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：随机游走算法优化综述

学　　号　M202173758

姓　　名　刘帅

专　　业　电子信息

课程指导教师　施展 童薇

院（系、所）　武汉光电国家研究中心

2022 年 1 月 5 日

随机游走算法优化综述

刘帅[1)]

[1)](华中科技大学 计算机科学与技术学院, 武汉 430074)

摘 要 最近，图上的随机游走作为一种用于图数据分析和机器学习的工具而广受欢迎。随机游走是许多图形处理、挖掘和学习应用程序中的强大工具。目前，随机游走算法是作为单独的实现而开发的，并且存在严重的性能和可扩展性问题，尤其是复杂游走策略的动态特性。为解决这一问题，研究人员从不同角度提出了很多的解决方案，KnightKing，这是第一个通用的分布式图随机游走引擎。为了解决静态图和许多动态步行者之间的独特交互，它采用了以步行者为中心的直观计算模型。相应的编程模型允许用户轻松指定现有的或新的随机游走算法，通过适用于流行的已知算法的新的统一边缘转移概率定义来促进。借助 KnightKing，这些不同的算法受益于其通用的分布式随机游走执行引擎，以创新的基于拒绝的采样机制为中心，可显着降低高阶随机游走算法的成本。而从内存访问角度进行优化，研究人员提出了 ThunderRW 的高效内存随机游走引擎。与现有的并行系统相比，在提高单图操作的性能方面，ThunderRW 支持大规模并行随机游走。针对大型图上的二阶随机游走，研究人员提出了一个遵循接受-拒绝范式以在内存和时间成本之间实现更好的平衡的采样方法，另一个优化用于快速采样自然图中存在的偏斜概率分布。其次，在成本模型的基础上提出了一种新颖的内存感知框架，在任意内存预算内实现二阶随机游走的高效率。该框架应用基于成本的优化器在内存预算内为图中的每个节点或边分配理想的节点采样方法，同时最小化随机游走的时间成本。最后，该框架为用户提供通用编程接口，以轻松定义新的二阶随机游走模型。

关键词 数据中心；随机游走；图数据分析；

中图法分类号

Review of random walk algorithm optimization

Shuai Liu[1)]

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

Abstract More recently, random walks on graphs have gained popularity as a tool for graph data analysis and machine learning. Random walk is a powerful tool in many graphics processing, mining, and learning applications. At present, random walk algorithm is developed as a separate implementation, and has serious performance and scalability problems, especially the dynamic characteristics of complex walk strategy. To solve this problem, researchers have proposed many solutions from different angles, KnightKing, which is the first general-purpose distributed graph random walk engine. To address the unique interaction between static graphs and many dynamic walkers, it employs a walkers centric intuitive computing model. The corresponding programming model allows users to easily specify existing or new random walk algorithms, facilitated by a new uniform definition of edge shift probability applicable to popular known algorithms. With KnightKing, these different algorithms benefit from its general-purpose distributed random walk execution engine, centered on an innovative repudiation-based sampling mechanism that significantly reduces the cost of higher-order random walk algorithms. To optimize the memory access, researchers propose ThunderRW's efficient memory random walk engine. ThunderRW supports massively parallel random walks in terms of improving the performance of single graph operations compared to existing parallel systems. For second-order random walks on large graphs, we propose a sampling method that follows the

accept-rejection paradigm to achieve a better balance between memory and time costs, and another optimization to rapidly sample skew probability distributions that exist in natural graphs. Secondly, based on the cost model, a novel memory awareness framework is proposed to achieve high efficiency of second-order random walk within any memory budget. The framework uses a cost-based optimizer to assign each node or edge in the graph an ideal node sampling method within the memory budget while minimizing the time cost of random walk. Finally, the framework provides users with a common programming interface to easily define new second-order random walk models.
Key words data center; random walk; graph data analysis;

1 引言

随机游走是一项基本且广泛使用的图处理任务。作为从图实体之间的集成路径中提取信息的强大数学工具，它构成了许多重要的图测量、排名和嵌入算法的基础，例如个性化 PageRank、SimRank 、DeepWalk 、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。它们服务于多种应用，例如节点/边缘分类、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。

基于随机行走的算法以图 G 为输入，同时有 w 个步行者。它启动步行者，每个人都从一个特定的顶点开始，然后他们独立地在图中漫游。在每一步中，步行者从其当前驻留的顶点的出边取样，并跟随它到下一站。当达到预设的路径长度或满足预设的例外条件时，每个步行者都有预设的终止概率退出。输出可以通过在随机行走过程中嵌入的计算产生，也可以通过转储产生的随机行走路径产生。这个过程可以重复多次。

直观地说，随机行走的计算量主要集中在边缘采样过程中，这也体现了个体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。随着随机漫步变得越来越流行，采样逻辑也变得越来越复杂，最近的算法执行动态采样，其中边缘转移概率取决于漫步者的当前状态、它访问过的前一个(或多个)顶点，以及当前驻留的顶点的边的属性。

