论文编码：RUC-BK-080901-2015201921

中国人民大学本科学生毕业论文
（毕业设计）

图数据库上的图节点相似度研究

作　　者：梁晓周

学　　院：信息学院

专　　业：计算机科学与技术

年　　级：2015 级

学　　号：2015201921

指导教师：魏哲巍

论文成绩：88

日　　期：2019 年 5 月 8 日

中国人民大学
RENMIN UNIVERSITY OF CHINA

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名： 

2019 年 5 月 6 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密□，在 10 年解密后适用本授权书

2、不保密☑。

（请在以上相应方框内打“√”）

作者签名： 

2019 年 5 月 6 日

导师签名：

年　　月　　日

中国人民大学本科生毕业论文简表

项目	内容
选题背景和意义	Personalized PageRank 作为衡量图节点间相似度的指标，可以被应用到用户或物品推荐的应用上，因此为图数据库平台提供一款快速计算 Personalized PageRank 的算法将会使众多应用受益。
问题提炼，主要想解决什么问题，或满足什么实际需求	作为计算 Personalized PageRank 的高效算法，FORA 和 All-Pair-Backward-Search 算法凭借其当前的实现还难以在更大范围的应用中做出贡献。同时，图数据库平台上 Personalized PageRank 的计算也迫切需要一个高效且有效限制误差范围的算法。
难点：从实际挑战中抽象出来的研究难题/实现新需求的关键之处	为了使得在图数据库上 FORA 及 All-Pair-Backward-Search 算法的实现能够与算法理论上的时间复杂度保持一致，我们需要深入探究图数据库底层如何支持相应功能并对其加以利用，从而解决算法实现在时间复杂度上的瓶颈问题。
解决方案主要思路或特点	在深入理解图数据库的底层机制的基础上，我将使用 Java 语言实现 Neo4j 图数据库上的 FORA 算法及 All-Pair-Backward-Search 算法，并保证算法的实现与其理论上的时间复杂度保持一致。
实验所基于的环境、平台、系统，比较的对象，自己在实验中所做的工作，以及实验结果	本毕业设计实验是在 2094.865MHz 8 核 CPU、500GB 内存的 Ubuntu 系统机器上进行的，我对 Monte-Carlo 算法、Forward Push 算法、Neo4j 的 PPR 算法、FORA 算法以及 All-Pair-Backward-Search 算法依次进行了预处理、Single-Source PPR 以及 Top-k PPR 实验。结果显示 All-Pair-Backward-Search 算法只需要以较低的时空开销对 PPR 结果进行预处理，此外在图较小或不允许预处理的情形下，FORA 算法最适合作为

	PPR 估计值计算的解决方案。
主要相关工作有哪些，存在的主要问题，本文工作相对于相关工作的创新点	本毕业设计的算法基础是王思博等人提出的FORA算法以及基于 Reid Anderson 等人提出的 Backward Search 算法的 All-Pair-Backward-Search 算法，它们为在海量数据下 Single-Source PPR 以及 Top-k PPR 估计值的计算提供了高效且有效限制误差范围的算法实现。而本毕业设计的创新点就是在当下最流行的Neo4j 图数据库中实现上述算法，并且根据 Neo4j 图数据库的实际情况对算法进行相应的调整，从而使得算法能在这个平台上发挥其独特的作用。

指导教师签名

摘要

近年来，越来越多的推荐算法通过计算图节点的 Personalized PageRank 来估计用户或物品间的相似度。然而，当前的许多 Personalized PageRank 算法或是存在时间开销的问题，或是存在准确度无法保证的问题，或是不符合应用需求。对此，王思博等人提出了综合 Forward Push 和 Monte-Carlo 算法的 FORA 算法，此外，我也在 Reid Anderson 等人提出的 Backward Search 算法基础上实现了 All-Pair-Backward-Search 算法。为了使得图能够存储更多信息，并且使得以上算法能够被更多的应用所使用，本毕业设计在 Neo4j 图数据库上实现了 FORA 算法以及 All-Pair-Backward-Search 算法。

在此基础上，我将 FORA 算法、All-Pair-Backward-Search 算法以及其他 Personalized PageRank 算法进行实验对比其性能，结果显示 All-Pair-Backward-Search 算法可以以相对较低的时间及空间开销进行预处理。而在不允许进行预处理或图中节点和边的数目较少的情形下，FORA 算法则可以高效地计算出 PPR 估计值并有效限制误差范围。

本毕业设计的创新点就是我们在当下最流行的 Neo4j 图数据库中实现了以上算法，并且根据 Neo4j 图数据库的实际情况对算法进行相应的调整，从而使得算法能在这个平台上发挥其独特的作用。

本毕业设计的代码已经开源，有兴趣可以查看网址 https://github.com/joezie/Personalized-PageRank-Algorithms-on-Neo4j。

关键词：Neo4j 图数据库，节点相似度，Personalized PageRank，Forward Push，Backward Search，随机游走

Abstract

Recently, an increasing number of recommendation algorithms use Personalized PageRank to evaluate similarity among users or objects. However, there are amounts of problems with many Personalized PageRank algorithms in being, such as high cost of time, no guarantee to precision and failing to meet the actual application demands. Aiming at solving these problems, Wang et al. proposed FORA, a PPR algorithm combining Forward Push and Monte-Carlo algorithms. Besides, based on Backward Search algorithm put forward by Reid Anderson et al., I implemented All-Pair-Backward-Search algorithm. To enable a graph to store more information and popularize the algorithms mentioned above, in this senior design I implemented FORA and All-Pair-Backward-Search on Neo4j Graph Database.

Based on this, I conducted experiments on FORA, All-Pair-Backward-Search, and other Personalized PageRank algorithms to compare their performances. The results showed that All-Pair-Backward-Search can conduct preprocessing at a relatively low cost of time and space. In the cases where preprocessing is prohibited or the numbers of nodes and edges are relatively small, FORA can efficiently compute the estimated value of Personalized PageRank and also restrict the error range effectively.

The innovation of this senior design is that we implemented the algorithms mentioned above on Neo4j Graph Database, the most popular graph database nowadays, and meanwhile, we made adjustments based on the actual situation of Neo4j Graph Database, so that these algorithms can contribute to this platform better.

The codes in this senior design are open source. Please refer to https://github.com/joezie/Personalized-PageRank-Algorithms-on-Neo4j if interested.

Key words: Neo4j Graph Database, Node Similarity, Personalized PageRank, Forward Push, Backward Search, Random Walk

目录

1 引言 1
1.1 问题介绍 1
1.2 算法介绍 2
1.3 论文章节安排 2
2 背景介绍 4
2.1 定义及约束条件 4
2.2.1 PPR 定义 4
2.2.2 误差约束条件 5
2.2 研究现状 6
2.2.1 Power Iterations 算法 6
2.2.2 Monte-Carlo 算法 7
2.2.3 Forward Push 算法 8
2.2.4 Backward Search 算法 9
2.2.5 BiPPR 算法 10
2.3 解决方案 11
2.4 研究目的及意义 11
3 Neo4j 介绍 13
3.1 Neo4j 概况 13
3.2 基于 Neo4j 的开发工具 13
3.2.1 数据导入方法 14
3.2.2 用户自定义过程 14
3.2.3 Java API 14
4 算法实现 17
4.1 算法介绍 17
4.1.1 FORA Single-Source PPR 算法介绍 17
4.1.2 FORA Top-k PPR 算法介绍 19

4.1.3 All-Pair-Backward-Search 算法介绍 21
4.2 实现过程 22
4.2.1 接口模块 22
4.2.2 辅助类模块 23
4.2.3 算法类模块 24
5 实验结果 27
5.1 实验设置 27
5.1.1 实验环境 27
5.1.2 数据集 27
5.1.3 实验方法 28
5.1.4 性能指标 29
5.1.5 参数设置 30
5.2 预处理实验结果 32
5.3 Single-Source PPR 实验结果 35
5.4 Top-k PPR 实验结果 37
5.5 图加载开销 39
6 结论 40
致谢 42
参考文献 43
附录 45

表格目录

表 1　Neo4j 数据导入方法 14
表 2　实验环境 27
表 3　数据集介绍 27
表 4　预处理实验参数设置 31
表 5　Single-Source PPR 实验参数设置 31
表 6　Top-k PPR 实验参数设置 32
表 7　GRQC 数据集预处理时间开销 33
表 8　BlogCatalog 数据集预处理时间开销 33
表 9　Flickr 数据集预处理时间开销 33
表 10　Com-Amazon 数据集预处理时间开销 33
表 11　GRQC 数据集预处理空间开销 34
表 12　BlogCatalog 数据集预处理空间开销 34
表 13　Flickr 数据集预处理空间开销 34
表 14　Com-Amazon 数据集预处理空间开销 34
表 15　GRQC 数据集 Single-Source PPR 运行时间开销 36
表 16　BlogCatalog 数据集 Single-Source PPR 运行时间开销 36
表 17　Flickr 数据集 Single-Source PPR 运行时间开销 36
表 18　Com-Amazon 数据集 Single-Source PPR 运行时间开销 36
表 19　GRQC 数据集 Top-k PPR 运行时间开销（以平均准确度为基准） 37
表 20　BlogCatalog 数据集 Top-k PPR 运行时间开销（以平均准确度为基准） 37
表 21　Flickr 数据集 Top-k PPR 运行时间开销（以平均准确度为基准） 38
表 22　Com-Amazon 数据集 Top-k PPR 运行时间开销（以平均准确度为基准） 38
表 23　GRQC 数据集 Top-k PPR 运行时间开销（以平均归一化折损累计增益为基准） ..38
表 24　BlogCatalog 数据集 Top-k PPR 运行时间开销（以平均归一化折损累计增益为基准）
..... 38
表 25　Flickr 数据集 Top-k PPR 运行时间开销（以平均归一化折损累计增益为基准） 39
表 26　Com-Amazon 数据集 Top-k PPR 运行时间开销（以平均归一化折损累计增益为基准）
..... 39
表 27　图加载平均时间开销 39

1 引言

1.1 问题介绍

近年来，越来越多的应用基于对用户或者物品的相似度分析，开发了许多受消费者欢迎的功能。网易云音乐的推荐歌单功能、淘宝商城的推荐商品功能、新浪微博的推荐关注功能等都很好地为用户提供了可能感兴趣的用户或物品推荐。各应用的推荐算法各不相同，其中一种应用甚广的算法是通过计算图节点的 Personalized PageRank 来估计用户或物品间的相似度。

Personalized PageRank 最先被用于 Google 搜索引擎中，以 Google 创始人之一的拉里•佩奇（Larry Page）的名字命名[1]。他的文章中介绍了 Personalized PageRank 的定义[2]，以下我们将其简称为 PPR。给定一个图 G 以及一对节点 s 和 t，节点 t 相对于节点 s 的 PPR 定义为从 s 开始一次随机游走在节点 t 处停止的概率。这反映了节点 t 相对于节点 s 的重要程度，因此可以作为推荐算法的依据。

音乐软件 Spotify 的歌单推荐功能“Discover Weekly”就是其中一个基于 PPR 的歌曲推荐算法。若用户 A 的歌单与用户 B 相似，那么用户 A 就很有可能对用户 B 歌单中的歌曲感兴趣，因此 Spotify 就会将用户 B 歌单中的歌曲推荐给用户 A。而对用户 A 和用户 B 的歌单相似度的分析便是基于对图节点的 PPR 的计算来进行估计的[3]。

Single-Source Personalized PageRank 则是 Personalized PageRank 的一个特殊情形，它给定了源节点 s，再进而计算图中其他点相对于 s 的 PPR。这选定了某一用户或物品作为算法考察的核心对象，然后考察图中其他所有节点相对于 s 的重要程度，更贴合了实际的用户推荐需求。以下我们将简称其为 Single-Source PPR。

通常来说，我们并不需要知道图中所有节点相对于源节点的重要程度，我们常常关注的是最重要的一些节点。因此，Single-Source PPR 还有一个应用十分普遍的变体是 Top-k PPR。它只关注最大的 k 个 Single-Source PPR，这也是实际应用中推荐功能的应用情形：推荐最有可能感兴趣的 k 个用户或物品。

此外，有时所有节点对之间的 PPR 也是人们所感兴趣的，亦即依次以所有节点为源节点的 Single-Source PPR，这经常用于有预处理的 PPR 应用中，通过提前计算好所有节点对之间的 PPR

来提高查询速度。以下我们将简称其为 All-Pair PPR。

1.2 算法介绍

目前已经有许多算法用于计算 PPR。Power Iterations 算法是 Larry Page 等人提出的计算 PPR 精确值的算法，该算法经过多次矩阵计算并最终在 PPR 值达到收敛时停止[2]。然而计算 PPR 精确值的时间复杂度过高，在海量数据的情形下耗时过长，不符合用户的使用需求。因此更多的前沿算法专注于计算 PPR 估计值，通过牺牲部分准确度的方式来达到减小时间开销的目的。Daniel Fogaras 等人提出的 Monte-Carlo 算法的方法是从源节点开始进行足够多次的随机游走，然后用停留在各节点的频率来估计节点的 PPR 值[4]。Reid Andersen 等人提出的 Forward Push 算法则是从源节点开始通过一系列的传递将概率数值分布给各个节点，从而得以估计各节点的 PPR 值[5]。Forward Push 算法的变体 Backward Search 也常常被用到，它是从目标节点开始传播概率数值到各个节点[6]。Peter Lofgren 等人提出的 BiPPR 算法用于计算一对节点之间的 PPR 值，它首先从目标节点开始进行 Backward Search，然后从源节点开始进行足够多次的随机游走，最后结合这两个步骤的结果得到目标节点相对于源节点的 PPR 估计值[7]。

然而，上述算法或是存在时间开销的问题，或是存在准确度无法保证的问题，或是不适用于 Single-Source PPR 及 Top-k PPR 的计算，对此，王思博等人提出了综合 Forward Push 和 Monte-Carlo 算法的 FORA 算法[8]，而我也在 Reid Anderson 等人提出的 Backward Search 算法[5]基础上实现了 All-Pair-Backward-Search 算法。目前这两个算法都是将图的节点以及边的信息存储在文本中，通过读文件的方式加载图。为了使得图能够存储更多信息，并且使得算法能够被更多的应用所使用，本毕业设计将在 Neo4j 图数据库上实现 FORA 算法以及 All-Pair-Backward-Search 算法。

1.3 论文章节安排

本文共由六个章节组成，组织结构如下：

第一章“引言”初步介绍 PPR 的含义及其应用，并概述现有的各 PPR 算法，同时提出我们的毕业设计方案，使读者对全文有初步的了解。

第二章“背景介绍”将会给出 PPR 的定义及约束条件，并且详细描述各 PPR 算法的概况及优

缺点，然后介绍我们的解决方案，并陈述其目的及意义。这一部分讲述了详尽的背景知识，为后文的方案阐述提供了知识准备。

第三章“Neo4j 介绍”将会介绍 Neo4j 图数据库的概况以及基于 Neo4j 的若干开发工具。Neo4j 图数据库是本毕业设计的实现基础，因此这一部分对其进行基本介绍以及部分技术细节的深入解读。

第四章“算法实现”将会分别介绍我们实现的算法 FORA Single-Source PPR、FORA Top-k PPR 以及 All-Pair-Backward-Search。然后我们会展示我们的实现过程，剖析代码结构，使得读者对代码实现有更直观的了解。

第五章“实验结果”将会首先讲述实验设置，然后分别给出预处理、Single-Source PPR 以及 Top-k PPR 的实验结果及分析，最后还会讲解算法中图加载的开销问题。这一部分是毕业设计成果的最终呈现，通过实验结果展示了本毕业设计的性能表现。

第六章“结论”将会给出对本毕业设计中存在问题的分析，并提出对未来的展望，同时也有我在毕业设计过程中收获的心得体会。

2 背景介绍

这一章节将会讲述详尽的背景知识，为接下来的方案阐述提供知识准备。我们将会分四个小节分别讲述 PPR 的定义及约束条件、PPR 算法的研究现状、我们实现的解决方案以及本毕业设计的目的及意义。

2.1 定义及约束条件

2.2.1 PPR 定义

在一个有向图$G=(V,E)$中，节点数目为$|V|=n$，边的数目为$|E|=m$，我们选定一个源节点$s\in V$以及衰减因子α，从 s 开始进行一次随机游走，在每一步中我们以α的概率停留在该点，或是以$1-\alpha$的概率移动到任一个出向邻居节点。对于所有节点$v\in V$，我们将 v 相对于 s 的 PPR 定义为从 s 开始的一次随机游走在 v 处停止的概率，我们将其记作$\pi(s,v)$[2]。由定义我们可以看到，PPR 可以用于衡量节点之间的相似度。

在以上定义中，若我们固定源节点为 s，那么我们得到的即为关于节点 s 的 Single-Source PPR；若我们进一步地加入参数 k，并且将 Single-Source PPR 的结果限定为最大的 k 个值，那么我们得到的是关于节点 s 的 Top-k PPR；而如若我们依次以所有节点$v\in V$作为源节点得到各自相应的 Single-Source PPR，那么我们就可以得到所有节点的 All-Pair PPR。

值得一提的是，在随机游走的过程中，我们采用 Yasuhiro Fujiwara 等人提出的方法，引入“重启机制”，亦即若在随机游走的某一步时，当前节点需要移动到某一出向邻居节点，但该节点没有出边，那么就回到源节点处重新开始随机游走[9]。引入这一机制可以保证算法在一些特殊情形下的健壮性，下面例子可以用来说明问题。

假设在有向图$G=(V,E)$中，$V=\{s,t\}$，$E=\{(s,t)\}$，我们给定衰减因子α并选定源节点为 s 进行一次随机游走。若不引入“重启机制”，那么我们的随机游走停在 s 的概率为α，而以$1-\alpha$的概率移动到 t 处。又由于 t 没有出边，因此随机游走将会在 t 处停止，亦即停在 t 的概率为$1-\alpha$。因此我们有$\pi(s,s)=\alpha$，$\pi(s,t)=1-\alpha$。当$\alpha<0.5$时，我们有$\pi(s,s)<\pi(s,t)$，亦即节点 s 与自身的相似度小于节点 t 与节点 s 的相似度，这显然是不符合实际情况的。因此我们在此引

入“重启机制”，那么随机游走的情形就会变成：从 s 开始，以α的概率停在 s 处，并以$1-\alpha$的概率移动到 t 处。然后会有$\alpha\cdot(1-\alpha)$的概率停在 t 处，并以$(1-\alpha)^2$的概率回到 s 处。然后再从 s 开始进行新一轮的随机游走。因此若随机游走触发“重启机制”的次数为 c（亦即随机游走返回源节点 c 次），那么随机游走最终停留在 s 的概率为

$$P(s,s)=\alpha\cdot\sum_{i=0}^{c}(1-\alpha)^{2i}$$

而停留在 t 的概率为

$$P(s,t)=\alpha\cdot\sum_{i=0}^{c}(1-\alpha)^{2i+1}$$

又由于$1-\alpha<1$，我们可以得到$\pi(s,s)>\pi(s,t)$，因此得以正确地描述实际情况。

2.2.2 误差约束条件

本文关注图节点的 Single-Source PPR 以及 Top-k PPR。然而，计算其精确值的时间开销极大，在现实的应用情景中不符合用户的使用需求。因此，本文将着重于描述计算 Single-Source PPR 及 Top-k PPR 估计值的算法，他们通过牺牲结果的部分准确度来减小时间开销。当然，算法的准确度不能是没有约束的，因此我们下面将分别描述 Single-Source PPR 及 Top-k PPR 估计值需要满足的约束条件。

