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：随机游走框架综述

学　　号 M202173693

姓　　名 宋启旺

专　　业 电子信息

课程指导教师 施展 童薇

院（系、所） 计算机科学与技术学院

2021 年 1 月 4 日

随机游走框架综述

宋启旺[1)]

[1)](华中科技大学计算机科学与技术学院，武汉 430074)

摘 要 图上的随机游走作为一种图数据分析和机器学习的工具，最近获得了巨大的人气。目前，随机游走算法是作为单独的实现而开发的，并且存在明显的性能和可扩展性问题，特别是在复杂游走策略的动态特性下。本文介绍的三种框架支持各种随机游走，并对随机游走的性能以及扩展性等问题做了不同程度和不同方面的优化。具体包括 KnightKing、GraphWalker 以及 ThunderRW。其中 KnightKing 是第一个通用的、分布式的图随机游走引擎，它采用了一个直观的以游走者为中心的计算模型以及拒绝采样方法来加快边缘采样速度，并且提出了一个统一的转变概率定义，允许用户轻松地指定现有的或新的随机游走算法。 GraphWalker 采用了一个新的状态感知的 I/O 模型，它利用每个随机游走的状态，优先将游走次数最多的图块从磁盘加载到内存，从而提高 I/O 的利用率；GraphWalker 还采用了基于重入法的异步游走更新方案，允许每个游走尽可能多地移动，以充分利用加载的子图，大大加快随机游走的进度；还提出了一个轻量级的以块为中心的索引方案来管理游走状态，并采用固定长度的游走缓冲策略来减少记录游走状态的内存成本。ThunderRW 采用了一个通用的以步骤为中心的编程模型 GMU，以抽象出不同的 RW 算法；基于 GMU 模型，开发了步骤交错技术，通过切换不同随机游走查询的执行来隐藏内存访问延迟。

关键词 随机游走；KnightKing；GraphWalker；ThunderRW；

中图法分类号 TP302

Overview of Random Walk Engine

Song Qiwang[1)]

[1)](School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074)

Abstract Random walk on graphs has recently gained immense popularity as a tool for graph data analytics and machine learning. Currently, random walk algorithms are developed as individual implementations and suffer significant performance and scalability problems, especially with the dynamic nature of sophisticated walk strategies. The three frameworks presented in this paper support a variety of random walks and are optimized for different degrees and aspects of random walk performance and scalability issues. Specifically, KnightKing, GraphWalker, and ThunderRW, of which KnightKing is the first general-purpose, distributed graph random walk engine that employs an intuitive walker-centric computation model and rejection sampling methods to speed up edge sampling, and proposes a uniform definition of transition probabilities that allows users to easily specify existing or new random walk algorithms. GraphWalker adopts a novel state-aware I/O model, which leverages the state of each random walk to preferentially load the graph block with the most walks from disk into memory, so as to improve the I/O utilization. GraphWalker also adopts an asynchronous walk updating scheme based on the re-entry method, which allows each walk to move as many steps as possible so as to fully utilize the loaded subgraph and greatly accelerate the progress of random walks, and A lightweight block-centric indexing scheme is also proposed to manage the walk states, and a fixed-length walk buffering strategy is used to reduce the memory cost of recording the walk states. ThunderRW adopts a generic step-centric programming model named GMU, to

abstract different RW algorithms; based on the GMU model, a step interleaving technique is developed that hides memory access latency by switching the execution of different random walk queries.

Key words Random Walk; KnightKing; GraphWalker; ThunderRW;

1 引言

本文将介绍三篇近三年的 CCF/A 类文章：文章 1：《KnightKing: A Fast Distributed Graph Random Walk Engine》，文章 2：《GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks》，文章 3：《ThunderRW: An In-Memory Graph Random Walk Engine》。下面对这三篇文章做一个总体性的概述。

随机游走是一个基本的和广泛使用的图处理任务，是提取图中实体之间关系的有效工具，被广泛用于许多应用中，如个性化网页排名（PPR）、SimRank、Random Walk Domination、Graphlet Concentration（GC）、Network Community Profiling（NCP）、DeepWalk 以及 Node2Vec。对于像 GC 和 NCP 这样的图分析任务，RW 查询通常在成本上占主导地位。即使对于图表示学习来说，RW 的采样成本也是非同小可的。此外，增加 RW 查询的数量可以提高 RW 算法的有效性。因此，加速 RW 查询是一个重要的问题。

RW 算法通常遵循算法 1 中说明的执行范式，它由大量的 RW 查询组成。每个查询 Q 从一个给定的源顶点开始。在每一步，Q 随机移动到当前所在顶点的一个邻居，并重复这一过程，直到满足特定的终止条件，例如，达到目标长度。尽管 RW 算法遵循类似的执行范式，但 RW 算法有相当多的变体，它们在邻居的选择上可能有很大的不同。最近出现的专门为 RW 算法设计的系统，包括 KnightKing、GraphWalker 和 ThunderRW。KnightKing 对随机游走进行了优化，并且可以在集群机器上运行；GraphWalker 对 I/O 效率进行了优化；ThunderRW 对 RW 查询对内存执行进行了优化。

Algorithm 1: Execution Paradigm of RW algorithms

```
Input: a graph G and a set ℚ of random walk queries;
Output: the walk sequences of each query in ℚ;
foreach Q ∈ ℚ do
    do
        Select a neighbor of the current residing vertex Q.cur at random;
        Add the selected vertex to Q;
    while Terminate(Q) is false;
return ℚ;
```

KnightKing 是第一个用于图随机行走对通用框架。它可以被看作是一个“分布式随机游走引擎”，是传统图引擎的随机游走对应物。KnightKing 采用以游走者为中心的视角，用 API 来自定义边缘转换概率，同时处理常见的随机游走基础设施。与图形引擎类似，KnightKing 隐藏了图形分区、顶点分配、节点间通信和负载平衡的系统细节。因此，它促进了一个直观的“像游走者一样思考”的观点。