因此，复杂的随机漫步算法以采样复杂度为代价，实现了更灵活的、特定于应用程序的漫步。例如，在流行的网络特征学习技术 node2vec 包含了动态图随机遍历和后续的 skip-gram 语言模型构建，据报道，一个 Spark 的 node2vec 实现的执行时间是前者的 98.8%。我们的实验表明，即使在最先进的图引擎 Gemini 之上实现，node2vec 也会因边缘采样而陷入困境，产生的顶点导航速率(每秒访问的顶点数)比 Twitter 图上的 BFS 慢 1434 倍。Node2Vec 的一个简单实现在 twitter 图上花费了 8 个多小时。此外，增加 RW 查询次数可以提高 RW 算法的有效性。因此，加速 RW 查询是一个重要的问题。

采样代价如此之高主要是由于其动态特性:在每一步，选择出边时需要重新计算所有出边的转移概率，其开销随着步行者当前居住顶点的度而增加。这样的开销在顶点之间是远远不平衡的，因为现实世界的图往往具有幂律度分布。更糟糕的是，具有更多关联边的顶点更有可能被访问，这进一步加剧了这些热顶点的采样开销。

大多数现有模型使用一阶随机游走，它假设下一个要访问的节点仅取决于当前节点。然而，一阶随机游走抽象了现实世界系统的过度简化。一个例子是 Web 上的用户轨迹，其中节点是网页，交互是用户从一个网页导航到另一个网页。用户的下一页访问不仅依赖于上一页，还受之前点击的顺序影响，称为高阶依赖。一阶随机游走无法捕捉到这种高阶依赖。二阶随机游走旨在对高阶依赖项进行建模，从而提高应用程序的准确性。二阶随机游走的一项流行和最新应用是帮助学习图分析任务的高质量嵌入，例如节点分类和链接预测。Node2vec 是最成功的图嵌入模型之一。它使用二阶随机游走将来自同一社区的节点紧密地编码在一起，并优于深度游走。除了图嵌入，二阶随机游走也成功地应用于寻找有意义的社区和模型序列数据 。

本文主要介绍了针对随机漫步的三份优化工作，一个是分布式的通用框架 KnightKing，一个是基于内存优化的 ThunderRW 框架，还有针对二阶随机漫步的内存感知框架及优化。

KnightKing，图随机游动的第一个通用框架。它可以被看作是一个“分布式随机游动引擎”，是传统图形引擎的随机游动对等体。KnightKing 采用了以步行者为中心的视图，使用 API 来定义自定义的边缘转移概率，同时处理常见的随机步行者基础设施。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等系统细节。因

此，它促进了一个直观的“像步行者一样思考”的视图，尽管用户可以灵活地添加可选的优化。

ThunderRW，一个通用的、高效的内存 RW 框架。采用了一个以步长为中心的规划模型，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。框架将 udf 应用到每个查询，并将查询的一个步骤视为一个任务单元，从而并行执行查询。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，提出了步进交错技术来解决软件预取时由于内存访问不规律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，step 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 RW 查询之间的内存级别并行性。

最后是一个可以作为各种二阶随机游走模型的中间件的内存感知框架。在这项工作中，实现了两个流行的二阶随机游走模型——node2vec 和自回归模型用于实验。此外，为了轻松受益于其设计的基于成本的优化器，该框架提供了灵活的编程接口，供用户定义自己的样本方法和二阶随机游走模型。

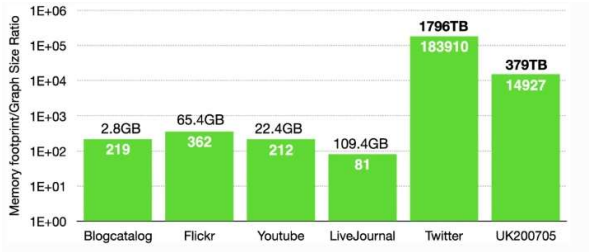

图 1 内存成本图

2 原理和优势

2.1 随机漫步

首先介绍了基于随机游动的算法的一般分类。它们遵循一个共同的框架:给定一个图，一定数量的步行者从一个给定的起点出发，然后根据给定的概率分布重复选择当前驻留顶点的一个邻居，并移动到这个邻居。直观地说，随机漫步算法的关键变化在于邻居选择步骤。

从边被选择的相对概率来看，随机游走算法可以分为无偏算法和有偏算法。无偏算法是指出边的转移概率不依赖于它们的权值或其他性质，而有偏算法是指在漫步中对边的邻居选择进行加权。

如果边缘转移的概率在整个过程中保持不变，我们就有一个静态随机游走。否则，我们有一个动态行走，其中过渡概率的确定涉及到行走状态，它在行走过程中不断演化。因此，在动态行走过程中，需要在每一步重新计算这些概率，而不是预先计算所有沿边转移的概率。我们进一步将算法按顺序分类，根据步行者最近的轨迹在更新其当前驻留顶点的转移概率时所考虑的距离。在一阶行走算法中，步行者对当前点之前访问过的顶点是不知道的，而在二阶算法中，在选择下一站时，步行者会考虑之前访问过的顶点，并从那里过渡到当前的顶点。请注意，高阶算法，包括二阶算法，根据定义是动态的。

