基于随机游走的图处理系统综述

汤贤林 [1)]

[1)](华中科技大学 武汉光电国家研究中心，湖北省武汉市 435400)

摘 要 图处理系统因为其在在各种应用中具有显著的有效性，受到学术界和工业界广泛的关注。在图的处理上，主流有 BFS 的方法和随机游走的方法，其中随机游走可以通过捕获所需的图属性来帮助学习、挖掘、分析和可视化大型图，并且大大减少原始图的大小，因此受到很大的欢迎。目前，随机游走仍然存在因为随机游走策略的不确定性带来的 I/O 问题，针对随机游走工作负载进行优化的系统也逐渐被开发。GraphWalker 部署一种具有异步行走更新的新型状态感知 I/O 模型。ThunderRW 是一个通用且高效的内存 RW 框架。GraSorw 是一种 I/O 高效的基于磁盘的大型图随机游走系统。NosWalker 是以随机游走序列为核心对边预采样的随机游走系统。SOWalker 通过优化加载数据块的策略和调整连续移动的范围优化了二阶随机游走。

关键词 图 图处理系统 随机游走 I/O 优化

Overview of Graph Processing Systems Based on Random Walks

Tang XianLin[1)]

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

Abstract Graph processing systems have garnered significant attention from both academia and industry due to their remarkable efficacy across various applications. In graph processing, the mainstream methods include BFS (Breadth-First Search) and random walk approaches. Random walks, particularly, aid in learning, mining, analyzing, and visualizing large graphs by capturing required graph properties and significantly reducing the size of the original graph. Hence, they are widely favored.

However, random walk strategies still face I/O challenges stemming from the uncertainties associated with the strategies. Consequently, systems optimized for random walk workloads are being progressively developed. For instance, GraphWalker deploys a novel state-aware I/O model with asynchronous walk updates. ThunderRW stands as a versatile and efficient in-memory RW framework. GraSorw represents an I/O-efficient disk-based system for large-scale graph random walks. NosWalker revolves around edge pre-sampling for random walk sequences. SOWalker optimizes second-order random walks by refining data block loading strategies and adjusting the range of consecutive movements.

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

1 引言

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。

从 14 年 DeepWalk 开始，基于图与自然语言都符合幂律分布的发现，随机游走开始应用于图处理系统并且使用自然语言处理的方法进行学习。随机游走解决图嵌入问题。具体来讲就是将节点的连接信息，关联信息，社群信息，结构信息，联通信息转化为低维连续稠密的向量，来帮助后续的分类或者预测工作。随机游走的重点在结构信息，与采样结果有关与具体节点信息无关，可以挖掘出邻近，局部，结构和死胡同的特征，并且在稀疏标注的场景上更有优势。

图随机游走算法是一种为了从图中的实体间提取路径信息的高效的、应用广泛的图处理工具。它为许多重要的图度量、排序和图嵌入算法奠定了基础，比如 PageRank、SimRank、DeepWalk 和 Node2Vec 等，这些算法可以独立工作，也可以作为机器学习任务的预处理步骤。基于对图随机游走算法应用的分析，发现基于随机游走的图计算过程具有如下特征：1）对图数据访问的随机性更强，2）并发随机游走的数量可能很大，3）随机游走的步长可能非常长。随机游走仍然存在因为随机游走策略的不确定性带来的 I/O 问题，针对随机游走工作负载进行优化的系统也逐渐被开发。

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。复杂的随机游走算法以采样复杂性为代价实现了更灵活的、特定于应用程序的游走。对于 node2vec 这类高阶随机游走算法，会因为边缘采样而陷入困境，产生巨大的采样成本：在每一步，出边选择都需要重新计算所有出边的转移概率，其开销随着 walker 当前所在顶点的度而增长。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。也因此出现了针对二阶随机游走进行优化的论文。

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。

2 背景

2.1 随机游走基础知识

游给定一个图$G = (V, E)$和一个起始顶点$u \in V$，随机行走者 w 进行如下的操作。u 的每一个邻居 v 有一个转移概率，该概率决定了 v 会被选为下一个顶点的可能性。典型地，转移概率$p(v|u)$只取决于 u，这种情况叫一阶随机游走。在高阶随机游走中，概率以$p(v|u, t, s, \ldots)$的形式指定，其中, s, t 是当前行走中 u 的前驱，计算出边的转移概率时要考虑 w 的行走历史。w 根据这个概率采样一条进行，并重复这一操作直到满足特定的终止条件。终止可以是确定性的(在进行了给定的步数之后)，也可以是随机的(行走者在每一步以固定的概率退出)。

基于随机游走的算法都遵循上述框架，不同随机游走算法的关键变化在于邻边的选择步骤。边被选择的相对概率上来看，随机游走算法可以分为无偏和有偏。无偏是指出边转移概率不依赖于它们的权值或者其他性质，而有偏是指在随机游走过程中对邻边的选择进行加权。如果边转移概率在整个过程中保持不变，则为静态随机游走，否则为动态随机游走，其中的转移概率的确定涉及到随机游走的状态，它在游走过程中不断变化。因此，在动态随机游走过程中，需要在每一步重新计算这些概率，而不是预先计算所有的边转移概率。

由于随机游走在节点嵌入应用中被广泛使用，在这里也给出节点嵌入的简单介绍。节点嵌入是图学习的基本组成部分。给定一个图$G = (V, E)$和一个维度$d << |V|$，节点嵌入是把每个节点$v \in V$用一个 d 维向量$Emb(v)$代表，使图 G 的结构信息得以保留。直观地说，如果一个节点对 u 和 v 有“相似”的邻域，那么它们的嵌入$Emb(u)$和$Emb(v)$也