KnightKing 的效率和可扩展性的核心是它能够快速选择下一条要跟随的边。我们首先提出了一个统一的转换概率定义，它允许用户直观地自定义随机游走算法的静态（与图相关）和动态（与游走者相关）转换概率。基于这个算法定义框架，我们建立了 KnightKing 作为第一个执行拒绝采样的随机游走系统。KnightKing 消除了扫描游走者当前所在顶点的所有出边以重新计算其转换概率的需要。直观地说，这个步骤对于动态随机游走是必要的，因为如果不评估其相对于其兄弟的转换概率，就不能继续对一条边进行采样。然而，通过拒绝抽样，我们只需在顶点 v 进行一些单独和快速评估的试验，就可以取代这种完整的 O(|Ev|)检查。

表 1 显示了区别：对于相同的节点 2vec 计算，KnightKing 在抽样一条边时平均只需要计算 0.79 条边的转换概率。这大大加快了边缘采样的速度，特别是在现实世界图中常见的幂律度分布的情况下：拥有数千甚至数百万条边缘的热门顶点可以以类似于其更不受欢迎的顶点的成本走入和走出，使它们免于成为频繁的、超昂贵的顶点。另外，与现有的近似优化不同，KnightKing 执行精确采样，在不牺牲正确性的情况下提高了性能。

Graphs	Degree mean	Degree variance	Full-scan average overhead	KnightKing's average overhead
Friendster	51.4	1.62E4	361 edges/step	0.77 edges/step
Twitter	70.4	6.42E6	92202 edges/step	0.79 edges/step

Table 1. Node2vec sampling overhead

最后，尽管拒绝采样本身是为动态采样而设计的，但 KnightKing 是一个通用的框架，也能有效地处理静态采样。除了上述算法创新，KnightKing 还包括系统设计选择和优化，以支持其以步行者为中心的编程模型。

为了解决 I/O 效率问题，以便有效地支持快速和可扩展的随机漫步，我们开发了 GraphWalker，这是一个 I/O 效率高且资源友好的图系统。GraphWalker 主要侧重于通过开发一个具有异步行

走更新功能的状态感知 I/O 模型来提高 I/O 效率。它还利用了一个轻量级的以块为中心的步行管理方案来提高内存效率。GraphWalker 的贡献如下。

- 开发了一个新的状态感知的 I/O 模型，它利用每个随机行走的状态，优先将行走次数最多的图块从磁盘加载到内存，从而提高 I/O 的利用率。我们还提出了一个考虑到步行的缓存方案来提高缓存效率。
- 开发了一个新的状态感知的 I/O 模型，它利用每个随机行走的状态，优先将行走次数最多的图块从磁盘加载到内存，从而提高 I/O 的利用率。我们还提出了一个考虑到步行的缓存方案来提高缓存效率。
- 提出了一个轻量级的以块为中心的索引方案来管理行走状态，并采用固定长度的行走缓冲策略来减少记录行走状态的内存成本。我们还开发了一个基于磁盘的行走管理方案，并使用异步分批 I/O 将行走状态写回磁盘，以支持在巨大的图上并行运行大规模的随机行走。
- 实现了一个原型，并进行了广泛的实验来证明其效率。结果显示，GraphWalker 与随机漫步专用系统 DrunkardMob 以及两个最先进的单机图系统 Graphene 和 GraFSoft 相比，可以实现超过一个数量级的速度提升。此外，GraphWalker 对资源更加友好，其性能甚至可以与在机器集群上运行的最先进的分布式随机行走系统 KnightKing 相媲美。

ThunderRW 是一个通用且高效的内存中 RW 框架。采用了一个以步骤为中心的编程模型，将计算从移动步行者的一个步骤的局部视图中抽象出来。用户通过在用户定义的函数（UDF）中“像游走者一样思考”来实现他们的 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视为一个任务单元来并行执行。此外，ThunderRW 还提供来不同的采样方法，以便用户可以根据工作负载的特点选择合适的方法。在以步骤为中心的编程模型的基础上，提出了步骤交错技术，以解决由不规则内存访问引起的缓存滞后问题。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，这就利用来不同 RW 查询之间的内存级并行性。

接下来，本文将系统的介绍这三个框架。

2 原理和优势

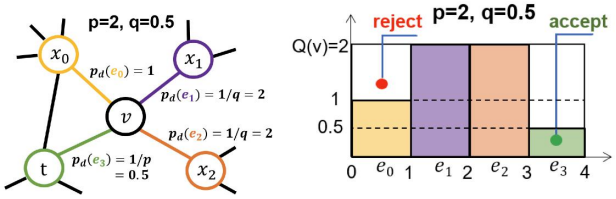
这一节将分别介绍 KnightKing、GraphWalker 以及 ThunderRW 的原理和优势。

2.1 KnightKing

首先确定一个定义边缘转换概率的一般框架，它适用于已知的随机游走算法。对于目前所在顶点 v 的游走者 w 来说，沿其边缘 e 的非标准化转换概率被定义为静态部分 Ps、动态部分 Pd 和扩展部分 Pe 的乘积。更具体来说，转换概率 P(e)=Ps(e)*Pd(e,v,w)*Pe(v,w)。注意，w 的状态带有必要的历史信息，如之前访问的 n 个顶点。

其次介绍一下 KnightKing 的关键创新：其统一的边缘采样机制，毫不费力地处理昂贵的动态/高阶游走，同时在静态游走中自动变形为别名解决方案。以拒绝采样为中心，它只需要计算几条边的非标准化转换概率，即使当前顶点有百万条边。同时，它仍然是一个精确的解决方案，在不失去正确性的情况下提供快速采样。

