# 图随机游走引擎综述

燕冉[1)]

1)(华中科技大学 计算机科学与技术学院，武汉 430074)

**摘 要** 图作为一种重要的数据结构，被广泛应用于各个领域。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。面对这样大规模的图数据，传统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。随机游走算法具备强大的图算法能力，同时保证了算法的准确性和高效性，被证明是分析大型图的有效办法。为了有效地支持快速和可扩展的随机游走，许多专门针对该算法进行优化的框架引擎逐渐被开发。本文对比了随机游走的五种框架支持，并描述了它们在随机游走的性能和扩展性等不同方面的关键优化。其中 GraphWalker 是单机外存图随机游走系统，它提出了一种基于游走状态感知的处理模型，有效提升 I/O 效率，并且设计了一种子图重入、异步更新的随机游走策略，有效地提升了随机游走的更新步长；KnightKing 是首个以游走为中心的分布式随机游走引擎，它使用拒绝采样技术使得运行效率得到提升；Flashmob 通过让内存访问更加具有顺序性和规律性，提高了缓存和内存带宽的利用率，并且将分区的问题映射成组合优化问题，使用动态规划解决，实现准确而高效的数据分区；ThunderRW 是一个高效的内存中随机游走引擎，使用步进交错技术，有效减少内存访问延迟；C-SAW 是一个新颖的 GPU 图形采样框架，支持广泛的采样和随机游走算法。

关键词 随机游走；GraphWalker；KnightKing；Flashmob；ThunderRW；C-SAW

## Overview of Graph Random Walk Engine

Ran Yan[1)]

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

**Abstract** As an important data structure, graphs are widely used in various fields. With the increase of social informatization and the advent of big data era, the scale and complexity of graph data are also growing. Facing such large-scale graph data, traditional iteration-based graph traversal algorithms face the problems of excessive computational complexity and high space overhead. The random walk algorithm has powerful graph algorithmic capabilities while ensuring accuracy and efficiency, and proves to be an effective approach for analyzing large graphs. To efficiently support fast and scalable random walks, many framework engines optimized specifically for this algorithm are gradually developed. This paper compares five frameworks supported by random walks and describes their key optimizations for different aspects of random walk performance and scalability. Among them, GraphWalker is a single-computer external storage graph random walk system, which proposes a processing model based on wander state awareness to effectively improve I/O efficiency, and designs a random walk strategy with subgraph reentry and asynchronous update to effectively improve the update step of random walk; KnightKing, the first wandering-centric distributed random walk engine, which uses rejection sampling techniques to enable improved operational efficiency; Flashmob improves cache and memory bandwidth utilization by making memory accesses more sequential and regular, and maps the problem of partitioning into a combinatorial optimization problem that is solved using dynamic programming to achieve accurate and efficient data partitioning; ThunderRW is an efficient in-memory random walk engine that uses step interleaving techniques to effectively reduce memory access latency; C-SAW is a novel, GPU graphics sampling framework that supports a wide range of sampling and random-walk algorithms..

**Key words** Random-Walk; GraphWalker; KnightKing; Flashmob; ThunderRW; C-SAW

# 1 引言

为了提高在单机上分析大型图形的性能，人们提出了许多核外图形处理系统。这些系统的一个主要工作是减少随机 I/O。当一个图形太大，无法装入内存时，这些系统将整个图形分割成许多子图，并将每个子图作为一个块存储在磁盘上。为了进行图分析，采用一个基于迭代的模型，在每个迭代中，块被依次加载到内存中，然后执行与加载的子图相关的分析。这样把大量的随机 I/O 变成了一系列顺序的 I/O。

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，可以利用图中节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 PPR（personalized PageRank），SimRank，DeepWalk，node2vec 等。这些算法既可以独立运行，也可以作为机器学习任务的预处理步骤。他们服务于各种应用场景，比如点边分类，社区检测，链接预测，图像处理，语言建模，知识发现，相似性测量和推荐系统等。

随机游走过程中的主要计算开销在于边采样过程，这也是不同的随机游走算法的不同之处，每个不同的随机游走算法定义其特有的边转移概率。随着随机游走的普及，边采样的逻辑也变得越来越复杂，近期的算法中更是提出了动态采样，即随机游走在每个节点处的采样概率不仅跟该节点的邻居信息有关，还与随机游走的状态有关。然而目前已经出现的传统图处理引擎，不能不能有效地支持随机游走算法。由于高度的随机性，许多游走不均匀地分散在图的不同部分，所以一些子图可能只包含少数游走。然而，基于迭代的模型并不知道这些游走的状态，它只是按顺序将所有子图加载到内存中进行分析，所以它的 I/O 利用率非常低。由于基于迭代的模型才用了同步更新策略，所有的游走在每个迭代中都只能移动一步，因此游走的更新效率也是有限的，这也进一步影响了 I/O 的效率。由于游走的随机性，每个顶点的游走数量都是动态变化的，所以现有的系统通常使用大量的动态数组来记录当前通过图中每条边或每个顶点的游走。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的情况，影响了可扩展性。

近年来，学者们提出了许多专门用于处理随机游走过程的计算引擎，以提高基于迭代模型的随机游走性能，如 GraphWalker、KnightKing、Flashmob、ThunderRW 以及 C-SAW，下文将针对它们的原理和优势、研究进展依次展开。

# 2 原理和优势

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

## 2.1 GraphWalker

为了支持数百亿条、数千步长的随机游走，GraphWalker 与基于迭代的模型盲目地按顺序加载图块不同，它采用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，从而支持快速和可扩展的随机游走，此外，即使一个图块中只有一个行走，它也必须将该图块加载到内存中进行分析。图 1 说明了它的整体设计，主要由三部分组成：

