，BFS 和 SSSP 的内存界限小于 45%，证明了缓存局部性比 PPR 和 DeepWalk 好得多。由于内存停滞比例大，PPR 和 DeepWalk 的退隐率都小于 10%。此外，作者还测量了这些算法的内存带宽利用率。测试显示最大内存带宽为 60 GB/s。从表中可以看出，BFS 和 SSSP 的带宽利用率相当高(分别为 86.2%和 63.6%)，而 PPR 和 DeepWalk 的带宽利用率非常低(分别为 2.3%和 9.3%)。

Method	Front End	Bad Spec	Core	Memory	Retiring	Memory Bandwidth
BFS	11.6%	9.1%	20.8%	40.6%	18.0%	51.7 GB/s
SSSP	9.1%	12.5%	24.9%	36.9%	16.6%	38.2 GB/s
PPR	0.6%	0.7%	15.8%	**73.1%**	9.7%	**1.4 GB/s**
DeepWalk	1.0%	3.9%	16.7%	**69.7%**	8.7%	**5.6 GB/s**
Node2Vec	11.5%	22.1%	24.3%	28.1%	14.1%	17.1 GB/s
MetaPath	6.2%	7.5%	29.7%	33.9%	22.7%	9.9 GB/s

表 1 流水线停滞对比

常见 RW 算法的内存计算由于缓存丢失和内存带宽利用不足而导致内存停滞，从而遭受严重的性能问题。对于高

阶 RW 算法，计算 p(e)和初始化用于采样的辅助数据结构和初始化用于采样的辅助数据结构主要是内存计算成本，其复杂性分别由 RW 算法和所选择的采样方法决定。

作者进一步检验了抽样方法的性能。通过执行 10^7 个 RW 查询，每个查询都从图中随机选择的顶点开始。目标长度为 80。分别采用三种类型的采样方式与 5 种具体的采样方法，得到的实验结果如下图所示。

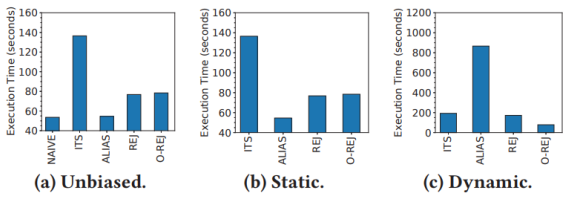


图 9 采样方式的影响

NAIVE 采样方法由于没有初始化阶段，在无偏 RW 上表现最好。在静态方法中，ALIAS 采样方法由于其生成阶段的时间复杂度较低而优于其他方法。然而，ALIAS 在动态 RW 上的运行速度比其他方法慢得多，因为它的初始化成本在实践中很高。由于没有初始化阶段，O-REJ 在动态 RW 上表现良好。可以观察到，由于每一步的初始化阶段(如果存在的话)，评估动态 RW 的成本比评估无偏静态 RW 的成本要高得多。

总而言之，采样方法对性能有重要的影响，没有一种采样方法可以在所有情况下都占主导地位。通常，动态读写比无偏读写和静态读写成本高。

2.2.3 ThunderRW 设计

（1）基本思想

首先需要开发机制来减少缓存停滞。分析结果表明，由于不规则的内存访问，常见 RW 算法的内存计算遭受了严重的性能问题。以前的随机游走框架都没有解决这个问题。另一方面，在随机游走工作负载中有大量的查询，但内存带宽没有得到充分利用。受之前使用预取加速数据库系统中多个索引查找的工作的启发，可以在不同查询之间进行预取和交错执行。其次，需要支持多种采样方法。现有框架只支持一种采样方法，一般认为所有 RW 都是动态的(如 C-SAW)，而采样方法对性能有重要影响，没有一种采样方法能在所有情况下都占主导地位。并且动态 RW 的评估成本通常比无偏和静态 RW 要高得多。

ThunderRW 的主要设计可以归结为两方面的内容，分别为以步长为中心的编程模型和步进交错技术。

（2）Step-centric 模型

为了抽象 RW 算法的计算，本文提出了步进中心模型。作者观察到 RW 算法是建立在许多 RW 查询之上的，而不是一个查询。尽管 RW 算法的查询内并行性有限，但由于每个 RW 查询都可以独立执行，因此在 RW 算法中具有丰富的查询间并行性。因此，以步长为中心的模型从查询的角度抽象了 RW 算法的计算，利用了查询间的并行性。

算法 2 给出了 ThunderRW 的概述。第 1-6 行逐个执行每个查询。第 3-5 行基于以步骤为中心的模型将一个步骤分解为三个函数。分别对应 Gather，Move 和 Update。

Algorithm 2: ThunderRW Framework

```
Input: a graph G and a set ℚ of random walk queries;
Output: the walk sequences of each query in ℚ;
1  foreach Q ∈ ℚ do
2      do
3          C ← Gather(G, Q, Weight);
4          e ← Move(G, Q, C);
5          stop ← Update(Q, e);
6      while stop is false;
7  return ℚ;
8  Function Gather(G, Q, Weight)
9      C ← {};
10     foreach e ∈ E_{Q.cur} do
11         Add Weight(Q, e) to C;
12     C ← execute initialization phase of a given sampling method on C;
13     return C;
14 Function Move(G, Q, C)
15     Select an edge e(Q.cur, v) ∈ E_{Q.cur} based on C and add v to Q;
16     return e(Q.cur, v);
```

算法 2 ThunderRW 算法

ThunderRW 将用户自定义函数应用于 RW 查询，并根据 RW 类型和选择的采样方法并行计算查询。因此，用户可以轻松地使用 ThunderRW 实现定制的 RW 算法，这大大减少了工程工作量。例如，用户只需编写大约 10 行代码来实现 Node2Vec，如图 10 所示。

```
WalkerType walker_type = WalkerType::Dynamic;
SamplingMethod sampling_method = SamplingMethod::O-REJ;
double Weight(Walker Q, Edge e) {
    if (Q.length == 0) return max(1.0 / a, 1.0, 1.0 / b);
    else if (e.dst == Q.prev) return 1.0 / a;
    else if (IsNeighbor(e.dst, Q.prev)) return 1.0;
    else return 1.0 / b;
}
bool Update(Walker Q, Edge e) {
    return Q.length == target_length;
}
double MaxWeight() {
    return max(1.0 / a, 1.0, 1.0 / b);
}
```

图 10 Node2vec 例子

RW 算法包含大量的随机游走查询，每个查询都可以独立快速地完成。因此，ThunderRW 采用静态调度的方式来保持 worker 之间的负载均衡。具体来说，我们将每个线程视为一个 worker，并将 Q 平均分配给 worker。worker 使用算法 2 独立执行分配的查询。实验结果表明，简单的调度方法取得了良好的性能。

（3）步进交织技术

受软件预取技术的启发，作者建议通过交替执行不同查询的步骤来隐藏内存访问延迟。具体来说，给定 Move 中的

一系列操作，我们将它们分解为多个阶段，以便一个阶段的计算消耗前一阶段生成的数据，并在必要时检索后续阶段的数据。我们同时执行一组查询。查询 Q 的一个阶段一旦完成，我们切换到组中其他查询的阶段继续执行。通过这种方式，我们可以在单个查询中隐藏内存访问延迟，并保持 CPU 繁忙。

下图给出了步进交织技术的一个示例，其中一个步骤被分为四个阶段。如果按顺序逐步执行查询，则由于内存访问，CPU 经常会被卡住。即使使用预取，阶段的计算也无法隐藏内存访问延迟。相反，步骤交错通过交替执行不同查询的步骤来隐藏内存访问延迟。

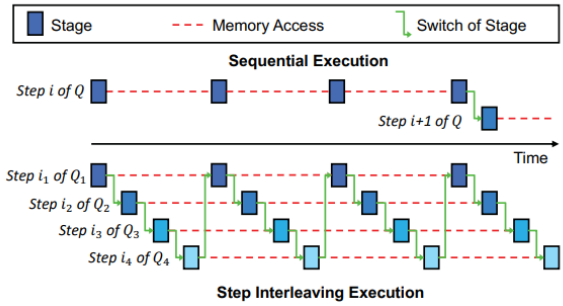


图 11 步进交织技术

作者使用状态依赖图（SDG）来对在一个步骤操作序列的各个阶段进行建模。SDG 中的每个节点都是一个包含一组操作的阶段，边表示它们之间的依赖关系。给定操作序列，我们分两步构建 SDG，抽象阶段(节点)和提取依赖关系(边)。

定义阶段:当我们通过切换查询的执行来隐藏内存访问延迟时，阶段的限制是每个阶段最多包含一个内存访问操作，并且使用数据的操作在后续阶段。

定义边:根据 SDG 中节点的依赖关系在节点之间添加边。

下图分别代表了偏置采样和拒绝采样的状态定义和 SDG 划分方法。

Stage	ALIAS
S_0	O_0: Load d_v.
S_1	O_1: Generate an int random num x in $[0, d_v)$.
	O_2: Generate a real random num y in $[0, 1)$.
	O_3: Load $C[x] = (H[x], A[x])$.
S_2	O_4: If $y < H[x]$, $v' = A[x].first$; Else $v' = A[x].second$.
	O_5: Add v' to Q and return $e(v, v')$.