2.2.2.1 Single-Source PPR 误差约束条件

假设 PPR 阈值为δ，误差阈值为ϵ，失效概率为p_f，给定一个有向图$G=(V,E)$，我们选定一个源节点 s，对于每个节点$v\in V$，若该节点的 Single-Source PPR 精确值$\pi(s,v)>\delta$，那么该节点的 Single-Source PPR 估计值$\hat{\pi}(s,v)$就应该满足条件

$$|\pi(s,v)-\hat{\pi}(s,v)|\le\epsilon\cdot\pi(s,v) \qquad (1)$$

且该条件以$1-p_f$的概率成立[8]。

约束条件(1)限定了 Single-Source PPR 精确值与其估计值的相对误差范围。

2.2.2.2 Top-k PPR 误差约束条件

假设 PPR 阈值为δ，误差阈值为ϵ，失效概率为p_f，结果个数为 k，给定一个有向图$G=(V,E)$，选定一个源节点 s，我们将 Top-k PPR 查询得到的估计值最大的 k 个节点按照数值大小依次记作$v_1, v_2, \cdots, v_k$，而其精确值最大的 k 个节点按照数值大小依次记作$v_1^*, v_2^*, \cdots, v_k^*$，若节点$v_i^*$的 Top-k PPR 精确值$\pi(s, v_i^*) > \delta, (i \in [1,k])$，那么 Top-k PPR 估计值$\hat{\pi}(s, v_i)$及其精确值$\pi(s, v_i)$就应该满足条件

$$|\pi(s, v_i) - \hat{\pi}(s, v_i)| \leq \epsilon \cdot \pi(s, v_i) \quad (2)$$

$$\pi(s, v_i) \geq (1-\epsilon) \cdot \pi(s, v_i^*) \quad (3)$$

且上述两个条件以$1-p_f$的概率成立[8]。

约束条件(2)限定了查询结果中节点的 Top-k PPR 估计值与其自身的精确值的相对误差范围。而约束条件(3)则限定了查询结果中节点的 Top-k PPR 精确值与实际排名在该位置上的节点的 Top-k PPR 精确值的相对误差范围。

2.2 研究现状

计算图节点的 PPR 算法大致可以分为两种：一种是计算精确值的算法，以 Power Iterations 算法为代表；另一种则是计算估计值的算法，包括 Monte-Carlo 算法、Forward Push 算法、Backward Search 算法、BiPPR 算法等。其中后者为本文讨论的重点。接下来我将逐一介绍这些 PPR 算法的概况及其优缺点。

2.2.1 Power Iterations 算法

作为 PageRank 的提出者，Larry Page 等人提供了 Power Iterations 算法计算 PPR 精确值。其主要思想是在每一次迭代中，各个节点都将自己的 PPR 值均分给出向邻居，每个节点又将从入向邻居收到的值加起来得到新的 PPR 值，如此进行多次迭代直至结果收敛。为了解决 PPR 值可能过度集中于个别节点的问题，Larry Page 等人在此基础上又引入了同比缩减和统一补偿，这对应了 PPR 定义中的衰减因子α。在每一次迭代前，我们对每个节点的 PPR 值进行同比缩减，即乘以$(1-\alpha)$。而在每一次迭代后，我们为所有节点进行统一补偿，亦即每个节点的 PPR 值都加上$\frac{\alpha}{n}$。

特别地，我们选定一个源节点 s，并将其 PPR 初始值设为 1，其余节点则初始化为 0，那么

这一算法就可以得到关于节点 s 的 Single-Source PPR。

上述过程中的每一次迭代都等价于一次矩阵运算，具体方法描述如下：给定一个有向图$G=(V,E)$以及源节点 s，假设衰减因子为α，邻接矩阵为$M\in\{0,1\}^{n\times n}$，对角矩阵$D\in R^{n\times n}$中对角线上第 i 行的元素d_{ii}值为对应的第 i 个节点v_i的出边数目，单位向量为$e\in R^n$，第 k 次迭代后得到的 PPR 值向量为$U_k\in R^n$，那么每次迭代的矩阵运算可以表示为

$$U_{k+1}=(1-\alpha)\cdot U_k\cdot D^{-1}M+\alpha\cdot e$$

其中U_0中 s 对应的元素初始化为 1，其余元素则置为 0[2]。

这一方法虽然直观简单，并且可以得到关于节点 s 的 Single-Source PPR 精确值，然而这一算法的时间及空间开销都很大。一方面，由于这一算法运用了矩阵乘法运算，因此即使是运用当今最前沿的矩阵乘法算法，单次矩阵乘法运算的时间复杂度也仍然达到了$O(n^{2.37})$[10]。另一方面存储矩阵需要用到$O(n^2)$的空间。在海量数据的情形下，Power Iterations 算法在时间以及空间上的开销都是难以接受的，这也是为什么现今大多数前沿的 PPR 算法都是计算其估计值来减小时间开销，此外也避免进行矩阵运算从而减小空间开销。

2.2.2 Monte-Carlo 算法

Daniel Fogaras 等人提出了利用 Monte-Carlo 算法来计算 Single-Source PPR 估计值的想法。给定一个有向图$G=(V,E)$以及源节点 s，假设衰减因子为α，PPR 阈值为δ，误差阈值为ϵ，失效概率为p_f，从源节点开始进行ω次随机游走，停止在节点v_i的次数记作c_i，那么v_i关于 s 的 PPR 估计值为$\hat{\pi}(s,v_i)=\frac{c_i}{\omega}$。为了满足约束条件(1)，随机游走的次数$\omega$需满足$\omega\geq\frac{3\log(2/p_f)}{\epsilon^2\delta}$。同时，若假设随机游走的每一步中，选择一个任意的出向邻居节点并移动到该节点只需要$O(1)$时间，那么由于随机游走的路径长度的数学期望为

$$E(len)=\sum_{len=0}^{\infty}len\cdot\alpha\cdot(1-\alpha)^{len}$$
$$=\frac{1-\alpha}{\alpha}$$

其中α为事先给定的常数，因此单次随机游走的时间复杂度是$O(1)$的，从而 Monte-Carlo 算法的时间复杂度是$O(\frac{\log(1/p_f)}{\epsilon^2\delta})$的[4]。

Monte-Carlo 算法是 PPR 定义的直观扩展，通过采样的方式来估计节点的 PPR 值，避免了矩阵运算的巨大开销的同时，也很好地限制了误差范围。然而，为了达到约束条件(1)的误差范围要求，算法所需的随机游走次数可能会很多。特别地，当$\delta = O(1/n)$，$p_f = O(1/n)$时，Monte-Carlo 算法的时间复杂度达到了$O(\frac{n \cdot \log n}{\epsilon^2})$，时间开销仍然颇大。

2.2.3 Forward Push 算法

Reid Anderson 等人提出了 Forward Push 算法计算 Single-Source PPR 估计值。与 Power Iterations 算法类似地，Forward Push 算法的主要思想也是通过多次迭代的方式从源节点开始传播 PPR 值。然而，他们对 PPR 值传播的过程进行了修改，引入了两个值 residue 以及 reserve。其中 residue 指的是被传播的价值，而 reserve 则是作为算法最后留存在节点中的价值，由 residue 转化而来。

给定一个有向图$G = (V, E)$以及源节点 s，假设衰减因子为α，residue 阈值为r_{max}，我们将节点 v 的 residue 记作$r(s, v)$，reserve 记作$\pi(s, v)$。在初始化时我们设置$r(s, s) = 1$，对于所有节点$v \in \{V \mid v \neq s\}$，我们设置$r(s, v) = 0$。而对于所有节点$v \in V$，我们设置$\pi(s, v) = 0$。在第 k 次迭代中，对于所有节点$v \in V$，若$r_k(s, v)/d_{out}(v) \geq r_{max}$，其中$d_{out}(v)$为节点 v 的出边数目，那么我们就执行以下步骤。

首先我们将该节点α部分的 residue 转化为自身的 reserve，亦即

$$\pi_{k+1}(s, v) = \pi_k(s, v) + \alpha \cdot r_k(s, v)$$

其中$\pi_k(s, v)$，$r_k(s, v)$分别为节点 v 在第 k 次迭代时的 reserve 和 residue 值。

然后我们将该节点剩余的$(1 - \alpha)$部分的 residue 均分给节点 v 的出向邻居节点，亦即对于所有节点$u \in \{w \in V \mid (v, w) \in E\}$，我们有

$$r_{k+1}(s, u) = r_k(s, u) + (1 - \alpha) \cdot r_k(s, v)/d_{out}(v)$$

值得一提的是，为了与 2.2.1 中提到的“重启机制”保持一致，若当前节点的出边数目为 0 时，我们将该节点剩余的$(1 - \alpha)$部分的 residue 传递给源节点 s，亦即

$$r_{k+1}(s, s) = r_k(s, s) + (1 - \alpha) \cdot r_k(s, v)$$

最后我们将该节点的 residue 置为 0，亦即

$$r_{k+1}(s, v) = 0$$

在每次迭代结束时。我们都检查是否存在节点v ∈ V满足$r_k(s,v)/d_{out}(v) \geq r_{max}$，若是，我们继续进行下一轮迭代。否则，我们就可以终止算法，此时留存在各节点 v 中的π(s, v)即可作为关于源节点 s 的 PPR 估计值[5]。

Forward Push 算法的时间复杂度是O($\frac{1}{r_{max}}$)的，若我们将 residue 阈值r_{max}设置得很小，那么 Forward Push 算法可以以很高的代价计算出精确的 PPR 值；若将r_{max}设置得较大，那么条件$r_k(s,v)/d_{out}(v) \geq r_{max}$就可以保证每一次迭代时都会有相当多的 residue 转化为 reserve，加快了算法的速度，同时也使得算法可以提前停止。但是算法得出的 PPR 估计值在最坏情形下却不能限制绝对或者相对误差的范围，因而也无法满足 Single-Source PPR 或者 Top-k PPR 的误差约束条件。

2.2.4 Backward Search 算法

Reid Anderson 等人还提出了 Backward Search 算法，总体看来，它可以看作是 Forward Push 算法的“逆向过程”，与后者不同的是，Backward Search 算法是从选定的目标节点 t 处开始传播 residue 值。

给定一个有向图G = (V, E)以及目标节点 t，假设衰减因子为α，residue 阈值为r_{max}，我们将节点 v 的 residue 记作r^b(v, t)，reserve 记作π^b(v, t)，residue 阈值为r^b_{max}。在初始化时我们设置r^b(t, t) = 1，对于所有节点v ∈ {V | v ≠ t}，我们设置r^b(v, t) = 0。而对于所有节点v ∈ V，我们设置π^b(v, t) = 0。在第 k 次迭代中，对于所有节点v ∈ V，若$r^b(v,t) > r^b_{max}$，那么我们就执行以下步骤。

首先我们将该节点α部分的 residue 转化为自身的 reserve，亦即

$$\pi^b_{k+1}(v,t) = \pi^b_k(v,t) + \alpha \cdot r^b_k(v,t)$$

其中$\pi^b_k(v,t)$，$r^b_k(v,t)$分别为节点 v 在第 k 次迭代时的 reserve 和 residue 值。

然后我们将该节点剩余的 residue 传递给节点 v 的入向邻居节点。需要注意的是，我们并不是将 residue 均分给入向邻居节点，由于 Backward Search 是 Forward Push 的“逆向过程”，对于入向邻居节点 u，我们应该将$\frac{1}{d_{out}(u)}$部分的 residue 传递给 u，其中$d_{out}(u)$为节点 u 的出边数目。因此，对于所有节点u ∈ {w ∈ V | (v, w) ∈ E}，我们有

$$r_{k+1}^{b}(u,t) = r_{k}^{b}(u,t) + (1-\alpha)\cdot r_{k}^{b}(v,t)/d_{out}(u)$$

值得一提的是，若当前节点的入边数目为 0，我们并不会像 Forward Push 中那样将 residue 传递给目标节点，因而在入边数目为 0 的节点处会出现 residue 流失的问题，这一缺陷也决定了 Backward Search 算法只能在无向图上运行。

最后我们将该节点的 residue 置为 0，亦即

$$r_{k+1}^{b}(v,t) = 0$$

在每次迭代结束时。我们检查是否存在节点v ∈ V满足$r^{b}(v,t) > r_{max}^{b}$，若是，我们继续进行下一轮迭代。否则，我们就可以终止算法，此时留存在各节点 v 中的$\pi^{b}(\mathrm{v},\mathrm{t})$即可作为关于目标节点 t 的 PPR 估计值[6]。

Peter Lofgren 等人还证明了 Backward Search 算法的均摊时间复杂度为$O(\frac{m}{n\cdot r_{max}^{b}})$的[7]。虽然 Backward Search 算法并不作为独立的算法被用于计算 Single-Source PPR 或 Top-k PPR，但它可以作为其它 PPR 算法的组成成分来为计算 PPR 估计值做贡献，2.2.5 中提到的 BiPPR 算法、4.1.3 中提到的 All-Pair-Backward-Search 算法都用到了 Backward Search 算法作为其算法的一部分。

2.2.5 BiPPR 算法

Peter Lofgren 等人提出了 BiPPR 算法用于计算图中给定的一对节点之间的 PPR 估计值。总体来说，BiPPR 算法包括 Backward Search 和随机游走两部分，其中第一部分从目标节点 t 开始进行的 Backward Search 可以看作是通过传播 residue 值的方式来找到一批“临近”t 的节点，而第二部分从源节点 s 开始的若干次随机游走则可看作是检测第一部分中找到的节点集合，最后将两部分的结果综合起来即可得到目标节点 t 关于源节点 s 的 PPR 估计值。

给定一个有向图$\mathrm{G} = (\mathrm{V},\mathrm{E})$，一个源节点 s，以及一个目标节点 t。假设衰减因子为α，PPR 阈值为δ，误差阈值为ϵ，失效概率为p_f，residue 阈值为r_{max}^{b}，我们将节点 v 的 residue 记作$r^{b}(\mathrm{v},\mathrm{t})$，reserve 记作$\pi^{b}(\mathrm{v},\mathrm{t})$。我们首先从目标节点 t 开始执行 Backward Search 算法得到各个节点 v 的$r^{b}(\mathrm{v},\mathrm{t})$以及$\pi^{b}(\mathrm{v},\mathrm{t})$，然后再从源节点 s 开始进行ω次随机游走，我们将第 i 次随机游走停止处的节点记作v_i，最后我们得到目标节点 t 关于源节点 s 的 PPR 估计值为

$$\hat{\pi}(\mathrm{s},\mathrm{t}) = \pi^{b}(\mathrm{s},\mathrm{t}) + \frac{1}{\omega}\cdot\sum_{i=1}^{\omega} r^{b}(v_i,t)$$

通过设置一个合理的 residue 阈值r_{max}^{b}，并将随机游走的次数设置为$\omega = \frac{c \cdot r_{max}^{b}}{\delta}$，其中参数 c 的设置依赖于算法的准确度，BiPPR 算法的时间复杂度可以达到$O(\frac{1}{\epsilon} \cdot \sqrt{\frac{m \cdot \log(1/p_f)}{n\delta}})$。特别地，当$\delta = O(1/n)$，$p_f = O(1/n)$时，BiPPR 算法的时间复杂度为$O(\frac{1}{\epsilon} \cdot \sqrt{m \cdot \log n})$ [7]。

相较于单纯的 Backward Search 算法以及 Monte-Carlo 算法，BiPPR 算法可以在满足约束条件(1)的同时有效减少 Backward Search 的迭代次数以及随机游走的次数，并使得两部分的时间开销达到平衡，显著提高了算法性能。然而，BiPPR 算法是用于计算一对节点对之间的 PPR 估计值，若只是简单地重复执行 BiPPR 算法来计算关于源节点 s 的 Single-Source PPR 以及 Top-k PPR，时间开销仍然很大。

2.3 解决方案

从 2.2 中各个算法的问题中我们可以看出，在海量数据下 Single-Source PPR 以及 Top-k PPR 估计值的计算迫切需要一个高效且有效限制误差范围的算法。对此，王思博等人提出了综合了 Forward Push 和 Monte-Carlo 算法的 FORA 算法[8]，而我也在 Reid Anderson 等人提出的 Backward Search 算法[5]基础上实现了 All-Pair-Backward-Search 算法。

到目前为止，FORA 和 All-Pair-Backward-Search 算法都只在 C++中实现，它们都是将图的节点以及边的信息存储在文本中，然后通过读文件的方式加载图之后再进行计算。若希望 FORA 和 All-Pair-Backward-Search 算法能够在更大范围的应用中做出贡献，那么我们就需要将它们嵌入到图数据库中。作为一个日益流行且功能强大的图数据库，Neo4j 可以成为 FORA 和 All-Pair-Backward-Search 算法的良好载体，我也将在第三章中对其进行基本介绍并对部分技术细节进行深入解读。

在本毕业设计中，我将使用 Java 语言实现 Neo4j 图数据库上的 FORA Single-Source PPR 算法、FORA Top-k PPR 算法以及 All-Pair-Backward-Search 算法，在第四章中将会有这三个算法的详细介绍及其实现过程的展示。

2.4 研究目的及意义

FORA 和 All-Pair-Backward-Search 算法作为计算图节点 PPR 的高效算法，尤其是它们在大

数据下优良的性能，很好地契合了当今的市场需求。本毕业设计的实现将使它们有机会被运用到 Neo4j 这样流行且强大的图数据库平台上，而由于图数据库中能储存许多额外的图信息，如节点的属性、边的属性等，PPR 被赋予了更加丰富的含义，从而得以结合这些信息来支撑最后的应用，如链接预测、节点分类等。因此，我们在图数据库中实现更高效的 PPR 算法可以使更多的用户及应用受益。

3 Neo4j 介绍

这一章节将会介绍 Neo4j 图数据库的概况以及基于 Neo4j 的若干开发工具。Neo4j 图数据库是本毕业设计的实现载体，因此我们需要对其进行基本介绍，并对部分技术细节进行深入解读。

3.1 Neo4j 概况

作为 NoSQL 数据库的一员，图数据库利用图结构来存储数据，并且提供对节点、边以及属性信息的语义查询。在处理关联紧密的数据以及表示海量数据之间的关系时，图数据库相比于传统的关系数据库或其他 NoSQL 数据库更加简单且具有表现力[11]。

而作为当今最流行的图数据库，Neo4j 是由 Neo4j 公司开发的一款开源图数据库产品。Neo4j 的底层开发是用 Java 语言实现的，它提供了 Cypher 查询语言以及丰富的 API，使得用其他语言编写的软件可以应用 Neo4j 的强大功能[12]。

在一个传统的模式数据库中，用户在添加数据之前必须定义数据的存储结构。比如在 MySQL 中，用户必须预先对表格的各列进行定义，然后才能添加一行数据。而 Neo4j 作为一个模式自由（Schema-free）的图数据库，其用户可以在没有预先定义的存储结构的前提下直接添加数据，甚至可以将拥有不同存储结构的数据进行聚合。Neo4j 的模式自由特性使得它更适合处理复杂且联系紧密的数据。此外，Neo4j 支持事务的 ACID 属性，亦即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）以及持久性（Durability）。

在接下来的实现中，Neo4j 中的每个节点将代表一个实体，如社交网络中的用户。而每条边则代表实体间的关系，如用户间的好友关系。在此基础上，节点和边还可以拥有若干属性，如用户的姓名、关系的紧密程度等。而这些丰富的属性通过与图节点的 PPR 值相结合，将会对最后的应用提供有力的支撑。

3.2 基于 Neo4j 的开发工具

作为一个日臻完善的开源项目，Neo4j 为开发者提供了丰富的开发工具，使得使用各种语言的开发者都有机会利用 Neo4j 的强大功能。而在本毕业设计中，我们将会用到 Neo4j 的多种数