1. 状态感知的图加载。
2. 异步游走更新。
3. 以块为中心的游走管理。

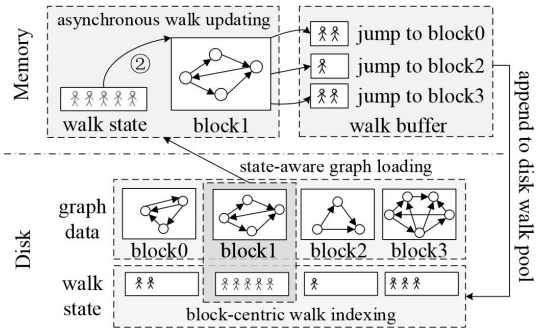



图 1 GraphWalker 整体设计

### 2.1.1 状态感知的图加载

**图数据组织和分区。**GraphWalker 用压缩稀疏行（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，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，然后再创建新的块。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 通过读取一次索引文件来记录每个块的起始顶点从而划分图形，使得它可以灵活地调整不同应用的块大小。

设置图块大小时，还存在一种权衡。即使用较小的块可以避免加载更多的数据，但是在更新随机游走时并不需要这些数据，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，因此需要不同的块大小。比如，具有少量行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 利用率可以得到改善。相反，具有大量行走的重量级任务更喜欢大块大小，因为大块大小可以提高行走更新率。

**图形加载和块缓存。**GraphWalker 在预处理阶段转换图形格式并划分图形块。在运行随机游走的阶段，GraphWalker 选择一个图块，并根据游走的状态将包含最大游走数量的图块加载到内存中。在完成对现有加载图块的分析后，再加载另一个图块对其分析。

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 2 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游走次数的块驱逐出去。缓存在内存中的块的最大数量取决于可用的内存大小。

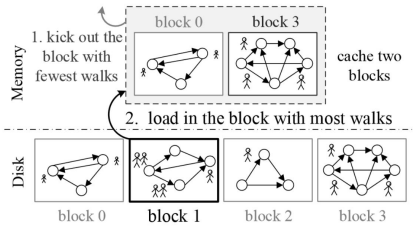



Figure 7: State-aware graph loading with block caching

图 2　状态感知模型

2.1.2　异步游走更新

在基于迭代的系统中，加载一个图块后，其子图中的每个游走只走一步，这导致游走更新率非常低。事实上，在走完一步之后，许多 walkers 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。然而，重新进入的方案可能会导致局部散兵游勇问题。

为了进一步提高 I/O 利用率和行走更新率，GraphWalker 采用了异步游走更新策略，允许每个游走持续更新，直到它到达加载图块的边界。在完成一个游走后，选择另一个游走来处理，直到当前图形块中的所有游走都被处理。然后，我们再根据前文所述的状态感知模型加载另一个图块。图 3 显示了在同一个图块中处理两个游走的例子。为了加速计算，作者还使用多线程来并行更新 Walk。作者强调，通过他们的异步游走更新模型，完全避免了对顶点的无用访问，并消除了局部散兵游勇问题。

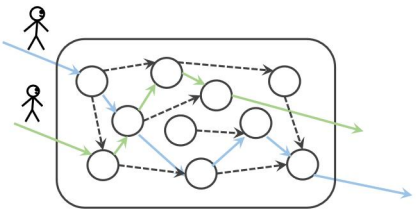


图 3　异步游走更新

然而，状态感知模型可能会导致全局的散兵游勇问题。也就是说，一些 walkers 所需要的图数据总是可以得到满足，所以可能移动得非常快，并取得了很大的进展，而另一些 walkers 可能被困在一些长期没有加载到内存的冷块中，所以可能移动得非常慢。

为了解决全局散兵游勇的问题，作者在 GraphWalker 的状态感知图加载过程中引入了一种概率性方法。想法是给散兵游勇一个机会，让他们移动一些步骤以赶上大多数游走的进度。具体来说，每次选择一个图块进行加载时，分配一个概率 p 来选择包含进度最慢的游走的图块，即给具有最小的游走步数分配概率 1-p，我们仍然加载具有最多行走的图块。值得注意的是，随着 p 的增加，全局散兵游勇问题将得到有效的缓解，但大多数游走的效率将下降。所以对 p 的设置要有所取舍。根据经验分析，发现 p=0.2 是一个合适的取值。

2.1.3　以块为中心的游走管理

为了减少管理所有游走状态的内存开销，作者提出了一个以块为中心的数据管理方案。对于每个图块，使用一个 walk pool 来记录当前在该块中的游走，参考图 4。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，默认情况下缓冲

区最多存储1024个步行，当一个区块中有超过1024个步行时把它们冲到磁盘上，并把它们存储为一个walk pool文件。在这里，作者用一个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walk状态的内存成本就非常低。

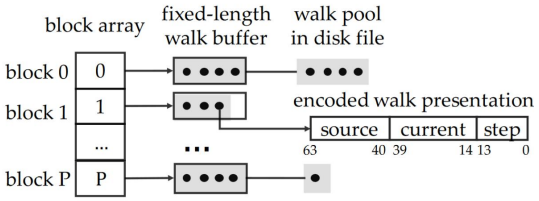



Figure 9: Block-centric walk management

图4　以块为中心的游走管理

具体实现流程，当把一个图块加载到内存中时，也把它的walk pool文件加载到内存中，并把将其与存储在内存中的walk pool中的walk合并。然后，进行随机游走并更新当前walk pool中的walk。在更新过程中，当walk pool满时，将walk pool中的所有walk追加到相应的walk pool文件中，并清除缓冲区。当完成对加载图块的计算后，清除当前的walk pool，并将每个图块的walk缓冲区和walk pool文件中的walk加起来，以便更新游走状态。

通过这种轻量级的游走管理，节省了大量用于存储游走状态的内存成本，从而使它能够支持大量的并发游走。此外，固定长度的游走缓冲策略将更新游走状态的许多小的I/O变成了几个大的I/O，这在很大程度上降低了提供游走状态的持久性存储I/O成本。