考虑图 2a 所示的样本图段上的无偏 Node2vec。假定一个游走者当前顶点为 v，v 之前已经访问过 t。动态非标准化概率分量 Pd(e)是 1/p，1，或 1/q，取决于 x 和 t 之间的距离。在样本参数设置为 p=2 和 q=0.5 时，四条边（分别通向 x0、x1、x2 和 t）的 Pd 值分别为 1、2、2 和 0.5。图 2b 用一个简单的离散概率分布图说明了这一点，条形高度对应于边的 Pd 值。



(a) Sample graph **(b)** Rejection sampling at v

Figure 2. Rejection sampling for unbiased node2vec

拒绝抽样的基本思想是找到一个覆盖所有条形的包络体 Q(v)，在包络体覆盖的区域内，将边缘间的一维抽样问题转换成二维问题。在 KnightKing 中，我们把 Q(v)定义为每个顶点的常数，直观地把它画成一条横线，与所有边上的最高 Pd 值相匹配。在这种情况下，Q(v) = max(1/p, 1,1/q) = 2，如图 2b 所示。

为了对一条边进行采样，我们在 y=Q(v)和 x=|Ev|这两条线所覆盖的矩形区域内，沿着 x 轴和 y

轴随机采样一个均匀分布的位置（x,y）。也就是说，想象在这个矩形区域内投掷飞镖。采样的 x 值给出一个候选边 e，而 y 值将与它相应的 Pd 进行比较。如果 y≤Pd(e)（飞镖击中了一个条形），e 被接受为一个成功的样本；否则（飞镖错过了所有的条形），e 被拒绝，这需要另一个采样试验，直到成功。

这种方法的优点在于，对于动态随机行走，转换概率 Pd 取决于行走者的状态，不能有效地预先计算，这种基于拒绝的取样允许人们先取样，然后检查取样是否能被接受。这个看似微小的差别消除了对当前顶点的所有边的昂贵扫描，这需要更新边之间的相对采样概率。在一个合理的包络中，有很大的机会在几次试验中取样成功（每次用 O(1)的时间来检查实际的动态概率，只针对取样的边）。由于现实世界中常见的幂律度分布，这种方法有效地消除了顶点之间的差异，极大地降低了有数千甚至数百万条边的顶点的采样成本。

KnightKing 迭代中的步骤：

1. 步行者产生用于拒绝采样的候选边并进行初步筛选。

2. 必要时，步行者根据采样结果发出步行者到顶点的状态查询。

3. 所有节点处理状态查询并发回结果。

4. 步行者检索状态查询结果。

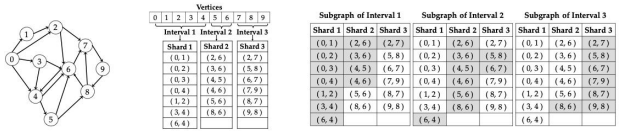
5. 步行者决定取样结果，如果成功则移动。

注意，以上描述的是 KnightKing 支持的随机行走的最一般情况。对于不那么复杂的算法，经常可以跳过一些步骤。例如，对于静态或一阶随机行走，第 2-4 步可以省略，因为不需要让其他顶点参与本地采样。对于这样的算法，所有的步行者可以锁步移动：在每个迭代中，步行者执行他们的采样和（局部）检查，直到一条边被成功采样（或步行终止）。在这些情况下，所有活跃的步行者都处于其步行序列的相同步骤。然而，对于像 node2vec 这样的高阶算法，跨行者的迭代是通过行者到顶点的两轮查询消息传递来同步的。采样成功的幸运步行者将继续向前走一步，而不太幸运的步行者则停留在他们当前的顶点进行下一次迭代。

2.2 GraphWalker

GraphWalker 的目标是不仅支持非常多的游走，比如几百亿次的游走，而且支持非常长的游走，比如每次游走都有几千步。为了实现这一目标，GraphWalker 采用一个状态感知模型，利用每个游走的状态，例如，游走所停留的当前顶点。简而言之，与基于迭代的模型不同，该模型盲目地按顺序加载图块，状态感知模型选择加载包含最大数量游走的图块，并使每个游走尽可能多地移动，直到它到达加载子图的边界。

通过这样做，游走可以在每次 I/O 中尽可能多地得到更新。因此，低 I/O 利用率和低步行更新率的问题都可以得到有效解决。



(a) Example graph (b) Data organization in shards (c) Graph loading with parallel sliding window

Figure 1: Storage and I/O model in GraphChi

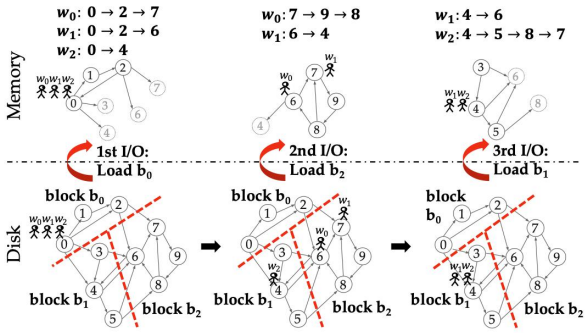


Figure 4: Main idea of the state-aware model

为了进一步说明上述想法并分析其好处，考虑图 1（a）中的例子图。假设我们要运行三个随机游走，这些游走从节点 0 开始，要移动四步。图 4 显示了用状态感知模型进行图加载和游走更新的过程。具体来说，在第一次 I/O 中，图块 b0 被加载到内存中，因为它包含了所有三个步行。在加载图块 b0 后，游走 w0 和 w1 移动了两步，w2 只移动了一步，因为它需要其他不在内存中的图块来游走更多的步骤。由于两个游走落入图块 b2，在第二次 I/O 中，图块 b2 被加载到内存中，游走 w0 完成，w1 可以移动一步。最后，剩下的两个游走都在块 b1 中，所以我们把 b1 加载到内存中，所有的游走都可以完成。注意，在这个例子中只需要三次 I/O。然而，对于基于迭代的模型，它可能需要 12 次 I/O，因为它使用了四个迭代，并且在每个迭代中产生三个 I/O。

