学号\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

密级\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

武汉大学本科毕业论文

基于Neo4j的图可达性算法设计与实现

院（系）名 称：计算机学院

专 业 名 称 ：计算机科学与技术

学 生 姓 名 ：柳泽宇

指 导 教 师 ：钟鸣 副教授

二○一九年六月

**郑 重 声 明**

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名： 日期：

摘要

在大图上的可达性查询是在图论上的一个经典问题。近些年来，越来越多的复杂数据被用图的方式存储和管理。这些数据包括交通网络、网页导航、语义网络、化学和生物数据库等系统。一个图可达性查询，给定两个节点是否有一条路径到达，是一种在图中最基本的关系。节点的可达到性也被常被用作其他图算法的基本算子。近些年来，随着互联网与大数据时代的来临，海量数据存储与处理给图数据的分析带来了很大的挑战。而过去以往的传统图算法在海量数据面前存在的一定的弊端，在大规模图上研究节点可达性查询拥有非同寻常的意义。

目前业界主流的图数据库有Neo4j、Amazon Neptune、JanusGraph等。但这些图数据库都在可达性查询上都没有提供直接相关的API，而是只是简单将其包含在最短路径算法之中。但是最短路径对于可达性查询并不是一个必要条件，因此还有非常多的优化空间。

此外，近些年在学术界也涌现了很多在大图上的可达性算法，其均在空间和时间上的均衡做了一定的取舍，可是这些优秀的算法在实际工业界级产品之中鲜有被直接运用。

Neo4j是一个开源的图数据库，是目前使用最多的图数据库。本文综合了最近发展的图可达性查询算法，将其分类汇总，并在基于Neo4j上用Java将其重写，权衡利弊，并在实际图数据库底层上提供了API供查询。

关键词：图算法；可达性查询算法；Neo4j；图索引

**ABSTRACT**

This paper is carried out on the basis of the 211 project-Ssmi-physical simulation system for ship motion control. ……

……

……

……

**Key words:** motion control； autopilot； neural ；GIS

目录

1绪论

1.1 研究背景 ………………………………………………………………………

1.2 图查询算法的研究现状 …………………………………………………

1.2.1 综述……………………………………………

1.2.2 Hop标签法…………………………………

1.2.3 传递闭包压缩法…………………………………

1.2.4 改良在线搜索法…………………………………

1.3 本课题的研究目标 …………………………………………………………

2图可达性算法

2.1 Hop标签：2-Hop标签法 ……………………………………………………… 35

2.2 传递闭包压缩法：Path-Tree…………………………………………………… 37

2.3 改良在线搜索法：Grail…………………………………………………… 39

3 Neo4J与实现

3.1 Neo4J架构简介……………………………………………………… 35

3.2 实验环境与数据集…………………………………………………… 37

3.3 实验结果…………………………………………………… 39

结论…………………………………………………………………………………… 57

参考文献……………………………………………………………………………… 59

致谢…………………………………………………………………………………… 62

附录…………………………………………………………………………………… 72

1 绪论

1.1研究背景

高效处理关于非常大的图的查询是一件非常有意义价值的研究话题，因为其具有许多现实意义，我们可以再XML数据库，GIS，网络信息挖掘，社交网络分析，知识图谱和生物信息挖掘上面经常碰到对潜在图信息的挖掘。例如，在公安局中构建犯罪嫌疑人员的人物关系网，通过图节点的可达性查询可以在海量数据中迅速锁定与犯罪嫌疑人有关的人员；在生物计算中，图是常见的信息存储架构，例如蛋白质分子间是否存在互斥、吸引关系，这就可以在蛋白质图上反应为可达性查询。

图的可达性查询的基本定义是给定有向图*,*对于节点和，是否有一条连续的路径可以从节点到达节点，即。在近些年来，引来了许多学者的研究兴趣。首先图的可达性查询作为在图算法中一个基本的问题值得研究，此外，图的可达性查询研究更是许多其他图算法的基石。例如，在回答图的配对问题时候，我们需要确定许多节点对之间的可达性才能回答最后的问题。可达性查询在一些算法也被运用做预处理问题，通过得到否定可达性查询从而快速剪枝，缩小搜索空间等。