## 2.2　KnightKing

复杂的随机游走算法由于它的动态性，实现了更灵活的、特定于应用程序的游走，而代价是采样的复杂性和高成本。这一挑战普遍存在于现有的许多随机游走算法，然而缺乏一个框架为高效和可扩展的图随机游走提供通用算法或系统支持。

作者提出了KnightKing，第一个提出的通用随机游走运算的框架引擎，是一个分布式的随机游走引擎。它提供了以游走游走者为中心的视图，使用自定义的API 来定义边的转移概率，同时处理其他随机游走的基础设置。与图计算引擎类似，它隐藏了图形分区、顶点分配、节点间通信和负载均衡等系统细节。因此，它提供了一个直观的“像游走游走者一样思考”的视图，同时用户可以添加可选的优化。

接下来描述KnightKing的系统设计和优化，为不同的随机游走算法提供一个统一的执行引擎，讨论集中在设计方面和随机游走执行的具体权衡上。

### 2.2.1　图的存储和分区

KnightKing使用压缩稀疏行(CSR)存储边缘，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过渡概率检查），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，一个顶点在随机游走中被访问的频率取决于用户定义的（可能是动态的）过渡概率，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，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游走处理或通信负载，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。

### 2.2.2　随机游走的执行和协调

**计算模型。**由于游走者是独立移动的，随机游走算法可能会出现不符合预期的并行和无协调，很容易扩展到更多线程/节点。然而，一个简单的实现，例如使用图形数据库作为存储后端，每个walker通过getVertex()和getEdges()等api检索信息，将导致过高的网络流量和过差的数据局域性。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。

在游走开始之前，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，使用提供的上限和可选的下限设置拒绝采样，以及游走实例化/初始化。

以游走者为中心的主要游走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进行任务调度，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

设置了并行处理，其线程数量与执行计算的可用核心数量相同，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，定义为游走者或消息的块，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。

**散兵游勇的处理。**其基于拒绝的核心策略将抽样复杂度降低到接近 O(1)， KnightKing 使得每一步的抽样成本不仅更低，而且更可预测。然而，它仍然可能面临长尾执行，少数散兵游勇比大部分游走者停留的时间要长。导致这种情况的一种情况是非决定性的终止，比如 PPR（其中游走者以用户设定的概率终止）。另一种是高阶算法，如 node2vec。

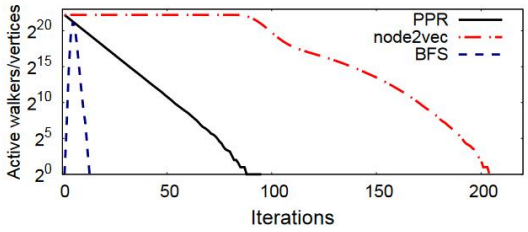



**Figure 5.** Tail behavior, random walk vs. BFS.

图 5　尾延迟

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，面临更长、更细的尾延迟，如图 5 中所示。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，有散兵游勇的随机游走“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 不能加快这些慢速游走者的速度，但是发现，在这种长尾期间，通过削减并发水平，系统性能可以得到明显的改善。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 节点的活动游走者数量低于阈值时，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）。

## 2.3　Flashmob

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，现有的图随机游走系统目前无法与 GPU 侧的节点嵌入训练速度相匹配。人们普遍认为随机漫步的固有随机性和图形的倾斜性质使得大多数内存访问是随机的。作者证明，通过仔细的分区、重新安排和批量操作，实际上可以获得大量的空间和时间定位。

由此提出 FlashMob 系统，通过使内存访问更有顺序和规律，提高了缓存和内存带宽的利用率，并且将分区的问题很好的映射成了一个经典的组合优化问题（多项选择背包问题），然后使用动态规划解决这个问题，主要优化如下。

### 2.3.1　频率感知的顶点分组

FlashMob 带来的一个关键设计变化是对图进行分区，并对每个分区中的步行者进行批处理。考虑到在随机游走中顶点程度和它们的受欢迎程度之间总体高度相关性，FlashMob 将输入图的顶点按其程度降序排列。排序后的顶点阵列被切割成不同大小的连续顶点分区（简称 VP），其大小根据图的特性和系统资源约束进行优化，对于相似度数的顶点分组，FlashMob 用不同的策略处理它们，并从不同的来源获得性能。FlashMob 还对步行者数据进行了划分，以便在每次步行迭代中，用连续的步行者数据块来处理 VP，以访问和更新步行者的位置。

### 2.3.2　边缘采样阶段

FlashMob 的随机游走迭代的边缘采样阶段执行中心任务：为每个行走者寻找一条边来移动。更具体地说，对于一个给定的 VP，FlashMob 选择一种采样策略，即预采样（PS）或直接采样（DS），并因此采取不同的数据组织和访问模式。

其中预取样（PS）主要思想是提前对许多边进行采样，这些边会被许多共处一地的步行者依次消耗。直接取样（DS）步行者当场从（通常很短的）邻接列表中选择一条出站边。

### 2.3.3　洗牌阶段

在每个采样阶段之后，步行者从每个 VP 中分散出来。在随后的洗牌阶段，所有线程并行工作，重新安排步行者，以便现在在同一 VP 内的步行者再次被连续存储。对于高阶行走，边缘采样所额外需要的每个行走者数据（例如，节点 2vec 的即时行走历史）与行走者的当前位置一起被洗牌。在整个 FlashMob 的工作流程中，反复遵循的设计准则是对常规、紧凑的数据结构（大多数情况下是一维阵列）进行顺序扫描。

## 2.4　ThunderRW

与现有的提高单一图操作性能的并行系统相比，ThunderRW 支持大规模的并行随机行走。它的核心设计源于作者的分析结果：普通的 RW 算法有高达 73.1%的 CPU 管线槽因不规则的内存访问而停