Stage	REJ
S_0	O_0: Load d_v.
S_1	O_1: Load the maximum value p_v^*.
S_2	O_2: Generate an int random num x in $[0, d_v)$.
	O_3: Generate a real random num y in $[0, p_v^*)$.
	O_4: Load $C[x] = p$.
S_3	O_5: If $y > C[x]$, jump to O_2; Else jump to O_6.
S_4	O_6: Load $e(v, v') = E_v[x]$.
S_5	O_7: Add v' to Q and return $e(v, v')$.

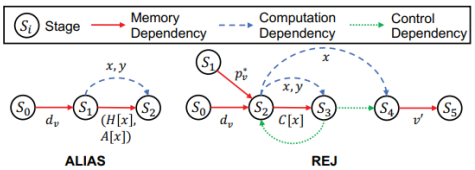


图 12 SDG

总之，SDG 是从 Move 中的一系列操作中抽象出各个阶段，并对它们之间的依赖关系进行建模的方法。

我们需要一个有效的切换机制。例如，禁止使用多线程，因为线程间上下文切换的开销以微秒为单位，而主内存延迟以纳秒为单位。由于每个线程倾向于接受许多 RW 查询，所以我们在单个线程中切换各个阶段的执行。

对于查询中不同阶段之间的数据通信，作者基于 SDG 创建了两种环形缓冲区，其中计算依赖边表示需要存储的信息。任务环用于跨查询的所有阶段的数据通信，而搜索环用于处理周期阶段。由于需要显式地记录周期阶段的状态并控制它们的切换，处理周期阶段不仅会导致实现的复杂性，而且会产生更多的开销。NAIVE 和 ALIAS 的 SDG 没有循环阶段，因为它们的生成阶段没有 for 循环，而 ITS、REJ 和 O-REJ 的 SDG 有。

2.2.4 实验部分

在实验中作者用以下方法比较了 ThunderRW 的性能。

BL:基线方法，首先将一个图完全加载到内存中，然后执行随机游走。

HG:从两个方面优化 BL:(1)为每个算法选择合适的采样方法;(2)将每个查询作为 OpenMP 的并行任务。

GW: GraphWalker，最先进的单机 RW 框架。为了便于比较，我们将 GraphWalker 配置为在内存中执行，不需要任何磁盘 I/O。

KK: KnightKing，最先进的分布式 RW 框架。它支持在单机上执行。

对于 RW 算法，GW 只支持无偏 RW。因此，我们执行 PPR 时不考虑边权值，只在 PPR 上评估 GW。尽管 KK 在原文中研究了 MetaPath，但其开源包无法处理标记图。

	PPR					DeepWalk				Node2vec				MetaPath			
Dataset	BL	HG	GW	KK	TRW	BL	HG	KK	TRW	BL	HG	KK	TRW	BL	HG	KK	TRW
am	0.06	0.008	0.42	0.012	**0.007**	2.16	0.21	0.44	**0.07**	9.97	0.26	2.08	**0.14**	0.22	0.018		**0.012**
yt	0.33	0.04	1.68	0.05	**0.015**	9.78	0.98	1.93	**0.26**	853.13	1.30	5.94	**1.03**	6.18	**0.23**		0.24
up	1.24	0.13	7.19	0.19	**0.07**	45.44	4.33	8.41	**0.95**	369.00	6.20	16.92	**4.01**	4.88	0.40		**0.24**
eu	0.16	0.02	0.99	0.03	**0.011**	8.16	0.82	1.56	**0.20**	2731.07	1.47	4.43	**1.14**	90.55	**3.18**		3.55
ac	4.84	0.51	19.31	0.65	**0.19**	173.66	17.86	31.88	**3.31**	6951.12	24.54	87.86	**6.26**	45.01	2.01		**1.69**
ab	8.86	0.94	26.74	1.09	**0.26**	212.80	22.24	40.07	**4.01**	26231.45	32.04	100.78	**7.87**	128.35	5.06		**4.47**
lj	1.69	0.19	7.90	0.23	**0.06**	55.63	5.44	10.67	**1.19**	2951.33	9.09	24.95	**6.20**	18.08	0.94		**0.75**
ot	1.49	0.16	5.25	0.19	**0.04**	38.54	3.70	7.97	**0.80**	5891.28	7.28	15.16	**4.82**	40.77	1.72		**1.57**
wk	21.86	2.21	47.05	3.07	**0.59**	502.27	49.67	95.17	**9.26**	*OOT*	68.43	216.24	**27.68**	5.98	**0.54**		0.55
uk	6.47	0.69	27.72	0.90	**0.24**	205.86	20.42	21.40	**4.56**	12630.01	54.36	94.69	**28.68**	322.66	12.84		**12.56**
tw	26.42	2.73	77.12	3.61	**1.16**	575.43	61.18	115.92	**11.13**	*OOT*	130.72	232.41	**91.00**	*OOT*	12300.32		**9780.20**
fs	79.14	8.20	223.81	10.72	**4.10**	1043.93	108.23	208.45	**17.67**	*OOT*	178.15	364.51	**120.16**	683.05	28.69		**25.01**

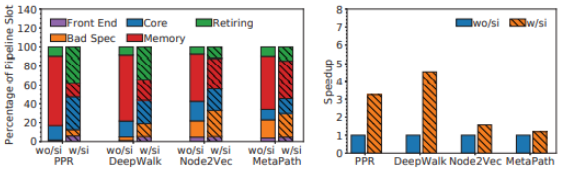
表 2 总体对比

表 2 给出了四种 RW 算法的竞争方法的总体比较。虽然 GW 是并行的，但它比顺序基线算法 BL 运行得慢。KK 的运行速度比 GW 和 BL 快，但比 HG 慢，因为框架比 HG 产生了额外的开销;HG 对每种算法都采用合适的采样方法。TRW 的运行速度分别比 GW 和 KK 快 54.6-131.7 倍和 1.7-14.6 倍。

总之，ThunderRW 明显优于最先进的框架和自主开发的解决方案(例如，BL 在 tw 上花费了超过 8 小时的 Node2Vec，而 TRW 在两分钟内就完成了算法)。此外，与 BL 和 HG 相

比，ThunderRW 在 RW 算法的实现和并行化上节省了大量的工程工作量。

下图展示了采用不同采样方法的 Lj 上的流水线停滞和加速。我们可以看到，步进交错技术显著地降低了五种采样方法的内存约束，并实现了显著的加速。结果证明了步进交织技术的通用性和有效性。



(a) Pipeline slot breakdown. (b) Speedup.

图 13 细节对比

2.1.5 总结

在本文中，作者提出了 ThunderRW，一种高效的内存 RW 引擎，用户可以轻松地在其上实现定制的 RW 算法。通过设计了一个以步骤为中心的模型，从移动查询的一个步骤的局部视图中抽象出计算。在此基础上，又提出了步进交错技术，通过交替执行多个查询来隐藏内存访问延迟。用当前框架实现了 PPR、DeepWalk、Node2Vec 和 MetaPath 四种有代表性的 RW 算法。实验结果表明，ThunderRW 比最先进的 RW 框架的性能提高了一个数量级，步长交错将内存约束从 73.1%降低到 15.0%。目前，作者在 ThunderRW 中通过显式地手动存储和恢复每个查询的状态来实现步进交错技术。一个有趣的未来工作是用协程实现该方法，这是一种支持交错执行的有效技术。

2.3 KnightKing

2.3.1 介绍

本论文出自 SOSP 19[3]，作为图形数据分析和机器学习的工具，图中的随机游走技术最近得到了广泛的欢迎。目前，随机行走算法是作为单独的实现开发的，并且存在显著的性能和可伸缩性问题，特别是复杂行走策略的动态特性。

基于此问题，作者提出了 KnightKing。这是一个通用的分布式图随机漫步引擎。为了解决静态图与许多动态 walker 之间的独特交互，它采用了直观的以 walker 为中心的计算模型。相应的编程模型允许用户轻松指定现有或新的随机行走算法，通过应用于流行已知算法的新的统一边缘转移概率定义来促进。使用 KnightKing，这些不同的算法受益于其通用的分布式随机漫步执行引擎，围绕创新的基于拒绝的抽样机制，极大地降低了高阶随机漫步算法的成本。作者通过实验评估证实，KnightKing 在执行算法方面带来了多达 4 个数量级的改进。

随机游走算法定义了它自己的边缘转移概率。随着随机行走变得越来越流行，采样逻辑也变得更加复杂，最近的算法执行动态采样，其中边缘转移概率取决于 walker 的当前状态，它访问的前一个顶点(或多个顶点)，以及它当前驻留顶点的边的属性。

与图处理不同的是，随机游走缺乏为高效和可扩展的提供通用算法或系统支持的通用框架。结果，应用程序用户开发了他们自己的随机游走实现，遭受了冗余劳动并导致了糟糕的性能。

本文提出了图随机游走的第一个一般框架 KnightKing。它可以被看作是一个“分布式随机漫步引擎”，是传统图引擎的随机漫步对应物。KnightKing 采用以 walker 为中心的视图，使用 api 定义自定义的边缘转移概率，同时处理常见的随机行走模型。与图引擎类似，KnightKing 隐藏了图划分、顶点分配、节点间通信和负载平衡等系统细节。因此，它促进了直观的“像 walker 一样思考”视图，尽管用户可以灵活地添加可选优化。