基于上述想法，开发了一个 I/O 高效的图系统 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) 状态感知的图加载，(2) 异步游走更新，以及(3) 以块为中心的游走管理。图 5 也说明了 GraphWalker 的整体设计。

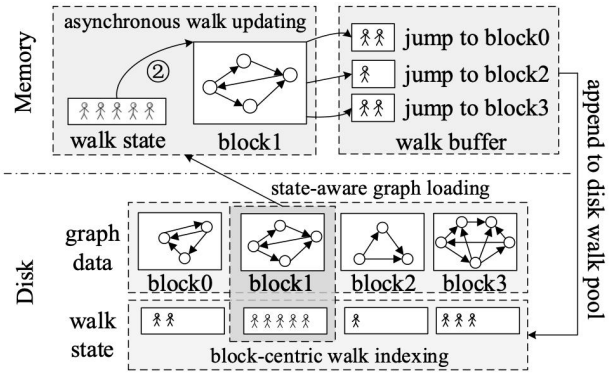


Figure 5: Overall design of GraphWalker

2.3 ThunderRW

为了抽象出 RW 算法的计算，我们在本文中提出了以步骤为中心的模型。我们观察到，RW 算法是建立在一些 RW 查询之上的，而不是单一的查询。尽管查询内的并行性有限，但由于每个 RW 查询可以独立执行，所以 RW 算法中存在大量的查询间并行性。因此，我们以步骤为中心的模型从查询的角度抽象出 RW 算法的计算，以利用查询间的并行性。

具体来说，我们从移动查询 Q 的一个步骤的局部视图来模拟计算。然后，我们把 Q 的一个步骤抽象为收集-移动-更新（GMU）操作，以描述 RW 算法的共同结构。通过以步骤为中心的模型，用户通过 "像步行者一样思考 "来开发 RW 算法。他们专注于定义设定 e ∈ Ev 转换概率的函数，并在每一步更新 Q 的状态，而该框架有利于将用户定义的面向步骤的函数应用于 RW 查询。

基于以步骤为中心的模型，ThunderRW 用 GMU 的操作来处理查询 Q 的一个步骤。在该模型下，随机内存访问的来源主要有两个。首先，Move 操作会随机挑选一条边，并沿着所选的边移动 Q。第二，用户定义的函数中的操作可以引入缓存缺失，例如 Node2Vec 中的距离检查操作。由于用户空间中的操作（即用户定义的函数）是由 RW 算法决定的，并且可以非常灵活，所以我们针对的是系统产生的内存问题（即移动操作）。在剖析结果的激励下，建议使用软件预完善技术来加速 ThunderRW 的内存计算。然而，查询 Q 的一个步骤并没有足够的计算工作量来隐藏内存访问延迟，因为 Q 的步骤有依赖关系。因此，我们建议通过交替执行不同查询的步骤来隐藏内存访问延时。

具体来说，给定 Move 中的操作序列，我们将其分解为多个阶段，这样一个阶段的计算会消耗之前阶段产生的数据，如果有必要，它还会为后续阶段检索数据。我们同时执行一组查询。一旦查询 Q 的一个阶段完成，我们就切换到该组中其他查询的阶段。当其他查询的阶段完成后，我们恢复 Q 的执行。通过这样的方式，我们把内存访问延迟隐藏在单个查询中，并保持 CPU 的忙碌。我们称这种方法为步骤交错。

例 5.1. 图 2 给出了一个例子，一个步骤被分为四个阶段。如果按部就班地执行一个查询，那么 CPU 就会因为内存访问而经常停滞。即使有预取，一个阶段的计算也不能掩盖内存访问的延迟。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。

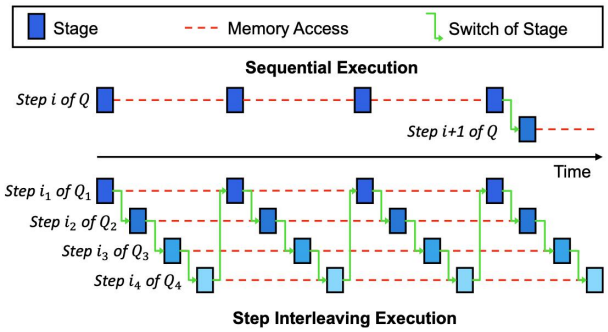


Figure 2: Sequential versus step interleaving.

3 研究进展

3.1 KnightKing

3.1.1 图形存储和分区

KnightKing 使用压缩稀疏行（CSR）存储边，这是一种在图系统中常用的紧凑数据结构。此外，考虑到游走者直接访问顶点的任何边是很重要的（例如，用拒绝采样执行局部动态转换概率检查），KnightKing 采用了顶点分区方案。每个顶点在分布式执行中被分配给一个节点，所有的有向边与它们的源顶点一起存储。无方向的边被存储两次，在两个方向。

一般来说，一个顶点在随机行走中被访问的频率取决于用户定义的（可能是动态的）转换概率，并涉及图的拓扑结构和游走者行为之间复杂的相互作用。在 KnightKing 中，我们粗略估计处理工作量为一个节点的本地顶点和边计数的总和，并在各节点间执行 1-D 分区平衡这个总和。虽然这可能不会产生均匀分布的随机行走处理或通信负载，但它实现了均匀的内存消耗，内存容量不足首先是分布式处理的主要原因。