滞，这比传统的图工作负载如 BFS 和 SSSP 遭受了明显的内存停滞，关键设计如下。

2.4.1　系统框架

**以步进为中心的模型。**

为了抽象出 RW 算法的计算，作者提出了以步进为中心的模型。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，以步进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，从移动查询的一个步骤 Q 的局部视图来模拟计算。然后，我们将 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。通过以步进为中心的模型，用户通过“像步行者一样思考”来开发 RW 算法。该框架有利于将用户定义的面向步进的函数应用于 RW 查询。

**以步进为中心的编程。**

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。

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，将每个线程视为一个工作者，并将 Q 平均分配给各工作者。

2.4.2　步进交错技术

步进交错技术可以总结如下。给定 Move 中的操作序列，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。同时执行一组查询。一旦查询 Q 的一个阶段完成，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，恢复 Q 的执行。通过这样的方式，将内存访问延迟隐藏在单个查询中，并保持 CPU 的工作状态。

### 2.5　C-SAW

许多应用需要学习、挖掘、分析和可视化大规模图。这些图往往太大，无法用传统的图处理技术有效解决。幸运的是，最近的研究工作发现了图抽样和随机行走，它们大大减少了原始图的大小，可以通过捕捉理想的图的属性来有利于学习、挖掘、分析和可视化大型图的任务。

为了能够支持广泛的采样和随机游走算法，本文作者介绍了 C-SAW，这是第一个在 GPU 上加速采样和随机行走框架的框架，关键设计如下。

2.5.1　以偏差为中心的采样框架

C-SAW 在 GPU 上卸载了采样和随机行走，目标是建立一个简单而富有表现力的 API 和一个高性能框架。特别是，简单意味着终端用户可以在不了解 GPU 编程语法的情况下对 C-SAW 进行编程。表达性要求 C-SAW 不仅要支持已知采样算法，还要准备支持新兴的算法。高性能的目标是框架设计。也就是说，编程的简单性并不妨碍 C-SAW 对主要的 GPU 和采样相关的优化进行探索。C-SAW 提供了三个用户定义的 API 函数，分别是 VERTEXBIAS、EDGEBIAS 和 UPDATE。

2.5.2　Multi-GPU C-SAW

随着信号源数量的不断增加，工作负载将使 GPU 饱和。在这种情况下，将 C-SAW 扩展到多个 GPU 将有助于加速采样性能。由于各种采样实例是相互独立的，C-SAW 简单地将所有的采样实例分为几个互不相干的组，每个组包含相同数量的实例。这里，互不相干组的数量与 GPU 的数量相同。之后，每个 GPU 将负责一个采样组。在采样过程中，每个 GPU 将执行如图 6 所示的相同任务，不需要 GPU 之间的通信。

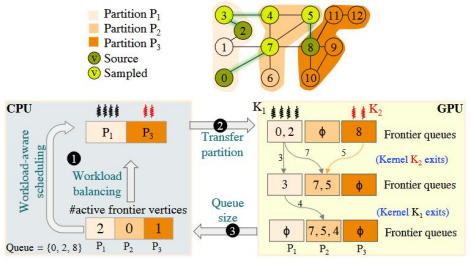



Fig. 8: Workload-aware scheduling of graph partition. The upper part shows the toy graph and its partition. We start sampling within partitions 1 and 3. The lower part shows an example for out-of-memory sampling. For simplicity, we hide InstanceID and CurrDepth from the frontier queue.

图 6　Multi-GPU

## 3　研究进展

### 3.1　GraphWalker

GraphWalker 的目标是提供快速和可扩展的随机游走，因此作者以最先进的单机随机游走图系统 DrunkardMob 为基准进行性能比较。还将 GraphWalker 与两个最先进的图系统 Graphene 和 GraFSoft 进行比较。为了进一步验证其可扩展性，还将 GraphWalker 与最新的分布式随机游走图系

统 KnightKing 进行比较。

3.1.1 实验设置

**Testbe.**实验在戴尔 Power Edge R730 机器上进行的，它有 64GB 内存和 24 个英特尔（R）Xeon(R) CPU E5-2650 v4 @ 2.20GHz 处理器。图形数据存储在由 7 个 500GB 的 SamSung 860 固态硬盘组成的 3.2TB RAID-0 上。对于分布式系统 KnightKing，它运行在一个 8 个节点的集群上，有 10Gbps 的以太网相互连接，每个节点配备两个 8-core Intel Xeon E5-2620 v4 处理器。

**Dataset.**图 7 列出了本文使用的六个图形数据集的统计数据。TT、FS、YW 和 CW 是真实世界的图。K30 和 K31 是用 Graph500 kronecker 生成的两个合成图。这些图都被广泛用于图系统的评估中。CSR（压缩稀疏行）大小表示通过以 CSR 格式存储图的最小存储成本，Text 大小表示以文本格式存储为边缘列表的数据集的大小。

| 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 |

Table 1: Statistics of Datasets

图 7 数据集

**Graph algorithms.**除了直接评估运行随机游走的性能，作者还考虑了以下四种基于随机游走的常见算法：Random Walk Domination (RWD)、Graphlet Concentration (Graphlet)、Personalized PageRank (PPR)、SimRank (SR)。

3.1.2 与 RW 特定系统比较

通过考虑不同的随机游走配置，从而来研究整个设计空间的性能。首先固定步长为 10，游走的数量从 $10^3$ 到 $10^{10}$ 不等，实验结果如图 8 所示，X 轴表示每个实验中配置的游走次数，Y 轴表示完成所有这些游走所需的时间。

我们可以看到，在所有设置下 GraphWalker 比 DrunkardMob 在不同的游走数量和不同的图数据集上都要快。一般来说，GraphWalker 在所有设置下实现了 16 至 70 倍的速度提升。而且，GraphWalker 能够支持巨大的图（例如 Kron30）和大规模的随机游走（例如 GraphWalker 大约在一个小时内就在大数据集上完成了 1010 游走）。