彼此接近。反之，具有不同邻域的两个节点的嵌入会相距更远。通过在正节点对集合 P 和负节点对集合 N 上训练深度学习模型来学习节点嵌入，使得$(u,v) \in P$的节点嵌入彼此更接近，而$(u,v) \in N$的节点嵌入则彼此相距较远。通常，N 是通过随机选择节点对构建的（考虑到现实世界图的稀疏性，它们不太可能连接），而 P 是通过随机游走构建的。

与传统的图计算框架从图数据的角度抽象计算相比，现有的随机游走框架采用了以游走步进为中心的模型，将每个查询视为并行任务。

广泛使用的节点嵌入算法有 DeepWalk 和 Node2Vec，它们在使用随机游走衡量邻域相似性的方式上有所不同，使用不同的转移概率定义。更具体地说，DeepWalk 执行一阶均匀随机游走，而 Node2Vec 则是一种带有超参数的二阶算法。通常，这两种算法都采用默认参数，从图中的每个节点开始，执行 10 次长度分别为 40 和 80 的随机游走。

2.1.1 DeepWalk

DeepWalk 是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。

2.1.2 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:

$$\alpha_{pq}(u,v,z) = \begin{cases} \frac{1}{p}, & d_{uz} = 0 \\ 1, & d_{uz} = 1 \\ \frac{1}{q}, & d_{uz} = 2 \end{cases}$$

图 1　Node2vec 公式

这里 p 和 q 是用户配置的超参数，其中 p 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。

2.2 随机游走算法流程

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。

不同的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 随机游走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随机游走分为静态和动态两类。如果在执行前确定了转移概率，则随机游走是静态的。否则是动态的，受随机游走查询状态的影响。

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

2.3 随机游走采样方法

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。主要有四种抽样技术，

包括朴素抽样、逆变换抽样、偏置采样和拒绝采样。其中拒绝采样目前受到更多的使用。

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，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$kq(z) \geq p\sim(z)$。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$u0 < p\sim(z0)$，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。

KnightKing [1]是第一个通用随机游走运算的框架，是一个分布式的随机游走引擎。KnightKing 的效率和可扩展性的核心是其快速选择下一条边的能力。其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的随机游走算法的静态和动态转移概率。基于这个算法定义框架，KnightKing 消除了扫描 walker 当前所在顶点所有出边以重新计算它们的转移概率的需要。

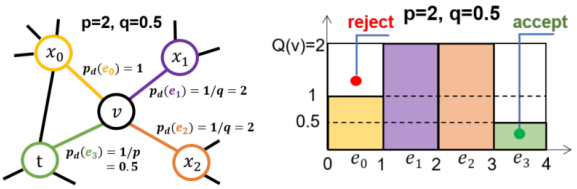


图 2 拒绝采样图

为了对一条边进行采样，随机采样一个位置(x, y)，该位置在由线 $y = Q(v)$ 和 $x = |E_v|$ 以及 x 和y 轴覆盖的矩形区域内均匀分布。即，想象一下在该矩形内投掷飞镖。采样的 x 值给出候选边 e，而 y 值将与其对应的 P_d 进行比较。如果 $y \leq P_d(e)$（飞镖击中木条），则认为 e 为成功样本；否则（飞镖未击中任何木条），e 被拒绝，这需要下一次抽样试验，直到成功。

3 相关研究

3.1 GraphWalker

GraphWalker[2]是一种针对单机随机游走的 I/O 高效和资源友好的设计。为支持非常大量的游走，比如数百亿次的游走，而且还支持非常长的游走，比如每次游走数千步。为了实现这一目标，主要思想是采用状态感知模型，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。总之，与基于迭代的模型盲目顺序加载图块不同，状态感知模型选择加载包含最大游走数的图块，并使每次游走尽可能多地移动步数，直到到达加载子图的边界。这样游走可以在每次 I/O 的时候尽可能地被更新。GraphWalker 可以支持快速且可扩展的随机游走。主要由三个部分组成（1）状态感知图加载；（2）异步游走更新；（3）以块为中心的游走管理。

3.1.1 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 3 所示的图结构对应的 CSR 格式为图 4 所示。

这种轻量级的图数据组织方式降低了每个子图的存储成本，从而降低了加载图的时间成本。因为 GraphWalker 通过简单地读取索引文件来记录每个块的起始顶点来对图进行分区，它还可以灵活地为不同的应用程序调整块的大小。GraphWalker 根据顶点 id 将图划分为块，根据 id 的升序排列依次添加顶点和它们的出边进入块中，直到块中的数据容量达到预设的容量大小，然后创造一个新的块。

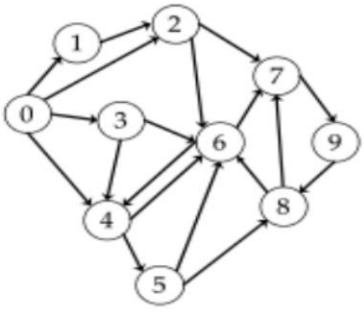


图 3 示例图

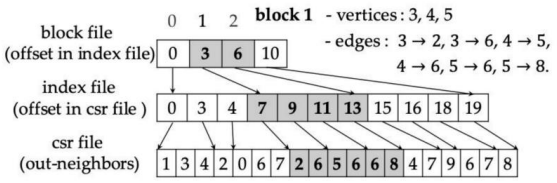


图 4　CSR 数据结构

GraphWalker 在预处理阶段转换图形格式并划分图形块。在运行随机遍历的阶段，GraphWalker 选择一个图块，并根据遍历的状态将其加载到内存中，特别是，它将加载包含最多遍历的块，在完成对加载的图形块的分析后，它然后选择另一个块以同样的方式加载。图加载过程在图 5 中进行了展示。为了减轻块大小的影响并提高缓存效率，GraphWalker 还支持块缓存，方法是开发一种行走感知的缓存方案，将多个块保存在内存中，其基本原理是具有更多随机游走访问的块更有可能在将来被访问。