3.1.2 随机游走的执行和协调

计算模型：由于游走者独立移动，随机游走算法可能会出现令人尴尬的并行和无协调，很容易扩展到更多的线程/节点。然而，一个天真的实现，如使用图数据库作为存储后端，每个步行者通过 getVertex()和 getEdges()等 API 检索信息，将产生过高的网络流量和糟糕的数据定位。相反，KnightKing 采用了图引擎中常见的 BSP（Bulk Synchronous Parallel）模型，这与它的迭代过程非常吻合。

在行走开始之前，KnightKing 按照用户指定或默认设置进行初始化。这包括建立每个顶点的别名表（如果定义了自定义静态分量 Ps），使用提供的上限和可选的下限（如果定义了自定义动态分量 Pd）设置拒绝采样，以及步行器实例化/初始化。

以游走者为中心的主要步行迭代的执行利用了类似于分布式图引擎中的支持：每个游走者（而不是每个顶点）的消息生成、目标节点查找和消息批处理，以及所有到所有的消息传递。还有一些常见的优化，如缓冲池管理和管道化，以使计算和通信重叠。

任务调度：KnightKing 以类似于 Gemini 这样的图引擎的方式执行任务调度，但以游走者为中心而不是以顶点为中心的方式工作。它在每个节点内设置并行处理，其线程数量与执行计算的可用核心数量相同，加上两个专门用于消息传递的线程。高阶随机游步中的两轮通信是在每个迭代中实现的，计算（生成传出的查询信息和处理传入的查询）与消息传递重叠。任务，定义为步行者或消息的块，被放入共享队列，供线程抓取。这种动态调度的粒度（块大小）被设定为 128，对游走者和消息都是如此。

落伍者处理：由于其基于拒绝的核心策略将采样复杂度降低到接近 O(1)，KnightKing 使每一步的采样成本大幅提高，不仅降低了可预测性。然而，它仍然可能面临长尾执行，少数散兵游勇比大部分步行者逗留的时间要长。导致这种情况的一种情况是非决定性的终止，比如 PPR（其中步行者以用户设定的概率终止）。另一种是高阶算法，如 node2vec。由于同步迭代包含了步行者到顶点的查询通信，步行者遇到样本拒绝时没有选择，只能留在原地为下一次迭代碰运气。

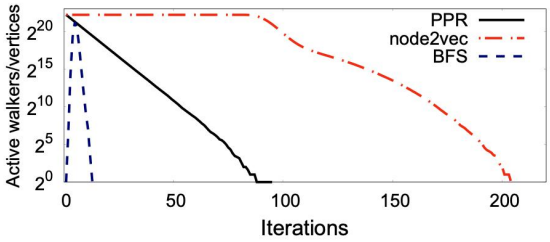


Figure 5. Tail behavior, random walk vs. BFS.

与必须处理不断减少的活动顶点集的图形处理算法不同，在这里，通过随机游走，我们面临更长、更细的尾巴，如图 5 中的 LiveJournal 图所示。BFS 有快速增长和缩小的活动顶点集，在 12 次迭代中完成。相比之下，有散兵游勇的随机行走 "收敛 "得更慢，极少数活跃的游走者远远落后于它。由于这种长尾是由算法引起的，KnightKing 不能加快这些慢速行走者的速度，但是我们发现，在这种长尾期间，通过削减并发水平，系统性能可以得到显著改善。当没有什么事情发生的时候，维护原始线程池的系统开销超过了并行处理的好处。

因此，当一个 KnightKing 节点的活动步行者数量低于阈值（在我们的实验中设定为 4000）时，它将切换到轻型模式，只保留三个线程（一个用于计算，两个用于信息传输）。结果显示，这种优化将长尾游走的总体执行时间减少了 57.5%。

3.2 GraphWalker

3.2.1 状态感知图形加载

图形数据组织和分区。GraphWalker 用广泛使用的压缩稀疏行（CSR）格式管理图形数据，该格式将顶点的外侧邻居依次存储为磁盘上的 csr 文件，并使用索引文件来记录 csr 文件中每个顶点的起始位置。GraphWalker 根据顶点 ID 将图分割成块。具体来说，我们根据顶点 ID 的升序将顶点和它们的外沿依次添加到一个块中，直到该块中的数据量超过预定的块大小，然后我们再创建一个新的块。图 6 显示了图 1（a）中示例图的数据布局。此外，这种轻量级的图数据组织减少了每个子图的存储成本，从而降低了图加载的时间成本。由于 GraphWalker 通过简单地读取一次索引文件来记录每个块的起始顶点来划分图形，它也可以灵活地调整不同应用的块大小。

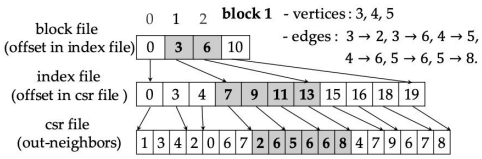


Figure 6: Graph data organization of the example graph in Figure 1(a). The graph is stored in CSR format and block partition ranges are recorded in the block file.

对于设置图块大小，我们发现存在一种权衡。也就是说，使用较小的块可以避免加载更多的数据，而这些数据在更新随机游走时并不需要，而使用较大的块可以在每次加载子图时有更多的游走被更新。此外，不同的分析任务需要不同的步行规

模，因此喜欢不同的块大小。

具有少量游走的轻量级任务更喜欢小的块大小，因为在这种情况下，I/O 利用率可以得到提高。相反，具有大量游走的重量级任务更喜欢大块大小，因为大块大小可以提高游走更新率。基于这种理解，我们使用经验分析，并将默认的块大小设置为$2^{(\log_{10}R+2)}$MB，其中 R 是随机游走的总数。例如，在运行 10 亿次游走的情况下，默认的块大小为 2GB，这通常小于商品机的内存容量，所以很容易在内存中保留一个图块。

图形加载和块缓存。GraphWalker 在预处理阶段转换图的格式并划分图块。在运行随机游走阶段，GraphWalker 选择一个图块，并根据游走的状态将其加载到内存中，尤其是加载包含最大数量游走的图块。在完成对加载的图块的分析后，它再选择另一个图块，以同样的方式加载。