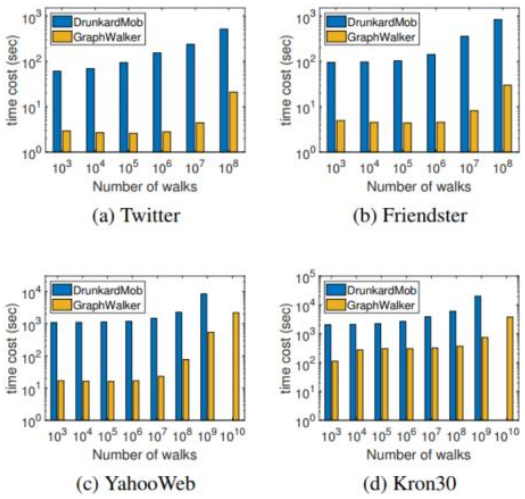



图 8 与 RW 特定系统比较，固定步长

作者还通过改变步长来评估性能。固定游走次数为 $10^5$，步长从 $2^2$ 到 $2^{10}$ 不等，实验结果如图 9 所示。

首先，我们可以看到 GraphWalker 总是比 DrunkardMob 快得多，在最好的情况下它甚至达到了三个数量级以上。虽然 GraphWalker 在处理非常大的图时，不能将其完全放在内存中，所以这种情况下需要在内存和磁盘之间交换和踢出块。即便如此，GraphWalker 的速度也比 DrunkardMob 快得多。例如，即使对 Kron30 来说，它也能达到 7-10 倍的加速。

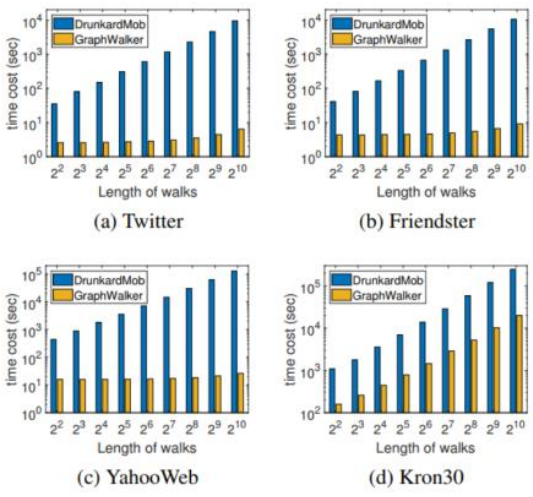
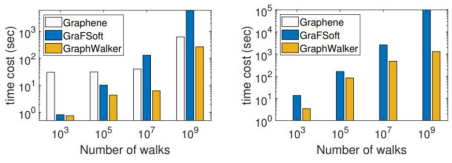



图 9 与 RW 特定系统比较，改变步长

3.1.3 与最先进的系统比较

与单机图形系统 Graphene、GraFSoft 比较，重点关注从单一来源开始运行随机游走的情况，实验结果如图 10 所示。

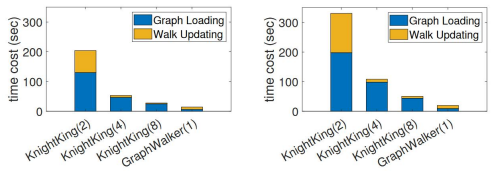
与 Graphene 相比，GraphWalker 可以扩展并且处理庞大的图数据，而 Graphene 因为它的半外部设计造成了高内存成本。所以 GraphWalker 的加速性能一直优于 Graphene。与 GraFSoft 相比，当游走数量较少时，GraphWalker 的改进是有限的，因为考虑到总数量较少，每个区块只能有几个游走，状态感知 I/O 模型带来的改进有限。当游走数量较大时，例如在 CrawlWeb 上运行 10 亿次随机游走时，GraphWalker 只花了二十几分钟，而 GraFSoft 甚至不能在 24 小时内完成这一任务。总的来说，GraphWalker 实现了 1-37 倍的速度提升。




(a) Friendster (b) CrawlWeb

图 10 与单机图形系统比较

与分布式随机游走系统 KnightKing 比较，在每个顶点开始一个游走，每个游走在每一步以概率 t 终止。我们设定 t=0.15。系统名称后面的数字表示正在使用的机器数量，实验结果如图 11 所示。




(a) Twitter (b) Friendster

图 11 与分布式随机游走系统

我们可以看到，对于 KnightKing 来说，随着集群规模的增加，计算时间，即更新游走的时间大大减少，但处理 I/O，即加载图块，仍然要花费大量的时间。这是因为 KnightKing 主要优化集中在计算效率，而不是磁盘 I/O。请注意，本实验中计算时间的结果与 KnightKing 论文中的结果一致。相比之下，GraphWalker 主要针对的是 I/O 效率问题，并且还根据其 I/O 模型对游走更新过程进行了相应的调整，所以它可以在图上实现非常快的随机游走。这里的行走更新时间也包括了行走索引的持续时间。我们还看到，GraphWalker 甚至与在 8 台机器上运行的 KnightKing 相比，也取得了相当的性能。此外，对于最大的图 CrawlWeb，KnightKing 在处理其他较小的图时，根据对其所用资源的估计，可能需要更大的集群来运行。因此，我们可以得出结论，GraphWalker 也是一个资源更友好的选择。

## 3.2 KnightKing

### 3.2.1 实验设置

Testbed:使用一个具有 40Gbps IB 互连的 8 节点集群，运行 Ubuntu 14.04。每个节点有 2 个 8-core Intel Xeon E5-2640 v2， 20MB L3 缓存， 94GB DRAM。

| Graph | $\|V\|$ | directed $\|E\|$ | undirected $\|E\|$ | Degree mean | Degree variance |
|---|---|---|---|---|---|
| LiveJournal [7] | 4.85M | 69.0M | 86.7M | 17.9 | 2.72E3 |
| Friendster [47] | 70.2M | 1.81B | 3.61B | 51.4 | 1.62E4 |
| Twitter [22] | 41.7M | 1.47B | 2.93B | 70.4 | 6.42E6 |
| UK-Union [8] | 134M | 5.51B | 9.39B | 70.3 | 3.04E6 |