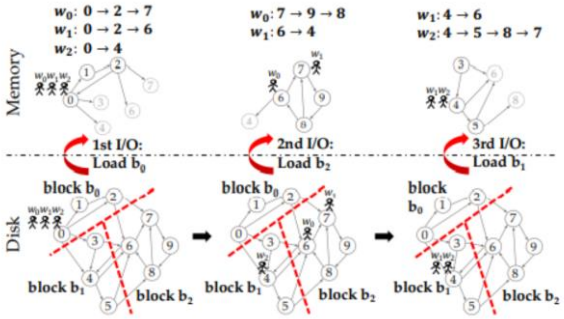


图 5　状态感知模型

首先根据状态感知模型选择一个候选块，为了加载这个块，需要检查它是否缓存在内存中。如果它已经在内存中，那么直接访问内存来执行分析。否则，将从磁盘加载它，并且如果缓存已满，还将取出内存中包含最少遍历的块，缓存在内存中的最大块数量取决于可用的内存大小。实验表明对子图分块敏感方案总是优于传统的缓存方案。

3.1.2 异步游走更新

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。

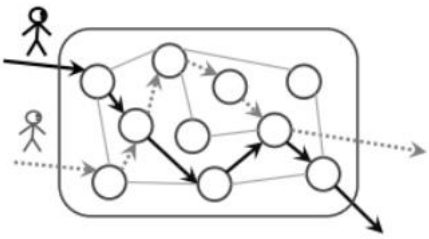


图 6　异步行走更新

然而，状态意识模型可能会导致全局滞后问题。也就是说 GraphWalker 可以快速地完成大多数遍历，但是要完成剩下的几个遍历需要很长时间。为了解决查问题 GraphWalker 引入了一种概率方法来处理状态感知的图加载过程。这样做是为了让掉队的随机游走有机会多走几步，这样他们就能跟上大多数随机游走的步伐。具体来说，每次选择一个图块来加载时，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$1-p$加载具有最多行走的图块。

3.1.3 以块为中心的游走管理

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。

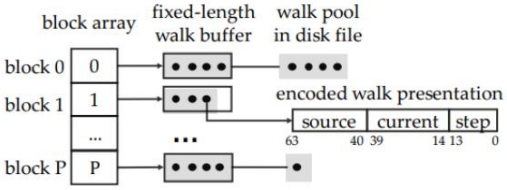


图 7　以块为核心管理组织结构

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。

3.1.4 主要贡献

GraphWalker 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。提出了（1）状态感知图加载；（2）异步游走更新；（3）以块为中心的游走管理。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。

3.2 ThunderRW

ThunderRW[3]是一个通用且高效的内存随机游走框架。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。是在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的背景下提出的一种高效的内存随机行走引擎。ThunderRW 的核心设计是由作者的分析结果驱动的:常见的随机游走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，以抽象不同的随机游走算法。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。

此外，ThunderRW 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。在以步进为中心的规划模型的基础上，ThunderRW 提出了步进交错技术来解决软件预取过程中由于不规律的内存访问而导致的缓存延迟问题。由于现代的 CPU 可以同时处理多个内存访问请求，步进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，这利用了不同随机游走查询之间的内存级别并行性。

3.2.1 以步进为中心的模型

随机游走算法建立在多个游走查询的基础上，而不是单个的查询。尽管随机游走算法内部查询的并行性有限，但由于每个随机游走查询都可以单独执行，因此存在大量查询并行性。所以 ThunderRW 以步进为中心的模型从查询的角度抽象了随机游走算法，以利用查询间的并行性。ThunderRW 从一个查询 Q 进行一步移动的局部视角，对计算进行建模。在以步进为中心的模型中，用户通过“像步行者一样思考”来开发随机游走算法，他们专注于定义函数设定转移概率并且更新在查询 Q 每一步中的状态，方便了用户定义面向步进的函数。

使用以 step 为中心的模型，用户通过“think-like-a-walker”来开发随机游走算法。用户只需要专注于定义算法、设置转移概率和更新每一步状态的函数，然后 ThunderRW 将帮助用户定义的面向 step 的函数应用于随机游走查询。

3.2.2 步交错技术

ThunderRW 创新提出了步交错技术，它减少了由随机内存访问引起的流水线停顿。ThunderRW 使用软件预取技术来加速 ThunderRW 的内存计算，然而对于一个查询 Q 的单独一步来说，没有足够的计算工作负载来隐藏内存访问延迟，因为查询 Q 的每一步之间相互独立，因此 ThunderRW 通过交替执行不同的查询步骤来隐藏内存访问延迟。当给定一组操作序列后，将它们分解为多个阶段，这样，一个阶段的计算将使用前一个阶段生成的数据，并在必要时检索后续阶段的数据。然后同时运行一组查询，一旦一个查询 Q 的一个阶段完成了，马上转移到一个组中的其他查询上，随后当其他查询完成后，继续执行查询 Q。通过这种方式，ThunderRW 可以在单个查询中隐藏内存访问延迟，并保持 CPU 繁忙，这种方法称为步交错技术。