据导入方法、用户自定义过程以及其 Java API。下面我将逐一介绍这些开发工具。

3.2.1 数据导入方法

在本毕业设计中，我们将会导入海量数据作为我们的测试数据集，因此我们需要一个简便高效的数据导入方法。Neo4j 为开发者提供了多种数据导入方法，导入速度不同的各种方法适用于不同的应用场景，同时它们也有各自的优缺点，具体情况详见表 1。

表 1 Neo4j 数据导入方法[13]

	CREATE语句	LOAD CSV语句	Batch Inserter	Batch Import	Neo4j-admin Import
适用场景	1 ~ 1万 nodes	1万 ~ 10万 nodes	千万以上 nodes	千万以上 nodes	千万以上 nodes
速度	很慢（1000 nodes/s）	一般（5000 nodes/s）	非常快（数万 nodes/s）	非常快（数万 nodes/s）	非常快（数万 nodes/s）
优点	使用方便，可实时插入。	使用方便，可以加载本地/远程CSV；可实时插入。	速度相比于前两个，有数量级的提升	基于Batch Inserter，可以直接运行编译好的jar包；可以在已存在的数据库中导入数据	官方出品，比Batch Import占用更少的资源
缺点	速度慢	需要将数据转换成CSV	需要转成CSV；导入时必须停止neo4j；只能在JAVA中使用	需要转成CSV；导入时必须停止neo4j	需要转成CSV；导入时必须停止Neo4j；只能生成新的数据库，而不能在已存在的数据库中插入数据。

在本毕业设计中，我们所用到的数据集包含的节点数目将会是千万级别的。此外，由于我们并只需要在最开始时导入数据，而不需要进行增量的数据导入，因此 Neo4j-admin Import 方法即可满足我们的需求。

3.2.2 用户自定义过程

与函数相比，Neo4j 中的过程（Procedure）可以进行更加复杂的操作并且以数据流作为返回结果，在 Cypher 查询中我们通过 CALL 语句来调用过程，并通过 YIELD 语句来选择输出结果的域。而作为一个具有可扩展性的图数据库，Neo4j 还为开发者提供了用户自定义过程（User-defined procedure）。开发者在实现用户自定义过程后需要将其打包成 JAR 文件并添加至相应数据库的 plugin 文件夹下，即可在 Cypher 查询中调用该过程[15]。

为了使 Neo4j 的用户能够更方便快捷地使用我们的算法来计算图节点的 PPR 值，我将使用用户自定义过程来包装我所实现的算法，并最终生成一个 JAR 文件供用户使用。

3.2.3 Java API

Neo4j 为 Java 开发者们提供了丰富的 API，我们只需要导入 Neo4j 的 JAR 文件并将 Neo4j 添加到依赖中即可使用，可以进行的操作包括创建及关闭数据库、为节点属性创建索引、访问节点及边等。此外，Neo4j 还提供了 neo4j-graph-algorithms 库，我们只需导入其相应的 JAR 文件即可使用其多样的图算法 API，如链接预测、社区发现、中心度计算等。其中，Neo4j 提供的 PPR 算法及其对 O(1)时间的随机游走的支持与本毕业设计有着紧密的联系，因此接下来我将对它们的部分技术细节进行深入解读。

3.2.3.1 Neo4j Single-Source PPR 算法

2018 年 7 月，Neo4j 发布了在该平台上计算 PPR 的图算法，并将其加入到了 3.4.4.0 版本的 neo4j-graph-algorithms 库中[15]，以下我们会将其记作 Neo4j Single-Source PPR 算法。为了更好地了解这一竞争对手，我深入研读了其 PPR 算法的源代码，了解到其算法实现基础是 Bundit Manaskasemsak 等人提出的基于分区的 PPR 并行算法[16]及 David Gleich 等人提出的线性系统 PPR 并行算法[17]，这两个算法本质上都是用 Power Iterations 算法来进行 PPR 计算。Neo4j Single-Source PPR 算法是首先从数据库中将图加载出来，然后利用 Power Iterations 算法计算图节点 PPR 值，这样便省去了计算 PPR 过程中频繁通过事务与数据库进行交互的时间开销，这一点也可以借鉴到我们的算法实现中。

需要注意的是，Neo4j Single-Source PPR 算法对于α的定义恰与我们相反，表示的是随机游走在每一步时移动到任一出向邻居节点的概率。此外，它们的 PPR 返回值并未进行归一化操作，亦即所有节点的 PPR 之和不为 1，因此还需要我们另行进行归一化操作才能得到我们想要的结果。

3.2.3.2 Neo4j 中 O(1)时间随机游走的实现

随机游走是 FORA 算法的重要组成部分，因此保证单次随机游走的时间复杂度是 O(1)的就变得至关重要，而这就要求在随机游走的每一步中，选择一个任意的出向邻居节点并移动到该节点的时间是$O(1)$的。

若我们只使用 Neo4j 提供的 API，那么选择一个任意的出向邻居节点的实现如下：首先利用 Node.getDegree()函数获得当前节点的出边数目 out_degree，然后生成一个 1 到 out_degree 之间

的随机整数 rand，再利用 Node.getRelationships().iterator()函数获得当前节点的所有出边的迭代器并赋值给 Iterator<Relationship>类型变量 iter，然后对 iter 调用 rand 次 Iterator.next()函数并将最后一次调用的返回值赋值给 Relationship 类型变量 rel，最后对 rel 调用 Relationship.getOtherNode()函数获得这一任意选择的出向邻居节点。

总的来说，我们只能用类似于“遍历邻接表”的方式去得到当前节点的一个任意的出向邻居，显然其时间复杂度是 O(out_degree)的，并不符合我们的要求。因此，我又深入研读了 Neo4j 的源代码以了解其图存储以及节点访问的底层实现机制，最后发现了一个名为 HeavyGraph 的类，其图存储机制是通过 AdjacencyMatrix 类型的成员变量 container 的两个不定长二维数组 outgoing 和 incoming 来分别存储各节点的出边邻居和入边邻居。一方面，不定长的二维数组结构相比于定长二维数组更加节省稀疏图的存储空间；另一方面，由于二维数组的本质是邻接矩阵，我们便可以利用 HeavyGraph.getTarget()函数在 O(1)时间访问当前节点的任意一个出向邻居。

由于在我们的算法中只需要用到图的节点以及边的关系，而不需要其属性信息，因此我们可以先利用 GraphLoader.load()函数加载 HeavyGraph 类型的图，然后基于它计算 PPR 值。在得到图节点的 PPR 值结果后，若还需要与图的属性信息相结合以支撑应用，可以进一步地通过数据库交互的方式获得这些信息。由此可见，这一解决方案是符合我们的实现目标的。

4 算法实现

这一章节将会依次对本毕业设计所实现的三个算法进行介绍，它们分别为 FORA Single-Source PPR、FORA Top-k PPR 以及 All-Pair-Backward-Search 算法。然后我会展示算法的实现过程，并对代码结构进行剖析，使得读者能够对代码实现有更直观的了解。

4.1 算法介绍

4.1.1 FORA Single-Source PPR 算法介绍

FORA（**FO**rward Push and **RA**ndom Walks）是由王思博等人提出来的用于计算 PPR 估计值的算法。由于该算法综合了 Forward Push 算法和 Monte-Carlo 算法的思想，它可以利用 Forward Push 部分所得到的信息显著地减少随机游走的次数从而提高计算速度，同时又满足 PPR 估计值的误差约束条件。

在 FORA Single-Source PPR 算法的实现中，给定一个有向图$\mathrm{G}=(\mathrm{V},\mathrm{E})$以及源节点 s，假设衰减因子为$\alpha$，PPR 阈值为$\delta$，误差阈值为$\epsilon$，失效概率为$p_f$，residue 阈值为$r_{max}$，我们将节点 v 的 residue 记作$r(\mathrm{s},\mathrm{v})$，reserve 记作$\pi(\mathrm{s},\mathrm{v})$。在第一阶段，我们从源节点 s 开始执行 Forward Push 算法得到各个节点 v 的$r(\mathrm{s},\mathrm{v})$以及$\pi(\mathrm{s},\mathrm{v})$，进一步地我们可以得到各节点 residue 的总和$r_{sum}=\sum_{v_i\in V} r(s,v_i)$。在第二阶段，我们总共需要进行$\omega$次随机游走，为了满足约束条件(1)，我们将随机游走的次数设置为$\omega=r_{sum}\cdot\frac{(2\epsilon/3+2)\cdot\log(2/p_f)}{\epsilon^2\delta}$。对于每个节点$v_i\in V$，若$\mathrm{r}(\mathrm{s},v_i)>0$，那么我们就以$v_i$为起点进行$\omega_i$次随机游走，其中$\omega_i=\lceil r(s,v_i)\cdot\omega/r_{sum}\rceil$，将单次随机游走停止处的节点记作 t，将$\frac{r(s,v_i)}{\omega_i}$累加到 t 的 reserve 值上。因此对于所有节点$\mathrm{t}\in\mathrm{V}$，我们最后得到 t 关于 s 的 PPR 估计值为

$$\hat{\pi}(\mathrm{s},\mathrm{t})=\pi(\mathrm{s},\mathrm{t})+\sum_{v_i\in S}\frac{r(s,v_i)\cdot\omega_i'}{\omega_i}$$

其中 S 为在节点 t 处停止的随机游走起点的集合，ω_i为以节点v_i为起点的随机游走次数，ω_i'为这些随机游走中最终在节点 t 处停止的次数。

为了使得总体的时间复杂度最小，我们将 residue 阈值设置为$r_{max}=\frac{\epsilon}{\sqrt{m}}\cdot\sqrt{\frac{\delta}{(2\epsilon/3+2)\cdot\log(2/p_f)}}$，这样在最坏情形下两个阶段的时间复杂度相同，此时算法总体的时间复杂度可以达到$O(\frac{1}{\epsilon}\cdot\sqrt{\frac{m\cdot(2\epsilon/3+2)\cdot\log(2/p_f)}{\delta}})$。特别地，当$\delta=O(1/n)$，$p_f=O(1/n)$时，算法的时间复杂度为$O(\frac{1}{\epsilon}\cdot\sqrt{m\cdot n\cdot\log n})$。进一步地，在无标度（Scale-free）网络中，节点和边的数目满足性质$\frac{m}{n}=O(\log n)$，此时我们的算法复杂度为$O(\frac{n\cdot\log n}{\epsilon})$，相较于 Monte-Carlo 算法计算速度提高了$\frac{1}{\epsilon}$倍[8]。

在此基础上，王思博等人又提出了进一步的优化方案。优化主要有以下两点：

第一点优化是省去了距离为 0 的随机游走。在 FORA 的随机游走部分，以每个节点 v 为源节点的随机游走都会有α部分停留在该节点上，亦即这部分随机游走的距离为 0。而本优化的思想是省去这α部分的随机游走，将α部分的 residue 直接转化为该点的 reserve，而剩余$(1-\alpha)$部分的 residue 再通过从 v 的任意一个出边邻居节点v'开始进行随机游走的方式来传播，而这些随机游走的距离至少为 1。

第二点优化是通过一个自适应的方法对两个阶段的用时进行平衡，从而减少算法的总用时。在原来的实现中，为了最小化算法的时间复杂度，我们是考虑了最坏情形对r_{max}进行取值的，但是实际情况下两个阶段的运行时间会有所差异，原r_{max}的取值并不能很好地平衡两个阶段的运行时间。由于理论上单次随机游走的用时是$O(\frac{1-\alpha}{\alpha})$的，它的平均用时$t_{rand}$只依赖于$\alpha$，同时在进行完 Forward Push 后我们也可以计算得到随机游走的总次数ω，因此我们可以提前估计第二阶段的总用时$t_2=\omega\cdot t_{rand}$。我们将r_{max}初始化为 1 并调用 Forward Push 算法得到所有节点$v\in V$的 residue 以及 reserve 值，在此过程中我们对 Forward Push 阶段进行计时并累加到第一阶段的总用时t_1中。然后我们利用得到的各节点的 residue 值更新r_{sum}以及ω，值得一提的是，由于我们在第一点优化中采用了直接将α部分的 residue 直接转化为该点的 reserve 的方法，因此在这里r_{sum}的计算应变为

$$r_{sum}=(1-\alpha)\cdot\sum_{v_i\in V}r(s,v_i)$$

我们将计算得到的t_2与t_1进行比较，若$t_1<t_2$则将r_{max}折半后再重复以上步骤调用 Forward Push 算法直至$t_1\geq t_2$。接下来我们再根据得到的r_{sum}、ω以及各节点$v\in V$的 reserve 值来进行随机游走并得到最终的 PPR 估计值。

4.1.2 FORA Top-k PPR 算法介绍

FORA Top-k PPR 算法的基础仍是 FORA Single-Source PPR，但是在其基础上进行了改进以提高计算效率。在 FORA Single-Source PPR 算法中，我们一般将δ的值设置为$\frac{1}{n}$，但是对于计算 Top-k PPR 而言这个取值过于保守，增加了许多不必要的随机游走次数。因此在 FORA Top-k PPR 算法的实现中，我们采取“尝试错误法”（Trial-and-error approach）来提高效率：给定一个有向图$\mathrm{G}=(\mathrm{V},\mathrm{E})$以及源节点 s，假设衰减因子为α，PPR 阈值为δ，误差阈值为ϵ，失效概率为p_f，residue 阈值为r_{max}，关心的 PPR 估计值最高的节点个数为 k。在第 j 次迭代中，我们令$p_f'=\frac{p_f}{n\log n}$，$\delta_j=\frac{1}{2^j}$，$r_{max}=\frac{\epsilon}{\sqrt{m}}\cdot\sqrt{\frac{\delta_j}{(2\epsilon/3+2)\cdot\log(2/p_f')}}$，然后调用 FORA Single-Source PPR 算法，我们将 PPR 估计值最高的 k 个节点按照 PPR 估计值递减顺序排列的集合记作$\mathrm{C}=\{v_1',\cdots,v_k'\}$，而将有可能是真正的 Top-k PPR 但不在集合 C 中的节点集合记作$\mathrm{U}=\{\mathrm{u}\in\mathrm{V}\backslash\mathrm{C}\,|UB_j(\mathrm{u})>(1+\epsilon)\cdot LB_j(v_k')\}$。若$LB_j(v_k')\geq\delta_j$且对于所有节点$v_i'\in C$，我们有$UB_j(v_i')<(1+\epsilon)\cdot LB_j(v_i')$，且不存在节点$\mathrm{u}\in\mathrm{U}$满足$UB_j(\mathrm{u})<(1+\epsilon)\cdot LB_j(\mathrm{u})/(1-\epsilon)$，那么集合 C 中节点的 PPR 估计值即为关于源节点 s 的 Top-k PPR 估计值。否则，我们进行下一轮迭代，若$\delta_j<\frac{1}{n}$则算法终止，因此调用 FORA Single-Source PPR 算法的次数最多为$\lfloor\log_2 n\rfloor$次。

我们将在第 j 次迭代中节点 v 相对于源节点 s 的 PPR 精确值和估计值分别记作$\pi(s,v)$和$\hat{\pi}_j(s,v)$，FORA Single-Source PPR 算法中 Forward Push 阶段得到的 reserve 值记作$\pi^{\circ}(s,v)$，residue 值总和记作r_{sum}，随机游走的总次数记作ω_j。在 FORA Top-k PPR 算法中，$UB_j(v)$及$LB_j(v)$分别表示π(s, v)在第 j 次迭代中的上界以及下界。为了保证 PPR 估计值结果满足约束条件(2)和(3)，我们令$UB_0(v)=1$，$LB_0(v)=0$，

$$UB_j(v)=\min\{1,\hat{\pi}_j(s,v)/(1-\epsilon_j),\hat{\pi}_j(s,v)+\lambda_j\}$$

$$LB_j(v)=\max\{0,\hat{\pi}_j(s,v)/(1+\epsilon_j),\hat{\pi}_j(s,v)-\lambda_j\}$$

其中，

$$\epsilon_j=\sqrt{\frac{3r_{sum}\cdot\log(2/p_f')}{\omega_j\cdot\max\{\pi^{\circ}(s,v),LB_{j-1}(v)\}}}$$

$$\lambda_j = \frac{2/3\log(2/p_f') + \sqrt{\frac{4}{9}r_{sum}^2 \cdot \log^2(2/p_f') + 8r_{sum} \cdot \omega_j \cdot \log(2/p_f') \cdot UB_{j-1}(v)}}{2\omega_j}$$

在判断算法能否结束的判断条件中，由于v_k'是集合 C 中 PPR 估计值最小的节点，因此$LB_j(v_k') \geq \delta_j$就保证了所有节点$v_i' \in C$均满足$LB_j(v_i') \geq \delta_j$，同时根据判断条件，它们还满足$UB_j(v_i') < (1+\epsilon) \cdot LB_j(v_i')$，因此约束条件(2)得到满足。此外，由于不存在节点$u \in U$满足$UB_j(u) < (1+\epsilon) \cdot LB_j(u)/(1-\epsilon)$，我们保证了约束条件(3)也得到满足[8]。

在最坏情况下，我们需要尝试$\delta = \frac{1}{2}, \cdots, \frac{1}{2^{\lfloor \log_2 n \rfloor}}$共$\lfloor \log_2 n \rfloor$次迭代，其时间复杂度为

$$\sum_{j=1}^{\lfloor \log_2 n \rfloor} \frac{1}{\epsilon} \cdot \sqrt{\frac{m \cdot \left(\frac{2\epsilon}{3}+2\right) \cdot \log\left(\frac{2}{p_f'}\right)}{\delta_j}} = \frac{1}{\epsilon} \cdot \sqrt{m \cdot \left(\frac{2\epsilon}{3}+2\right) \cdot \log\left(\frac{2n\log n}{p_f}\right)} \cdot \sum_{j=1}^{\lfloor \log_2 n \rfloor} \sqrt{2}^j$$

$$= \frac{1}{\epsilon} \cdot \sqrt{m \cdot \left(\frac{2\epsilon}{3}+2\right) \cdot \log\left(\frac{2n\log n}{p_f}\right)} \cdot \frac{\sqrt{2}\left(1-\sqrt{2}^{\lfloor \log_2 n \rfloor}\right)}{1-\sqrt{2}}$$

$$< \frac{2+2\sqrt{2}}{\epsilon} \cdot \sqrt{m \cdot n \cdot \left(\frac{2\epsilon}{3}+2\right) \cdot \log\left(\frac{2n\log n}{p_f}\right)}$$

特别地，当$p_f = O(1/n)$时，算法的复杂度为$O(\frac{1}{\epsilon} \cdot \sqrt{m \cdot n \cdot \log n})$。因此即使在最坏的情形下，FORA Top-k PPR 算法的时间复杂度也与 FORA Single-Source PPR 算法相同。而在实际实验中，算法的迭代次数往往会小于$\lfloor \log_2 n \rfloor$，因此 FORA Top-k PPR 算法的计算效率往往是显著高于 FORA Single-Source PPR 算法的。

王思博等人对 FORA Top-k PPR 算法也提出了进一步的优化方案，主要有以下两点：

第一点优化是减少最坏情形下的迭代次数。在利用“尝试错误法”的过程中，我们令$\delta_j = \frac{1}{k \cdot 2^{j-1}}$，$\epsilon' = \frac{\epsilon}{2}$，$p_f' = \frac{p_f}{n\log_2(n/k)}$，$r_{max} = \frac{\epsilon'}{\sqrt{m}} \cdot \sqrt{\frac{\delta_j}{(2\epsilon'/3+2) \cdot \log(2/p_f')}}$，这样 reserve 阈值的初始值$\delta_1$就不再是$\frac{1}{2}$，而是改为了$\frac{1}{k}$，从而使得最坏情形下的迭代次数变为了$\left\lfloor \log_2 \frac{n}{k} \right\rfloor$次。