**Table 2.** Real-world graph datasets

图 12 测试集

Input graphs:图给出了四个真实世界图数据集的规范，广泛用于图处理和随机游走评估。这些图涵盖了不同的图尺寸，Twitter 图和 UK-Union 图比其他两个图呈现出更明显的幂律度分布。使用它们的无向版本，并通过将边权值分配为[1,5)随机抽样的实数来进一步创建它们的加权版本(用于有偏向行走)。

Random walk applications:评估了前面讨论的四种流行的随机行走算法，即 DeepWalk、PPR、Meta-path 和 node2vec。前两者是静态的，而后两者是动态的。所有的测试都部署了|V |的步行者。对于 Meta-path，有 5 种边缘类型和 10 种循环路径方案，长度=5。每个步行者被随机分配一个方案。对于 PPR，将终止概率设定为 1/80，因此其预期行走长度也是 80。

Systems for comparison:由于缺乏通用的随机游走引擎，将 KnightKing 与 Gemini 的随机游走适应版本进行比较，Gemini 是最先进的分布式图计算系统，用 C++实现。它采用了双模式（推/拉）更新传播模型，并针对图的处理进行了密集的系统优化。与 GAS 模型一样，在 Gemini 中，一个顶点不能直接访问它的所有入射边，而是要与分布在不同节点上的镜像互动。因此，为它实现了一个两阶段的采样算法：在每一步，一个步行者首先使用 ITS 采样到哪个节点，然后它在该节点上的镜像采样到一个特定的边。对于动态游走，过渡概率是以临时的方式计算的，第二阶段也使用 ITS（由于其高初始化成本，别名为低效率）。对于静态行走，过渡概率和相应的数据结构是预先计算的，ITS 和 alias 都是为第二阶段评估的。这样的顶点复制和边缘分布也使 Gemini 无法采用拒绝采样，因为游走者读

取任何特定的边缘需要两次迭代（向镜像发送请求并等待其响应）。

Evaluation methodology:报告的执行时间包括初始化游走者和采样相关数据结构，但不包括图加载、分区或行走轨迹收集。然后使用线性回归估计整个执行时间。

3.2.2 整体表现

图 13 和图 14 给出了算法和输入图（分别为非加权和加权版本）组合的总体实验执行结果。

| Time in seconds | | Gemini | KnightKing | Speedup ×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* |

图 13 非加权

| Time in seconds | | Gemini | KnightKing | Speedup ×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* |

图 14 加权

KnightKing 通过静态行走（DeepWalk 和 PPR）执行了其统一的采样工作流程，但没有实际执行拒绝采样。因此它对 Gemini 的性能优势来自其系统方面。总的来说，KnightKing 领先 Gemini 高达 16.94 倍。

除了 Gemini 固有的图分区的限制（一个顶点通过分散在不同节点上的镜像访问其边缘），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。当边被分配了权重（图），两个系统的执行时间都有适度的增加。

在所有的输入图和两个系统中，PPR 比 DeepWalk 慢得多，这是因为它的非决定性行为。虽然它的预期游走长度也是 80，与 DeepWalk 的（固定）长度相匹配，但 PPR 的最长行走长度超过 1000，产生了较长的执行时间。

当涉及到动态算法时，KnightKing 的拒绝采样带来了压倒性的优势。使用传统抽样的 Gemini，看到 Meta-path 的游走执行时间大幅增长，而 node2vec 则爆炸性增长。对于这两种算法，Twitter 图的行走不能在 6 小时内完成，而 node2vec 在 UK-Union 上估计需要 500 小时以上（8 个节点上共有 128 个线程）。

KnightKing 执行精确采样，并将这种具有似乎无法驯服的固有成本的执行减少到接近静态算法的时间长度。在两种算法和所有输入图（加权或未加权）中，KnightKing 在 200 秒内完成，将 O(n) 边缘采样减少到 O(1)水平。

## 3.3 Flashmob

3.3.1 实验设置

Test platform.实验使用戴尔 PowerEdge R740 服务器，运行 Ubuntu 20.04.1 LTS。它有两个 2.60GHz Xeon Gold 6126 处理器，每个都有 12 个内核和 296GB DRAM。每个核心都有一个私人的 32KB L1 和 1MB L2 缓存，而一个插座内的所有核心共享一个 19.75MB L3 缓存（LLC）。

Graph Datasets.用图系统中常用的 5 个真实世界的图数据集进行测试（图 15），包括三个社交网络（YouTube、Twitter 和 Friendster），以及两个网络图（UK-Union 和 YahooWeb）。

| Graphs | \|V\| | \|E\| | CSR Size |
|---|---|---|---|
| YouTube (YT) [64] | 1.14M | 4.95M | 50.8MB |
| Twitter (TW) [50] | 41.65M | 1.47B | 11.4GB |
| Friendster (FS) [93] | 65.61M | 1.81B | 14.2GB |
| UK-Union (UK) [7] | 131.81M | 5.51B | 42.5GB |
| YahooWeb (YH) [92] | 720.24M | 6.64B | 57.5GB |