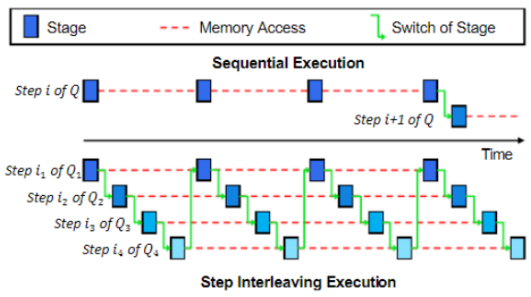


图 8 顺序交错与步交错

图 8 给出了一个示例，其中一个步骤分为四个阶段。如果按顺序逐步执行查询，则 CPU 会因内存

访问而频繁停滞。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。

ThunderSW 用阶段依赖图(SDG) 来对步骤中一系列操作的阶段进行建模。如图 7 所示，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。给定操作顺序，分两步构建 SDG，抽象阶段（节点）和提取依赖关系（边）。

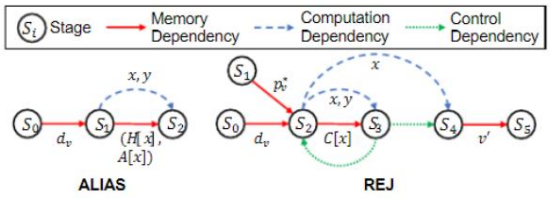


图 9　阶段依赖图

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。

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。

3.2.3 主要贡献

作者提出了 ThunderRW，一种高效的内存随机游走引擎，用户可以轻松地在其上实现定制的随机游走算法。ThunderRW 设计了一个以步长为中心的模型，将计算从查询移动一步的局部视图中抽象出来，在此基础上，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 四种有代表性的随机游走算法。实验结果显示 ThunderRW 的性能比目前最先进的随机游走框架高出一个数量级，并且步进交错将内存限制从 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。

3.3 GraSorw

GraSorw [4]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。由于二阶随机游走模型的复杂性和内存限制，单机运行基于二阶随机游走的应用不具有可扩展性。现有的基于磁盘的图系统只对一阶随机游走模型友好，并且在执行二阶随机游走时会遇到昂贵的磁盘 I/O 开销。首先，为了消除大量轻量级顶点 I/O，开发了一个双块执行引擎，通过应用新的三角形双块调度策略、基于桶的遍历管理和倾斜遍历存储，将随机 I/O 转换为顺序 I/O。其次，为了提高 I/O 利用率，我们设计了一个基于学习的块加载模型，以利用全载入和按需加载方法的优势。

3.3.1 GraSorw 基本流程

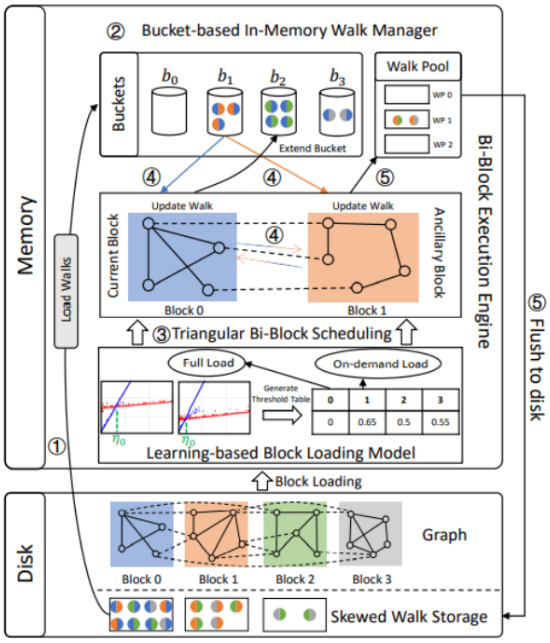


图 10　GraSorw 执行流程

引擎利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至内存，并与内存中已有的中间行走池进行整合，形成当前的行走。

然后，管理器将当前行走切分成桶，每个桶储存着具有相同块集的行走，其中每个块都包含它们之前和当前所驻留的顶点。这种基于桶的内存中行走管理方式将大量的顶点 I/O 合并为单个块 I/O。

在 GraSorw 中，考虑到之前和当前所涉及的顶点跨越了两个块，在处理每个桶之前，我们需要将另一个块加载到内存中，称为辅助块。在这个系统中，每个辅助块对应于一个桶，而当前块则在所有桶之间共享。在每个时间段内，双块执行引擎采用三角双块调度方法来确定辅助块的加载顺序，并利

用基于学习的块加载模型来实现块的加载。

在加载块之后，引擎会异步地更新桶中的行走。在这些行走中，当前顶点可能位于当前块或者辅助块中，如图中所示的蓝色和橙色箭头所指示的那样。此外，由于存在一些连接着两个块的边，行走也可以跨越这两个块进行更新。当行走到达内存中不属于块的任何顶点或者达到了终止条件时，遍历更新过程将停止。

对于前一种情况需要保持中间步行的持久性信息，以便将来进行更新。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。

3.3.2 双块调度策略

双块执行引擎的基本思想是保证当前和以前的顶点都在内存中，在内存中保留当前块和辅助块两个块。

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。

获得当前块 I/O 的最小数量是一个 NP 困难的问题，因为二阶随机游走的块访问序列是未知的。为了解决这个问题，我们需要设计一个在线算法。然而，大多数现有的启发式在线解决方案在最优解方面存在着较大的误差界限，要么没有最优解，要么误差较大。因此，我们进行了对不同当前块调度策略的实证研究，发现没有一种方法适用于所有数据集，同一方法在不同数据集上的性能可能差异很大。然而，基于迭代的方法通常在大多数情况下表现最佳。基于这些观察，本文选择采用基于迭代的方法调度当前块，并着重开发一种新的调度策略，以优化辅助块 I/O。