除了转移概率定义外，不同的随机游走算法可能会采用不同的终止策略。常见的策略包括在给定的步数上截断行走(导致行走序列的长度一致)，或者让每个步数在每一步上以给定的概率结束行走。

2.2 转移概率

我们将介绍五种具有代表性和流行的随机漫步算法，并给出它们的边缘转移概率。

2.2.1 PPR

众所周知的 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，特别是完全 PPR(对所有顶点进行个性化)，需要耗费大量的时间或空间才能有效地计算，特别是对于大的图。因此，基于随机行走的解决方案成为一个常见的近似，行走序列生成并保存为未来 PPR 查询。我们在本文中的讨论是基于一个使用全个性化模型的随机漫步实现，这是一个有偏见的静态算法，模拟 web 浏览中的个人偏好。

2.2.2 DeepWalk

另一个有偏静态随机漫步算法的例子是 DeepWalk，它是一种广泛应用于机器学习的重要的图嵌入技术。它利用了图形分析的语言建模技术，使用截断的随机遍历生成许多遍历序列。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。这种表示在许多应用中都很有用，如多标签分类、链接预测和异常检测。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行走。

与 PPR 一样，当 DeepWalk 在加权图上运行时，

一条边的转移概率也与其权重成正比。主要的区别在于终止组件:与 PPR 不同，PPR 可以有随机的“早期终止”，DeepWalk 会一直持续到给定的路径长度。

2.2.3 Meta-path

接下来，我们引入了基于元路径的算法，用来捕捉顶点和边的异质性背后的语义。在这些算法中，每个步行者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，在出版物图中，为了探究引文关系，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 → authoredBy^{-1}”。通过重复这样的模板可以创建更长的行走，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)关系来生成长引文链。

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 N 个用户提供的元路径方案中随机为每个步行者分配一个元路径。对于指定元路径方案 S 的步行者，在 k^{th} 步:

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

这给出了一个动态一阶随机漫步算法的例子。在同一顶点，对于不同方案或不同步数的步行者，其边缘转移概率分布是不同的，不能在执行之初就预先计算。同时，下一条边的选择仍然是一阶的，只涉及到指定方案的当前位置，不考虑之前访问过哪些顶点。

2.2.4 node2vec

最后，介绍一个高阶算法。这样的算法是强大的，因为步行者在选择下一站时记录了他们最近的行走历史，这在许多使用场景中反映了现实。到目前为止，在实际应用中看到的绝大多数高阶算法都是二阶的。其中，我们引入了非常流行的 node2vec，它具有与 DeepWalk 类似的应用，但更灵活，表达能力更强。

在一个给定的无向图上，一个步数 w(记得它的最后一站是 last(w) = t)在它当前驻留的顶点 v 处有如下的动态边转移概率，对于连接 v 和顶点 x 的边 e 来说:

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

2.2.5 自回归模型

分布如下

$$p_{uvz} = \frac{p'_{uvz}}{\sum_{t \in N(v)} p'_{uvt}},$$

其中$p'_{uvz} = (1-\alpha)p_{vz} + \alpha p_{uz}, p_{vz} = \frac{w_{vz}}{W_v}$，$0 \le \alpha < 1$

3 研究进展

3.1 KnightKing

它的统一边缘采样机制可以轻松地处理昂贵的动态/高阶行走，同时在静态行走中自动变形为别名解决方案。以拒绝采样为中心，只需要计算多条边的未归一化转移概率，即使是在有上百万条边的顶点。同时，它仍然是一个精确的解决方案，提供快速采样而不失去正确性。

3.1.1 工作流和编程模型

与传统图形引擎在迭代中协调多个顶点(沿着多个边)的更新类似，KnightKing 拥有一个迭代计算模型，可以同时协调多个步行者的行动。与顶点和边一样，步行者也会根据当前驻留顶点的分配分配给每个节点/线程。当然，它与传统图引擎的明显区别在于，它是沿采样边的概率游走，而不是沿所有/活动边的确定性更新传播。一个更微妙的不同之处在于每个迭代的执行都使用了分布式随机行走引擎。

虽然图形引擎可以通过一轮的顶点到顶点消息来推送/拉取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。因为每个步行者可能需要执行边缘选择基于最近访问的顶点,可能是“步行者-顶点”查询的原因(例如 node2vec 的情况下,检查是否前一点 t 匹配后一点 x)。在分布式环境下,顶点和边跨节点分区,需要传递消息来发送这样的查询并收集它们的结果。