KnightKing 的效率和可扩展性的核心在于其快速选择下一个跳顶点的能力。作者首先提出了一个统一的转移概率定义，它允许用户直观地定义自定义随机行走算法的静态(依赖于图)和动态(依赖于 walker)转移概率。基于该算法定义框架，作者构建了 KnightKing 作为第一个执行拒绝抽样的随机游走系统。虽然拒绝抽样本身是为动态抽样而设计的，但 KnightKing 是一个通用框架，它也能有效地处理静态抽样。除了上述算法创新，KnightKing 还包括系统设计选择和优化，以支持其以 walker 为中心的编程模型。

2.3.2 研究背景

本文是一篇发表较早的论文，在此之前关于随机游走的研究相对较少。现有的对于静态随机游走的优化是现有随机行走实现采用的替代方法是别名方法。它将每条边分成一个或多个碎片，碎片总数不大于 2n，放入 n 个桶中，限制每个桶最多存放 2 个碎片，并且每个桶中碎片的重量之和(P_s)完全相同。这些桶及其内容构成别名表。为了对一条边进行采样，我们首先对一个桶进行均匀采样，然后根据桶的权重对桶的一个片段进行采样，返回该片段所属的边缘。在这里，一条边被采样的概率与它相应部分的权重之和成正比，而相应部分的权重又等于它自己的权重。

下图展示了 4 条边的分割，使用与上面相同的例子，并将它们分配到 4 个桶中。建立别名表的预处理时间和空间开销为 O(n)，使得边缘采样的复杂度为 O(1)。KnightKing 利用别名方法来处理静态转换概率组件。

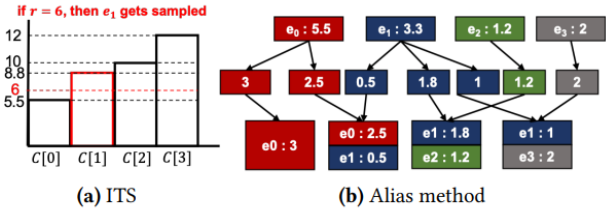


图 14 静态随机游走采样

上述优化并不适用于动态随机漫步，因为它带来了大量的时间和空间开销来预计算和存储 ITS 数组或枚举可能的 walker 状态的别名表。例如，在 11 GB 的 Twitter 图上，使用 CDF 或 alias 精确计算 node2vec 分别需要大约 970TB 或 1.89PB 内存。

针对动态随机游走的算法优化是存在的。例如，元路径实现执行预处理来构建每个边类型的 ITS 数组或别名表，在不增加预处理时间/空间开销的情况下实现快速采样，因为边按类型划分为不连接的集。然而，这不能推广到所有的动态随机游走。对于 node2vec, Fast-Node2Vec 使用诸如缓存流行顶点的边缘列表等优化来减少数据传输，并牺牲 walker 处理并发性以节省内存消耗。

在写作本文时，作者所知道的唯一关于随机行走的系统研究是 DrunkardMob，目标是多核处理器上的高速随机行走。然而，它只关注静态行走，并被设计为在单机上运行内核外。通用图计算系统通常针对传统的图算法进行深度优化和评估，而不解决随机游走工作负载。

与这些系统不同的是，KnightKing 能够以 O(1)成本进行精确边缘采样，预处理开销不超过 O(n)水平，用于常见的随机行走算法，包括高阶随机行走算法。

2.3.3 KnightKing 设计

与传统图形引擎在迭代中协调多个顶点(沿多个边)的更新类似，KnightKing 拥有迭代计算模型，同时协调多个步行者的动作。像顶点和边一样，walker 也被分配给每个节点/线程，基于其当前驻留顶点的分配。当然，与传统图引擎的明显区别是沿着采样边的概率行走与沿着所有/活动边的确定性更新传播。随之而来的，更微妙的区别是每次迭代执行分布式随机漫步引擎。

虽然图引擎可以通过一轮点到顶点消息来推送/拉更新和执行顶点状态更新，但在处理高阶遍历时，KnightKing 需要打破这种迭代，以包含两轮消息传递。由于每个 walker 都可能需要根据最近访问的顶点进行边缘选择，因此可能会发出 walker-to-vertex 查询。在分布式设置下，顶点和边是跨节点分区的，需要通过消息传递来发送此类查询并收集其结果。

为了促进这种分布式查询操作的高效批处理和协调，KnightKing 扮演了类似于邮局的角色，其中 walker 根据其本地抽样候选对象提交查询消息，发送到涉及动态检查的顶点。所有这些查询都是根据点到节点的分配来交付的，所有节点都并行地处理从所有 walker 接收到的查询。然后，另一轮消息传递将集体返回结果，查询 walker 将一起检索结果。

以下是 KnightKing 迭代中的步骤:

1、walker 生成候选边缘进行拒绝抽样并进行初步筛选。

2、 必要时，walker 根据采样结果发出行走者到顶点的状态查询。

3、所有节点都处理状态查询并返回结果。

4、walker 检索状态查询结果。

5、walker 决定采样结果，如果成功就移动。

请注意，上面描述的是 KnightKing 支持的随机游走的最一般情况。通常步骤可以跳过，例如，对于静态或一阶随机游走，步骤 2-4 被省略，因为在局部采样中不需要涉及其他顶点。对于这样的算法，所有的 walker 都可以同步移动:在每次迭代中，walker 执行采样和检查，直到一条边被成功采样。在这些情况下，所有活跃的 walker 在他们的行走序列中都处于相同的步长。然而，对于 node2vec 这样的高阶算法，跨 walker 的迭代是通过两轮行走者到顶点的查询消息传递来同步的。成功采样的幸运 walker 将继续向前走一步，而不太幸运的 walker 将停留在当前的顶点，等待下一次迭代。这样，walker 可能以不同的速度前进，可能会产生掉队者。

KnightKing 会自动执行默认的 walker 状态维护，比如更新当前驻留的顶点和行走的步数。此外，用户可以使用提供的 api 执行自定义 walker 属性的初始化和更新。

KnightKing 中的系统设计选择和优化，为各种随机游走算法提供了统一的执行引擎。KnightKing 用了大约 2500 行 c++代码实现，使用 OpenMPI 在节点间传递消息。它的底层与分布式图引擎有很多共同之处，在分布式图引擎中，作者采用了成熟系统中的基础设施和技术。我们的讨论集中在设计方面和特定于随机漫步执行的权衡。

（1） 图存储和划分

KnightKing 存储边使用压缩稀疏行(CSR)。KnightKing 采用了顶点分区方案。在分布式执行中，每个顶点被分配给一个节点，所有有向边都存储在它们的源顶点中。无向边在两个方向上存储两次。

一般来说，在随机行走中访问顶点的频率取决于用户定

义的(潜在的动态)转移概率，并涉及图拓扑和行走者行为之间复杂的相互作用。在 KnightKing 中，粗略地将处理工作量估计为节点的局部顶点和边缘计数的总和，并在节点之间执行 1-D 分区来平衡这个总和。

（2） 计算模型

当 walker 独立移动时，随机行走算法可能会出现令人尴尬的并行和无协调，容易扩展到更多的线程/节点。每个 walker 通过 getVertex()和 getEdges()等 api 检索信息。

在 walk 开始之前，KnightKing 根据用户指定或默认设置执行初始化。这包括如果定义了自定义静态组件 Ps，则构建逐顶点别名表;如果定义了自定义动态组件 Pd，则使用提供的上界和可选下界设置拒绝抽样;以及 walker 实例化/初始化。

（3） 任务调度

KnightKing 执行任务调度的方式与 Gemini 等图形引擎类似，但以 walker 为中心，而不是以顶点为中心。它在每个节点中设置并行处理，其线程数与可用的执行计算的核数相同，另外还有两个线程专门用于消息传递。高阶随机游走中的两轮通信在每次迭代中实现，计算(生成传出查询消息和处理传入查询)与消息传递重叠。

因为基于拒绝的核心策略将抽样复杂性降低到近 O(1)，KnightKing 使每一步的抽样成本大大提高了可预测性。然而，它仍然可能面临长尾执行，少数掉队者比大部分 walker 停留的时间要长得多。解决的方案是当一个节点的活跃 walker 数量低于阈值时，它通过仅保留三个线程切换到 light 模式。

2.2.4 实验部分

下表给出了算法和输入图组合的总体执行时间(分别为未加权和加权版本)。所有结果都来自执行 8 个节点，每个节点使用 16 个线程。通过静态行走(DeepWalk 和 PPR)，KnightKing 执行统一的采样工作流，但不实际执行拒绝采样。因此，它相对于 Gemini 的性能优势来自于它的系统方面。总的来说，KnightKing 在这两个静态算法中领先 Gemini 高达 16.94 倍(平均 8.22 倍)。

除了前面提到的 Gemini 固有的图划分的限制外，性能差距还来自于图处理系统和图随机游走工作负载之间的设计不匹配。例如，它的“密集”模式使用基于内角的拉式通信，而 walker 自然地向外工作，迫使它保持“稀疏”模式。在传统的图处理中，顶点通常会更新它的所有邻居。因此 Gemini 通过广播执行这样的“推送”操作，让顶点同时通知它的所有镜像。这为随机行走的执行带来了巨大的浪费，因为 walker 只需要沿着单一的边缘行走。当给边分配权重时(表 4)，结果相似，两个系统的执行时间都有适度增加。