GraSorw 提出支持三角形双块调度策略的倾斜行走存储和基于桶的内存行走存储有助于将随机顶点 I/ O 聚集到块中。

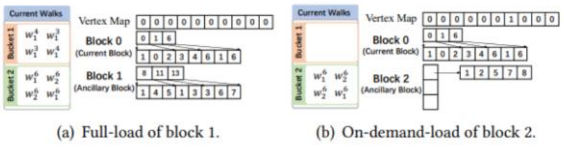
传统的步长存储方法将步长与其当前顶点所属的块关联起来。这在三角形双块调度策略下进行更新时带来了限制。GraSorw 中的倾斜行走存储同时考虑了行走的前一个顶点和当前顶点来安排行走。与传统的 walk 存储相比，这种存储将当前顶点属于同一块的 walk 分成两组。一组包含当前顶点属于比以前顶点所属的 ID 更大的块的行走;另一组包含剩余的步行。因此，在三角双块调度策略中，当包含当前顶点的块被加载为辅助块时，将处理第一组行走；而当包含当前顶点的块被加载为当前块时，将处理第二组。

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。

3.3.3 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。

满载模式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。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。



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

图 11 不同加载模式对比

图 11 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。由于如图 11(a)所示，决定使用全负载方法加载块 1，因此索引文件和 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 1 中的行走更新后，块 1 的内存被释放。在执行桶 2 之前，需要扫描其中的所有行走来计算块 2 的激活顶点。在图 11(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-demandload 方法可节省 18.8%的块 I/O。

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，因此很难开发出一个启发式方法来构建成本模型。在本文中，作者开发了一个基于学习的模型，用于预测每种加载方法的成本。在扩展新激活的顶点时（执行阶段），遍历更新会引发新的磁盘 I/O。相比全负载模式，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。

3.3.4 主要贡献

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。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。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。

3.4 NosWalker

NosWalker[5]采用了一种基于行走者的调度（walker oriented scheduling）系统架构，代替了传统的面向图的调度方式。传统的图分析算法通常使用通用的分布式图处理框架，但这些系统并未充分利用随机游走应用的独特特性。随机游走的采样过程可以与行走者（walkers）的处理过程分离，这使得系统只需在内存中保留预先采样的结果，通常远远小于整个边集。此外，在随机游走中，主导整体性能的是每秒移动的步数，而不是行走者的数量。因此，使用独立的行走者时，无需同时处理所有行走者。

文章首先提出两个定理 1.随机游走的抽样过程只与边缘数据有关；2. walkers 之间相互独立。因此在 NosWalker 中以 walkers 为核心而不是图，并且在原先以块为核心的调度方法上使用了对边预采样的方法进行了 I/O 上的优化。NosWalker 基于即让行走者保持移动的关键思想设计。换句话说，除了尝试加载更多、更快的图数据外，NosWalker 还尽最大努力确保内存中始终有足够的数据来推进少量行走者。

此外，文章中提到长尾问题，如图 12，即在执行接近结束时，行走者的稀疏程度非常高，因此在没有优化的情况下，出现停滞的可能性很大。为了缓解这个问题，NosWalker 设计了几种机制来自适应地调整行为，包括自适应块粒度、从已完成的行走者中回收内存以保留更多的预先采样边，以及为经常访问的顶点优先分配预先采样边。

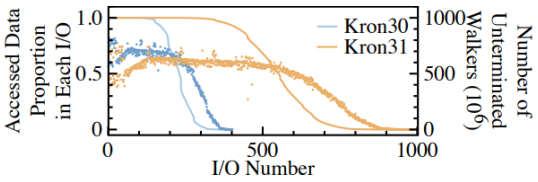


图 12　长尾问题

3.4.1 边采样技术

在 NosWalker 中以 walkers 为核心而不是图，因此不同于 GraphWalker 中以块为中心的 walk 策略与 I/O 方法，只需要对与顶点相关的边进行采样即可。此外，对边采样时也并不是将与顶点相关的所有边加载进内存，而是以一定概率加载部分边。因此，相同大小的内存空间中，这样的边采样策略可以保存更多可能被利用的边信息，避免了以块存储后加载进内存的大部分边实际并没有使用的缺

陷。这种机制的关键优势有两个方面：1.预采样边的大小要比顶点的原始出边集小得多；2.缓冲的预采样结果的分布不受块分割的限制。

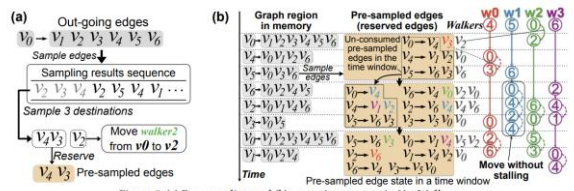


Figure 5: (a) Pre-sampling and (b) execution process in NosWalker.