为了促进这种分布式查询操作的高效批处理和协调，KnightKing 扮演了一个类似于邮局的角色，其中步行者根据他们的本地抽样候选者提交查询消息，并将其发送到涉及动态检查的顶点。所有这些查询都是根据顶点到节点的分配来交付的，所有节点都并行处理从所有步行者接收到的查询。然后，另一轮消息传递将共同返回结果，查询的步行者将一起检索结果。以下是 KnightKing 迭代的步骤:

1. 步行者生成拒绝抽样的候选边并进行初步筛选。

2. 在必要时，步行者根据抽样结果向顶点发出步行者状态查询。

3. 所有节点处理状态查询并返回结果。

4. 步行者检索状态查询结果。

5. 步行者决定取样结果，如果成功就移动。

注意，上面描述的是 KnightKing 支持的最一般的随机游走情况。用更少的复杂算法，通常步骤可以跳过。例如，在静态或一阶随机游走的情况下，由于在局部采样时不需要涉及其他顶点，因此省略了步骤 2-4。对于这样的算法，所有的步行者都可以锁步移动:在每次迭代中，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(或行走终止)。在这些情况下，所有活跃的步行者都沿着他们的行走顺序走相同的步骤。但是，对于像 node2vec 这样的高阶算法，步行者之间的迭代是通过两轮从步行者到顶点的查询消息传递实现同步的。成功采样的幸运者将继续向前走一步，而不那么幸运的人则停留在当前的顶点等待下一次迭代。这样，步行者可以以不同的速度前进，潜在地产生掉队者。

3.2 THUNDERRW

3.2.1 Step-centric 模型

为了抽象 RW 算法的计算，这篇文章提出了步进中心模型。其观察到 RW 算法是建立在多个 RW 查询而不是单个查询的基础上的。尽管 RW 算法的查询内部并行性有限，但由于每个 RW 查询都可以独立执行，因此在 RW 算法中存在大量的查询间并行性。因此，他们用步进中心模型从查询的角度抽象了 RW 算法的计算，以利用查询间的并行性。

Algorithm 1 ThunderRW Framework

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 $C \leftarrow$ Gather$(G, Q, \textit{Weight})$;
4 $e \leftarrow$ Move(G, Q, C);
5 $stop \leftarrow \textit{Update}(Q, e)$;
6 **while** $stop$ *is false*;
7 **return** $\mathbb{Q}$;
8 **Function** Gather$(G, Q, \textit{Weight})$
9 $C \leftarrow \{\}$;
10 **foreach** $e \in E_{Q.cur}$ **do**
11 Add $\textit{Weight}(Q, e)$ to C;
12 $C \leftarrow$ execute initialization phase of a given sampling method on C;
13 **return** C;
14 **Function** Move(G, Q, C)
15 Select an edge $e(Q.cur, v) \in E_{Q.cur}$ based on C and add v to Q;
16 **return** $e(Q.cur, v)$;

Algorithm 2 Preprocessing for Static Random Walk

Input: a graph G;
Output: the transition probabilities C_v on E_v for each vertex v;
1 **foreach** $v \in V(G)$ **do**
2 $C_v \leftarrow \{\}$;
3 **foreach** $e \in E_v$ **do**
4 Add $\textit{Weight}(null, e)$ to C_v;
5 $C_v \leftarrow$ execute initialization phase of a given sampling method on C_v;
6 Store C_v for the usage in query execution.

3.2.2 步进交错技术

基于以步骤为中心的模型，ThunderRW 使用 GMU 操作处理查询Q的步骤。该模型下的随机内存访问可能有两个主要来源。首先，移动操作随机选择一条边并沿着选定的边移动。其次，自定义函数中的操作会引入缓存未命中，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）由 RW 算法决定，并且非常灵活，因此我们针对系统引起的内存问题（即 Move 手术）。受分析结果的启发，我们建议使用软件优化技术来加速 ThunderRW 的内存计算。然而，查询Q 的步骤没有足够的计算工作量来隐藏内存访问延迟，因为Q的步骤具有依赖关系。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。一旦查询Q 的 一个阶段完成，我们就会切换到组中其他查询的阶段。当其他查询的阶段完成时，我们恢复Q的执行。通过这种方式，我们在单个查询中隐藏了内存访问延迟并保持 CPU 忙碌。我们称这种方法为步骤交错。

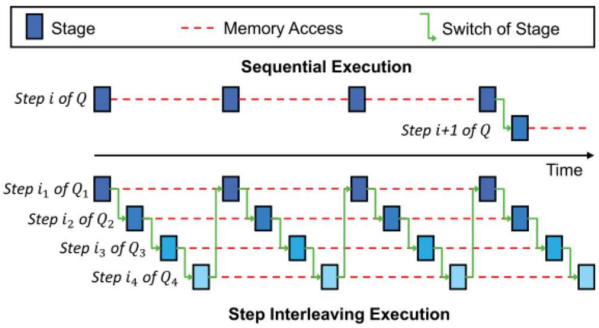

图 2 顺序交错与步骤交错

3.2.3 阶段依赖图

我们设计阶段依赖图(SDG) 来对步骤中一系列操作的阶段进行建模。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。给定操作顺序，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：由于我们通过切换查询的执行来隐藏内存访问延迟，阶段的约束是每个阶段最多包含一个内存访问操作，而消耗数据的操作在后续阶段。请注意，为了便于切换的实现，我们将包含跳转操作的操作视为单个阶段。