Time in seconds		Gemini	KnightKing	Speedup $\times times$
DeepWalk	LiveJ	17.64	2.22	7.93
	FriendS	182.09	21.15	8.61
	Twitter	96.90	12.76	7.60
	UK-Union	223.88	38.73	5.78
PPR	LiveJ	110.14	6.50	16.94
	FriendS	297.51	30.82	9.65
	Twitter	201.55	20.27	9.94
	UK-Union	351.99	49.56	7.10
Meta-path	LiveJ	63.59	2.74	23.20
	FriendS	691.07	32.28	21.41
	Twitter	24165*	20.98	1152.03*
	UK-Union	537438*	66.87	8037.50*
node2vec	LiveJ	168.55	14.12	11.93
	FriendS	1467.07	69.80	21.02
	Twitter	97373*	44.14	2206.12*
	UK-Union	1822207*	163.59	11138.85*

表 3 无权图

Time in seconds		Gemini	KnightKing	Speedup $\times times$
DeepWalk	LiveJ	17.73	3.14	5.65
	FriendS	193.47	30.47	6.35
	Twitter	102.12	17.29	5.91
	UK-Union	233.76	63.24	3.70
PPR	LiveJ	107.52	7.21	14.92
	FriendS	306.10	39.22	7.80
	Twitter	211.99	24.69	8.59
	UK-Union	352.99	70.50	5.01
Meta-path	LiveJ	68.65	3.38	20.32
	FriendS	770.09	47.81	16.25
	Twitter	51783*	30.31	1711.62*
	UK-Union	932536*	97.44	9570.07*
node2vec	LiveJ	170.46	15.34	11.11
	FriendS	1483.33	78.68	18.85
	Twitter	101095*	49.35	2048.53*
	UK-Union	1917205*	189.34	10126.20*

表 4 有权图

在所有输入图和两个系统中，由于 PPR 的不确定性行为，它比 DeepWalk 慢得多。然它的预期步行长度也是 80，与 DeepWalk 的(固定)长度相当，但 PPR 的最长步行长度超过 1000，产生了更长的执行时间和前面讨论的离散情况。

在动态算法方面，KnightKing 的拒绝抽样带来了压倒性的优势。使用传统采样的 Gemini 发现，行走执行时间随着元路径的增加而显著增加，并且随着 node2vec 的增加而激增。对于这两种算法，Twitter 图行走都不能在 6 小时内完成，UK-Union 上的 node2vec 估计需要>500 小时(8 个节点上共有 128 个线程)。这些结果解释了目前对这种高阶算法采用近似解的原因。

2.1.5 总结

在本文中提出了 KnightKing，这是第一个通用的分布式图随机行走引擎。它提供了一个直观的以 walker 为中心的计算模型，以支持随机行走算法的简单规范。作者提出了一个统一的边缘转移概率定义，适用于流行的已知算法，以

及新颖的基于拒绝的采样方案，极大地降低了昂贵的高阶随机游走算法的成本。通过对 KnightKing 的设计和评估表明，在精确边缘采样中，无论当前顶点的出边数如何，都有可能实现接近 O(1)的复杂度，而不会损失精度。

2.4 FlashMob

2.4.1 介绍

本论文出自 SOSP 21[4]，以随机访问大型工作集为主导的数据密集型应用程序无法利用现代处理器的计算能力。图随机游走是许多重要的图处理和学习应用中不可缺少的主力，是这类应用的一个突出案例。

但是现有的图随机游走系统目前无法匹配 GPU 侧节点嵌入的训练速度。本文揭示了现有的方法不能有效地利用现代 CPU 内存层次结构，因为人们普遍认为随机游走中固有的随机性和图的歪斜性质使大多数内存访问是随机的。作者证明，通过仔细划分、重新安排和批处理操作，实际上有大量的空间和时间局部性可以获取。由此产生的系统 FlashMob 通过使内存访问更加有序和规则，提高了缓存和内存带宽的利用率。经典的组合优化问题(及其精确的伪多项式解)可以应用于复杂的决策，以实现准确而高效的数据/任务划分。在不同图形上的综合实验表明，FlashMod 比现有最快的系统实现了数量级的性能改进。它处理 58GB 真实图的每一步速度比现有系统在 L2 缓存中 600KB 图上的速度要快。

现代计算机具有为数据密集型应用程序设计的复杂内存层次结构。多级 CPU 缓存，以及高 DRAM 带宽，以及硬件预取和内存行缓冲区等特性，透明地帮助程序从时间和空间局部性中获利。前者允许数据重用。后者有利于大型顺序访问，这往往更快。但是在大型工作集上大量执行随机访问的应用程序从最先进的 CPU 硬件中获得的好处要低得多。它们经常花费大量的执行时间等待数据，浪费昂贵的数据中心或服务器资源。

许多图随机行走执行将边或路径样本提供给图嵌入训练，通常使用随机梯度下降(SGD)方法。因此，最近的图嵌入框架同时在 CPU 上进行图随机游走，在 GPU 上进行嵌入训练(加上下游应用程序)。

在行走的大型图的明显随机性质下，有大量的空间和时间局部性可以通过仔细的划分、重新安排和批处理操作来获取。本文提出了一种新的图形随机游走设计方法 FlashMob。它在很大程度上支持顺序的内存访问，以在不同的缓存级别中有策略地切割图形分区来处理。

FlashMob 对包含相似度顶点的分区进行流处理，随后可以进行多次优化。对于少量的高度点，通过对碰巧同时位于这些热点上的步行者进行批处理，来获取它们的访问密度。对于低次数据，利用它们在程度上的规律性，采用简化的数据结构和直接索引来减少处理过程中的随机内存访问。

2.4.2 研究背景

目前最先进的随机游走方法集中在算法改进上，例如改进的边缘采样(KnightKing)或大图上的核外行走(GraphWalker)，这大大降低了复杂随机行走算法的内存需求，并支持非常大的图的内存处理。

现有系统无一例外地逐个处理 walker。在迭代过程中，所有 walker 轮流到每个样本，并从其当前顶点的邻接表中跟踪一条边。这种常见的做法虽然直观，但却造成了巨大的资源浪费。与图处理任务不同，图随机游走通过从潜在的许多候选边中选择一条边来执行更稀疏的处理。

因此，来自采样缓存线的大部分内容将被丢弃。这在多个方面滥用了内存层次结构:快速私有 L2 缓存中的缓存数据利用率低，共享 L3 缓存中的数据重用率低，并且增加了 DRAM 总流量。作者的工作旨在减少这种浪费。

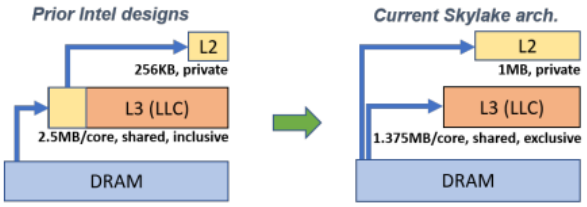


图 15 英特尔处理器缓存设计的改变

如图 15 所示，最近的英特尔处理器保留了现代多核处理器的分层缓存层次结构:每个内核都有其私有的 L1 和 L2 缓存，而插槽中的所有内核共享位于内核和主存之间的 LLC。早期的 Intel 采用了一个比 L2 大一个数量级的 LLC，具有包容性的 LLC 管理。这意味着所有进入小得多的 L2 缓存的数据也驻留在 L3 中。使用英特尔目前的可扩展系列(Skylake)处理器，L3 和 L2 之间的相对大小比例显著降低。全新的 L3 设计进一步提升了更大的 L2:缓存失误将直接将数据带入 L2 而不是 L3，后者用于保存从 L2 中驱逐出来的数据，从而实现更好的整体缓存容量利用率，并促进核心之间的数据共享。此外，在非包容的 L3 设计中，虽然合并缓存大小实际上比以前的 Broadwell 架构缩小了(从每个核 2.5MB 减小到 1.375MB)，但现在更多的这样的空间在更快的 L2 上，并保存一组不相连的数据作为共享 L3。

作者测量了使用不同的访问模式从内存层次结构的不同位置加载单个单词的延迟。表 5 中的结果证实:(1)尽管这

种内存硬件具有随机访问的性质，但在顺序访问和随机访问之间存在很大的延迟差距;(2)序列流带来了可承受的延迟，甚至从远程内存跨 NUMA 节点，而序列-随机性能差距增长迅速，因为我们向下层次结构;(3)指针跟踪是昂贵的，它的成本使得 L3 缓存中的访问比简单的随机访问 DRAM 要慢。

Location	L1C	L2C	L3C	LocalMem	RemoteMem
Sequential read	0.42ns	0.41ns	0.44ns	**0.76ns**	**1.51ns**
Random read	**0.77ns**	**0.95ns**	**2.60ns**	**18.35ns**	24.35ns
Pointer-chasing	**1.69ns**	**5.26ns**	19.26ns	116.90ns	194.26ns

表 5 加载延时

2.4.3 FlashMob 设计

（1） 基本思想

图 16 描述了 FlashMob 架构概述。与现有的随机游走实现不同，FlashMob 不会跟随 walker 访问整个图，即使有足够的内存空间承载后者。它将所有顶点按程度降序排序，并将它们切割成许多顶点分区。图中显示了两个这样的分区，一个具有几个高次顶点，另一个具有许多低次顶点。一个线程一次只处理一个任务，将当前在一个分区上的所有 walker 移动一步。