第二点优化是简化了算法结束的判断条件。在原来的实现中，判断算法是否结束需要考量各节点的上下界UB(v)及 LB(v)，而本优化的思路则是通过将第 k 大的 PPR 估计值，即$\hat{\pi}(s, v_k')$与$(1+\epsilon) \cdot \delta_j$进行比较来取而代之。这样在保证满足约束条件(2)和(3)的前提下也大大简化了每一次迭代结束时的条件判断。

4.1.3 All-Pair-Backward-Search 算法介绍

在 Reid Anderson 等人提出的 Backward Search 算法[6]的基础上，我将其应用于 All-Pair PPR 的计算上并实现了 All-Pair-Backward-Search 算法，它通过预处理建立索引，从而利用索引提高了图节点的 PPR 查询效率。

All-Pair-Backward-Search 算法的总体思想是先在预处理阶段依次以所有节点作为目标节点执行 Backward Search 算法，然后我们将结果通过倒排表的方式归纳起来并存储到文件中，当我们查询关于源节点的 Single-Source PPR 时只需要通过查询文件即可获得相应的结果。

在 All-Pair-Backward-Search 算法的实现中，给定一个有向图$\mathrm{G}=(\mathrm{V},\mathrm{E})$，假设衰减因子为$\alpha$，residue 阈值为$r_{max}^{b}$。我们依次以所有节点$\mathrm{t}\in\mathrm{V}$作为目标节点执行 Backward Search 算法，我们将所得到的节点 v 的 residue 记作$r^{b}(\mathrm{v},\mathrm{t})$，reserve 记作$\pi^{b}(\mathrm{v},\mathrm{t})$。接下来对于所有节点$\mathrm{v}\in\mathrm{V}$，我们对$\pi^{b}(\mathrm{v},\mathrm{t})\geq r_{max}^{b}$的 reserve 值进行排序，然后以$\left(\mathrm{t},\pi^{b}(\mathrm{v},\mathrm{t})\right)$元组的形式记录在节点 v 所对应的文件中。当我们查询关于源节点 s 的 Single-Source PPR 时，我们将节点 s 所对应的文件中各目标节点 t 及其所对应的 reserve 值$\pi^{b}(\mathrm{s},\mathrm{t})$返回，以作为各节点相对于源节点 s 的 PPR 估计值。

在一次 Backward Search 中，对于选定的源节点 s 和目标节点 t，我们有

$$\pi(\mathrm{s},\mathrm{t})=\pi^{b}(s,t)+\sum_{v\in V}\pi(s,v)r^{b}(v,t)$$

其中$\pi(\mathrm{s},\mathrm{t})$为节点 t 关于源节点 s 的 PPR 精确值。因此以$\pi^{b}(s,t)$作为 PPR 估计值的误差为

$$\begin{aligned}\sum_{v\in V}\pi(s,v)r^{b}(v,t)&\leq\sum_{v\in V}\pi(s,v)\cdot r_{max}^{b}\\&=r_{max}^{b}\cdot\sum_{v\in V}\pi(s,v)\\&=r_{max}^{b}\end{aligned}$$

而在 Backward Search 算法的时间及空间开销上，由于对于所有节点 $\mathrm{t}\in\mathrm{V}$，我们都只存储了$\pi^{b}(\mathrm{v},\mathrm{t})\geq r_{max}^{b}$的节点 v，而我们知道$\sum_{v\in V}\pi^{b}(v,t)=1$，故每个节点 t 最多只需要存储$\frac{1}{r_{max}^{b}}$个元组，因此算法的预处理空间开销为 $\mathrm{O}(\frac{n}{r_{max}^{b}})$的。而由于一次 Backward Search 的均摊时间复杂度是 $\mathrm{O}(\frac{m}{n\cdot r_{max}^{b}})$的，因此算法的预处理时间开销是 $\mathrm{O}(\frac{m}{r_{max}^{b}})$的。

此外，2.2.4 中指出了 Backward Search 的缺陷导致其只能作用于无向图上，因此以其为基础的 All-Pair-Backward-Search 算法也只能作用于无向图上。

4.2 实现过程

在 3.2.1 及 3.2.3 中，我们讲述了 Neo4j 的数据导入方法及其 Java API 的安装使用方法。以此作为参考，我首先安装配置了 Neo4j 的运行环境并在 Java 工程中导入了相应的 JAR 文件，然后依照 Neo4j-admin Import 方法对 csv 文件的格式要求，编写格式转换程序自动地将存储了图节点和边信息的 txt 格式文件转换为符合条件的 csv 格式文件，并利用 Neo4j-admin Import 方法将其导入到 Neo4j 数据库中。以此为基础，我使用 Java 语言实现了 Neo4j 图数据库上的 Power Iterations、Monte-Carlo、Forward Push、Backward Search、FORA Single-Source PPR、FORA Top-k PPR、All-Pair-Backward-Search 以及 Neo4j Single-Source PPR 算法。在这一章节中，我将对工程中的各个模块进行介绍。同时，为了使读者对代码结构有一个更直观的认识，我还在附录中附加了本毕业设计的 UML 类图以及对各模块的详细介绍。

4.2.1 接口模块

针对 Single-Source PPR、Top-k PPR 以及 All-Pair PPR 三类 PPR 算法，我分别设计了三个接口来描述各类 PPR 算法所需要实现的功能。借助接口我们得以对各类算法的行为进行统一，从而使得通用的算法使用成为可能。

4.2.1.1 Whole_Graph_Util_Interface.interface

作为 Single-Source PPR 算法所需要实现的接口，Whole_Graph_Util_Interface 接口共定义了三个方法，它们的功能分别为对给定源节点的 Single-Source PPR 进行计算、获得结果以及打印结果。实现本接口的算法有 Power Iterations、Monte-Carlo、Forward Push、Neo4j Single-Source PPR、FORA Single-Source PPR 以及 All-Pair-Backward-Search 算法。

4.2.1.2 Topk_Util_Interface.interface

作为 Top-k PPR 算法所需要实现的接口，Topk_Util_Interface 接口共定义了三个方法，它们的功能分别为对给定源节点的 Top-k PPR 进行计算、打印结果以及获得 Top-k 节点的集合。实现本接口的算法有 Power Iterations、Monte-Carlo、Forward Push、Neo4j Single-Source PPR、FORA Top-k PPR 以及 All-Pair-Backward-Search 算法。

4.2.1.3 Preprocessing_Interface.interface

作为通过预处理计算 All-Pair PPR 的算法所需要实现的接口，Preprocessing_Interface 接口共定义了四个方法，它们的功能分别为计算 All-Pair PPR、读取给定源节点的预处理 PPR 结果、获取预处理文件大小以及删除预处理文件。实现本接口的算法有 Monte-Carlo、Forward Push、Neo4j Single-Source PPR、FORA Single-Source PPR 以及 All-Pair-Backward-Search 算法。

4.2.2 辅助类模块

我们设计了若干服务于各 PPR 算法类的辅助类，它们所定义的方法为 PPR 计算以及性能测试提供了便利。

4.2.2.1 Algo_Conf.class

Algo_Conf 类用于为各个 PPR 算法配置参数并新建相应类的对象。对于不同的算法我们需要对各自特定的参数进行赋值，为了方便起见，我们设计了此类用于设置个算法的参数并且通过调用相应类的构造函数新建对象。

4.2.2.2 Algo_Util.class

Algo_Util 类被各个 PPR 算法类继承，其中定义了若干供 PPR 计算使用的实用方法，包括获取节点名、计算第 k 大的 PPR 值等，这些方法对于调试以及 Top-k PPR 计算都有很大帮助。

4.2.2.3 Gen_Util.class

Gen_Util 类是为我们的性能测试服务的，它所定义的方法功能包括随机生成源节点数组、

计算 PPR 结果误差、新建 PPR 算法类的对象、进行单独或者批量的 PPR 测试等。Gen_Util 类提高了我们的实验效率，使得性能测试的批量操作更加简便。

4.2.2.4 PPR.class

PPR 类是我们的主类，它继承了 Gen_Util 类，其定义的方法功能包括创建和关闭数据库、加载图、设置实验的基本参数等。此外，我们在其主方法中创建数据库，然后从数据库中加载图，并对算法进行批量的性能测试。

4.2.3 算法类模块

本模块中的算法类是本毕业设计的核心，它们各自实现了计算 Single-Source PPR、Top-k PPR 或者 All-Pair PPR 的算法。同时由于这些类都实现了相应的接口，因此我可以很方便地针对它们进行统一的性能测试。

4.2.3.1 Power_Method.class

Power_Method 类中实现了计算 PPR 精确值的 Power Iterations 算法，其定义的方法功能包括计算 Single-Source PPR 或 Top-k PPR 的准确结果等，在性能测试中该结果作为参考值被用于计算其他 PPR 算法的结果误差。值得一提的是，其 Top-k PPR 算法本质上是在计算完 Single-Source PPR 结果后截取了前 k 大的结果。

4.2.3.2 Monte_Carlo.class

Monte_Carlo 类中实现了 Monte-Carlo 算法，其定义的方法功能包括对 All-Pair PPR 结果进行预处理、计算 Single-Source PPR、计算 Top-k PPR 等，其中 All-Pair PPR 算法其实是依次以每个节点作为源节点计算 Single-Source PPR 算法并将结果记入文件中，而 Top-k PPR 算法则是在计算完 Single-Source PPR 结果后截取了前 k 大的结果。

4.2.3.3 Forward_Push.class

Forward_Push 类中实现了 Forward Push 算法，其定义的方法功能包括对 All-Pair PPR 结果进行预处理、计算 Single-Source PPR、计算 Top-k PPR 等。与 Monte_Carlo 类相似地，其 All-Pair PPR 结果也是通过依次调用 Single-Source PPR 算法得到的，而 Top-k PPR 算法也是在计算完 Single-Source PPR 结果后截取了前 k 大的结果。

4.2.3.4 Backward_Search.class

Backward_Search 类中实现了 Backward Search 算法，其定义的方法功能包括计算和返回 Single-Source PPR 结果，它将作为 All-Pair-Backward-Search 算法的组成部分用于计算 PPR 估计值。

4.2.3.5 Fora_Whole_Graph.class

Fora_Whole_Graph 类中实现了 FORA Single-Source PPR 算法，其定义的方法功能包括计算 Single-Source PPR、对 All-Pair PPR 结果进行预处理等。其中在 Single-Source PPR 算法中它调用了 Monte_Carlo 类以及 Forward_Push 类的方法作为 FORA 算法的组成部分，而其 All-Pair PPR 算法也是通过依次调用 Single-Source PPR 算法来实现的。

4.2.3.6 Fora_Topk.class

Fora_Topk 类中实现了 FORA Top-k PPR 算法，其定义的方法功能包括计算、返回和打印 Top-k PPR 估计值结果，它同样也调用了 Monte_Carlo 类以及 Forward_Push 类的方法作为 FORA Top-k 算法的组成部分。

4.2.3.7 Base_Whole_Graph.class

Base_Whole_Graph 类中实现了 All-Pair-Backward-Search 算法，其定义的方法功能包括计算 Single-Source PPR 和 Top-k PPR、对 All-Pair PPR 结果进行预处理等。其中在预处理算法中，它通过依次以各节点为目标节点调用 Backward Search 算法得到 All-Pair PPR 估计值，并以倒排表的形式将结果记入文件中。其 Single-Source PPR 算法就是以读文件的形式得到各节点相对于给

定源节点的 PPR 估计值，而 Top-k PPR 算法的不同点就在于，在它的预处理结束后只有以各节点为源节点的前 k 大的 PPR 估计值被存入了文件中。

4.2.3.8 Neo4j_Method.class

Neo4j_Method 类中实现了 Neo4j Single-Source PPR 算法，其定义的方法功能包括对 All-Pair PPR 结果进行预处理、计算 Single-Source PPR、计算 Top-k PPR 等。其中在其 Single-Source PPR 的实现中，我们调用了 Neo4j Graph Algorithm 库提供的 PPR 计算方法并且对其返回的结果进行了归一化操作。另外，与 Monte-Carlo 算法类似地，其预处理算法以及 Top-k PPR 算法本质上都是通过调用 Single-Source PPR 算法来实现的。

5 实验结果

这一章节将会针对各个 PPR 算法进行实验来测试其性能，共有以下三种实验：预处理、Single-Source PPR 以及 Top-k PPR 实验。我们将首先介绍实验设置，然后展示各实验的算法性能测试结果并进行分析，最后将会讲解图加载的开销问题。

5.1 实验设置

5.1.1 实验环境

具体的实验环境情况详见表 2。

表 2 实验环境

环境	属性
CPU	8 核；2094.865MHz
内存	500GB
操作系统	Ubuntu 14.04.1 LTS
JDK	1.8.0
Neo4j	3.5.1

5.1.2 数据集

在 4.1.3 中我们指出了目前 All-Pair-Backward-Search 算法只能在无向图上计算 All-Pair PPR 估计值，因此本实验选取了四个无向图作为我们的数据集，具体情况详见表 3。

表 3 数据集介绍

名称	节点数	边数	数据来源
GRQC	5242	28968	http://networkrepository.com/ca-GrQc.php

BlogCatalog	10312	667966	http://socialcomputing.asu.edu/datasets/BlogCatalog3
Flickr	80513	11799764	http://socialcomputing.asu.edu/datasets/Flickr
Com-Amazon	334863	1851745	https://snap.stanford.edu/data/com-Amazon.html

5.1.3 实验方法

在 3.2.3 中，我们发现 Neo4j Single-Source PPR 算法是先从数据库中将图加载出来，然后再计算节点的 PPR 估计值。在我们的实验中，我们也采用相同的方法，在开始时先利用 Neo4j 提供的 GraphLoader.load()函数加载 HeavyGraph 类型的图，然后基于它进行各个算法的性能测试。在本实验中，Neo4j Single-Source PPR 算法只进行图节点 PPR 估计值的计算，而其图加载部分已经被我们独立出来成为所有 PPR 算法开始前的准备步骤。对于图加载的开销问题我们会在 5.5 中进行进一步的探讨。

本实验将会分为预处理、Single-Source PPR 以及 Top-k PPR 实验三部分进行，每次实验中我们都会随机选取 50 个节点作为源节点运行算法，它们的性能指标平均值将会作为各算法的性能指标。同时，各算法的性能又取决于我们所给定的误差参数，通过调整误差参数值，我们即可研究在不同的误差要求的情况下各算法的性能表现。各算法的误差参数各不相同，其中 Forward Push 与 All-Pair-Backward-Search 算法的误差参数均为 residue 阈值r_{max}，Monte-Carlo、FORA Single-Source PPR 以及 FORA Top-k PPR 算法的误差参数则为误差阈值ϵ，Neo4j Single-Source PPR 算法的误差参数为迭代次数 iterations。我们将会分别在 5.1.4 和 5.1.5 中介绍算法的性能指标以及误差参数的具体设置。

在预处理实验部分，我们对 Forward Push、Monte-Carlo、FORA Single-Source PPR、All-Pair-Backward-Search 以及 Neo4j Single-Source PPR 算法进行实验，其中除了 All-Pair-Backward-Search 算法外，其余算法的预处理方法均是依次以所有节点为源节点运行 Single-Source PPR 算法得到所有节点对之间的 PPR 估计值。我们会将 PPR 估计值非零的结果存入文件中，当我们进行以某个节点 s 作为源节点的 Single-Source PPR 查询时，我们只需要读相应的文件即可获得 PPR 估计值结果，而不存在于文件中的 PPR 值我们则将其视为 0。由于各算法进行 Single-Source PPR 查询的运行时间开销相差无几，因此在这部分，我们将以各算法的预处理时空开销以及最大绝对误差作为性能指标。

在 Single-Source PPR 实验部分，我们对 Forward Push、Monte-Carlo、FORA Single-Source PPR、All-Pair-Backward-Search 以及 Neo4j Single-Source PPR 算法进行实验，其中比较特殊的是只有 All-Pair-Backward-Search 算法会进行预处理。而在这一部分，我们将以各算法查询 Single-Source PPR 的运行时间开销及最大绝对误差作为性能指标。

在 Top-k PPR 实验部分，我们对 Forward Push、Monte-Carlo、FORA Top-k PPR、All-Pair-Backward-Search 以及 Neo4j Single-Source PPR 算法进行实验，其中除了 FORA Top-k 算法外，其余算法计算 Top-k PPR 的方法实质上都是利用 Single-Source PPR 算法得到结果后再截取其中的 Top-k 结果。在这一部分，我们将以各算法查询 Top-k PPR 的运行时间开销、准确率以及归一化折损累计增益作为性能指标。

5.1.4 性能指标

为了衡量 PPR 算法的性能优劣，我们将以预处理时间开销、预处理空间开销、运行时间开销以及结果误差作为算法的性能指标。

5.1.4.1 预处理时间开销

给定算法类型与误差参数，我们将利用该算法计算 All-Pair PPR 估计值以及将结果存入文件的时间作为预处理时间开销。

5.1.4.2 预处理空间开销

给定算法类型与误差参数，我们将利用该算法进行预处理得到的 All-Pair PPR 估计值结果所占的文件系统空间作为预处理空间开销。

5.1.4.3 运行时间开销

给定算法类型、误差参数以及源节点，我们可以测量利用该算法对源节点进行的 Single-Source PPR 或 Top-k PPR 查询的时间，对于多个源节点依次进行实验后我们取查询时间的平均值作为算法的运行时间开销。

5.1.4.4 结果误差

在性能测试中，我们采用 Power Iterations 算法计算 PPR 精确值，并以此为基准计算各算法 PPR 估计值的误差。

对于预处理实验以及 Single-Source PPR 实验，我们采用最大绝对误差的平均值作为结果误差的指标。给定源节点 s，对于所有$v \in V$，我们将其相对于源节点 s 的 PPR 精确值和估计值分别记作$\pi(s, v)$和$\hat{\pi}(s, v)$，那么最大绝对误差为

$$err_{max} = \max_{v \in V} |\pi(s, v) - \hat{\pi}(s, v)|$$

而对于 Top-k PPR 实验，我们采用准确率（Precision）和归一化折损累计增益（Normalized Discounted cumulative gain）的平均值作为结果误差的指标，其中准确率表示该算法对 Top-k 节点预测正确的比率，而归一化折损累计增益则是一个基于预测结果的关联度以及正确性的归一化指标，其思想是高关联度的结果比一般关联度的结果更影响最终的指标，且有高关联度的结果出现在更靠前的位置的时候，指标会越高。给定 k 及源节点 s，我们将 Power Iterations 算法计算出来的 Top-k 节点集合记作$S_{truth} = \{v_1, \cdots, v_k\}$，而待测试的 Top-k PPR 算法计算出来的 Top-k 节点集合记作$S_{algo} = \{v_1', \cdots, v_k'\}$。对于所有$v \in S_{truth}$，我们将其相对于源节点 s 的 PPR 精确值记作$\pi(s, v)$。那么准确率为

$$\text{precision} = \frac{|S_{truth} \cap S_{algo}|}{k}$$

而归一化折损累计增益则为

$$\text{NDCG} = \frac{1}{Z_k} \sum_{i=1}^{k} \frac{2^{\pi(s, v_i')} - 1}{\log(i+1)}$$