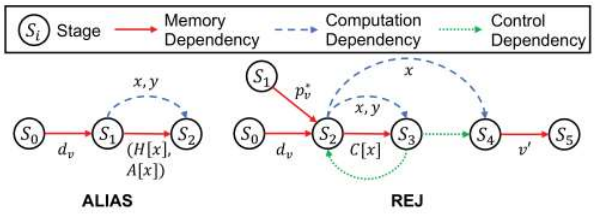

图 3 阶段依赖图

3.2.4 状态切换机制

下面介绍 SDG 下步进交织的实现。此时需要一个高效的开关机制。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，而主存延迟以纳秒为单位。由于每个线程往往会进行许多 RW 查询，因此我们在单个线程中的各个阶段之间切换执行。

我们根据它们是否属于 SDG 中的周期将 SDG 的阶段分为两类。对于不在循环中的阶段（称为非循环阶段），查询只访问它们一次以完成 Move 。给定一组查询 Q，我们以耦合方式执行它们。特别地，一旦一个查询$Q_i \in$ Q 完成一个阶段S，我们就会切换到下一个查询$Q_{i+1} \in$ Q 来处理S。在所有查询完成后S，我们进入下一阶段。相反，周期中的阶段（称为周期阶段 ）可以访问不同查询的不同时间。为了处理不规则性，我们以解耦的方式处理它们。具体来说，每个查询Q 都 记录了要执行的阶段S。当切换到Q，我们执行S，设定下一阶段的Q完成后，基于 SDG，并切换到下一个查询S。因此，每个查询都是异步执行的。

针对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，其中计算依赖边指示需要存储的信息。特别地，任务环用于跨查询所有阶段的数据通信，而搜索环用于处理循环阶段。由于我们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。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 的 SDG 有。

3.3 二阶随机游走的记忆感知框架

3.3.1 具有拒绝的二阶随机游动

拒绝方法由目标分布 P、建议分布 Q 和包围常数 c 组成。在二阶随机游走的情况下，假设前一节点和当前节点分别为 u 和 v, P 为 e2e 转移概率分布，Q 为 n2e 转移概率分布。

形式上：

$$P = \{\frac{w_{vz}^{`}}{W_v'}\},\quad Q = \{\frac{w_{vz}}{W_v}\}$$

其中$p_i \le Cq_i$

则边界常数：

$$C_{uv} = max\left\{\frac{P}{Q}\right\} = max_{z\in N(v)}\left\{\frac{w_{vz}^{`}}{w_{vz}}\frac{W_v}{W_v'}\right\}$$

在采样过程中，接受比为

$$\beta_{uvz} = \frac{1}{C_{uv}}\frac{w_{vz}^{`}}{w_{vz}}\frac{W_v}{W_v'} = \frac{w_{vz}^{`}}{w_{vz}}min_{t\in N(v)}\left\{\frac{w_{vt}}{w_{vt}'}\right\}$$

3.3.1.1 node2vec 的拒绝法

假设前一个节点 u 和当前节点 v 有共同邻居，那么，边界常数为

$$C_{uv} = max_{z\in N(v)}\left\{\frac{W}{W'}\frac{w_{vz}'}{w_{vz}}\right\} = \frac{W}{W'}max\left\{\frac{1}{a},\frac{1}{b},1\right\}$$

顶点 z 的接受率为:

$$\beta_{uvz} = \frac{w_{vz}'}{W'}\frac{W}{C_{uv}*w_{vz}} = \frac{w_{vz}'}{w_{vz}}min\{1,a,b\}$$

3.3.1.2 自回归模型的拒绝法

同样，边界常数为

$$C_{uv} = max_{z\in N(v)}\{\frac{p_{uvz}}{p_{vz}}\} = max_{z\in N(v)}\{\frac{p_{uvz}'}{p_{vz}W'}\}$$
$$= max_{z\in N(v)}\{\frac{p_{uvz}'}{p_{vz}\sum_{l\in N(v)}p_{uvl}'}\}$$
$$= max_{z\in N(v)}\{\frac{(1-\alpha)p_{vz}+\alpha p_{uz}}{p_{vz}\sum_{l\in N(v)}((1-\alpha)p_{vl}+\alpha p_{ul})}\}$$
$$= \frac{max_{z\in N(v)}\{(1-\alpha)+\alpha\frac{p_{uz}}{p_{vz}}\}}{(1-\alpha)+\alpha\sum_{l\in N(v)}p_{ul}}$$