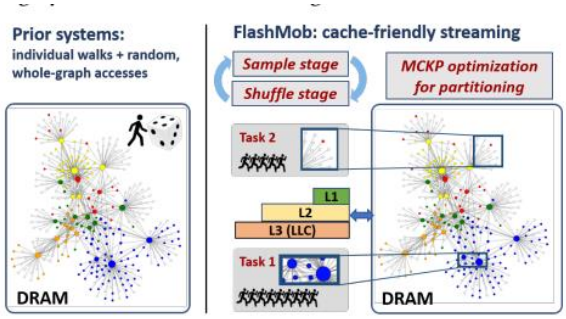


图 16 与现有 RW 系统对比

结合顶点划分和 walker 批处理，FlashMob 通过(1)在缓存中拟合每个任务的工作集和(2)促进顺序内存访问，极大地提高了内存访问效率。本质上，它在 CPU 缓存中执行“核外”处理，并使用 DRAM 进行快速数据流

体行走性能取决于许多因素的复杂相互作用，例如图的大小和度分布，每个缓存级别的大小/速度，以及图上的相对行走者“密度”。FlashMob 不是硬编码，也不是让用户来配置重要参数，如分区和采样策略设置(何时启用预采样)，而是通过近似将其优化映射到 MultiChoice KnapSack (MCKP)问题。

（2） 频率感知顶点分组

将排序后的顶点数组切割成不同大小的相邻顶点分区(简称 vp)。下图展示了一个示例图，其中 8 个顶点按程度排序，并切割成 3 个 vp。

图 17 顶点划分

vp 构成基本任务单元，其大小基于图特征和系统资源约束进行优化。将相似度的顶点分组，FlashMob 用不同的策略处理它们，并从不同的来源获得性能。

FlashMob 还对 walker 数据进行了分区，这样在每次步行者迭代过程中，VP 都会使用连续的 walker 数据块来访问和更新 walker 位置。为此，FlashMob 试图拥有简单而紧凑的 walker 状态存储，它可以作为“消息”，在 walker 的步伐之间进行洗选，以及 walker 的路径历史。

（3） 边采样阶段

FlashMob 随机行走迭代的边缘采样阶段执行核心任务:为每个 walker 找到一条可以移动的边缘。更具体地说，对于给定的 VP, FlashMob 选择一种抽样策略，要么是预抽样(PS)，要么是直接抽样(DS)，因此采用不同的数据组织和访问模式。

预采样(PS)为访问频率更高的顶点设计的，通过在同一顶点上批量行走，最大限度地提高缓存利用率。主要思想是提前采样许多边，这些边被许多共存的行走器依次消耗。对于采用 PS 策略的 VP，每个顶点分配一个预采样边缘缓冲区，由行走者检索边缘样本。使用 PS，可以从本质上解耦边缘样本的生产和消耗阶段。虽然这会产生额外一轮边样本存储/检索操作，但通过批处理类似操作，整体内存访问本身的效率显著提高。

直观地说，预采样的好处随着低次顶点的减少而减少。在极端情况下，对于只有一条边的顶点，如图 5 中的 VP3，根本不需要边采样。对于那些度为 2 的顶点(它们占顶点的 5.0%-14.7%)，如图 5 中的 VP2，从它们的两条边存储和采样既节省了空间又节省了时间。

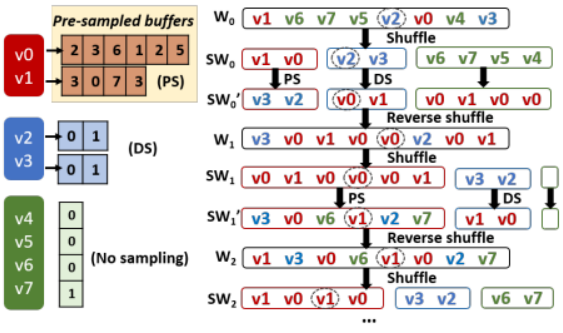


图 18 walker 数据管理

因此，FlashMob 提供了 DS 选项，walker 在现场投掷骰子，从(通常是短的)邻接表中选择出边。在图 5 中，每当 walker 在 VP2 内时，它都会从顶点的两条外向边随机采样。

每个 FlashMob 线程处理一个 walker 的子集。主要的区别是它的缓存友好的边缘采样，也分区图以适应缓存中的任务。以前系统中的随机行走实现需要随机访问一个顶点，并随机选择它的一条边，要么在整个图中，要么在核外/分布式处理的情况下在分阶段到本地内存的整个子图中。使用 DS, FlashMob 不会改变直接采样语义或操作复杂性，而是将范围限制在当前 VP 内的邻接表。

FlashMob 缓存友好的边缘采样的有效性主要由两个因素决定。首先,随机访问的工作集适合缓存，这主要取决于顶点分区的大小。注意，尽管 PS 和 DS 都涉及随机读取，但 PS 的工作集更小。相比之下，DS 需要在 VP 中缓存所有的边。因此，为了适应相同的缓存级别(例如，L2)， PS 允许比 DS 在高度顶点上更大的分区。

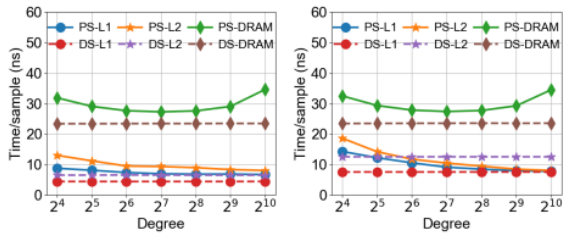
图 19 给出了具有统一度的合成 vp 的样本灵敏度结果，范围从 1024 到 16。图中给出了 walker/边缘密度为 1 和 0.25 时的样本阶段性能，在 PS(实线)和 DS(虚线)下，每个 vp 的大小分别适合 L1、L2、L3 和 DRAM(设置为 8×L3 容量)的工作集。从这些数字中可以得出以下几点结论:

1、这两种策略都受益于将工作集放入更快的缓存中。

2. 度数越高，PS 工作越快，因为度数越高的顶点会吸引更多的 walker，从而带来更高的顺序读取缓存线利用率。同时，所有的 DS 线对顶点度数不敏感。

3.当数据适合缓存时，这两种策略都受益于较高的 walker 密度，从而提高了缓存利用率。

4. 在具体的策略上，DS-L1 的效果最好，因为它涉及的访问较少，但是需要将一个大的图切割成大量的 vp。PS-L1 紧随其后，特别是在高阶顶点时，用于样本生产和消费的缓存线都得到了更好的利用。PS-DRAM 比任何其他组合都要慢得多



(a) 1 walker/edge (b) 0.25 walker/edge

图 19 实验对比

（4） 洗牌阶段

在每个样本阶段之后，walker 从每个 VP 中散开。在随后的 shuffle 阶段，所有线程并行工作以重新排列 walker，以便现在位于同一 VP 中的那些线程再次连续存储。对于高阶步行，边缘采样所需的额外每个 walker 数与步行者的当前位置一起洗牌。

通过这样做，FlashMob 交换了廉价的计算(流内存访问)，将 walker 数组的占用空间减少了一半。它节省了采样和洗牌阶段的主存带宽，以及大量的 DRAM 空间。后者反过来允许 FlashMob 在每一轮随机行走中容纳更多的 walker，最大化 walker 密度以更好地重用数据。

（5） 动态规划步行优化

作者提出了一种划分顶点和分配采样策略的原则方法，通过将其表述为组合优化问题并获得有效的最优动态规划解决方案。

主要的权衡在于 VP 的大小和数量:较小的 VP 适合更快的缓存，但收益很大程度上取决于 VP 内的平均度和行走密度等因素;更多的 vp 增加了 shuffle 开销，要么不适合缓存，要么增加了 shuffle 级别。

（6） Cross-socket walk

如今的普通服务器拥有具有多个插槽的 NUMA(非统一内存访问)体系结构，其中每个插槽上的 CPU 核心可以更快地访问“本地”内存。通过使用动态规划优化单套接字采样策略和 VP 大小，最后检查跨套接字 walk。更新许多独立步行者的独特性质使我们在分解工作时更灵活。为此，FlashMob 研究了以下两种发行模式。

使用第一种模式 flashmobp(“P”表示图分区)，vp 分布在套接字和 walker 数组之间。例如，在一个 2-socket 服务器上，socket 0 上的线程将始终对 vp 的前一半进行采样，并对 walker 的前一半进行洗牌。它唯一的远程内存访问发生在示例阶段，当它的 vp 需要处理分配给 socket 1 的当前驻留在 socket 0 vp 上的 walker 时。由于 walker 已经洗牌，这样的远程访问是严格的仅流访问，避免了超级昂贵的远程随机访问。由于洗牌阶段比样本阶段执行更多的扫描，保持其访问本地可以减少总体远程内存流量。