其中

$$Z_k = \sum_{i=1}^{k} \frac{2^{\pi(s, v_i)} - 1}{\log(i+1)}$$

5.1.5 参数设置

在本次实验中，为了使得各算法结果误差的区间尽可能地相近，同时体现出各算法的性能指标关于误差参数的变化趋势，我们将分别按照表 4、表 5 和表 6 中的相应数值对预处理、Single-Source PPR 以及 Top-k PPR 实验的误差参数依次进行设置。在 Top-k PPR 实验中，我们将选取

k=50 进行实验。

表 4 预处理实验参数设置

算法	误差参数	GRQC	BlogCatalog	Flickr	Com-Amazon
Forward Push	r_{max}	1E-4, 1E-5, 1E-6, 5E-7, 1E-7	1E-6, 7E-7, 5E-7, 3E-7, 1E-7	5E-5, 1E-5, 5E-6, 1E-6, 7E-7	1E-5, 5E-5, 1E-6, 5E-7, 3E-7
Monte-Carlo	ϵ	1.0, 0.5, 0.3, 0.2, 0.1	5.0, 1.0, 0.7, 0.5, 0.3	20, 10, 7, 5, 3	50, 20, 10, 7, 5
FORA Single-Source PPR	ϵ	10, 5, 0.5, 0.3, 0.1	50.0, 10.0, 5.0, 1.0, 0.5	500, 200, 100, 20, 10	500, 200, 70, 50, 30
All-Pair-Backward-Search	r_{max}	1E-3, 5E-4, 5E-5, 1E-6, 5E-7	1E-3, 7E-4, 5E-4, 1E-4, 5E-5	5E-3, 1E-3, 5E-4, 1E-4, 7E-5	1E-4, 5E-5, 4E-5, 3E-5, 2E-5
Neo4j Single-Source PPR	iterations	5, 40, 100, 200, 300	1, 5, 10, 40, 100	1, 5, 10, 40, 100	1, 3, 5

表 5 Single-Source PPR 实验参数设置

算法	误差参数	GRQC	BlogCatalog	Flickr	Com-Amazon
Forward Push	r_{max}	1E-4, 1E-5, 1E-6, 1E-7, 1E-8	1E-6, 5E-7, 1E-7, 7E-8, 5E-8	5E-5, 1E-6, 5E-8, 1E-8, 5E-9	1E-6, 5E-7, 3E-7, 5E-8, 1E-8
Monte-Carlo	ϵ	1.0, 0.5, 0.3, 0.1, 0.05	5.0, 1.0, 0.5, 0.3, 0.1	10.0, 5.0, 1.0, 0.5, 0.3	1.0, 0.7, 0.5, 0.3, 0.1
FORA Single-Source PPR	ϵ	10.0, 5.0, 0.5, 0.1, 0.05	50.0, 10.0, 5.0, 1.0, 0.5	500.0, 50.0, 5.0, 1.0, 0.5	50, 10, 5, 3, 1
All-Pair-	r_{max}	1E-3, 5E-4,	1E-3, 5E-4,	5E-3, 1E-3,	1E-4, 5E-5,

Backward-Search		5E-5, 1E-6, 5E-7	1E-4, 5E-5, 7E-4	5E-4, 1E-4, 7E-5	4E-5, 3E-5, 2E-5
Neo4j Single-Source PPR	iterations	5, 40, 100, 200, 300	1, 5, 10, 40, 100	1, 5, 10, 40, 100	1, 5, 10, 40, 100

表 6 Top-k PPR 实验参数设置

算法	误差参数	GRQC	BlogCatalog	Flickr	Com-Amazon
Forward Push	r_{max}	1E-4, 1E-6, 5E-8, 7E-9, 7E-10	1E-6, 5E-7, 1E-7, 5E-8, 1E-8	5E-7, 1E-7, 5E-8, 1E-8, 5E-9	5E-5, 1E-5, 5E-6, 1E-6, 5E-7
Monte-Carlo	ϵ	3.0, 1.0, 0.2, 0.1, 0.05	5.0, 1.0, 0.5, 0.1, 0.05	5.0, 1.0, 0.3, 0.1, 0.05	10.0, 5.0, 1.0, 0.5, 0.3
FORA Top-k PPR	ϵ	10.0, 0.5, 0.1, 0.01, 0.001	1.0, 0.5, 0.1, 0.05, 0.01	50.0, 10.0, 1.0, 0.1, 0.05	50.0, 10.0, 5.0, 1.0, 0.5
All-Pair-Backward-Search	r_{max}	1E-3, 5E-4, 5E-5, 1E-7, 5E-8	1E-3, 7E-4, 5E-4, 1E-4, 5E-5	5E-4, 3E-4, 1E-4, 7E-5, 5E-5	1E-3, 5E-4, 1E-4, 5E-5
Neo4j Single-Source PPR	iterations	5, 40, 300, 500, 1000	1, 5, 10, 40, 100	5, 10, 40, 100, 200	5, 10, 100, 200, 300

5.2 预处理实验结果

在预处理实验中，各算法的预处理时间开销及平均最大绝对误差关于误差参数的变化情况如表 7 至表 10 所示。其中 Neo4j Single-Source PPR 算法的平均最大绝对误差始终是远大于其他算法的，因此该算法并不适用于图节点 PPR 的预处理，在下面的分析中我们也不再讨论它。实际上，在 5.3 中我们将会看到，该算法也不适用于 Single-Source PPR 查询。

我们可以看到，在同样的平均最大绝对误差下，All-Pair-Backward-Search 算法的预处理速度几乎总是快于其他算法。

表 7 GRQC 数据集预处理时间开销

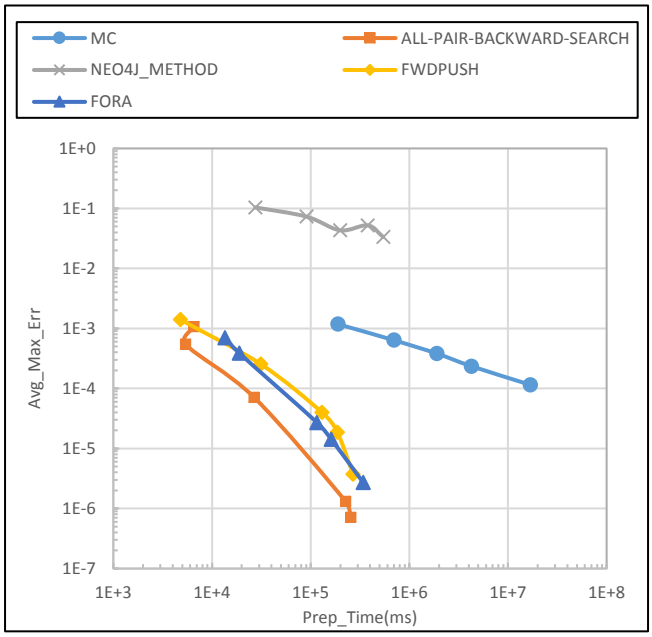


表 8 BlogCatalog 数据集预处理时间开销

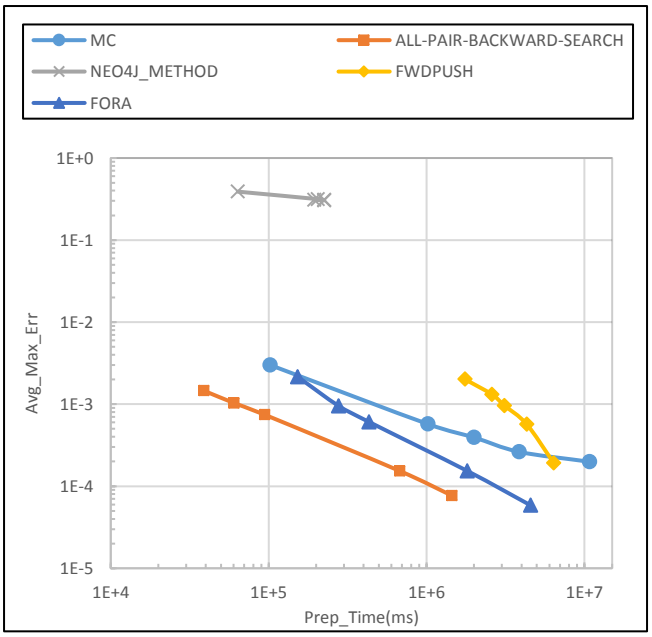


表 9 Flickr 数据集预处理时间开销

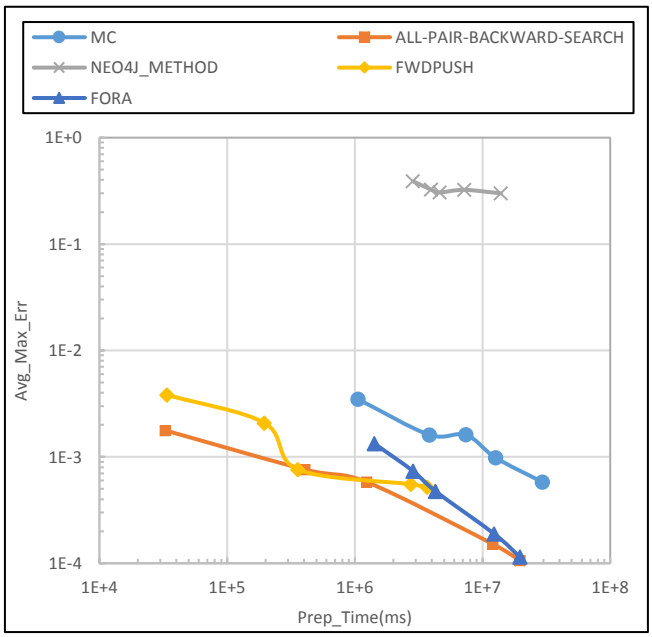
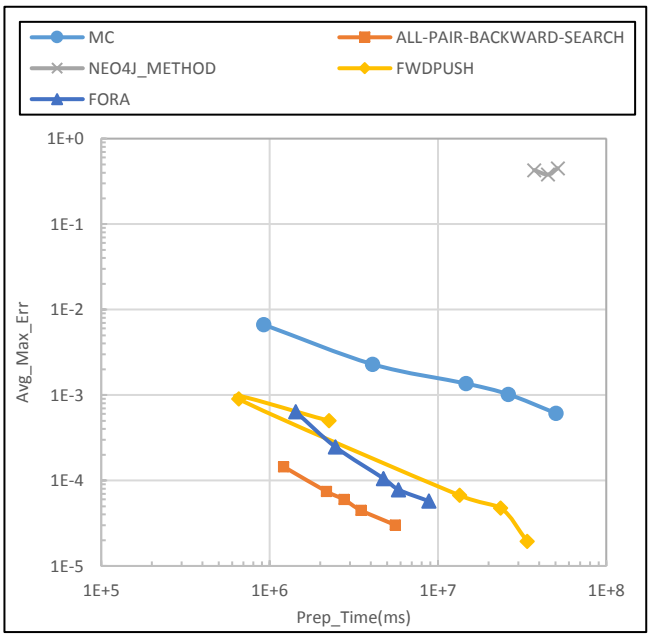


表 10 Com-Amazon 数据集预处理时间开销



讨论完各算法的预处理时间开销，我们接下来继续观察各算法的预处理空间开销及平均最大绝对误差关于误差参数的变化情况。如表 11 至表 14 所示，在同样的平均最大绝对误差下，All-Pair-Backward-Search 算法的预处理空间开销是远小于其他算法的。

综上所述，利用 All-Pair-Backward-Search 算法进行图节点 PPR 预处理可以在消耗相对较少的文件系统空间的同时以相对较快的速度完成预处理工作。它在时空开销上的优异表现要归功

于其误差参数r_{max}：在对各节点进行 Backward Search 时，以r_{max}作为 residue 阈值的 Backward Search 算法相较于其他算法可以较早地运行完毕；而将 PPR 预处理结果存入文件时，我们只将 reserve 值大于r_{max}的结果存入文件中，从而也节省了空间开销。由此可见，在允许对图节点 PPR 进行预处理的情形下，All-Pair-Backward-Search 算法是最适合的预处理算法。

表 11 GRQC 数据集预处理空间开销

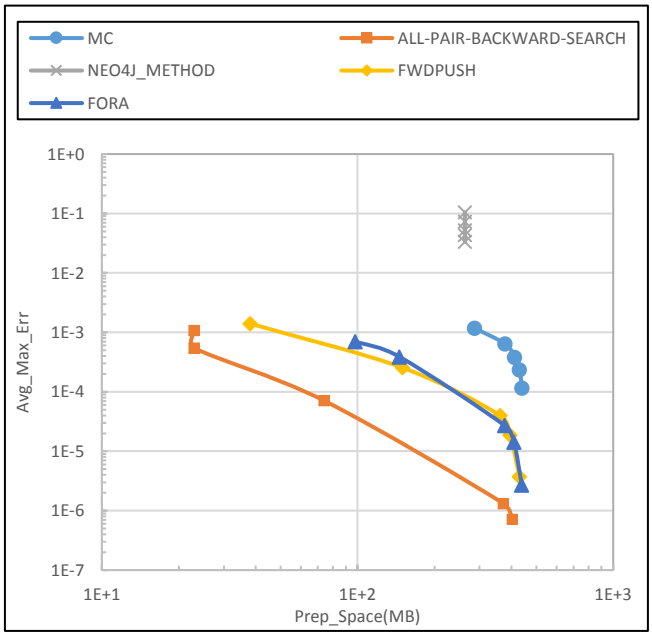


表 12 BlogCatalog 数据集预处理空间开销

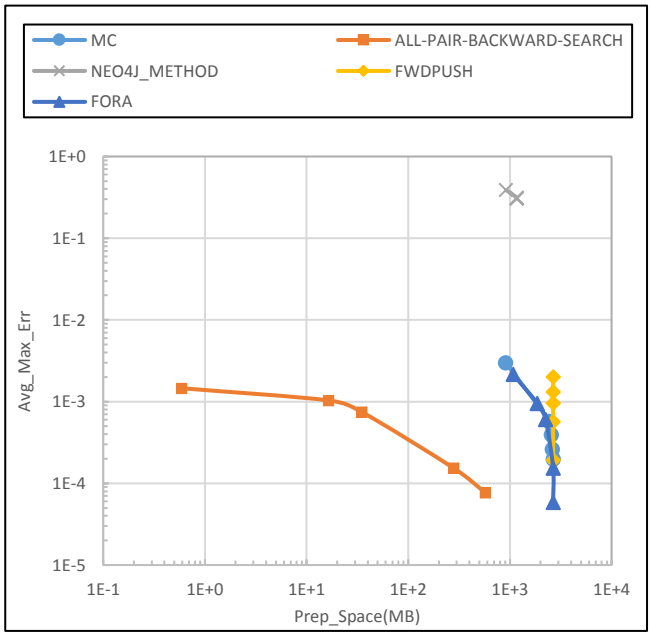


表 13 Flickr 数据集预处理空间开销

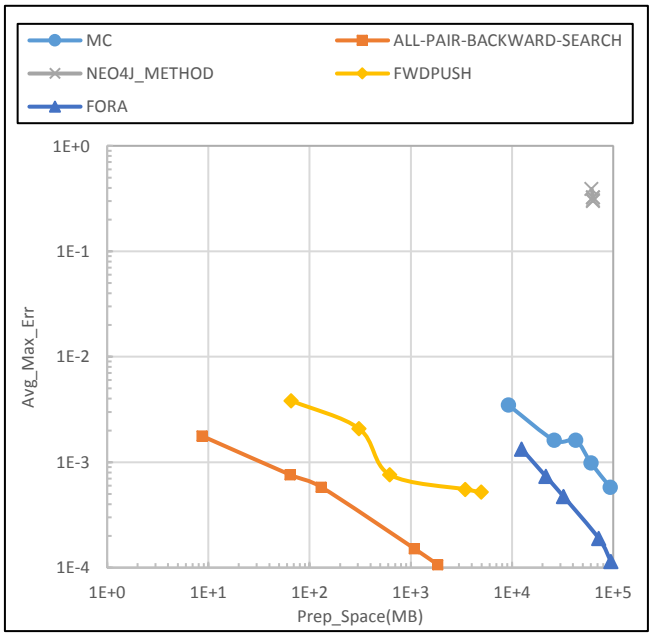
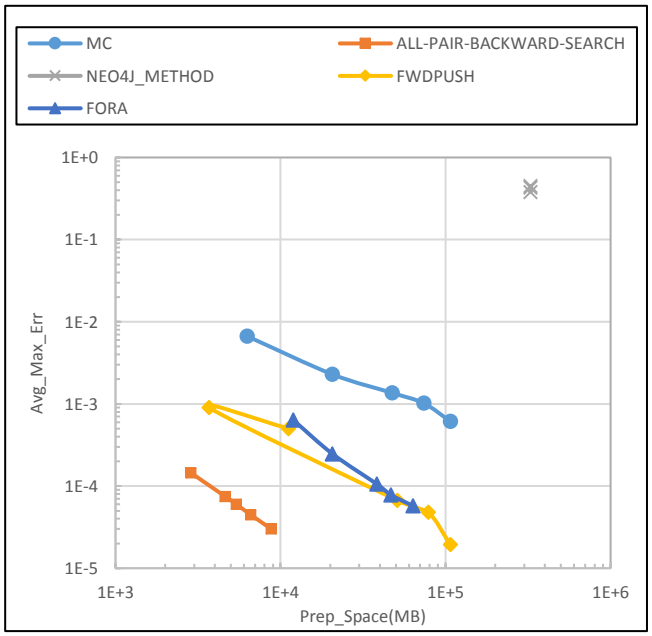


表 14 Com-Amazon 数据集预处理空间开销



5.3 Single-Source PPR 实验结果

在 Single-Source PPR 实验中，各算法的平均运行时间开销及平均最大绝对误差关于误差参数的变化情况如表 15 至表 18 所示。从图中我们可以看到，Neo4j Single-Source PPR 算法在平均最大绝对误差上的表现依然不佳。这是由于 Neo4j Single-Source PPR 算法可能是致力于得到所有节点 PPR 值的相对大小关系，但是不要求 PPR 取值严格遵循 PPR 定义的。因而若我们利用其计算一个节点和边的数目较多的图的 PPR 估计值，那么我们将其与按照 PPR 定义得到的精确值比较得出的绝对误差也自然会很大。

但与此同时，参考 5.1.4 中 Top-k PPR 实验的两个结果误差指标准确率以及归一化折损累计增益的定义，我们可以看出 Top-k 节点的 PPR 估计值是否严格遵循 PPR 定义对于这两项指标的影响并不大，大体来说只需要算法得到的 Top-k 节点的相对大小关系与实际相符即可。因此，Neo4j Single-Source PPR 算法可能适用于计算 Top-k PPR，而在 5.4 中的实验结果也印证了这一猜想。

接下来我们将对除 Neo4j Single-Source PPR 算法以外的各算法结果进行进一步的讨论。在这些算法中，比较特殊的是 All-Pair-Backward-Search 算法，它已经通过预处理将 PPR 估计值结果存储到文件中，因此其 Single-Source PPR 查询的运行时间即为其读文件的时间开销。而其他算法则均是在进行 Single-Source PPR 查询时运行算法得到所有节点的 PPR 估计值。

从表中我们可以看到，All-Pair-Backward-Search 算法的运行时间几乎不随着其误差参数r_{max}的改变而改变。虽然由 5.2 我们可以看到预处理文件的大小会随着误差参数r_{max}的减小而增大，但是因为文件访问的大部分时间是耗费在磁盘寻址的，因而算法的运行时间基本不变。因此，当平均最大绝对误差较小时，All-Pair-Backward-Search 算法的运行时间会显著低于其他算法。然而，当平均最大绝对误差较大时，由于其他算法的计算都是在内存中进行的，因而没有磁盘访问的时间开销，此时 All-Pair-Backward-Search 算法就不是运行速度最快的算法了。