顶点 z 的接受率为:

$$\beta_{uvz} = \frac{p'_{uvz}}{\sum_{l\in N(v)} p'_{uvl}} \frac{1}{C_{uv} * p_{vz}}$$
$$= \frac{(1-\alpha)+\alpha\frac{p_{uz}}{p_{vz}}}{\sum_{l\in N(v)} p'_{uvl} * max_{t\in N(v)}\{\frac{p'_{uvt}}{p_{vt}\sum_{l\in N(v)} p'_{uvl}}\}}$$
$$= \frac{(1-\alpha)+\alpha\frac{p_{uz}}{p_{vz}}}{max_{t\in N(v)}\{(1-\alpha)+\alpha\frac{p_{ut}}{p_{vt}}\}}$$

3.3.2 倾斜概率分布的基于组的节点采样器

拒绝法的效率由边界常数决定。虽然边界常数通常是小于相应的节点度，在最坏的情况下，它仍然可以任意接近节点度。特别是，在大型图上，如 Twitter, UK200705，很少有超级节点具有非常大的节点度，并带来高度倾斜的端到端概率分布。这些偏态分布进一步导致了大的边界常数(几乎等于超节点的程度)，从而破坏了排斥法的效率。在本节中，通过首先分析大型真实世界图中的倾斜端到端概率分布，提出一个基于组的节点采样器，用于对倾斜概率分布进行有效采样。

文章中的偏态概率分布是指少数元素的概率远高于其他元素的概率分布。基于二阶随机漫步模型中 e2e 概率分布的定义，我们发现节点的三角形数量对 e2e 概率分布的形状影响很大。更具体地说，假设当前节点为度为 dv 的 v，前一个节点为 u，节点 u 与 v 之间的共同近邻个数为 θ_{uv}，我们可以将节点 v 的近邻分为三组。

组 I:组中仅包含前一个节点 u，称为第一组;

组 II:组中包含节点 u 和 v 的共同邻居，其大小为 θ_{uv}。这叫做第二组;

组 III:组包含其余 $d_v-1-\theta_{uv}$ 节点。这叫做第三组。

由于同一组节点受模型超参数的影响相同，因此对于特定的超参数构型，边(u, v)的 e2e 概率分布的形状仅受 θ_{uv} 的影响。

Algorithm 3 Group-Based Node Sampler

Input: previous node: u, current node: v
Output: next node: z
$G_I = \{u\}$
$G_{II} = N(v) \cap N(u)$
$G_{III} = N(v) - G_I - G_{II}$
Naive Sampler: draw group G_i from $\{G_I, G_{II}, G_{III}\}$, where the probability of a group is the sum of probabilities of nodes belonging to the group.
if $G_i == G_I$ **then**
  $z = u$
else if $G_i == G_{II}$ **then**
  Rejection Sampler: draw node z from $p(z|u,v,G_{II}) = \frac{p(z|u,v)}{\sum_{y\in G_{II}} p(y|u,v)}$.
else if $G_i == G_{III}$ **then**
  repeat
    draw node z' from $N(v)$ uniformly.
    if $z' \in G_{III}$ **then**
      $r \leftarrow rand(0,1)$
      if $r \le \frac{p(z'|u,v)}{\max_{y\in G_{III}} p(y|u,v)}$ **then**
        $z = z'$
        break
      end if
    end if
  until FALSE
end if
return z

假设采用 node2vec 模型，I 组节点的概率由超参数 a 控制，III 组节点的概率由超参数 b 控制，超参数越大，概率越小。

为了有效地从倾斜概率分布中抽取节点，我们提出了一种基于分组的节点抽样器。其基本思想是，我们首先对组进行抽样，然后从选定的组中对节点进行抽样。因为同一组概率是不偏的，这种方法可以避免较大的边界常数。在基于组的节点采样器中，我们需要识别节点所属的组。然而，如果我们记录这三个组的所有成员关系，端到端分布的额外内存成本将与当前节点的度成比例，这对于二阶随机漫步模型来说是昂贵的。为了避免这种情况，对于边(u, v)的 e2e 分布，我们在内存中只保留第 II 族的成员，其大小为三角形的个数 θ_{uv};另外两组可以从第二组的信息中自动推断出来。

3.3.3 记忆感知的二阶随机漫步框架

图 4 展示了感知内存的二阶随机遍历框架的概述。该框架的核心是一个基于成本的优化器。该优化器保证了在各种内存预算下的二阶随机遍历的高效率。该算法不仅从零开始贪婪地寻找高效的节点采样分配，而且能够在内存预算在线变化时快速更新分配。在执行过程中，框架(1)首先初始化代价模型，并计算拒绝节点采样器的边界常数。(2)然后，在不违反内存约束的情况下，执行基于代价的优化器生成高效的节点采样器分配。(3)在分配的基础上，对整个图中的每个节点采样器进行初始化。(4)