第二种模式是 FlashMob-R(“R”代表图形复制)，尽可能避免远程内存访问。当整个图以及预采样边缘缓冲区等辅助数据适合于单插座 DRAM 时，人们可以简单地在每个插座复制这样的图数据，并同时运行多个独立的随机游走实例。对于中等大小的图，这消除了所有远程数据访问和跨套接字同步。然而，通过复制图形本身，FlashMob-R 用于行走者数组的内存更少，导致行走密度降低和缓存数据的重用率降低。

2.4.4 实验部分

图 20 给出了 5 张真实世界图上的整体游走性能，在每

步时间内，将 FlashMob 与两个基线系统进行比较。图 20a 显示了 DeepWalk 的结果，KnightKing 比 GraphVite 快 2.2-3.8 倍，因为它的边缘访问更有效。与此同时，FlashMob 的速度比 KnightKing 提高了 5.4 到 13.7 倍，在 40 秒内完成一个行走步骤。对于更小的图，它可以通过将更多的采样任务放入更快的缓存中来实现更快的速度。此外，除了最小的图(YT)， KnightKing 在其他图上提供了非常相似的性能。相比之下，FlashMob 在更小的图形上获得了额外的优化，逐步将其每步时间从 YH 上的 37 纳秒降低到 YT 上的 21 纳秒。

图 20b 给出了 node2vec 结果。二阶游走使得边缘采样更加复杂。尽管如此，FlashMob 还是比 KnightKing 加速了 3.9 到 19.9 倍。较小的收益空间主要是由于较低的访问位置，因为 node2vec 涉及 walker 的采样目的地(下一站的候选目的地)与其前一站之间的连通性检查。虽然 FlashMob 再次批量查找，但计算不再局限于单个 VP。

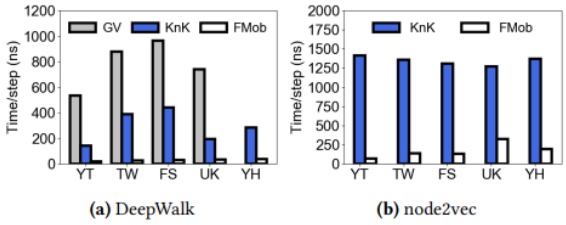


图 20 总体比较

接着作者评估 FlashMob 的 DP 算法的自动顶点划分和采样策略优化，结果如图 21a 所示。“其他”类别包括采样和洗牌阶段之外的所有成本，例如初始化和最终输出。时间分割表明了我们的优化面临的非平凡挑战:分区和变换通过在缓存中拟合工作集来实现快速采样，这反过来使得变换本身的相对成本与采样相当。

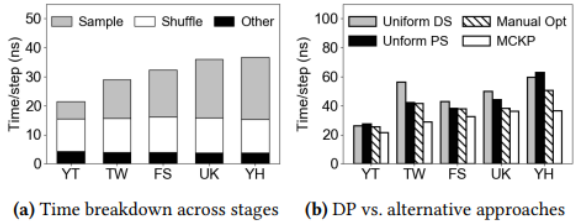


图 21 DP 算法优化

图 21b 比较了 dp 识别的解决方案与几个备选方案。作者测试了两种统一的划分策略，分别采用预采样(PS)和直接采样(DS)，将图切割成 2048 个等大小的 vp(以允许单层洗牌)。此外，还包括了“Manual Opt”，这是在确定 MCKP 映射之前在原型中采用的最佳划分和样本策略优化。结果表明，基于 dp 的 MCKP 解决方案的总体性能明显优于统一策略，证实了自适应调整数据和任务带来的改进幅度。正如预期的那样，它也显著优于我们的人工优化，人工优化是基于对低行走者密度或高度点采用 PS，对其他顶点采用 DS 的启发式方法。

2.4.5 总结

在这项工作中，作者推翻了长期以来的假设，即大图上的随机游走需要用随机 DRAM 访问来解决。作者提出的系统 FlashMob，通过仔细的划分、重新排列和批处理操作，成功地利用了随机游走语义中隐藏的空间和时间局部性。研究结果显示，现代服务器的缓存即使对于随机和不规则计算的数据驱动程序也能实现高效的“缓存外”处理。作者还发现，由于顺序和随机 DRAM 访问之间的巨大差异，通过执行更多的扫描来增加访问量是值得的，这可能最终会节省时间和实际的 DRAM 流量。

除了将 FlashMob 扩展到游走在磁盘上的图形之外，作者还在探索其他涉及抽样的途径，在不影响统计保证的情况下，将随机访问转换为顺序访问。

2.5 GraSorw

2.5.1 介绍

本论文出自 VLDB 22[5]，随机游走被广泛应用于许多图分析任务中，尤其是一阶随机游走。然而，作为现实世界问题的简化，一阶随机游走在建模数据中的高阶结构时表现不佳。最近，基于二阶随机行走的应用程序(例如 Node2vec，二阶 PageRank)变得越来越有吸引力。由于二阶随机游走模型的复杂性和内存限制，在一台机器上运行基于二阶随机游走的应用程序是不可扩展的。现有的基于磁盘的图系统只对一阶随机漫步模型友好，并且在执行二阶随机漫步时遭受昂贵的磁盘 I/O。本文介绍了一种 I/O 高效的基于磁盘的大型图可伸缩二阶随机游走图系统 GraSorw。首先，为了消除大量的轻顶点 I/O，作者开发了一个双块执行引擎，通过应用一种新的三角形双块调度策略、基于桶的行走管理和倾斜行走存储，将随机 I/O 转换为顺序 I/O。其次，为了提高 I/O 利用率，作者设计了一个基于学习的块加载模型，以充分利用满载和按需加载方法的优点。通过在六个大型真实数据集以及几个合成数据集上进行了广泛的实验。实证结果表明，与现有的基于磁盘的图形系统相比，GraSorw 中流行任务的端到端时间成本降低了不止一个数量级。

作者精心实现了 GraSorw，用于在单机上高效地处理大图上的二阶随机游走任务。在 6 个大型数据集上的实验结

果表明，在常见的二阶随机任务(如随机行走生成和使用 Node2vec 随机行走模型查询 PageRank)中，GraSorw 的效率提高了一个数量级以上。可以将作者的贡献总结如下：

（1）本文确定了在现有的基于磁盘的图处理系统上运行二阶随机游走模型的 I/O 效率低，并提出了一个 I/O 效率高的系统 GraSorw。

（2）本文提出了一个高效的双块执行引擎，它配备了一个三角形的双块调度策略，倾斜的行走存储和基于桶的内存行走管理，以消除大量的顶点 I/ O。

（3）本文提出了一个基于学习的块加载模型，以提高块 I/O 利用率，当一些行走仍然在桶中。

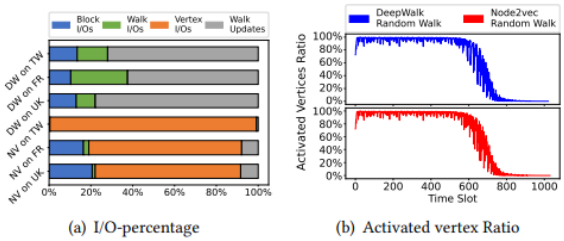
（4）将 GraSorw 与 SOGW 和 SGSC 在真实世界和合成大图上进行比较。结果表明，GraSorw 显著缩短了二阶随机游走任务的端到端时间，提高了 I/O 效率。

2.5.2 研究背景

如今，许多真实世界的图形以 CSR 格式占用了数百 Gb，这超过了大多数商用机器的 RAM 大小。由于内存的限制，使用基于内存的框架在单机上运行大型图上的随机游走模型是不可扩展的。许多通用的基于磁盘的图系统被提出用于在大图上进行一阶随机游走。他们最初将整个图划分为几个块，即子图。在执行过程中，这些系统将一个块加载到内存中，更新当前块中所有激活的顶点和边，并重复这一操作，直到满足某个终止条件。

然而，现有的基于磁盘的图形系统都没有考虑到二阶随机游走模型。在本文中，作者旨在设计一个可扩展的基于磁盘的图系统，用于在大型图上执行二阶随机游走模型。采用了将整个图处理成块的思想来解决内存不足的问题。设计这样一个系统的主要挑战是处理极端的 I/O 开销，这是双重的。

首先是海量的顶点 I/O。在现有的基于块的图系统中生成随机行走时，行走与包含其当前顶点的块相关联。由于块在处理遍历之前被加载到内存中，当前顶点及其邻居都在块中，并且在没有磁盘 I/O 的情况下更新第一个存储器随机遍历是有效的。但是，当处理二阶随机游走，需要当前顶点和前一个顶点的信息，虽然当前顶点很容易从内存中检索，但前一个顶点可能从磁盘中的任何其他块中检索，导致顶点 I/O。这些顶点 I/O 是随机且轻的，使得在现有的图系统上实现二阶随机游走模型的 I/O 成本非常高。图 22(a)用三个大型图数据集可视化了 SOGW 系统运行 DeepWalk(即一阶)和 Node2vec(即二阶)随机游走的代价，我们将代价分解为块 I/ O、行走 I/ O、顶点 I/ O 和行走更新代价。可以清楚地看到，在二阶随机游走任务中，效率瓶颈是顶点 I/ O 的代价。