**Table 4.** Graphs used (0-degree vertices removed)

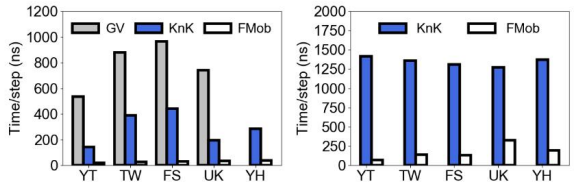
图 15 测试集

Systems for Comparison.与两个最先进的系统进行比较。GraphVite 是第一个用于共享内存环境的高性能 CPU-GPU 混合节点嵌入系统，使用 CPU 进行随机漫步的边缘采样，使用 GPU 进行节点嵌入训练，报告称比主要的基于 CPU 的节点嵌入系统的速度提高了 50 倍。这里将 FlashMob 与 GraphVite 的随机行走组件进行比较。KnightKing 是一个通用的随机行走引擎，与之前的系统相比，速度提高了几个数量级。

Random Walk Algorithms.用两种流行的节点嵌入算法来评估 FlashMob：DeepWalk，一种具有静态过渡概率的一阶行走，随机均匀地选择节点；node2vec，一种具有动态过渡概率的二阶行走，在 BFS 和 DFS 之间插值。

### 3.3.2 整体表现

图 16 给出了在 5 个真实世界的图形上的整体游走性能，以每步时间为单位，比较了 FlashMob 和两个基准系统。




(a) DeepWalk (b) node2vec

**Figure 8.** Overall speed. DeepWalk with FlashMob (short bars): 21.5, 29.0, 32.4, 36.1, and 36.7ns/step, respectively.

图 16 整体速度

图 8a 显示了 DeepWalk 的结果，其中 KnightKing 比 GraphVite 快 2.2-3.8 倍，因为它的边缘访问效率更高。同时，FlashMob 比 KnightKing 的速度提高了 5.4-13.7 倍，在 40ns 内完成一个游走步骤。对于较小的图形，它能够实现更快的速度。相比之下，FlashMob 设法对较小的图形进行了额外的优化，逐渐将其每步时间从 YH 上的 37ns 降低到 YT 上的 21ns。

图 8b 给出了 node2vec 的结果。这里省略了 GraphVite，因为它明显滞后，会严重影响数据的呈现。二阶游走使边缘采样变得更加复杂。尽管如此，FlashMob 比 KnightKing 实现了 3.9-19.9 倍的速度提升。尽管 FlashMob 再次对这种查找进行了批处理，但计算不再受限于单个 VP 内。

## 3.4 ThunderRW

### 3.4.1 实验设置

Datasets.图 17 列出了实验中 12 个真实世界图形的统计数据。这些数据集来自不同的类别，如网络、社会和引文，并且具有不同的密度。顶点的数量从几十万到几千万不等，而边的数量从几百万到几十亿不等。

**Table 5: Properties of real-world datasets.**

| Dataset | Name | $\|V\|$ | $\|E\|$ | $d_{avg}$ | $d_{max}$ | Memory |
|---|---|---|---|---|---|---|
| amazon | *am* | 0.55M | 1.85M | 3.38 | 549 | 0.01GB |
| youtube | *yt* | 1.14M | 2.99M | 5.24 | 28754 | 0.03GB |
| us patents | *up* | 3.78M | 16.52M | 8.74 | 793 | 0.17GB |
| eu-2005 | *eu* | 0.86M | 19.24M | 44.74 | 68963 | 0.15GB |
| amazon-clothing | *ac* | 15.16M | 63.33M | 4.18 | 12845 | 0.35GB |
| amazon-book | *ab* | 18.29M | 102.12M | 5.58 | 58147 | 0.52GB |
| livejournal | *lj* | 4.85M | 68.99M | 28.45 | 20333 | 0.54GB |
| com-orkut | *ot* | 3.07M | 117.19M | 76.34 | 33313 | 0.89GB |
| wikidata | *wk* | 40.96M | 265.20M | 6.47 | 8085513 | 1.29GB |
| uk-2002 | *uk* | 18.52M | 298.11M | 32.19 | 194955 | 2.30GB |
| twitter | *tw* | 41.66M | 1.21B | 58.08 | 2997487 | 9.27GB |
| friendster | *fs* | 65.61M | 1.81B | 55.17 | 5214 | 13.71GB |

图 17 数据集

Workloads.研究了 PPR、DeepWalk、Node2Vec 和 MetaPath 来评估性能和通用性。

### 3.4.2 整体表现

图 18 给出了四个 RW 算法的竞争方法的总体比较。尽管 GW 是并行的，但它比 BL，即顺序基线算法运行得慢。KK 比 GW 和 BL 跑得快，但比 HG 慢，因为与 HG 相比，该框架产生了额外的开销；而且 HG 为每种算法采用了适当的采样方法。TRW 比 GW 和 KK 分别快 54.6-131.7 倍和 1.7-14.6 倍。

**Table 6: Overall performance comparison (seconds).**

| | PPR | | | | | DeepWalk | | | | Node2vec | | | | MetaPath | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | *BL* | *HG* | *GW* | *KK* | *TRW* | *BL* | *HG* | *KK* | *TRW* | *BL* | *HG* | *KK* | *TRW* | *BL* | *HG* | *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** | 203.86 | 20.42 | 21.40 | **4.56** | 12630.01 | 34.36 | 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** |

图 18

受益于并行化，HG 在 PPR 和 DeepWalk 上比 BL 实现了 7.5-10.5 倍的速度提升。此外，HG 在 Node2Vec 和 MetaPath 上的运行速度分别比 BL 快 38.3-1857.9 倍和 11.1-28.5 倍，因为 HG 对 Node2Vec 采用了 O-REJ 采样，避免了每一步对 Q 邻居的扫描，并对 MetaPath 采用 ITS 采样，其初始化阶段比实际的 ALIAS 更有效率。TRW 的运行速度比 BL 快 8.6-3333.1 倍。即使与 HG 相比，TRW 也实现了高达 6.1 倍的速度，这得益于以步进为中心的模型和步进交错技术。由于 MetaPath 是动态的，而且 TRW 和 HG 都使用 ITS 采样，每一步的收集操作在成本上占主导地位。但在 12 个图中，ThunderRW 上的 MetaPath 仍优于 HG 上的 MetaPath，而在另外三个图上则稍慢一些。因此，MetaPath 对 tw 的执行时间要比对其他图形的执行时间长得多。