另外需要指出的是，在 All-Pair-Backward-Search 算法的实验过程中，我发现当误差参数r_{max}减小到一定程度时，算法运行时会出现内存不足的问题，这是由于我们目前的实现是在 All-Pair PPR 计算结束后才统一地将结果写入文件中。为了解决这一问题，我们在将来的实现中需要每隔一段时间将中间结果写入文件中，从而释放内存空间供进一步的 PPR 计算使用。

除去 All-Pair-Backward-Search 算法，若我们只关注无预处理的 Single-Source PPR 算法，我们可以看到 FORA 算法的表现十分优异，尤其是在节点和边数目较多的图中，它在同样的平均最大绝对误差下的运行速度要快于其他算法。

综上所述，在节点和边的数目较多且允许预处理的情形下，All-Pair-Backward-Search 算法更适合 Single-Source PPR 查询；否则，FORA 算法更符合使用需求。

表 15 GRQC 数据集 Single-Source PPR 运行时间开销

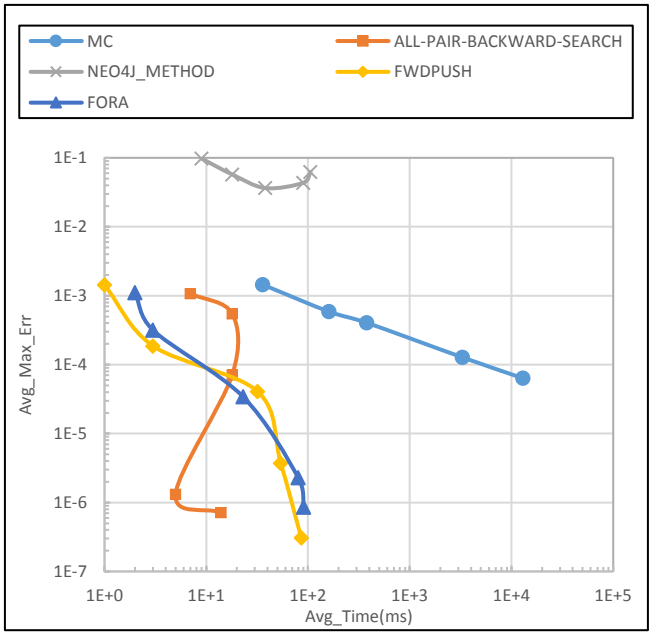


表 16 BlogCatalog 数据集 Single-Source PPR 运行时间开销

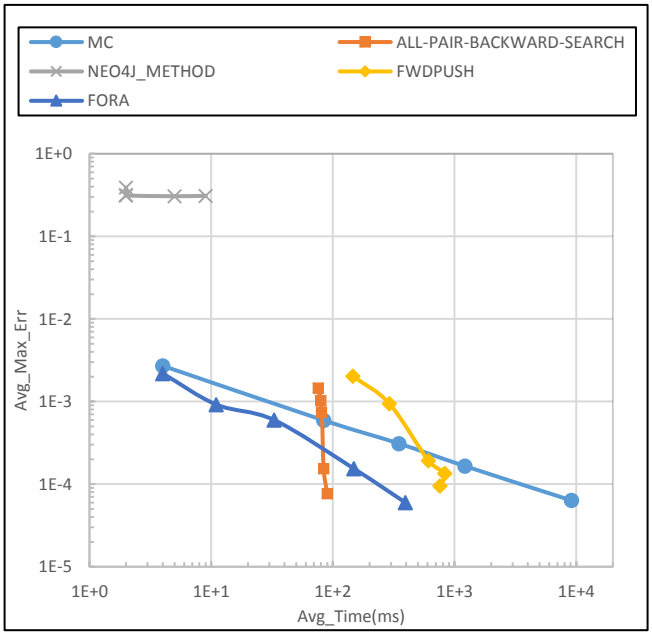


表 17 Flickr 数据集 Single-Source PPR 运行时间开销

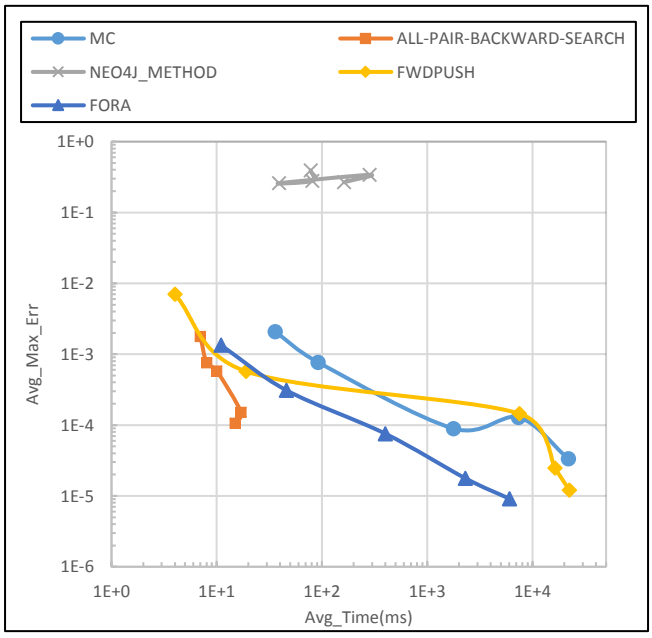
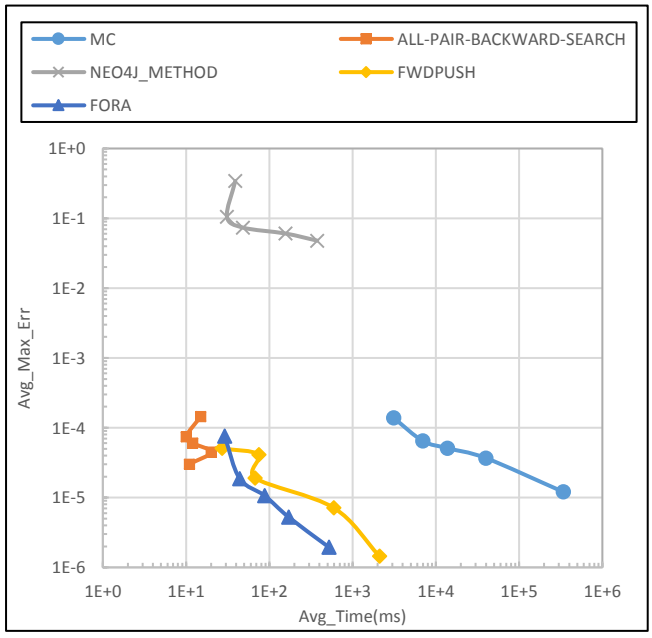


表 18 Com-Amazon 数据集 Single-Source PPR 运行时间开销



5.4 Top-k PPR 实验结果

表 19 至表 22 和表 23 至表 26 分别展示了 Top-k PPR 实验中各算法的平均运行时间开销与准确度及归一化折损累计增益的平均值关于误差参数的变化情况。其中我们可以看到 Neo4j Single-Source PPR 算法在四个数据集中的平均准确度均可以达到 0.65 以上，平均归一化折损累计增益则是均可以达到 0.955 以上，虽然与其他算法的误差精度仍有差距，但是较之其在 5.2 和 5.3 实验中的表现而言已经是非常大的性能提升了。

与 5.3 中 Single-Source PPR 实验的情况类似的，All-Pair-Backward-Search 算法的平均运行时间几乎不随误差参数r_{max}的变化而变化，总体而言其速度都是快于其他算法的。

而在无预处理的 Single-Source PPR 算法中，Monte-Carlo 算法的平均运行时间开销随着误差参数ϵ的减小而迅速增大，当平均准确度或平均归一化折损累计增益的数值接近 1 时运行时间的增长速度尤其迅猛。另外，我们可以看到，在同样的平均准确度或平均归一化折损累计增益下，FORA Top-k 算法的运行速度总体是比 Forward Push 算法快的，特别地，FORA Top-k 算法可以在 1 秒内达到平均准确度 0.93 以上或平均归一化折损累计增益 0.997 以上的水平。

综上所述，在允许预处理的情形下，我们可以采用 All-Pair-Backward-Search 算法计算 Top-k PPR 估计值；否则，我们可以采用 FORA Top-k 算法进行计算。

表 19 GRQC 数据集 Top-k PPR 运行时间开销

（以平均准确度为基准）

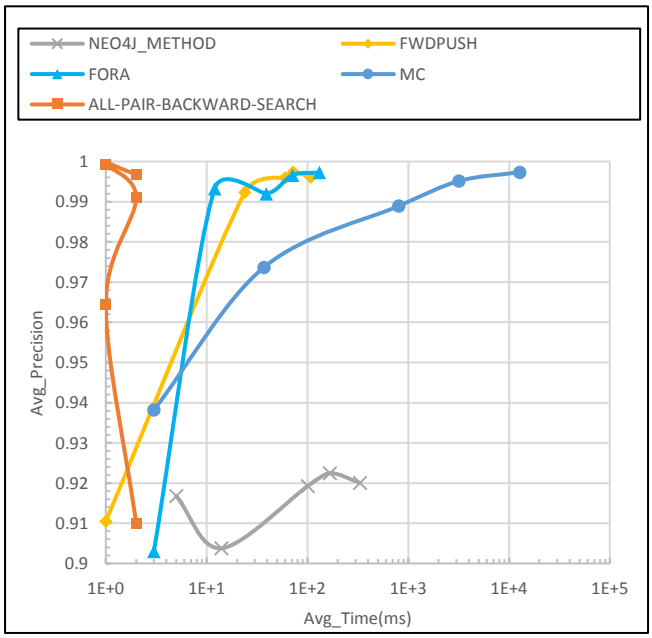


表 20 BlogCatalog 数据集 Top-k PPR 运行时间开销

（以平均准确度为基准）

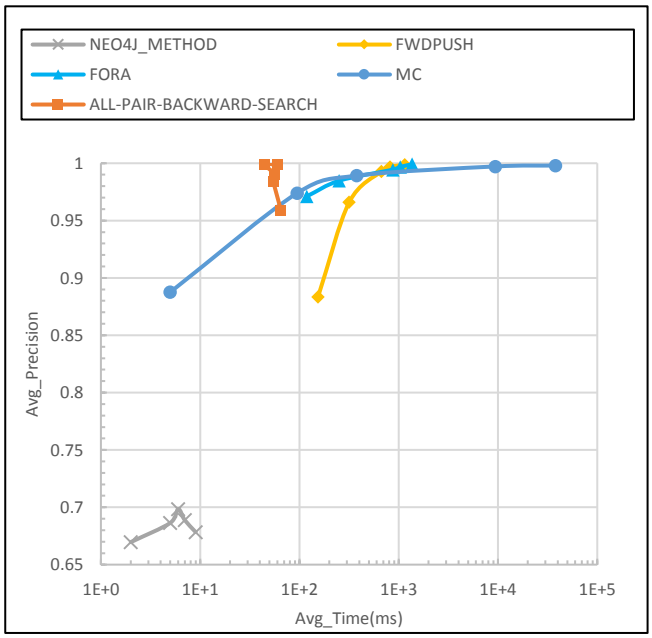


表 21 Flickr 数据集 Top-k PPR 运行时间开销

（以平均准确度为基准）

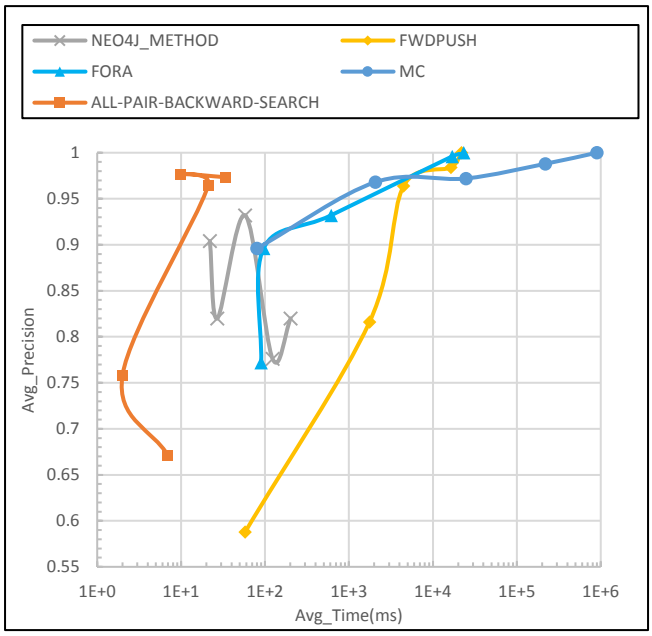


表 22 Com-Amazon 数据集 Top-k PPR 运行时间开销

（以平均准确度为基准）

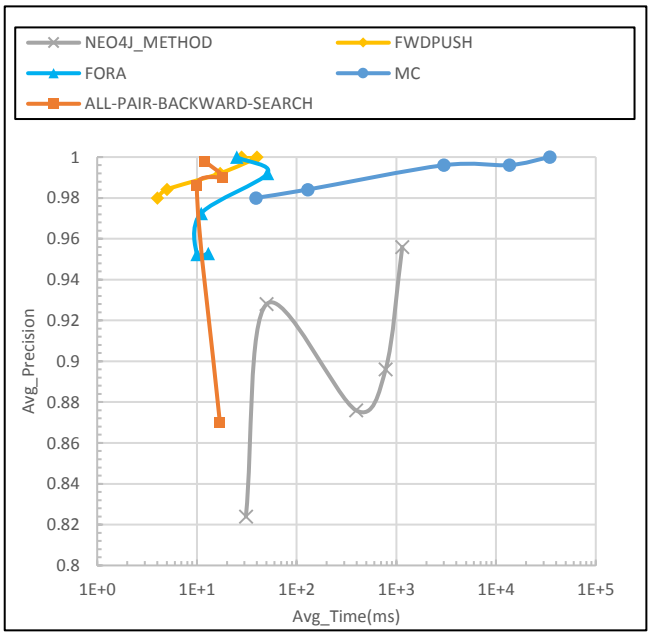


表 23 GRQC 数据集 Top-k PPR 运行时间开销

（以平均归一化折损累计增益为基准）

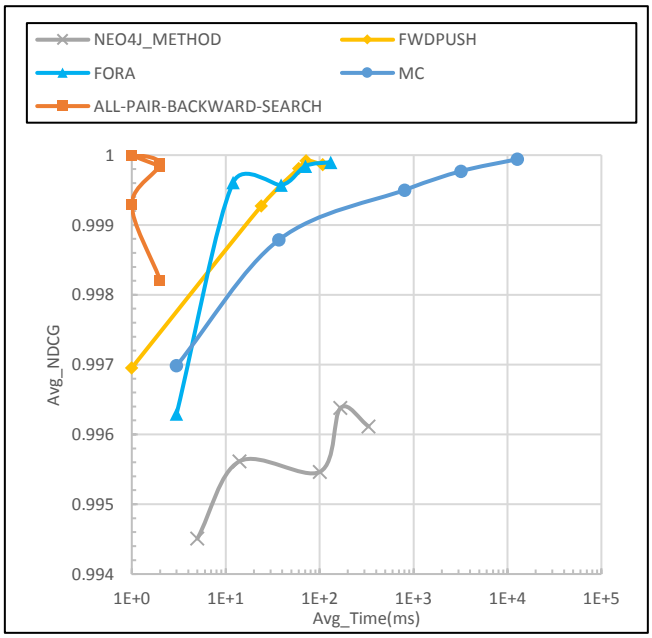


表 24 BlogCatalog 数据集 Top-k PPR 运行时间开销

（以平均归一化折损累计增益为基准）

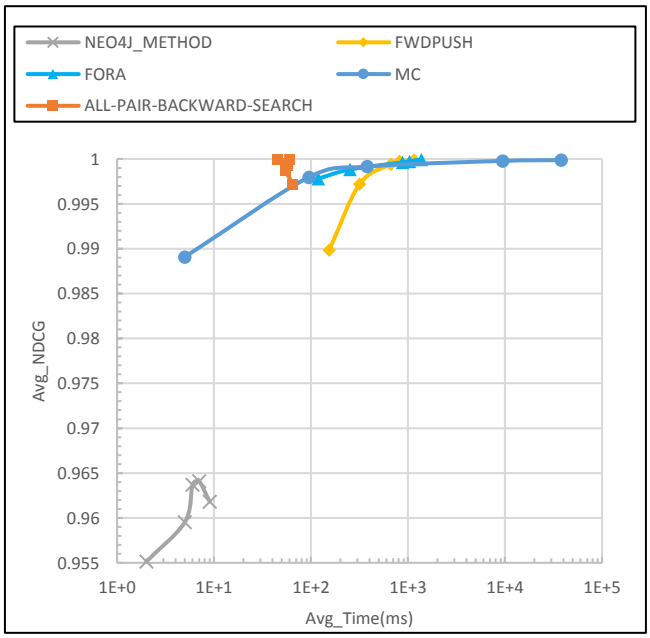


表 25 Flickr 数据集 Top-k PPR 运行时间开销

（以平均归一化折损累计增益为基准）

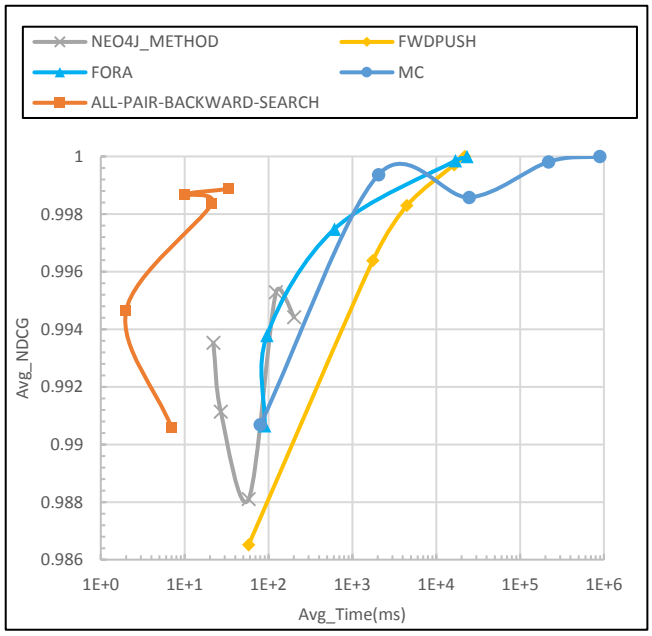
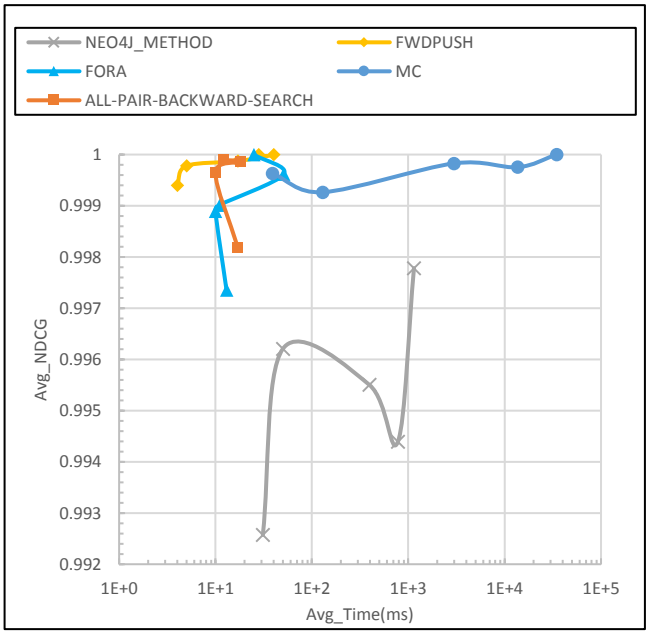