(a) I/O-percentage　　(b) Activated vertex Ratio

图 22 SOGW 运行结果

第二个问题就是块 I/ O 利用率低。大多数现有的基于块的图形系统一次加载整个块。但是，当随机游走任务的工作量较轻，或者任务即将结束时，激活的顶点可能只是整个块的一小部分，导致块 I/ O 的浪费。图 22(b)显示了使用 LiveJournal 数据集在 SOGW 系统中运行 DeepWalk 和 Node2vec 随机行走任务时，激活顶点与时隙的比值。我们可以看到，在任务的最后(即大约最后 20%的时间段)，比值接近于零。为了解决低块 I/O 利用率的问题，DynamicShards 和 Graphene 动态调整了图块的布局，以减少无用数据的加载，但它们没有考虑随机游走特征。GraphWalke 根据随机行走的总次数来确定合适的块大小，以提高块 I/O 的利用率，但这种解决方案是静态的，当任务即将结束，块中只剩下少量的行走时就会失败。

2.5.3 GraSorw 设计

（1）基本思想

GraSorw 是一种 I/O 高效的基于磁盘的图形系统，用于可扩展的二阶随机游走。与前面的工作类似，图形和中间行走都存储在磁盘上。该图被划分为多个块，每个块与一个存储中间步行的步行池相关联。不同之处在于:为了减少大量的顶点和块 I/ O，我们设计了一个双块执行引擎和一个基于学习的块加载模型。图 23 描述了 GraSorw 的高级执行流。在执行过程中，双块执行引擎迭代地选择一个块作为当前块，使用基于学习的块加载模型将辅助块加载到内存中，并更新与当前块相关的中间遍历。接下来详细介绍了 GraSorw 在一个时隙中的执行流程。

在每个时隙中，①引擎使用基于桶的内存行走管理器将与当前块相关的中间行走加载到内存中，并将它们与内存行走池中的中间行走合并，形成当前行走。②然后，管理器将当前的步行分割成桶，每个桶存储具有相同块集的步行，其中对块包含它们先前和当前驻留的顶点。这样基于桶的内存中行走管理将大量的顶点 I/O 合并为单个块 I/O。

考虑到先前和当前驻留的顶点涉及到两个块，在处理一个桶之前，我们需要加载另一个块到内存中，称为辅助块。在 GraSorw 中，每个辅助块对应一个桶，而当前块在所有桶

之间共享。③在每个时隙中，双块执行引擎使用三角双块调度方法来确定辅助块的加载顺序，并使用基于学习的块加载模型来加载块。

加载块之后，④引擎异步更新桶中的行走。在这个桶中行走的当前顶点可以在当前块中，也可以在辅助块中，如蓝色和橙色箭头所示。此外，由于有一些边连接两个块，步行也可以更新跨越两个块。当移动到内存中不属于块的任何顶点或达到终止条件时，遍历的更新将停止。⑤对于前一种情况，需要步行持久性来保存这些中间步行的信息，以便将来更新。中间步行有两个地方可去。

其中大部分存储在带有倾斜行走存储的内存行走池中(，其他可能被移动到桶中，这是桶扩展策略引起的。当步行池的大小达到预定义的阈值时，内存中的步行池将被刷新到磁盘。在桶中的所有遍历都结束或持续之后，使用三角形双块调度选择下一个辅助块，并迭代执行相应的桶。

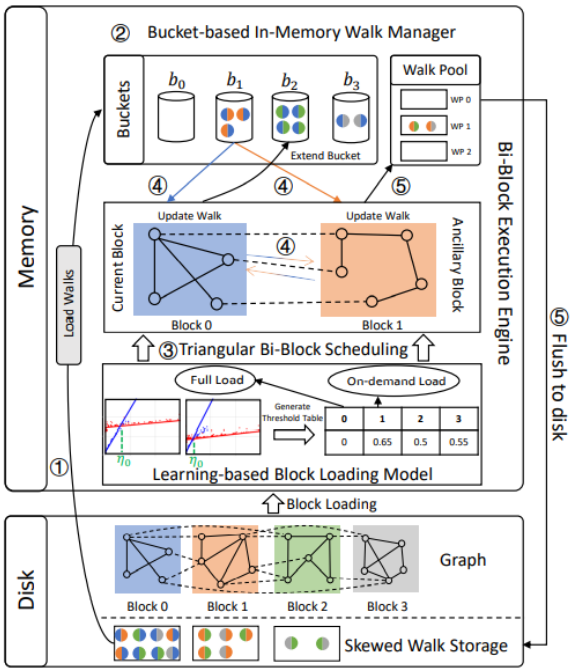


图 23 GraSorw 执行流程

GraSorw 中基于学习的块加载模型是为了在电流行走次数较小时提高 I/O 利用率而提出的。它使用一个线性回归模型，通过学习历史日志来预测成本，在模型的基础上，我们为每个区块推导阈值，并使用阈值来选择区块加载方法。详细信息将在第 5 节中介绍。在图 23 中，块 0 是完全加载的，块 1 是按需加载方法加载的。

（2）双块执行引擎

双块执行引擎的基本思想是在内存中保留两个块(即当前块和辅助块)，从而保证当前和以前的顶点都在内存中。

块 I/O 总数与两个因素有关:当前块 I/ O(即时隙)的数量和每个时隙中辅助块 I/ O 的数量。最小化块 I/ O 总数可以通过分别减少当前块 I/O 和辅助块 I/ O 来实现。然而获得当前块 I/ O 的最小数量是一个 np 困难的问题。

二阶随机游走的块访问序列是未知的，我们需要设计一个在线算法来解决上述问题。大多数现有的启发式在线解的最优解没有(或差)近似误差界。因此，我们对当前块的不同调度策略进行了实证研究，没有一种方法在所有数据集上都是最优的，同一方法在不同数据集上的性能可能有很大差异。但一般来说，基于迭代的方法，在大多数情况下达到最佳效果。基于这些观察，本文采用基于迭代的方法调度当前块，然后重点开发一种新的调度策略，优化辅助块 I/ O。

接下里首先介绍了支持三角形双块调度策略的倾斜行走存储，然后介绍了基于桶的内存行走存储，它有助于将随机顶点 I/ O 聚集到块中。

传统的步长存储方法将步长与其当前顶点所属的块关联起来。这在三角形双块调度策略下进行更新时带来了限制。GraSorw 中的倾斜行走存储同时考虑了行走的前一个顶点和当前顶点来安排行走。与传统的 walk 存储相比，这种存储将当前顶点属于同一块的 walk 分成两组。一组包含当前顶点属于比以前顶点所属的 ID 更大的块的行走;另一组包含剩余的步行。因此，在三角双块调度策略中，当当前顶点所在的块作为辅助块加载时，处理第一组，当当前顶点所在的块作为当前块加载时，处理第二组。

为了将随机顶点 I/ O 合并到块 I/ O 中，基于桶的内存行走管理器将当前的行走分割成桶，每个桶存储具有相同块集的行走，其中对块包含它们先前和当前驻留的顶点。也就是说，桶收集也依赖于行走的当前顶点和前一个顶点。此外，结合倾斜行走存储，如果行走按照其前一个顶点收集到 bucket 中，则其当前顶点所属块的 ID 小于前一个顶点的 ID，反之亦然。此步行管理支持三角形双块调度策略。

（3）基于学习的块加载模型

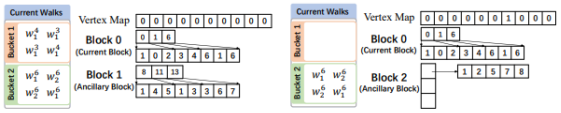
大部分的块 I/ O 是由辅助块加载引起的。当一个块中只有一小部分顶点在行走时，可能会导致块 I/ O 的浪费。为了提高 I/ O 利用率，作者在 GraSorw 中引入了满负载和按需负载两种块加载方法，并提出了一种基于学习的模型，根据运行时统计数据自动选择辅助块的块加载方法。

作者在这里提出了两种块加载模型。第一种是满载模式。这种方法在现有的基于磁盘的图形系统中已经被广泛使用，它意味着将整个块一次性加载到内存中。在 GraSorw 中，对应块的 Index File 和 CSR File 的切片被加载到内存中。

第二种是按需加载模式，这个方法意味着只有对应块

中的激活顶点被加载到内存中。加载一个块 B 时，首先检查行走中每个行走的当前顶点和前一个顶点，并记录属于该块的所有顶点。这些顶点是激活顶点，将用于更新行走。然后只加载与激活顶点相关的 CSR 分段。在 GraSorw 中，按需加载发生在每个桶的执行之前。

如图 24 所示，通过一个例子比较了满负荷和按需负载之间的 I/O 差异。假设当前有 8 次遍历，每个存储在 Index 文件和 CSR 文件中的值在磁盘上占用 4 个字节。系统决定以按需加载方式加载块 2，以满载方式加载块 1。顶点映射用于记录激活的顶点。由于 block 1 被决定用 full-load 方法加载，如图 24(a)所示，索引文件和 CSR 文件的整个切片被加载到内存中，导致 32 字节的 I/O。在桶 1 中执行遍历更新后，块 1 的内存被释放。在执行桶 2 之前，将扫描桶 2 中的所有行走，以计算区块 2 的激活顶点。在图 24(b)所示的例子中，只需要顶点 6 的信息，所以系统只需要将顶点 6 的 CSR 分段加载到内存中，只需要 20 个字节的 I/O。加载辅助块的 CSR 信息总共需要 52 个字节的 I/O。但是，如果使用纯 full-load 方法同时加载 block 1 和 block 2，则会产生 64 字节的磁盘 I/O。在本例中，混合使用 full load 和 on-demand load 方法可节省 18.8%的块 I/O。



(a) Full-load of block 1. (b) On-demand-load of block 2.