为了缓解区块大小的影响，提高缓存效率，GraphWalker 还通过开发一个有意识的游走缓存方案来实现区块缓存，将多个区块保留在内存中。其原理是，具有更多游走次数的块更有可能在不久的将来被再次需要。因此，带有块缓存的图形加载过程如下。

如图 7 所示，我们首先根据状态感知模型选择一个候选块，为了加载这个块，我们检查它是否被缓存在内存中。如果它已经在内存中，那么我们就直接访问内存来进行分析。否则，我们就从磁盘上加载它，如果缓存已满，也会将内存中包含最少走动的块驱逐出去。缓存在内存中的块的最大数量取决于可用的内存大小。

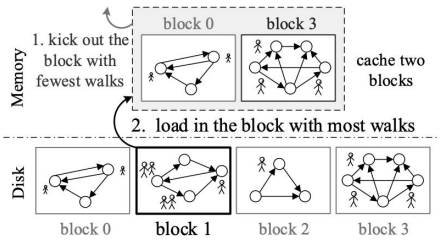


Figure 7: State-aware graph loading with block cachin

我们强调，这个有游走意识的块缓存方案在以下几个方面与传统的页缓存不同。首先，我们不采用预取，因为状态感知模型不喜欢按顺序访问图块。其次，页缓存以页的粒度管理内存中的数据，而我们以块的粒度进行管理，以适应基于块的图形加载和计算。最后但并非最不重要的是，驱逐策略也利用了游走状态，这与 LRU 不同。我们的实验表明，有游走意识的块缓存方案总是优于传统的页缓存方案。

3.2.2 同步游走更新

请注意，在基于迭代的系统中，在加载一个图块后，加载的子图中的每个行只走一步，这导致行的更新率非常低。事实上，在走完一步后，许多行仍然停留在当前子图的顶点上，因此它们可以通过更多的步骤进一步更新。为了提高 I/O 效率，一些作品使用了加载数据重入，这使得行走可以重复使用加载的数据。Lumos 使用跨迭代值传播的方式，为后续迭代正式使用加载的数据，以提供同步保证。其思路是再次重新进入子图，通过再次遍历子图中的顶点，再走一步。此外，我们还可以继续重新进入子图，直到所有的行走都到达子图的边界。

然而，重新进入的方案可能会导致局部掉队者问题。也就是说，当图块刚刚被加载时，许多行能够在第一时间移动一步，随着重新进入次数的增加，大多数行可能会到达子图的边界，只有少数行留在子图中，它们需要花费多次重新进入才能完成。我们的实验表明，一个区块中最后 20%的行走可能要花费 60%的再访问。这些重访的利用率很低，并花费了大量的时间。我们还发现，在某些重访之后，简单地停止在当前加载的子图上行走，也不能解决局部掉队者的问题，也不能减少完成时间，因为最后几个行走仍然留在子图中，我们仍然需要用额外的 I/O 重新加载子图来完成行走。

为了进一步提高 I/O 利用率和行走更新率，GraphWalker 采用了异步行走更新策略，允许每个行走持续更新，直到它到达加载图块的边界。在完成一个步行后，我们选择另一个步行来处理，直到当前图形块中的所有步行都被处理。然后，我们根据上述的状态感知模型加载另一个图块。图 8 显示了在同一个图块中处理两个步行的例子。为了加速计算，我们还使用多线程来并行更新 Walk。我们强调，通过我们的异步走更新模型，我们完全避免了顶点的无用访问，并消除了局部掉队者问题。

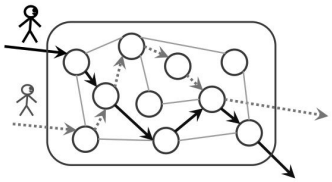


Figure 8: Asynchronous walk updating in parallel

然而，状态感知模型可能会导致全局的掉队者

问题。也就是说，一些游走可能移动得非常快，并取得了很大的进展，因为它们所需要的图数据总是可以得到满足，而另一些游走可能移动得非常慢，因为它们可能被困在一些长期没有加载到内存的冷块中。因此，GraphWalker 可以快速完成大多数行走，但需要很长时间才能完成剩下的少数行走。我们的实验表明，少数几个走动经常会产生近一半的总 I/O。

为了解决全局掉队者的问题，我们在 GraphWalker 的状态感知图加载过程中引入了一种概率性方法。我们的想法是给掉队者一个机会，让他们移动一些步骤，以便赶上大多数行走的进度。具体来说，每次当我们选择一个图块来加载时，我们分配一个概率 p 来选择包含进度最慢的行走的图块，即具有最小的行走步数，并且以概率 1-p，我们仍然加载具有最多行走的图块。请注意，随着 p 的增加，全局掉队者问题将得到更有效的缓解，但大多数行走的效率将下降。所以对 p 的设置要有所取舍。根据我们的经验分析，我们发现 p=0.2 是一个合适的设置，在某些情况下我们可以得到 20%的改进。

3.2.3 以块为中心的游走管理

我们用三个变量来记录每个行走，即源、当前和步骤，它们分别表示起始顶点、当前顶点在块中的偏移量和移动的步骤数。我们用 64 位来记录每个行走。为每个变量分配的比特数如图 9 所示。这种数据结构可以支持同时在 224 个源顶点开始随机行走，而且它还允许每个行走最多移动 214 步。请注意，由于我们可以在每个顶点启动许多次行走，所以对行走的总数没有限制。