最后，框架准备进行二阶随机遍历。此外，该框架为用户提供了灵活的编程接口，以定义新的采样器和面向应用程序的随机漫步模型。因此，该框架可以作为现有的基于二阶随机步的应用程序的中间件，以提高采样效率。下面将介绍基于成本的优化器中定义以节点为中心的节点采样器分配问题，然后引入两类贪婪算法来解决分配问题。同时还引入了以边缘为中心的分配策略，从而使优化器对小内存预算更友好。在本节的最后，介绍了针对动态内存预算的节点采样器分配问题的自适应解决方案。

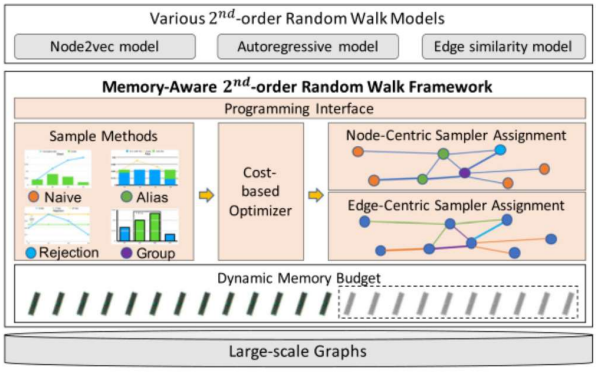

图 4 内存感知框架

3.3.3.1 基于成本的优化器

基于成本的优化器的目标是找到一个采样效率最大化的节点采样器分配，而内存占用受到给定预算的限制。为了形式化地分析节点采样器分配问题，我们将其定义为:

(以节点为中心的节点采样器分配问题)。给定一个图 G = (V, E) 和一组节点采样 $NS = \{ns_j\}$, $1 \le j \le S$,成本模型 C M = {(T_{ij}, M_{ij})},其中 T_{ij} 是节点使用采样器 ns_j 的时间成本，M_{ij} 是相应的内存成本,该问题的目标是为图 G 中的每个节点分配一个采样器，使总时间开销最小化，且内存开销不超过预定义的内存预算 M

目标是：

$$\text{minimize} \sum_{i=1}^{|V|} \sum_{j=1}^{j=S} T_{ij} x_{ij},$$

$$\text{subject to} \sum_{i=1}^{|V|} \sum_{j=1}^{j=S} M_{ij} x_{ij} \le M,$$

$$\sum_{j=1}^{j=S} x_{ij} = 1, i = 1, 2, ..., |V|,$$

$$x_{ij} = \{0, 1\}, i = 1, 2, ..., |V|; j = 1, 2, ..., S.$$

可以证明具有新内存约束的分配问题是一个精确的 0-1 MCKP 问题。

3.3.3.2 节点采样器分配的贪婪算法

原来的 0-1 MCKP 是 NP-hard 问题。它有伪多项式算法，伪多项式算法是一个动态规划解;动态规划方法的时间复杂度较高，不能处理大型图。在这里，我们关注贪婪算法，这通常是有效的大数据场景。首先介绍了一种近似有界的算法——LP 贪婪算法，然后介绍了一种启发式算法——基于度的贪婪算法作为基线。

LP 贪婪算法：

为了求解 0-1 MCKP，经典的解决方法是通过放宽约束并允许 x_{ij} 为实值，将原始问题转化为线性 MCKP (LMCKP)。LMCKP 的最优解是 0-1 MCKP 的下界。原问题通过枚举，或通过四舍五入找到近似值求解得到最优解。在这里，我们使用舍入技术来实现一个高效的节点采样器分配。

Algorithm 4 LP Greedy Algorithm

Input: Graph: $G(V, E)$, Cost Model: $CM = \{(T_{ij}, M_{ij})\}$, Memory Budget: M
Output: Assignment: $assignDict$, Trace: path
1: $usedMem = 0$
2: $path = []$
3: **for** each v_i in V **do**
4: eliminate P-Domination and LP-Domination of node samplers of v_i according to Property 1 and 2.
5: assign node sampler with smallest memory cost (NS_{i1}) to node v_i, i.e., $assignDict[v_i] = NS_{i1}$
6: $usedMem = usedMem + M_{i1}$
7: **end for**
8: compute the gradients $\frac{T_{ij+1}-T_{ij}}{M_{ij+1}-M_{ij}}$ for each node.
9: sort the gradient array q in ascending order.
10: **while** q.notEmpty() **do**
11: $NS_{ik} = (T_{ik}, M_{ik})$, i.e., select the minimal $\frac{T_{ij+1}-T_{ij}}{M_{ij+1}-M_{ij}}$ from q.
12: $(T_i, M_i) \leftarrow assignDict[v_i]$
13: **if** $usedMem - M_i + M_{ik} > M$ **then**
14: break;
15: **end if**
16: $assignDict[v_i] = NS_{ik}$
17: $usedMem = usedMem - M_i + M_{ik}$
18: $path.append(NS_{ik})$ //maintain the assignment trace for update.
19: **end while**