图 24 不同加载方式对比

此外，不需要分配内存来存储 block 2 的索引文件的切片。这个例子表明在两个块加载方法之间进行权衡是值得的。

为块自动选择加载方法的关键是估算相应的成本。然而，由于激活顶点的数量与任务相关，并且两种块加载方法之间的数据结构不同也会影响效率，因此很难开发成本模型的启发式方法。在本文中，作者开发了一个基于学习的模型来预测每种加载方法的成本。当扩展新激活的顶点时(执行阶段)，遍历更新会招致新的磁盘 I/ O。与 full-load 模式相比，按需加载的加载阶段可能会更短，但由于增加了新的磁盘 I/ o，执行阶段可能会变长。因此，我们将这两个阶段作为一个整体来进行成本估算。具体的训练方法详见原文。

（4） 图形划分

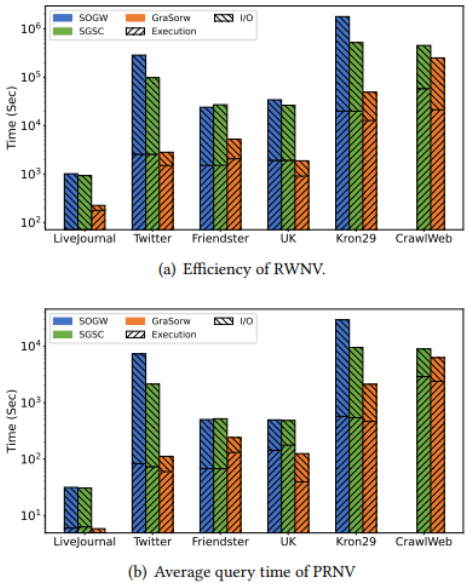
GraSorw 默认的图分区方法是顺序分区，它将顶点按照 ID 的顺序划分到不同的块中，并确保所有的块都符合预定义的块大小。GraSorw 还支持自定义图分区，用户需要提供相应的边表文件和块文件。边列表文件由图中的所有边组成，块文件表示每个顶点所属的块。

对于随机行走任务，最好在一个块中更新尽可能多的步骤。增加块的密度提供了帮助，因为在这个块中顶点之间有更多的边，并且与稀疏块相比，行走更有可能停留在块内部。METIS 是一种流行且经典的图划分算法，旨在增加每个块的密度。

2.5.4 实验部分

作者在实验部分详细比较在 GraphWalker 上实现的两个变体来展示 GraSorw 的端到端性能。然后分别讨论了本文提出的两种主要技术。对于双块执行引擎，作者将其性能与普通桶模型进行比较，后者是基于简单的基于桶的遍历收集思想实现的。对于基于学习的块加载模型，作者在 GraSorw 中使用该模型验证了 I/O 利用率和执行效率的提高。然后测试了不同图划分方法的影响。他们还展示了 GraSorw 的参数敏感性，包括行走分布和块大小的变化。此外，作者还了 GraSorw 在一组具有不同图分布的合成数据集上的效率。在这里我简单列举一下端到端性能的比较。

作者首先评估 GraSorw 与两个基线系统 SOGW 和 SGSC 的整体性能。SOGW 和 SGSC 由于效率低，在处理除 LiveJournal 外的大型图时，无法在合理的时间约束下完成标准参数的任务。对除 LiveJournal 之外的图进行了如下估计：根据 GraphWalker 中的实证研究，当无法将整个图放入内存时，总时间随步行长度线性增加。此外作者还发现在 SOGW 上运行二阶随机行走任务时，由于之前的顶点信息需要从磁盘中检索，这占了大部分时间，因此总时间也随着行走次数线性增加。



(a) Efficiency of RWNV.

(b) Average query time of PRNV

图 25 总体性能对比

图 25 给出了 RWNV 和 PRNV 在各种图上的结果。CrawlWeb 上的 SOGW 结果缺失，因为用于估计总时间的小规模任务无法在两天内完成。根据这个小范围任务执行时间的下限，作者估计对于 RWNV 和 PRNV, SOGW 都不能在两周内完成完整的任务。对于 SGSC，顶点缓存初始化的时间包含在 I/O 时间中。在三个系统中，可以看到 GraSorw 在所有这些图上的两个任务中都达到了最好的性能。特别是在 Twitter 上，SOGW 完成 RWNV 任务需要两天多的时间，而 GraSorw 只需要 47 分钟，速度提高了 95 倍。在占用数百 GB 的 CSR 格式(如 Kron29)的更大的图上，执行二阶随机行走任务更具挑战性，因为根据 SOGW 的评估，传统的基于磁盘的方法大约需要 20 天。幸运的是，在 GraSorw 的帮助下，半天就可以完成这样的任务，这是更加合理的。

在 CrawlWeb 上，与 SGSC 相比，GraSorw 的 RWNV 和 PRNV 的速度分别提高了 1.81 倍和 1.43 倍，而 CSR 格式的 CrawlWeb 占用了近 900GB 的内存。在大多数图中，SGSC 比 SOGW 略快，这是因为内存中存在静态顶点缓存，这使得从内存中检索一些重要顶点的信息成为可能，而不是通过调用顶点 I/ O。然而，SGSC 在 Friendster 上运行这样的任务需要更多的时间。一个可能的原因是，对于 Friendster 来说，SGSC 中的缓存命中率很低。初始化静态顶点缓存的时间比从它的好处中节省的时间长。从 SGSC 和 GraSorw 的结果对比可以看出，对于固定的内存大小，使用三角双块调度来加载块(即辅助块)比利用内存空间来存储尽可能多的大次顶点更有效。总体而言，GraSorw 在 RWNV 任务中实现了 1.81× ～95×的性能提升，在 PRNV 任务中实现了 1.43 × ～19.1×的性能提升。

在图中，作者还展示了每个结果的时间成本细分，显示为执行时间和 I/O 时间。可以看到，在 GraSorw 中，每个任务在所有图上的 I/O 时间成本都显著降低了。GraSorw 在 Twitter 上最大程度地降低了 I/O 开销，RWNV 和 PRNV 的效率分别比 SOGW 提高了 213 倍和 138 倍。在 SOGW 中，昂贵的 I/O 成本来自于大量的轻顶点 I/O，而在 GraSorw 中，通过桶和辅助块，将这些用于检索行走的前一个顶点信息的顶点 I/O 转换为块 I/O，效率更高。

2.5.5 总结

二阶随机游走是数据中高阶依赖关系建模的重要方法。现有的基于磁盘的图系统不能有效地支持二阶随机游走。作者提出了一个 I/O 高效的基于磁盘的二阶随机游走系统。为了减少大量的轻顶点 I/ O，作者开发了一个具有三角形双块调度策略的双块执行引擎，它巧妙地将小的随机 I/ O 转换为大的顺序 I/ O。为了提高 I/ O 利用率，引入了一种基于学习的块加载模型来自动选择合适的块加载方法。最后，在五个大图上对提出的系统进行了经验评估，结果表明 GraSorw 显著优于现有的基于磁盘的随机游走系统。此外，考虑到大多数应用程序中二阶随机行走的处理是一个独立的阶段，GraSorw 可以很容易地嵌入或集成到现有的基于二阶随机行走的应用程序中。

3. 总结

随机游走技术是一种提取实体之间信息的强力工具，值得我们去深入研究。本文中我主要介绍了五种随机游走引擎技术，他们各有特色。GraphWalker 是一种用于随机行走的 I/O 高效图形系统，通过部署一种具有异步行走更新的新型状态感知 I/O 模型，可以有效地处理由数千亿条边组成的非常大的圆盘图，而且它还可以扩展到运行数百亿个具有数千步长的随机行走。ThunderRW 是一个通用且高效的内存 RW 框架。通过采用了一个以步长为中心的编程模型，从步行者移动一步的局部视图中抽象出计算，同时还提出了步进交错技术来解决缓存失速问题。KnightKing 是一个通用的分布式图随机漫步引擎，采用了直观的以 walker 为中心的计算模型，围绕创新的基于拒绝的抽样机制，极大地降低了高阶随机漫步算法的成本。FlashMob 通过使内存访问更加有序和规则，提高了缓存和内存带宽的利用率。GraSorw 是一种 I/O 高效的基于磁盘的大型图随机游走系统。通过开发了一个双块执行引擎与基于学习的块加载模型，与现有的基于磁盘的图形系统相比提升不少性能。

4. 参考文献

[1] Wang, R. , et al. "GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks." USENIX Annual Technical Conference 2020.

[2] Sun, S. , et al. "ThunderRW: An In-Memory Graph Random Walk Engine (Complete Version)." (2021).

[3] Yang K , Zhang M , Ma X , et al. KnightKing: a fast distributed graph random walk engine. 2019.

[4] Ke Y, Xiaosong Ma , Kang Chen , et al. Random Walks on Huge Graphs at Cache Efficiency. 2021.

[5] Li, H. , et al. "An I/O-Efficient Disk-based Graph System for Scalable Second-Order Random Walk of Large Graphs." (2022).