图 13 NosWalker 基本流程

图 13 中的 a 展示了对边采样的过程，确定顶点$v0$后，在$v0$可以到达的下一个顶点的集合中随机挑选顶点，如图中随机选择的是$v2$，$v3$，$v4$，然后除了当前直接会使用的$v0$到$v2$的边，并且将当前的边消耗掉不再使用，剩余的边会留在内存中，以达到预采样的效果，帮助后续以 walker 为核心的行走策略。利用内存中存在的边进行连续的走动，正如 GraphWalker 中利用内存中加载的块进行连续的走动。

图 13 中的 b 展示了 NosWalker 行走的流程，以$w0$举例，首先他从$v4$开始，图中采样了$v0$，$v2$，$v1$，首先从$v4$到$v0$，并消耗了这条边，然后发现内存中存在因为$w2$的行走，而预采样的$v0$出发的边，因此利用内存中存在的边进行连续的走动，$w0$继续从$v0$到$v3$，并消耗了这条边。再关注$w1$，可以看出他利用内存中存在的边连续行走了很多步，但并没有重新去进行 I/O，只需要利用内存中存在的预采样的边的数据。

总之，边采样技术是在以 walker 为核心的行走策略上设计出来的方法。具体来说，在读取完一个顶点的全部出边集之后，系统可以利用预采样技术计算多个采样结果，并将它们存储在内存中以备将来使用。在满足 walker 可以连续行走的同时，减少了内存中需要存储数据的规模大小。

3.4.2 自适应块粒度技术

在图 12 的长尾效应中，最后 30%的时间实际上只剩下 3%的 walkers 仍在活跃。并且因为 walkers 后期分布分散，导致 I/O 效率低。

当活跃 walkers 与图相比足够密集时，NosWalker 使用粗粒度块模式，连续加载最热门的粗粒度块。在高性能的 Linux 本地异步 I/O 访问库的帮助下，一个单线程就足以实现 SSD 的最大顺序读取吞吐量。当活跃 walkers 变得稀疏时，只有大块数据中的一部分是需要的。在这种情况下，NosWalker 切换到细粒度块模式。在这种模式下，NosWalker 试图确定应该加载哪些细粒度块，并发起针对这些已确定块的精确 I/O 请求。由于 SSD 的底层硬件通常将 4 KiB（一个 SSD 页面）作为 I/O 操作中可读的最小单位，NosWalker 在概念上将每个粗粒度块划分为 4 KiB 的细粒度块，并以 4 KiB 的块粒度发起 I/O 操作。这确保了 SSD 的高 IOPS 可以得到充分利用，同时尽可能准确地绕过不需要的数据。

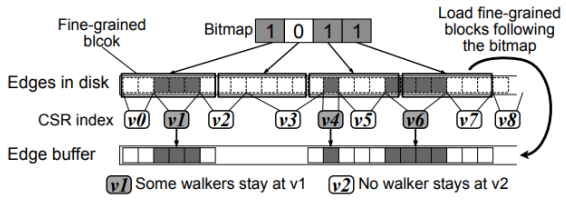


图 14 标记并加载细粒度块

为了支持细粒度块模式，NosWalker 使用位图来指示是否应加载细粒度块。当一个顶点的预采样边全部被消耗时，一些行走者可能会停滞在这些顶点，因此无法移动。例如，在图 14 中，第一个块由于$v1$而被标记，第三和第四个块由于$v4$和$v6$而被标记。其他未标记的块，比如第二个块，因此被跳过以节省 I/O。如图所示，这些细粒度块被加载到边缓冲区中相应的地址。因此，NosWalker 可以像往常一样通过 CSR 索引访问边。这简化了系统实现并提高了边的访问效率。NosWalker 仅在活跃行走者稀疏时使用细粒度块模式。在这种情况下，最初占用大部分内存的行走者池释放了大量内存。因此，NosWalker 可以利用这些内存来容纳位图数组，并将 CSR 索引保留在内存中。

此外，还有动态内存空间的缓冲区设计方法，NosWalker 不是一开始就用小尺寸初始化所有缓冲区并逐渐扩大它们的尺寸，而是通过首先将缓冲区大小设置为足够大且固定，然后逐个分配新缓冲区来支持动态增长的预采样缓冲区。它避免了占用大部分内存的小缓冲区的元数据。当部分内存被释放时，只需要进行一次内存分配来分配一个新缓冲区，而不是为每个缓冲区重新分配内存空间。

对于图中存在的低度顶点，预采样边不是一种高效的做法。低度顶点的边通常只是整个图的一小部分。例如，在 Kron30 中大约有 9%的度为 1 的顶点，而这些顶点仅占大约 0.3%的边。因此，

NosWalker 在应该为低度（1 到 4，由图的大小确定）顶点进行预采样时，直接在内存中保留低度顶点的边。

3.4.3 主要贡献

NosWalker 提出了一种新颖的以 walker 为导向的调度方法，从而设计了一种解耦图加载和预采样过程与行走者处理的架构。它主要提出了 walker 为核心的调度方法下，对边预采样的方法。并且为了缓解长尾问题，它使系统能够自适应地生成行走者，并灵活调整内存中保留的采样结果的分布，使用了自适应块粒度技术。这些优化大幅减少了顶点数据和边数据的磁盘 I/O，分别导致了更高的 I/O 利用率。根据我们的评估，NosWalker 相比于最先进的离线随机游走系统，包括 DrunkardMob、GraphWalker 和 GraSorw，可以实现高达两个数量级的加速。

3.5 SOWalker

SOWalker[6]是一种面向 I/O 优化的核外图处理系统，用于进行二阶随机游走。首先它提出了一个 Walk 矩阵，以避免加载不可更新的行走，并消除无用的行走 I/O。其次，它开发了一个基于 Benefit-aware I/O 模型，使用了模拟退火方法，得到加载具有最大累积可更新行走的多个块的较优解，从而提高了 I/O 的利用率。最后，它采用了基于块集的行走更新策略，允许每次加载的块集中的每个行走移动尽可能多的步数，从而显著提高了行走更新速率。与两个最先进的随机游走系统 GraphWalker 和 GraSorw 相比，SOWalker 取得了显著的性能提升。