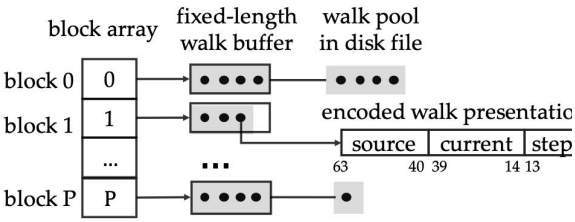


Figure 9: Block-centric walk management

为了减少管理所有行走状态的内存开销，我们提出了一个以块为中心的方案。对于每个图块，我们使用一个行走池来记录当前在该块中的行走，参考图 9。我们将每个步行池实现为一个固定长度的缓冲区，默认情况下最多存储 1024 个步行，以避免动态内存分配成本。当一个区块中有超过 1024 个步行时，我们把它们冲到磁盘上，并把它们存储为一个叫做步行池文件的文件。请注意，我们用一个 64 位的长数据类型来编码每个 walk，所以每个 walk pool 只需要 8KB。这样一来，管理步行状态的内存成本就非常低了。例如，在 YahooWeb 中运行 10 亿次行走时，如果 GraphWalker 使用 100 个图块，它的成本只有 800KB。然而，DrunkardMob 的成本超过 4GB，因为每个行走至少使用四个字节。此外，这些行走在 1120 万个动态数组中跳跃，从而导致频繁的内存重新分配，带来额外的时间成本。

当我们把一个图块加载到内存中时，我们也把它的步行池文件加载到内存中，并把步行与存储在内存步行池中的步行合并。然后，我们进行随机走动并更新当前走动池中的走动。在更新过程中，当行走池满时，我们将行走池中的所有行走追加到相应的行走池文件中，将其冲到磁盘上，并清除缓冲区。当完成对加载的图块的计算后，我们清除当前的步行池，并将每个图块的步行缓冲区和步行池文件中的步行加起来，以便更新步行状态。

通过这种轻量级的行走管理，我们节省了大量用于存储行走状态的内存成本，因此它能够支持大量的并发行走。此外，固定长度的步行缓冲策略将许多用于更新步行状态的小 I/O 变成了几个大的 I/O，这在很大程度上降低了提供步行状态持久性存储的 I/O 成本。

3.3 ThunderRW

3.3.1 以步骤为中心编程

框架。算法 2 给出了 ThunderRW 的概述。第 1-6 行逐一执行每个查询。第 3-5 行根据以步骤为中心的模型将一个步骤分成三个函数。Gather 收集与 Q.cur 相邻的边的转换概率。它在$E_{Q.cur}$上循环，对每条边 e 应用 Weight（一个用户定义的函数），并将 e 的转换概率加到 C（第 10-11 行）。然后，第 12 行执行给定采样方法的初始化阶段，以更新 C。Move 根据 C 挑选一条边，并将 Q 沿着选定的边移动（第 14-16 行）。由于系统空间（即不包括用户定义的函数的框架）中的随机内存访问主要在 Move 中进行，我们应用步骤交错技术来优化其性能。最后，第 5 行调用用户定义的函数 Update，根据移动情况更新 Q 的状态。Update 的返回值决定了 Q 是否应该被终止。