表 26 Com-Amazon 数据集 Top-k PPR 运行时间开销

（以平均归一化折损累计增益为基准）



5.5 图加载开销

在我们所实现的算法的最初版本中，我们将图导入到 Neo4j 数据库中，在计算图节点 PPR 估计值时则通过创建事务与数据库交互的方式来获取节点或边的信息。PPR 算法中需要频繁获取节点或边的信息，这样就会导致数据库交互的时间开销非常大。此外，我们在 3.2.3 中也分析过，只凭借数据库交互的手段是不能够实现 O(1)时间复杂度的随机游走的。

因此我们参考 Neo4j Single-Source PPR 算法中采用的方式，先将图从数据库中加载出来并存储在类型为 HeavyGraph 的对象中，然后再运行 PPR 算法。对于空间开销，我们在 3.2.3 中就分析过，其不定长的二维数组结构有利于节省稀疏图的存储空间，使得我们可以加载节点和边的数目较大的图到内存中。而对于时间开销，各个数据集的图加载平均时间开销如表 27 所示，在实际的应用情景下，我们常常是进行图加载后进行多次 PPR 查询，因此总体而言，从 Neo4j 数据库中加载图的时间开销是比较小的。

表 27 图加载平均时间开销

数据集	GRQC	BlogCatalog	Flickr	Com-Amazon
图加载时间	786ms	1290ms	9003ms	4530ms

6 结论

在本毕业设计中，我实现了 Neo4j 图数据库上的 FORA Single-Source PPR 算法、FORA Top-k PPR 算法以及 All-Pair-Backward-Search 算法，为在海量数据下 Single-Source PPR 以及 Top-k PPR 估计值的计算提供了一个高效且有效限制误差范围的算法实现。FORA 算法综合了 Forward Push 和 Monte-Carlo 算法的思想，利用 Forward Push 部分所得到的信息显著减少了随机游走的次数从而提高计算速度，同时又满足 PPR 估计值的误差约束条件。而 All-Pair-Backward-Search 算法则是基于 Backward Search 算法预先计算出 All-Pair PPR 估计值并存储到文件中，当我们查询关于源节点的 PPR 时只需要通过查询文件即可获得相应的结果，有效提高了查询效率。同时，我还对各 PPR 算法进行了性能测试，其结果表明 All-Pair-Backward-Search 算法只需以较低的时间及空间开销即可实现对图节点 PPR 的预处理，在节点和边的数目较多且允许预处理的情形下，我们可以采用 All-Pair-Backward-Search 算法作为计算图节点 Single-Source PPR 及 Top-k PPR 的解决方案；否则，我们采用 FORA 算法进行 Single-Source PPR 查询或 Top-k PPR 查询。

与此同时，我们也需要指出本实验设计的不足之处。首先，目前 All-Pair-Backward-Search 算法还只能计算无向图的节点 PPR 估计值，若要将其推广到有向图上还需要我们对其进行进一步的研究。其次，为了使 Neo4j 的用户能够更方便快捷地使用我们的高效算法来计算图节点的 PPR 值，我接下来还需要使用 Neo4j 提供的用户自定义过程来包装我所实现的算法。最后，在 All-Pair-Backward-Search 算法的实验过程中我们遇到了内存不足的问题，我们在将来的实现中需要适时地将中间结果写入文件中从而释放内存空间。

通过这次毕业设计，我第一次有机会去了解当今最前沿的理论，并且利用自己在课堂上所学到的知识去实现它。FORA 以及 All-Pair-Backward-Search 作为计算图节点 PPR 的高效算法，尤其是它们在大数据下优良的性能，很好地契合了当今的市场需求。而我利用自己在课堂上所学的 Java 编程设计知识、算法知识等实现这一算法，使其有机会运用到 Neo4j 这样流行的图数据库平台上，从而使更多的用户以及软件收益。这让我体会到了探索并解决现实问题的趣味。

其次，我通过这次机会学习到了新的编程语言并对其进行深入理解。出于毕业设计的需求，我从零开始学习了 Neo4j 的基本操作。此外为了满足算法的复杂度需求，我还仔细研读了 Neo4j 的 API 文档以及其实现代码，这个过程加深了我对这一图数据库的理解。一门新语言不仅可以

运用于当下的毕业设计中，更是成为了我在未来道路上的一个有力的武器！

此外我还体会到了“站在巨人的肩膀上”的重要性。诚然，以我现在的知识储备量，让我自己思索出一个全新的高效算法是不现实的。因此我更加体会到了学习借鉴他人经验的重要性。我仔细研读了最前沿的论文，并且与魏哲巍老师探讨其中的算法，希冀在这个过程中可以将他人的理论经验融汇到我的知识储备中。如同幼苗需要汲取养分才可以茁壮成长，我在求学的道路上还需要站在巨人的肩膀上，才得以看得更远，走得更远！

致谢

很荣幸能够在魏哲巍老师的指导下完成本人的本科毕业设计，衷心感谢魏哲巍老师在这个过程中为我指明方向并且对我的设计及实践进行耐心指导，使我如沐春风，受益匪浅！同时也衷心感谢本科这四年来中国人民大学以及信息学院对我的栽培，教会了我要以国民表率、社会栋梁的标准来要求自己。感谢在这个过程中对我给予帮助的老师、同学、家人，也希望我在未来的路上能够不忘初心，继续前进，不辜负大家对我的期望！

参考文献

[1] Google Press Center. Fun Facts [EB/OL]. https://web.archive.org/web/20010715123343/https://www.google.com/press/funfacts.html. 2019 年 4 月 11 日访问.

[2] Page, L., Brin, S., Motwani, R., & Winograd, T. The PageRank citation ranking: Bringing order to the web [R]. Stanford, CA: Stanford InfoLab, 1999.

[3] Cornell University. Personalized PageRank in Recommendation System [EB/OL]. https://blogs.cornell.edu/info2040/2017/10/25/personalized-pagerank-in-recommendation-system/. 2019 年 4 月 11 日访问.

[4] Fogaras, D., Rácz, B., Csalogány, K., & Sarlós, T. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments [J]. Internet Mathematics, 2005, 2(3):333-358.

[5] Andersen, R., Chung, F., & Lang, K. Local graph partitioning using pagerank vectors [J]. 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06), 2006, 475-486.

[6] Andersen, R., Borgs, C., Chayes, J., Hopcraft, J., Mirrokni, V. S., & Teng, S. H. Local computation of PageRank contributions [J]. International Workshop on Algorithms and Models for the Web-Graph, 2007, 150-165.

[7] Lofgren, P., Banerjee, S., & Goel, A. Personalized pagerank estimation and search: A bidirectional approach [J]. Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, 2016, 163-172.

[8] Wang, S., Yang, R., Xiao, X., Wei, Z., & Yang, Y. Fora: Simple and effective approximate single-source personalized pagerank [J]. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, 505-514.

[9] Fujiwara, Y., Nakatsuji, M., Yamamuro, T., Shiokawa, H., & Onizuka, M. Efficient personalized pagerank with accuracy assurance [J]. Proceedings of the 18th ACM SIGKDD

international conference on Knowledge discovery and data mining, 2012, 15-23.

[10] Le Gall, F. Powers of tensors and fast matrix multiplication [J]. Proceedings of the 39th international symposium on symbolic and algebraic computation, 2014, 296-303.

[11] Yoon, B. H., Kim, S. K., & Kim, S. Y. Use of graph database for the integration of heterogeneous biological data [J]. Genomics & informatics, 2017, 15(1):19.

[12] Todd Hoff. Neo4j - A Graph Database That Kicks Buttox [EB/OL]. http://highscalability.com/neo4j-graph-database-kicks-buttox. 2019 年 4 月 11 日访问.

[13] 张龙斌. 如何将大规模数据导入 Neo4j 及导入具体步骤及 Demo [EB/OL]. https://my.oschina.net/zlb1992/blog/918243. 2019 年 4 月 11 日访问.

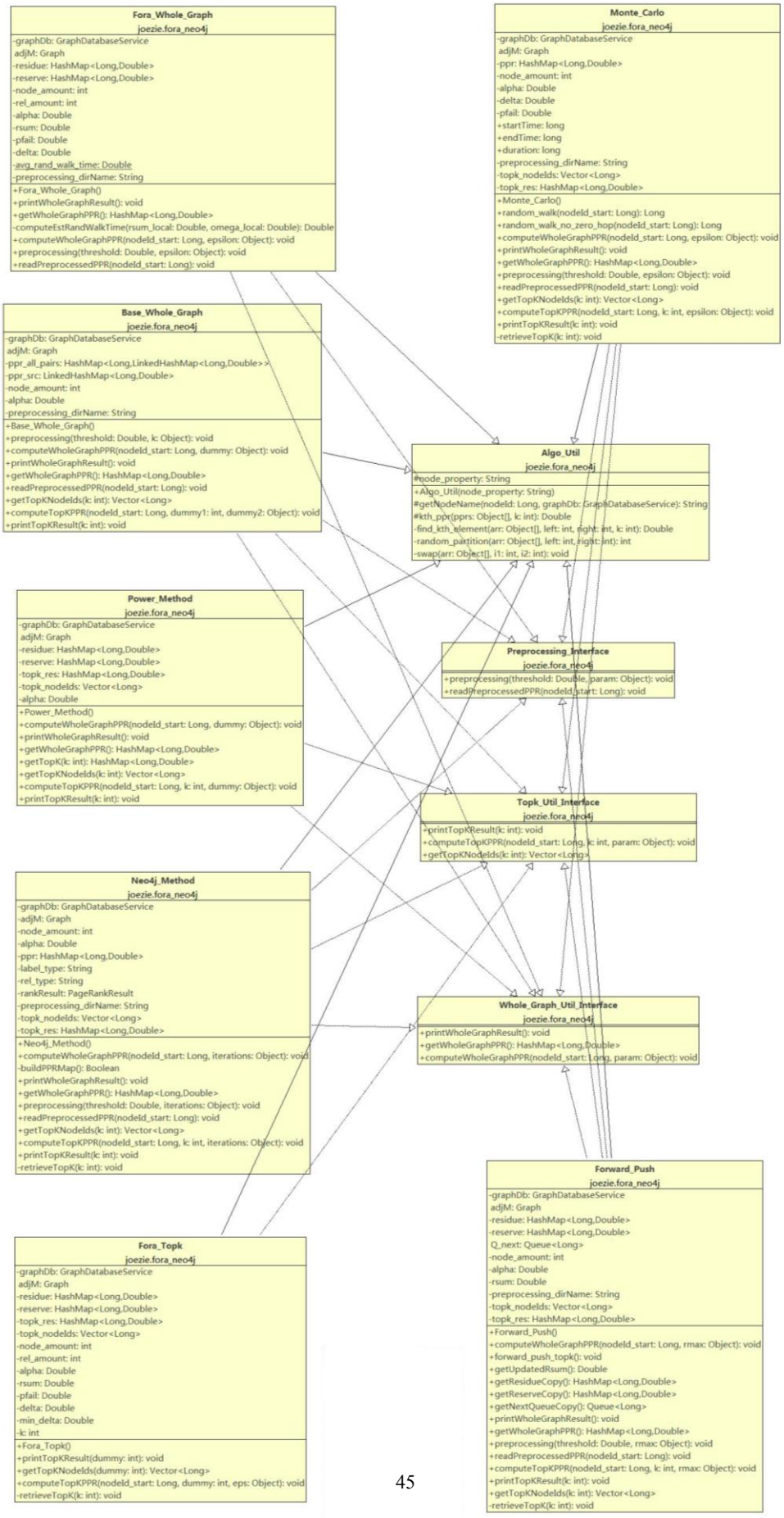
[14] Neo4j, Inc. User Defined Procedures and Functions [EB/OL]. https://neo4j.com/developer/procedures-functions/. 2019 年 4 月 11 日访问.

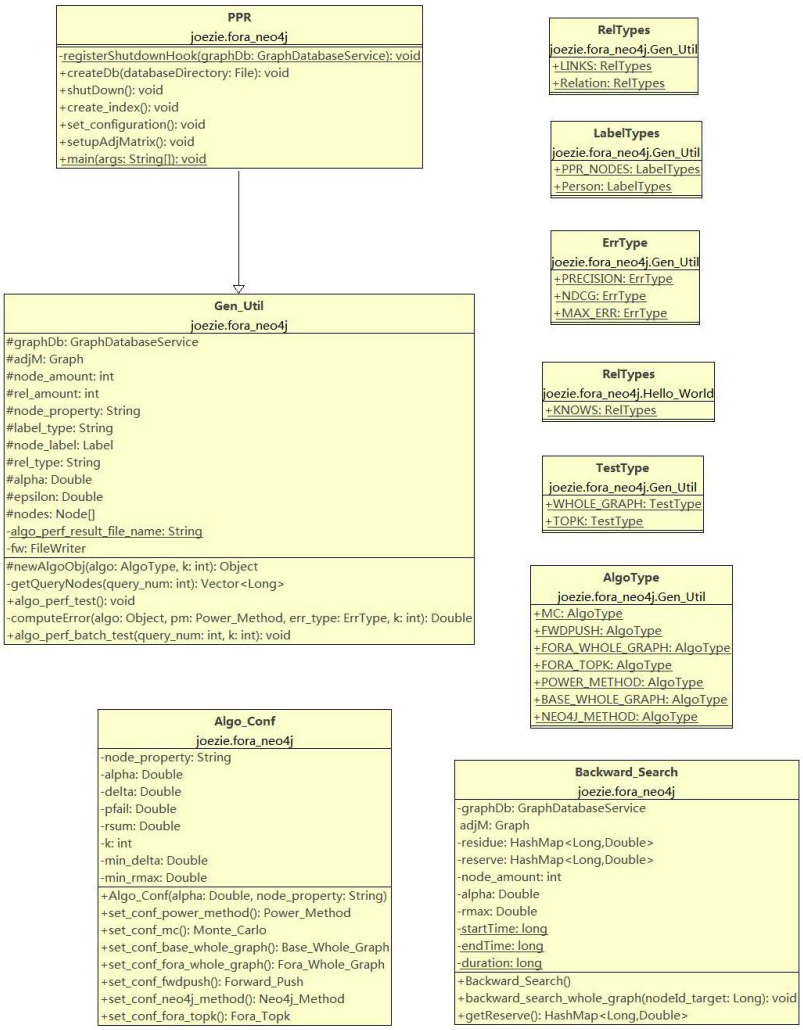
[15] Mark Needham. Neo4j Graph Algorithms Release—Random Walk and Personalized PageRank [EB/OL]. https://medium.com/neo4j/graph-algorithms-release-random-walk-and-personalized-pagerank-80160db3757. 2019 年 4 月 11 日访问.

[16] Manaskasemsak, B., & Rungsawang, A. An efficient partition-based parallel PageRank algorithm [J]. 11th International Conference on Parallel and Distributed Systems (ICPADS'05), 2005, 1:257-263.

[17] Gleich, D., Zhukov, L., & Berkhin, P. Fast parallel PageRank: A linear system approach [R]. Sunnyvale, CA: Yahoo!, 2004.

附录 1　UML 类图


Fora_Whole_Graph
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-residue: HashMap<Long,Double>
-reserve: HashMap<Long,Double>
-node_amount: int
-rel_amount: int
-alpha: Double
-rsum: Double
-pfail: Double
-delta: Double
-avg_rand_walk_time: Double
-preprocessing_dirName: String
+Fora_Whole_Graph()
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
-computeEstRandWalkTime(rsum_local: Double, omega_local: Double): Double
+computeWholeGraphPPR(nodeId_start: Long, epsilon: Object): void
+preprocessing(threshold: Double, epsilon: Object): void
+readPreprocessedPPR(nodeId_start: Long): void
Monte_Carlo
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-ppr: HashMap<Long,Double>
-node_amount: int
-alpha: Double
-delta: Double
-pfail: Double
+startTime: long
+endTime: long
+duration: long
-preprocessing_dirName: String
-topk_nodeIds: Vector<Long>
-topk_res: HashMap<Long,Double>
+Monte_Carlo()
+random_walk(nodeId_start: Long): Long
+random_walk_no_zero_hop(nodeId_start: Long): Long
+computeWholeGraphPPR(nodeId_start: Long, epsilon: Object): void
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+preprocessing(threshold: Double, epsilon: Object): void
+readPreprocessedPPR(nodeId_start: Long): void
+getTopKNodeIds(k: int): Vector<Long>
+computeTopKPPR(nodeId_start: Long, k: int, epsilon: Object): void
+printTopKResult(k: int): void
-retrieveTopK(k: int): void
Base_Whole_Graph
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-ppr_all_pairs: HashMap<Long,LinkedHashMap<Long,Double>>
-ppr_src: LinkedHashMap<Long,Double>
-node_amount: int
-alpha: Double
-preprocessing_dirName: String
+Base_Whole_Graph()
+preprocessing(threshold: Double, k: Object): void
+computeWholeGraphPPR(nodeId_start: Long, dummy: Object): void
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+readPreprocessedPPR(nodeId_start: Long): void
+getTopKNodeIds(k: int): Vector<Long>
+computeTopKPPR(nodeId_start: Long, dummy1: int, dummy2: Object): void
+printTopKResult(k: int): void
Algo_Util
joezie.fora_neo4j
#node_property: String
+Algo_Util(node_property: String)
#getNodeName(nodeId: Long, graphDb: GraphDatabaseService): String
#kth_ppr(ppprs: Object[], k: int): Double
-find_kth_element(arr: Object[], left: int, right: int, k: int): Double
-random_partition(arr: Object[], left: int, right: int): int
-swap(arr: Object[], i1: int, i2: int): void
Power_Method
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-residue: HashMap<Long,Double>
-reserve: HashMap<Long,Double>
-topk_res: HashMap<Long,Double>
-topk_nodeIds: Vector<Long>
-alpha: Double
+Power_Method()
+computeWholeGraphPPR(nodeId_start: Long, dummy: Object): void
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+getTopK(k: int): HashMap<Long,Double>
+getTopKNodeIds(k: int): Vector<Long>
+computeTopKPPR(nodeId_start: Long, k: int, dummy: Object): void
+printTopKResult(k: int): void
Preprocessing_Interface
joezie.fora_neo4j
+preprocessing(threshold: Double, param: Object): void
+readPreprocessedPPR(nodeId_start: Long): void
Topk_Util_Interface
joezie.fora_neo4j
+printTopKResult(k: int): void
+computeTopKPPR(nodeId_start: Long, k: int, param: Object): void
+getTopKNodeIds(k: int): Vector<Long>
Neo4j_Method
joezie.fora_neo4j
-graphDb: GraphDatabaseService
-adjM: Graph
-node_amount: int
-alpha: Double
-ppr: HashMap<Long,Double>
-label_type: String
-rel_type: String
-rankResult: PageRankResult
-preprocessing_dirName: String
-topk_nodeIds: Vector<Long>
-topk_res: HashMap<Long,Double>
+Neo4j_Method()
+computeWholeGraphPPR(nodeId_start: Long, iterations: Object): void
-buildPPRMap(): Boolean
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+preprocessing(threshold: Double, iterations: Object): void
+readPreprocessedPPR(nodeId_start: Long): void
+getTopKNodeIds(k: int): Vector<Long>
+computeTopKPPR(nodeId_start: Long, k: int, iterations: Object): void
+printTopKResult(k: int): void
-retrieveTopK(k: int): void
Whole_Graph_Util_Interface
joezie.fora_neo4j
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+computeWholeGraphPPR(nodeId_start: Long, param: Object): void
Forward_Push
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-residue: HashMap<Long,Double>
-reserve: HashMap<Long,Double>
Q_next: Queue<Long>
-node_amount: int
-alpha: Double
-rsum: Double
-preprocessing_dirName: String
-topk_nodeIds: Vector<Long>
-topk_res: HashMap<Long,Double>
+Forward_Push()
+computeWholeGraphPPR(nodeId_start: Long, rmax: Object): void
+forward_push_topk(): void
+getUpdatedRsum(): Double
+getResidueCopy(): HashMap<Long,Double>
+getReserveCopy(): HashMap<Long,Double>
+getNextQueueCopy(): Queue<Long>
+printWholeGraphResult(): void
+getWholeGraphPPR(): HashMap<Long,Double>
+preprocessing(threshold: Double, rmax: Object): void
+readPreprocessedPPR(nodeId_start: Long): void
+computeTopKPPR(nodeId_start: Long, k: int, rmax: Object): void
+printTopKResult(k: int): void
+getTopKNodeIds(k: int): Vector<Long>
-retrieveTopK(k: int): void
Fora_Topk
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-residue: HashMap<Long,Double>
-reserve: HashMap<Long,Double>
-topk_res: HashMap<Long,Double>
-topk_nodeIds: Vector<Long>
-node_amount: int
-rel_amount: int
-alpha: Double
-rsum: Double
-pfail: Double
-delta: Double
-min_delta: Double
-k: int
+Fora_Topk()
+printTopKResult(dummy: int): void
+getTopKNodeIds(dummy: int): Vector<Long>
+computeTopKPPR(nodeId_start: Long, dummy: int, eps: Object): void
-retrieveTopK(k: int): void