3.5.1 Walk 矩阵

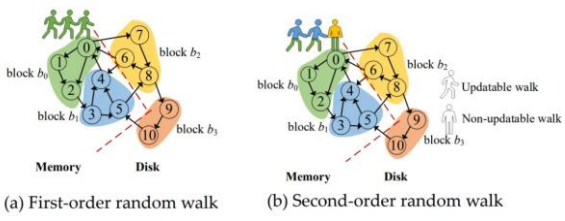


图 15 两种随机游走可更新行走的差异

图 15 展示了一阶随机和二阶随机游走的可更新行走上的差异。在一阶随机游走中当前节点在加载块中，可以直接更新。而二阶随机游走中上一个节点可能不在加载块内，不在内存中，无法更新，需要额外 I/O。一阶随机游走的 3 个 walks 都可以更新，但二阶随机游走中只有两个可更新。当无法更新的 walk 较多时，将会带来无效的 I/O。

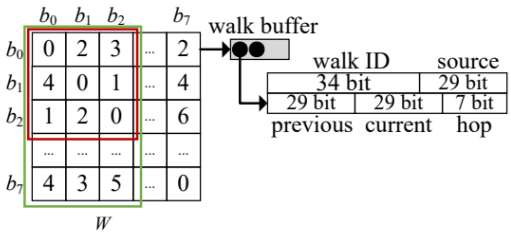


图 16 行走矩阵数据结构

为了跳过加载不可更新的行走并消除无用的行走 I/O，我们使用一个行走矩阵来表示这些行走，数据结构如图 16。矩阵的维度是块的数量。矩阵中的每个元素 (i,j) 表示其前一个顶点属于第 i 块，当前顶点属于第 j 块的行走。所有元素的总和即为未完成行走的数量。行走矩阵根据行走池中的行走状态进行创建和更新。当内存中的所有行走已完成或已到达加载的块集边界时，SOWalker 检查行走状态以获取前一个顶点和当前顶点的块 ID。如果 ID 不同，意味着该行走穿越了块。统计跨块的行走数量并更新行走矩阵的相应元素。基于行走矩阵，SOWalker 可以轻松地检查行走是否可以更新，判断前一个顶点和当前顶点是否都在内存中，从而跳过加载不可更新的行走并消除无用的行走 I/O。需要注意的是，$W(i,i)$ 中的行走数量始终为 0，因为前一个和当前顶点在同一个块中的行走可以在不需要额外的块 I/O 的情况下被更新。

3.5.2 Benefit-aware I/O 模型

二阶随机游走原本方法会加载 walk 上一个节点所在的块作为辅助块，I/O 利用率低。GraphWalker 加载块时，加载块内 walk 数目最多的块，但是一阶上加载块内 walk 数目最多的块有效，但是二阶上不一定有效。二阶上需要前一个块同时也需要当前块，需要加载拥有最多累计可更新 walk 的多个块形成的块集而不能仅仅看当前块。

为了提高 I/O 利用率，本文提出了一个 Benefit-aware 的 I/O 模型，以加载具有最大累积可更新行走的多个块。由于二阶随机游走的前一个和当前顶点可能属于不同的块，我们需要考虑块之间的依赖关系。因此，我们同时安排多个块，

而不是一个块。为此，我们将块调度问题建模为最大边权重团问题。我们还采用了一种高效的启发式算法，以极小的成本提供可比较的 I/O 性能。

线性规划方法，准确但是耗时。而启发式算法中的模拟退火算法，在经过实验后发现是一种有效且高效的方法。

Model	Execution time (s)	Block I/O time (s)	Block I/O number	Computation time (s)
Random	4970	3234	9868	-
Max-*m*	3871	2162	6391	-
Exact	14311	548	1484	12097
BA	2133	575	1537	10

图 17 不同块加载策略耗时的差异

图 17 中可以看出，对比随机选择加载进内存的块的策略其他策略各自有所提升。但是对比根据块内 walk 数目选择最多的 m 个块和线性规划(Exact)的块加载数目，发现加载块内 walk 数目最多的块的方法效果实际上并不是特别好，仍然存在较大优化空间。但是线性规划的方法在计算时间又存在极大的劣势，导致执行总时间很长。对比使用模拟退火方法(BA)获得较优解的方法，发现模拟退火这种启发式的方法在很低的计算开销下可以达到很不错的效果，可以有更好的效果和更快的运行时间

3.5.3 基于块集的行走更新策略

现有的随机游走系统将图分割成多个块。块的加载和行走管理是以块粒度进行的，导致行走更新被限制在单个块内，以块为核心的行走更新策略。一旦一个 walk 到达当前块的边界，其更新将会停止。然而，这样的策略会阻碍行走的更新，并潜在增加块 I/O 的数量。当内存中存在多个块时，基于块的行走更新策略将无法充分利用内存中其他块的节点，当 walk 在一个块的边缘停下后，可能临近的块实际上也在内存中，但是缺没能继续走下去。为了对此进行优化，提出了基于块集的行走更新策略，具体如图 18。倘若以块为核心，当此 walk 从$b1$到$b0$的那条边$(4,0)$时 walk 就会停止行走，但如果以块集为核心，那它将可以继续走到$b0$，在进一步回到$b1$再到$b2$，直到内存中不在存在下一个顶点才停止。很明显，基于块集的行走更新策略比起基于块的行走更新策略能够在同一个阶段内行走更多的步数。