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;
foreach Q ∈ ℚ do
    do
        C ← Gather(G, Q, Weight);
        e ← Move(G, Q, C);
        stop ← Update(Q, e);
    while stop is false;
return ℚ;
Function Gather(G, Q, Weight)
    C ← {};
    foreach e ∈ E_{Q.cur} do
        Add Weight(Q, e) to C;
    C ← execute initialization phase of a given sampling method on C;
    return C;
Function Move(G, Q, C)
    Select an edge e(Q.cur, v) ∈ E_{Q.cur} based on C and add v to Q;
    return e(Q.cur, v);
```

算法 2 中描述的框架可以支持无偏、静态和动态 RW 的不同采样方法。此外，我们根据 RW 的类型和所选择的采样方法来优化 ThunderRW 的执行流程。在执行过程中，静态 RW 的过渡概率是固定的。在这种情况下，ThunderRW 省略了 Gather 操作，但引入了一个预处理步骤以降低运行时间成本，该步骤在系统初始化时获得了转换概率。算法 3 介绍了静态 RW 的预处理方法。给定一个顶点 v，第 3-4 行在E_v的每条边 e 上循环，并对 e 应用 Weight 函数以获得转换概率。由于该概率不依赖于查询，我们将 Q 设置为空。之后，第 5-6 行对C_v进行给定抽样方法的初始化阶段，并存储C_v以便在查询执行中使用。因此，我们可以直接加载$C_{Q.cur}$，在算法 2 中对于静态 RW 可以省略 Gather 函数。

此外，NAIVE 和 O-REJ 采样方法没有初始化阶段。因此，我们不需要收集初始化的转变概率。因此，如果使用 NAIVE 或 O-REJ，ThunderRW 会跳过预处理步骤和执行中的 Gather 操作。

应用程序编程接口（API）。ThunderRW 提供了两种 API，其中包括超参数和用户定义的函数。用户通过两个步骤来开发他们的 RW 算法。首先，通过超参数 walker_type 和 sampling_method 分别设置 RW 的类型和采样方法。其次，定义权重和更新函数。Weight 函数指定了一条边被选中的相对机会。Update 函数给定被选中的边，修改 Q 的状态。如果它的返回值为真，那么该框架就会终止 Q。否则，Q 继续在 G 上行走。当使用 O-REJ 时，用户需要实现 MaxWeight 函数来设置过渡概率的最大值。

ThunderRW 将用户定义的函数应用于 RW 查询，并根据 RW 类型和选定的采样方法并行地评估查询。因此，用户可以通过 ThunderRW 轻松实现定制的 RW 算法，这大大降低了工程工作量。

平行化。RW 算法包含大量的随机行走查询，每个查询都可以独立快速完成。因此，ThunderRW 采用了静态调度方法来保持工作者之间的负载平衡。具体来说，我们将每个线程视为一个工作者，并将 Q 平均分配给各工作者。一个工作者通过算法 2 独立执行分配的查询。实验结果表明，这种简单的调度方法取得了良好的性能。

3.3.2 阶段依赖图

我们设计了阶段依赖图（SDG）来模拟一个步骤中的操作序列的阶段。SDG 中的每个节点都是一个包含一组操作的阶段，而边则代表它们之间的依赖关系。给定操作序列，我们分两步建立 SDG，抽象出阶段（节点）和提取依赖关系（边）。

定义阶段：由于我们通过切换查询的执行来隐藏内存访问延迟，对阶段的约束是每个阶段最多包含一个内存访问操作，而消耗数据的操作是在后续阶段。请注意，为了便于实现切换，我们将包含跳转操作的操作视为一个阶段。

定义边：接下来，我们在 SDG 中的节点之间根据它们的依赖关系来添加边。给定阶段 S 和S'，如果 S 和S'之间存在依赖关系，我们就从 S 向S'添加一条边。这些边被分为三种类型，内存依赖、计算依赖和控制依赖。我们把前两种关系称为数据依赖关系。更具体地说，如果S'消耗了 S 从内存中加载的数据，那么这个边的类型就是内存依赖关系。否则，S'依赖于由 S 计算的数据，则边缘类型为计算依赖。导致依赖性的数据作为属性附加到每个边上。此外，如果 S 包含跳转到S'的操作，我们将控制依赖从 S 添加到S'。SDG 允许节点之间存在多条边（即依赖关系）。如果我们只考虑数据依赖关系，SDG 是一个有向无环图（DAG），而控制依赖关系可以在 SDG 中产生循环。

总之，SDG 是一种从 Move 的操作序列中抽象出阶段的方法，并对它们之间的依赖关系进行建

模。请注意，MOVE 的阶段设计不需要用户输入，但它是在系统空间中实现的。

3.3.3 状态切换机制

在本小节中，我们将介绍 SDG 下的步骤交错的实现。基于公式 2，我们需要一个高效的切换机制。例如，使用多线程是不允许的，因为线程间上下文切换的开销是微秒级的，而主内存的延迟是纳秒级的。由于每个线程都倾向于采取许多 RW 查询，我们在单个线程中的阶段之间进行切换执行。

我们根据 SDG 的阶段是否属于 SDG 的周期，将其分为两类。对于不属于循环的阶段（称为非循环阶段），一个查询正好访问一次，以完成移动。给定一组查询 Q′，我们以一种耦合的方式执行它们。特别是，一旦一个查询$Q_i \in Q'$完成了一个阶段 S，我们就切换到下一个查询$Q_{i+1} \in Q'$来处理 S。在所有的查询完成 S 后，我们转到下一个阶段。相比之下，循环中的阶段（称为循环阶段）可以为不同的查询访问不同的时间。为了处理这种不规则性，我们以解耦的方式处理它们。具体来说，每个查询 Q 记录要执行的阶段 S。当切换到 Q 时，我们执行 S，根据 SDG 设定 Q 的下一个阶段，并在完成 S 后切换到下一个查询。因此，每个查询的执行是异步的。

对于查询中不同阶段之间的数据通信，我们基于 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。特别是，任务环用于查询中所有阶段的数据通信，而搜索环则用于处理周期阶段。由于我们需要明确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施的复杂性，而且还会产生更多的开销。请注意，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 则有。

4 总结与展望

本文介绍了三种随机游走引擎 KnightKing、GraphWalker 以及 ThunderRW，每一种 RW 框架针对 RW 算法不同的性能问题做了相应优化。

KnightKing 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行走者为中心的计算模型，以支持随机行走算法的简单规范。它采用了一个统一的边缘转换概率定义，适用于流行的已知算法，以及新颖的基于拒绝的采样方案，大大降低了昂贵的高阶随机行走算法的成本。对 KnightKing 的设计和评估表明，无论当前顶点有多少条出边，都有可能在精确的边抽样中达到接近 O(1)的复杂度，而不会失去准确性。

GraphWalker 是一个 I/O 效率高的系统，用于支持在单台机器上对大型图进行快速和可扩展的随机行走。GraphWalker 仔细管理图数据和行走索引，并通过使用状态感知的图加载和异步行走更新来优化 I/O 效率。原型实验结果表明，GraphWalker 的性能优于最先进的单机系统，而且它还取得了与运行在集群机器上的分布式图系统相当的性能。在未来的工作中，将考虑把 GraphWalker 中的状态感知设计理念扩展到分布式集群，以便并行处理大量的分析任务。

ThunderRW 是一个高效的内存中 RW 引擎，用户可以在其上轻松实现定制的 RW 算法。提出者设计了一个以步骤为中心的模型，将计算从移动查询的一个步骤的局部视图中抽象出来。基于该模型，提出了步骤交错技术，通过交替执行多个查询来隐藏内存访问延迟。提出者用框架实现了四种代表性的 RW 算法，包括 PPR、DeepWalk、Node2Vec 和 MetaPath。实验结果表明，ThunderRW 的性能比最先进的 RW 框架高出一个数量级，而且步骤交织将内存约束从 73.1%降低到 15.0%。目前，提出者在 ThunderRW 中通过明确地、手动地存储和恢复每个查询的状态来实现步骤交错技术。一个有趣的未来工作是用 coroutines 来实现这个方法，coroutines 是一种支持交错执行的有效技术。

5 参考文献

[1]. Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: a fast distributed graph random walk engine. In Proceedings of the VLDB Endowment 7, 14 (2014), 1981-1992.

[2]. Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. Graph-Walker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In 2020 USENIX Annual Technical Conference (USENIX ATC 20). 559–571.

[3]. Shixuan Sun, Yuhang Chen, Shengliang Lu,

Bingsheng He, and Yuchen Li. 2021. ThunderRW: An in-memory graph random walk engine.. In Proc. VLDB Endow., Vol. 14. 1992--2005.