基于度的贪婪算法：

解决节点抽样器分配问题的一种直觉是尽可能多地使用别名节点抽样器。从代价模型可以看出:(1)将节点采样器由拒绝/天真改为别名时，节点度越大的节点通常效率提高越大。这促使我们首先分配别名节点采样器的大程度节点。(2)度越小的节点占用的内存越少，在一定的内存预算内，更多的节点可以使用别名节点采样器。这促使我们首先分配程度小的节点作为别名节点采样器。

在上述观察的基础上，我们引入了一种基于基本度的贪心算法，其工作原理如下:首先，图中节点

按度递增(递减)排序;然后，对于每个节点，在内存预算内，它试图以别名、拒绝和朴素顺序分配节点采样器。

与 LP 贪心算法相比，基于度的求解要简单得多。但是，它没有明确地考虑时间代价，并且很难得到近似界。实验结果表明，基于度的算法只有在内存预算较大的情况下才能很好地工作。

3.3.3.3 以边为中心的节点采样器分配

之前引入的以节点为中心的节点采样器分配策略允许使用同一类型的节点采样器获得不同的端到端概率分布。当内存预算较小时，采样效率较低。具体来说，在二阶随机游走模型中，尽管两个 e2e 分布拥有相同的当前节点，但由于先前节点和当前节点之间的共同邻居数量不同，它们之间可能存在很大的差异。由于以节点为中心的分配策略不能以细粒度的方式使用内存预算，所以当内存预算很小时，它会分配相同的(低效的)节点

采样器对于两个不同分布的节点，从而达到了较差的效率。

为了避免上述问题，提高内存感知框架在内存预算较小的情况下的效率，我们将节点为中心的策略扩展为边缘为中心的策略，为每条边分配一个节点采样器。边缘中心策略可以细粒度地利用内存预算。当内存预算较小时，可以优先为倾斜概率分布分配有效的节点采样器。这种策略尤其适用于十亿边图。

3.3.3.4 动态内存预算的自适应解决方案

在实际环境中，例如云服务、公共集群，可用内存是动态的。这要求节点采样器的分配能够在内存预算变化时自适应调整。由于贪婪算法的线性特性，我们可以很容易地扩展所提出的解决方案是自适应的。

在原始的贪心过程中，我们保持贪心选项的跟踪。当内存预算发生变化时，我们执行以下两种策略来更新节点采样器分配。

内存预算增加：当可用内存增加时，之前的分配不会受到新的总预算的影响，所以我们只是从跟踪中的最后一个状态开始重新启动贪婪算法。

内存预算减少：在这种情况下，我们需要撤销一些节点分配以减少总内存。由于原始的贪心算法按照内存大小递增的顺序进行贪心选择，所以我们执行相反的顺序以减少使用的内存。特别地，我们从跟踪中取出贪婪的选择，直到总内存满足新的预算。

4 总结与展望

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步行者为中心的计算模型，以支持随机步行者算法的简单规范。其中提出的统一的边缘转移概率定义，适用于流行的已知算法，以及新的基于拒绝的采样方案，极大地降低了昂贵的高阶随机漫步算法的代价。在对 KnightKing 的设计和评估表明,在精确的边采样中，无论当前顶点出边的数量如何，都可以实现接近 O(1)的复杂度，而不损失精度。

ThunderRW 是一个高效的内存 RW 引擎，用户可以轻松地实现定制的 RW 算法。其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。在此框架中实现了 PPR、DeepWalk、Node2Vec 和 MetaPath 等四种典型的 RW 算法。实验结果表明，ThunderRW 的性能比目前最先进的 RW 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，在 ThunderRW 中通过显式和手动存储和恢复每个查询的状态来实现步进交错技术。未来一个有趣的工作是用协程来实现该方法，协程是一种支持交叉执行的有效技术。

二阶随机漫步是建模数据中高阶依赖关系的重要工具。第三篇文章研究了在不同内存预算下如何有效地支持二阶随机游动的问题。通过开发一个基于成本的优化器，提出了一个内存感知框架。优化器为图中的每个节点分配不同的节点采样器，并确保在内存预算下效率最大化。其设计了一种有效的贪婪算法来生成这种高效的分配。其实验结果证明了内存感知框架的优势。此外，在二阶随机游动的情况下，图的效率和结构性质之间的复杂关系仍然是一个有待解决的问题，还需要在未来使用所提出的技术来优化深度学习计算框架中的图形采样功能。

这三篇文章分别从不同的角度对随机游走进行了优化与改进，大大提高了其可用性和性能。未来，随着数据量的不断增大，图计算将越来越重要，更高效可靠的算法模型将会不断涌现。

5 参考文献

[1]. Yang K , Zhang M X , Chen K , et al.

KnightKing: a fast distributed graph random walk engine[C]// the 27th ACM Symposium. ACM, 2019.

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

[3]. Shao Y, Huang S, Li Y, et al. Memory-aware framework for fast and scalable second-order random walk over billion-edge natural graphs[J]. The VLDB Journal, 2021: 1-29.