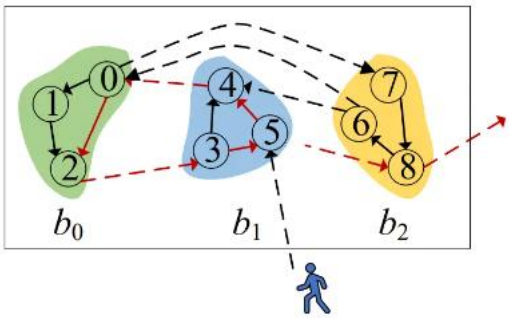


图 18 以块集为核心的行走更新策略

尽管一个图被分割成多个块，但行走可以通过这些块之间的切割边在块之间移动。因此，如果一个行走移动到内存中属于某个块的任何顶点，它可以进一步被更新，直到达到内存中块集的边界并移动到磁盘中的块。以块集为核心行走更新允许行走重复访问内存中的块，这提高了行走更新速率并加速了随机游走过程。最近，GraSorw 也允许行走在块之间进行更新。然而，它将内存中的块数量限制为 2，这对于不同规模的图和随机游走来说不够灵活。

3.5.4 主要贡献

SOWalker 是一种面向 I/O 优化的核外图处理系统，主要对二阶随机游走进行了优化。1.使用 Walk 矩阵避免加载不可更新的 walks。2.使用 Benefit-aware I/O 模型加载拥有最多累计可更新 walk 的块集。3.使用基于块集的行走调度策略使 walk 在已加载块集内，可以跨越块连续移动，可以尽可能多得移动。这些优化相比最先进的随机游走系统带来了显著的性能优势，并极大降低了 I/O 成本。

4 总结

随机游走技术是一种强大的信息提取工具，值得我们深入探究。在本文中，我介绍了五种随机游走引擎技术，每种技术都有其独特之处。GraphWalker 是一种高效处理 I/O 的图形系统，专为随机游走而设计。它采用了一种新型的状态感知 I/O 模型，可以有效处理包含数十亿条边的大型图形数据集，并能扩展到支持数百亿步长的随机游走。ThunderRW 是一个高效的内存随机游走引擎，用户可以轻松地实现定制的随机游走算法。其设计基于一个以步长为核心的模型，将计算从查询移动一步的局部视图

中抽象出来。此外，ThunderRW 还引入了步进交错技术，通过交替执行多个查询来隐藏内存访问延迟。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。NosWalker 提出了一种新颖的以 walker 为导向的调度方法，它主要提出了 walker 为核心的调度方法下，对边预采样的方法。并且为了缓解长尾问题，它使系统能够自适应地生成行走者，并灵活调整内存中保留的采样结果的分布，使用了自适应块粒度技术。SOWalker 是一种面向 I/O 优化的核外图处理系统，使用了 Walk 矩阵避免无效 I/O，Benefit-aware I/O 模型加载有效块集，并使用基于块集的行走调度策略进行了优化，主要对二阶随机游走进行了优化。

对比 ThunderRW 比 NosWalker，如果仅考虑计算时间，ThunderRW 比 NosWalker 快约 1.5 倍。另一方面，如果将数据加载时间也计入 ThunderRW 的执行时间，那么 ThunderRW 的总时间约比 NosWalker 慢约 32%。这是因为 ThunderRW 大约 75% 的时间用于图加载，而 NosWalker 可以通过流水线方式加载图并移动行走者。此外随机游走中的图分区问题和缓存停顿也是一个性能瓶颈，因此对随机游走内存优化的研究也是比较吸引人的。该如何在内存中处理图，该怎么协调远端内存和本地内存的，如果考虑分布式系统又该怎么优化，是否能将 NosWalker 中以 walker 为核心的思想在分布式系统上做出有趣的东西呢。

附录

问：二阶随机游走起源于哪?

答：二阶随机游走最早在Node2Vec中被提出。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。

问：walk矩阵中对角矩阵为何都是 0?

答：W(i,i)中的行走数量始终为 0，因为前一个和当前顶点在同一个块中的行走可以在不需要额外的块I/O的情况下被更新。

问：基于块集的行走更新策略有什么好处？

答：以块集为核心行走更新允许行走重复访问内存中的块，这提高了行走更新速率并加速了随机游走过程。倘若以块为核心，当此walk从b1到b0 的那条边(4,0)时walk就会停止行走，但如果以块集为核心，那它将可以继续走到b0，在进一步回到b1 再到b2，直到内存中不在存在下一个顶点才停止。很明显，基于块集的行走更新策略比起基于块的行走更新策略能够在同一个阶段内行走更多的步数。

参 考 文 献

[1] Yang K, Zhang M X, Chen K, et al. Knightking: a fast distributed graph random walk engine[C]//Proceedings of the 27th ACM symposium on operating systems principles. 2019: 524-537.

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

[3] Sun S, Chen Y, Lu S, et al. ThunderRW: an in-memory graph random walk engine[J]. 2021.

[4] Li H, Shao Y, Du J, et al. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs[J]. arXiv preprint arXiv:2203.16123, 2022.

[5] Wang S, Zhang M, Yang K, et al. NosWalker: A Decoupled Architecture for Out-of-Core Random Walk Processing[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. 2023: 466-482.

[6] Wu Y, Shi Z, Huang S, et al. SOWalker: An I/O-Optimized Out-of-Core Graph Processing System for Second-Order Random Walks[C]//2023 USENIX Annual Technical Conference (USENIX ATC 23). 2023: 87-100.