PPR
joezie.fora_neo4j
-registerShutdownHook(graphDb: GraphDatabaseService): void
+createDb(databaseDirectory: File): void
+shutDown(): void
+create_index(): void
+set_configuration(): void
+setupAdjMatrix(): void
+main(args: String[]): void
RelTypes
joezie.fora_neo4j.Gen_Util
+LINKS: RelTypes
+Relation: RelTypes
LabelTypes
joezie.fora_neo4j.Gen_Util
+PPR_NODES: LabelTypes
+Person: LabelTypes
ErrType
joezie.fora_neo4j.Gen_Util
+PRECISION: ErrType
+NDCG: ErrType
+MAX_ERR: ErrType
Gen_Util
joezie.fora_neo4j
#graphDb: GraphDatabaseService
#adjM: Graph
#node_amount: int
#rel_amount: int
#node_property: String
#label_type: String
#node_label: Label
#rel_type: String
#alpha: Double
#epsilon: Double
#nodes: Node[]
-algo_perf_result_file_name: String
-fw: FileWriter
#newAlgoObj(algo: AlgoType, k: int): Object
-getQueryNodes(query_num: int): Vector<Long>
+algo_perf_test(): void
-computeError(algo: Object, pm: Power_Method, err_type: ErrType, k: int): Double
+algo_perf_batch_test(query_num: int, k: int): void
RelTypes
joezie.fora_neo4j.Hello_World
+KNOWS: RelTypes
TestType
joezie.fora_neo4j.Gen_Util
+WHOLE_GRAPH: TestType
+TOPK: TestType
AlgoType
joezie.fora_neo4j.Gen_Util
+MC: AlgoType
+FWDPUSH: AlgoType
+FORA_WHOLE_GRAPH: AlgoType
+FORA_TOPK: AlgoType
+POWER_METHOD: AlgoType
+BASE_WHOLE_GRAPH: AlgoType
+NEO4J_METHOD: AlgoType
Algo_Conf
joezie.fora_neo4j
-node_property: String
-alpha: Double
-delta: Double
-pfail: Double
-rsum: Double
-k: int
-min_delta: Double
-min_rmax: Double
+Algo_Conf(alpha: Double, node_property: String)
+set_conf_power_method(): Power_Method
+set_conf_mc(): Monte_Carlo
+set_conf_base_whole_graph(): Base_Whole_Graph
+set_conf_fora_whole_graph(): Fora_Whole_Graph
+set_conf_fwdpush(): Forward_Push
+set_conf_neo4j_method(): Neo4j_Method
+set_conf_fora_topk(): Fora_Topk
Backward_Search
joezie.fora_neo4j
-graphDb: GraphDatabaseService
adjM: Graph
-residue: HashMap<Long,Double>
-reserve: HashMap<Long,Double>
-node_amount: int
-alpha: Double
-rmax: Double
-startTime: long
-endTime: long
-duration: long
+Backward_Search()
+backward_search_whole_graph(nodeId_target: Long): void
+getReserve(): HashMap<Long,Double>

附录 2　接口及类介绍

1　Whole_Graph_Util_Interface.interface

方法名	功能介绍
+printWholeGraphResult(): void	输出各节点的 Single-Source PPR 计算结果，需要实现类实现
+getWholeGraphPPR(): HashMap<Long, Double>	返回存储各节点 Single-Source PPR 结果的 HashMap，需要实现类实现
+computeWholeGraphPPR(nodeId_start: Long, param: Object): void	计算各节点的 Single-Source PPR，需要实现类实现

2　Topk_Util_Interface.interface

方法名	功能介绍
+printTopKResult(k: int): void	输出各节点的 Top-k PPR 计算结果，需要实现类实现
+getTopKNodeIds(k: int): Vector<Long>	返回存储 Top-k 节点 ID 的 Vector，需要实现类实现
+computeTopKPPR(nodeId_start: Long, k: int, param: Object): void	计算各节点的 Top-k PPR，需要实现类实现

3　Preprocessing_Interface.interface

方法名	功能介绍
+preprocessing(threshold: Double, param: Object): void	预计算 All--Pair PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long):	从文件中读取 Single-Source PPR 预处理结

void	果
+getPrepSize(): Long	返回预处理文件所占空间大小
+deletePrepDir(): void	删除预处理文件

4 Algo_Conf.class

方法名	功能介绍
+Algo_Conf(alpha: Double, node_property: String)	构造方法
+set_conf_power_method(graphDb: GraphDatabaseService, adjM: Graph): Power_Method	为 Power Iterations 算法配置参数并新建 Power_Method 类的对象
+set_conf_mc(node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, adjM: Graph): Monte_Carlo	为 Monte-Carlo 算法配置参数并新建 Monte_Carlo 类的对象
+set_conf_base_whole_graph(node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, adjM: Graph): Base_Whole_Graph	为 All-Pair-Backward-Search 算法配置参数并新建 Base_Whole_Graph 类的对象
+set_conf_fora_whole_graph(node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, adjM: Graph): Fora_Whole_Graph	为FORA Single-Source PPR 算法配置参数并新建 Fora_Whole_Graph 类的对象
+set_conf_fwdpush(node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, adjM: Graph): Forward_Push	为 Forward Push 算法配置参数并新建 Forward_Push 类的对象

+set_conf_neo4j_method(graphDb: GraphDatabaseService, adjM: Graph, label_type: String, rel_type: String, node_amount: int): Neo4j_Method	为 Neo4j Single-Source PPR 算法配置参数并新建 Neo4j_Method 类的对象
+set_conf_fora_topk(node_amount: int, rel_amount: int, k: int, graphDb: GraphDatabaseService, adjM: Graph): Fora_Topk	为 FORA Top-k PPR 算法配置参数并新建 Fora_Topk 类的对象

5 Algo_Util.class

方法名	功能介绍
+Algo_Util(node_property: String)	构造方法
#getNodeName(nodeId: Long, graphDb: GraphDatabaseService): String	返回节点 ID 对应的节点名
#kth_ppr(pprs: Object[], k: int): Double	返回所有节点中第 k 大的 PPR 值
-find_kth_element(arr: Object[], left: int, right: int, k: int): Double	返回数组下标 left 至 right 之间第 k 大的数值
-random_partition(arr: Object[], left: int, right: int): int	返回一个任意选择的标兵元素在分割数组后的下标，此时在数组中下标小于它的元素均不小于标兵元素，下标大于它的元素均不大于标兵元素
-swap(arr: Object[], i1: int, i2: int): void	交换数组中两个元素的位置

6 Gen_Util.class

方法名	功能介绍
#newAlgoObj(algo: AlgoType, k: int): Object	返回一个相应算法的新建对象

-getQueryNodes(query_num: int): Vector<Long>	返回存储了随机选择的源节点样本的 Vector
+algo_perf_test(algoType: AlgoType, query_num: int, k: int, param: Object, threshold: Double, to_be_preprocessed: Boolean, testType: TestType): void	对给定算法进行单次性能测试，测定其运行时间及误差并将测试结果存入文件中
-computeError(algo: Object, pm: Power_Method, err_type: ErrType, k: int): Double	通过将 PPR 算法得到的 PPR 估计值与 Power Iterations 算法得到的 PPR 精确值相比较得到其误差
+algo_perf_batch_test(query_num: int, k: int): void	对所有算法批量地进行 Single-Source PPR、Top-k PPR 及预处理的性能测试

7 PPR.class

方法名	功能介绍
-registerShutdownHook(graphDb: GraphDatabaseService): void	注册一个监听数据库库关闭动作的钩子方法（Hook）
+createDb(databaseDirectory: File): void	打开一个已有的数据库或创建一个新的数据库
+shutDown(): void	关闭数据库
+create_index(): void	为节点属性创建索引
+set_configuration(alpha: Double, epsilon: Double, node_property: String, label_type: String, rel_type: String): void	根据用户输入的属性及数据集的属性设置参数
+setupAdjMatrix(): void	将图中节点和边的信息从数据库中加载出来
+main(args: String[]): void	主方法。创建数据库、加载图、对各算法进

	行性能测试。

8 Power_Method.class

方法名	功能介绍
+Power_Method(graphDb: GraphDatabaseService, alpha: Double, adjM: Graph, node_property: String)	构造方法
computeWholeGraphPPR(nodeId_start: Long, dummy: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，基于 Power Iterations 算法计算所有节点关于源节点的 Single-Source PPR 精确值
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照递减顺序输出所有节点关于源节点的 Single-Source PPR 精确值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回所有节点关于源节点的 Single-Source PPR 精确值
+getTopK(k: int): HashMap<Long,Double>	返回关于源节点的 Top-k PPR 精确值
+getTopKNodeIds(k: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并按照 PPR 精确值递减排序
+computeTopKPPR(nodeId_start: Long, k: int, dummy: Object): void	重写 Topk_Util_Interface 接口的同名函数，将不小于第 k 大的 PPR 值的 PPR 结果存储起来
+printTopKResult(k: int): void	重写 Topk_Util_Interface 接口的同名函数，输出关于源节点的 Top-k PPR 精确值

9 Monte_Carlo.class

方法名	功能介绍
+Monte_Carlo(alpha: Double, node_amount: int, graphDb: GraphDatabaseService, pfail: Double, delta: Double, adjM: Graph, node_property: String)	构造方法
+random_walk(nodeId_start: Long): Long	从源节点开始一次随机游走
+random_walk_no_zero_hop(nodeId_start: Long): Long	从源节点的任一出边邻居节点开始一次随机游走
+computeWholeGraphPPR(nodeId_start: Long, epsilon: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，基于 Monte-Carlo 算法计算所有节点关于源节点的 Single-Source PPR 估计值
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照递减顺序输出所有节点关于源节点的 Single-Source PPR 估计值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回所有节点关于源节点的 Single-Source PPR 估计值
+preprocessing(threshold: Double, epsilon: Object): void	重写 Preprocessing_Interface 接口的同名函数，基于 Monte-Carlo 算法预计算 All-Pair PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long): void	重写 Preprocessing_Interface 接口的同名函数，从文件中读取 Single-Source PPR 预处理结果
+getTopKNodeIds(k: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并

	按照 PPR 估计值递减排序
+computeTopKPPR(nodeId_start: Long, k: int, epsilon: Object): void	重写 Topk_Util_Interface 接口的同名函数，实质是调用了 computeWholeGraphPPR 方法
+printTopKResult(k: int): void	重写 Topk_Util_Interface 接口的同名函数，输出关于源节点的 Top-k PPR 估计值
-retrieveTopK(k: int): void	从 Single-Source PPR 结果中截取 Top-k PPR 结果

10 Forward_Push.class

方法名	功能介绍
+Forward_Push(alpha: Double, rsum: Double, node_amount: int, graphDb: GraphDatabaseService, adjM: Graph, node_property: String)	构造方法
+computeWholeGraphPPR(nodeId_start: Long, rmax: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，计算所有节点关于源节点的 Single-Source PPR 估计值的算法中的 Forward Push 部分
+forward_push_topk(nodeId_start: Long, Q: Queue<Long>, min_rmax: Double, isFirstFwdpush: boolean, rmax: Double): void	计算所有节点关于源节点的 Top-k PPR 估计值的算法中的 Forward Push 部分
+getUpdatedRsum(): Double	返回所有节点 residue 的总和
+getResidueCopy(): HashMap<Long,Double>	复制并返回存储着所有节点 residue 值的 HashMap
+getReserveCopy(): HashMap<Long,Double>	复制并返回存储着所有节点 reserve 值的 HashMap

+getNextQueueCopy(): Queue<Long>	复制并返回存储着在下一次 Forward Push 中可能继续进行 Forward Push 的节点的 Queue
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照 reserve 值递减顺序输出所有节点关于源节点的 reserve 以及 residue 值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回存储着所有节点关于源节点 reserve 值的 HashMap
+preprocessing(threshold: Double, rmax: Object): void	重写 Preprocessing_Interface 接口的同名函数，基于 Forward Push 算法预计算 All-Pair PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long): void	重写 Preprocessing_Interface 接口的同名函数，从文件中读取 Single-Source PPR 预处理结果
+getTopKNodeIds(k: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并按照 PPR 估计值递减排序
+computeTopKPPR(nodeId_start: Long, k: int, rmax: Object): void	重写 Topk_Util_Interface 接口的同名函数，实质是调用了 computeWholeGraphPPR 方法
+printTopKResult(k: int): void	重写 Topk_Util_Interface 接口的同名函数，输出关于源节点的 Top-k PPR 估计值
-retrieveTopK(k: int): void	从 Single-Source PPR 结果中截取 Top-k PPR 结果

11 Backward_Search.class

方法名	功能介绍

+Backward_Search(alpha: Double, rmax: Double, node_amount: int, graphDb: GraphDatabaseService, adjM: Graph)	构造方法
+backward_search_whole_graph(nodeId_target: Long): void	计算所有节点关于源节点的 Single-Source PPR 估计值的算法中的 Backward Search 部分
+getReserve(): HashMap<Long,Double>	返回存储着所有节点 reserve 值的 HashMap

12 Fora_Whole_Graph.class

方法名	功能介绍
+Fora_Whole_Graph(alpha: Double, rsum: Double, pfail: Double, delta: Double, node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, adjM: Graph, node_property: String)	构造方法
+computeWholeGraphPPR(nodeId_start: Long, epsilon: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，基于 FORA 算法计算所有节点关于源节点的 Single-Source PPR 估计值
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照递减顺序输出所有节点关于源节点的 Single-Source PPR 估计值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回存储着所有节点关于源节点 Single-Source PPR 估计值的 HashMap
-computeEstRandWalkTime(rsum_local: Double, omega_local: Double): Double	用于估算随机游走部分的总耗时

+preprocessing(threshold: Double, epsilon: Object): void	重写 Preprocessing_Interface 接口的同名函数，基于 FORA 算法预计算 All-Pair PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long): void	重写 Preprocessing_Interface 接口的同名函数，从文件中读取 Single-Source PPR 预处理结果

13 Fora_Topk.class

方法名	功能介绍
+Fora_Topk(alpha: Double, rsum: Double, pfail: Double, delta: Double, node_amount: int, rel_amount: int, graphDb: GraphDatabaseService, min_delta: Double, k: int, adjM: Graph, node_property: String)	构造方法
+computeTopKPPR(nodeId_start: Long, dummy: int, eps: Object): void	重写 Topk_Util_Interface 接口的同名函数，基于 FORA 算法计算关于源节点的 Top-k PPR 估计值
+printTopKResult(dummy: int): void	重写 Topk_Util_Interface 接口的同名函数，按照递减顺序输出关于源节点的 Top-k PPR 估计值
+getTopKNodeIds(dummy: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并按照 PPR 估计值递减排序
-retrieveTopK(k: int): void	从 Single-Source PPR 结果中截取 Top-k PPR 结果

14 Base_Whole_Graph.class

方法名	功能介绍
+Base_Whole_Graph(alpha: Double, node_amount: int, graphDb: GraphDatabaseService, adjM: Graph, node_property: String)	构造方法
+computeWholeGraphPPR(nodeId_start: Long, dummy: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，从文件中读取 Single-Source PPR 预处理结果
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照递减顺序输出所有节点关于源节点的 Single-Source PPR 估计值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回所有节点关于源节点的 Single-Source PPR 估计值
+preprocessing(threshold: Double, k: Object): void	重写 Preprocessing_Interface 接口的同名函数，基于 All-Pair-Backward-Search 算法预计算各节点的 Single-Source PPR 或 Top-k PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long): void	重写 Preprocessing_Interface 接口的同名函数，从文件中读取 Single-Source PPR 预处理结果，实质是调用了 computeWholeGraphPPR 方法
+getTopKNodeIds(k: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并按照 PPR 估计值递减排序
+computeTopKPPR(nodeId_start: Long, dummy1: int, dummy2: Object): void	重写 Topk_Util_Interface 接口的同名函数，实质是调用了 computeWholeGraphPPR 方法

+printTopKResult(k: int): void	重写 Topk_Util_Interface 接口的同名函数，输出关于源节点的 Top-k PPR 估计值

15 Neo4j_Method.class

方法名	功能介绍
+Neo4j_Method(graphDb: GraphDatabaseService, alpha: Double, label_type: String, rel_type: String, adjM: Graph, node_amount: int, node_property: String)	构造方法
+computeWholeGraphPPR(nodeId_start: Long, iterations: Object): void	重写 Whole_Graph_Util_Interface 接口的同名函数，基于 Neo4j Single-Source PPR 算法计算所有节点关于源节点的 Single-Source PPR 估计值
-buildPPRMap(): Boolean	基于 Neo4j Single-Source PPR 算法计算出来的结果进行归一化得到符合我们的 PPR 标准的结果
+printWholeGraphResult(): void	重写 Whole_Graph_Util_Interface 接口的同名函数，按照递减顺序输出所有节点关于源节点的 Single-Source PPR 估计值
+getWholeGraphPPR(): HashMap<Long,Double>	重写 Whole_Graph_Util_Interface 接口的同名函数，返回所有节点关于源节点的 Single-Source PPR 估计值
+preprocessing(threshold: Double, iterations: Object): void	重写 Preprocessing_Interface 接口的同名函数，基于 Neo4j Single-Source PPR 算法预计算 All-Pair PPR 结果并存储到文件中
+readPreprocessedPPR(nodeId_start: Long):	重写 Preprocessing_Interface 接口的同名函

void	数，从文件中读取 Single-Source PPR 预处理结果
+getTopKNodeIds(k: int): Vector<Long>	重写 Topk_Util_Interface 接口的同名函数，返回关于源节点的 Top-k PPR 的节点 ID 并按照 PPR 估计值递减排序
+computeTopKPPR(nodeId_start: Long, k: int, iterations: Object): void	重写 Topk_Util_Interface 接口的同名函数，实质是调用了 computeWholeGraphPPR 方法
+printTopKResult(k: int): void	重写 Topk_Util_Interface 接口的同名函数，输出关于源节点的 Top-k PPR 估计值
-retrieveTopK(k: int): void	从 Single-Source PPR 结果中截取 Top-k PPR 结果