综上所述，ThunderRW 的性能大大超过了最先进的框架和自制的解决方案。此外，与 BL 和 HG 相比，ThunderRW 在 RW 算法的实现和并行化方面节省了大量的工作量。

### 3.5 C-SAW

#### 3.5.1 实验设置

Dataset.使用表中的图数据集来研究 C-SAW。这个数据集包含了广泛的应用，如社交网络（LJ、OR、FR 和 TW）、论坛讨论（RE 和 YE）、网上购物（AM）、引文网络（CP）、计算机路由（AS）和网页（WG）。

| Dataset | Abbr. | Vertex Count | Edge Count | Avg. degree | Size (of CSR) |
|---|---|---|---|---|---|
| Amazon0601 [56] | AM | 0.4M | 3.4M | 8.39 | 59 MB |
| As-skitter [56] | AS | 1.7M | 11.1M | 6.54 | 325 MB |
| cit-Patents [56] | CP | 3.8M | 16.5M | 4.38 | 293 MB |
| LiveJournal [56] | LJ | 4.8M | 68.9M | 14.23 | 1.1 GB |
| Orkut [56] | OR | 3.1M | 117.2M | 38.14 | 1.8 GB |
| Reddit [12], [57] | RE | 0.2M | 11.6M | 49.82 | 179 MB |
| web-Google [56] | WG | 0.8M | 5.1M | 5.83 | 85 MB |
| Yelp [12], [57] | YE | 0.7M | 6.9M | 9.73 | 111 MB |
| Friendster [56] | FR | 65.6M | 1.8M | 27.53 | 29 GB |
| Twitter [58] | TW | 41.6M | 1.5M | 35.25 | 22 GB |

TABLE II: Details of evaluated graphs.

图 19 数据集

Metrics.作者引入了一个新的指标－每秒采样边数（SEPS）来评估采样和随机游走的性能，而不是经典图分析学中的每秒遍历边数（TEPS）。内核执行时间被用来计算 SEPS，即生成样本的时间，除了内存外的情况下还包括传输分区的时间。

Test Setup.与 GraphSAINT 类似，为随机游走算法生成 4000 个实例，为采样算法生成 2000 个实例。

#### 3.5.2 整体表现

将 C-SAW 与 KnightKing 和 GraphSAINT 进行比较。分析结果显示，GraphSAINT 和 KnightKing 都使用多线程来进行计算。由于 KnightKing 只支持随机游走的变化，将 C-SAW 与 KnightKing 进行比较，以获得有偏见的随机游走。GraphSAINT 提供了 Python 和 C++两种实现方式。作者选择 C++实现，表现出更好的性能。

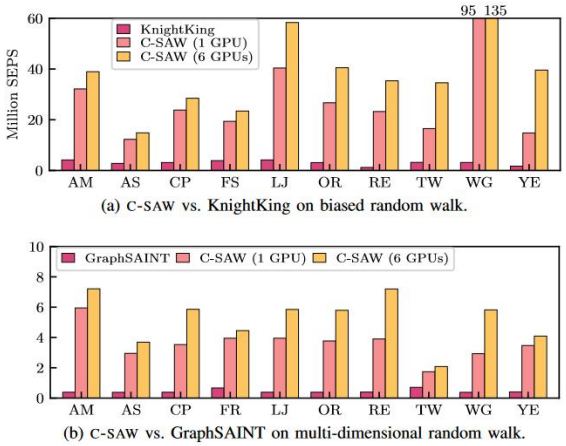



图 20 C-SAW 实验对比

## 4 总结与展望

作为图数据分析和机器学习的一种工具，图随机游走逐渐受到更多的欢迎，专门针对其进行优化的计算框架引擎也逐渐提出，本文介绍了其中两种随机游走引擎。

本文介绍了五种随机游走引擎 GraphWalker、KnightKing、Flashmob、ThunderRW 以及 C-SAW，每一种随机游走框架针对随机游走算法不同的性能问题做了相应优化。

GraphWalker 为了有效地支持快速和可扩展的随机游走，开发一个具有异步游走更新功能的状态感知 I/O 模型。这些用于随机游走过程的框架引擎极大优化和遍历了随机游走的计算过程。KnightKing 是第一个通用的、分布式图随机行 走引擎，它提供了一个统一的框架来支持各种随机游走，并集中优化了游走过程。Flashmob 指出了现有的算法不能有效地利用 CPU 内存层次结构的问题，并由此解决了内存访问的随机性、数据依赖和加载到缓存里的数据重用率不高的问题。ThunderRW 设计了一个以步进为中心的模型，将计算从移动查询的一个步进的局部视图中抽象出来。进一步提出了步进交错技术，通过交替执行多个查询来解决内存访问延迟。C-SAW 引入了新的以偏差为中心的框架、双区域搜索、工作负载感知的 GPU 和多 GPU 调度。

## 参 考 文 献

[1] Wang R, Li Y, Xie H, et al. Graphwalker: An i/o-e ff icient and resource-friendly graph analytic system for fast and scalable random walks[C]//2020 USENIX Annual Technical Conference (USENIXATC 20). 2020: 559-571.

[2] Yang K , Zhang M X , Chen K , et al. KnightKing: a fast distributed graph random walk engine[C]// the 27th ACM Symposium. ACM, 2019.

[3] Yang K , Ma X S , Thirumuruganathan S , et al. Random Walks on Huge Graphs at Cache Efficiency[C]// the 28th ACM Symposium. ACM, 2021.

[4] Sun S, Chen Y, Lu S, et al. ThunderRW: An inmemory graph random walk engine[J]. 2021.

[5] Pandey S, Li L, Hoisie A, et al. C-SAW: A framework for graph sampling and random walk on GPUs[C]//SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020: 1-15.