在过去，图的可达性查询主要的思想是根据可达性标签构建图的索引。也就是说，在一张图中，每个顶点被赋予特定标签，所以在图中任意两个顶点的可达性可以被他们的标签所决定。这些算法在中小型图中表现的非常好，有非常快速的速度和可以接受的索引体积，但随着信息时代的来临，图变得越来越大，一张有几百万个节点和边的图变得寻常，而到这时候，以往的算法变会有一定的瓶颈性。在图变得越来越大的情况下，如何在数百万节点下建立索引，如何把过去的算法的可扩展性给实现开来是一件非常有意义的事。

在这篇课题中，我们会首先介绍过去传统处理图可达性查询的快速算法，再介绍最前沿的关于扩展图可达性算法到大图上应用的算法。有些图可达性算法已经被提出了很久，但在实际工业界上却很少被应用，本论文最后会结合目前最成熟最热门的开源图数据库Neo4J，并尝试将这些算法应用到Neo4J自身上，并给出性能分析，评估其工业实际价值。

**1.2图查询算法研究现状**

节点之间的可达性查询对有向图来说是一个基本的算子。它回答了一个节点是否有一条路径可以到达节点，这条路径是简单路径，也就是说不会有重复走节点的可能性。可达性计算在计算机科学领域被研究很广泛，包括软件工程，编程语言和分布式计算。最早期的在可达性计算领域的研究是在数据库领域，可以追溯到其在操作符号的回溯和知识系统管理上。最近出现的“大数据”热潮（富数据——从生物、社交网络、软件分析和语意网络）对挖掘图数据产生了全新的挑战，重新点燃了深挖可达性计算的热情。

假设我们有一个成为我们的可达性查询的DAG，我们有个顶点和条边。

|  |  |  |
| --- | --- | --- |
| 时间/类 型 | BFS/DFS法 | 传递闭包 |
| 索引构造时间 |  |  |
| 查询时间 |  |  |
| 索引大小 |  |  |

表**1. BFS/DFS与传递闭包**

在最近几年中，在数据库系统中，有很多给图加索引的方法被提出来用来加速回答可达性查询。所有这些方法是存在于两种极端的可达性查询方法中，如表1，也就是DFS（深度优先遍历）/ BFS（广度优先遍历）和完全传递闭包。所有的工作都是在努力在查询时间、索引大小、构造代价之间寻求一个巧妙的平衡。

BFS/DFS是最直接的算法，我们根据深度和宽度一层一层去搜索我们想要的节点。但这种方法在节点、边数量变得很大时候效率非常低下，纵使有剪枝策略也还是会在很多情况下扫描全部节点，效率很差。

构建传递闭包法可以达到的查询速度，但是其相当于把整个图可能的配对都提前计算了一遍然后存储到另外一个图中。这样构建索引的开销太大，而且需要很大的额外存储空间。

所以，普通DFS、BFS和构建传递闭包这些方法在海量数据组成的图面前都面临一个可扩展性的瓶颈。海量数据来源于像社交网络（脸书、推特以及微博），海量语义网络以及知识图谱。这些方法大部分只能处理中小型的图数据，差不多在前到万级别的节点和边数目就已经达到处理极限。像在线搜索的方法，像DFS/BFS，总是能在任意大小的图上能进行，他们的查询时间也随着图的增大而增大，以至于在图非常大的时候难以进行，代价花费太高。而构建传递闭包在非常大的图基本很难实现。

近些年来，有很多针对可达性查询计算的方法已经被提了出来，在这里我们把他们大致分成三类：Hop标签法，传递闭包压缩法，改良在线搜索法。

此外，由于有向图都可以通过构造强连通分量(经典的Tarjan算法)从而转换成有向无环图，在下面我们都以给定有向无环图为大前提。

**1．2．1 Hop标签法**

这一种类的方法应用中间的节点作为“跳板”来编码可达性。比如每个节点记录两个包含中间节点的列表。一个列表中包含了能到达此节点的中间顶点，另一个列表包含了从此节点能到达其他中间节点的中间顶点。为了回答可达性查询，我们需要在入度节点集合和出度节点集合做一个Join操作，Join操作用来寻找他们是否有共同顶点。使用两种集合的标签，Hop标签法在某种程度上也可以被视为传递闭包法。但是和传递闭包压缩法相比，Hop标签法更慢一些但是它的索引会小很多。

目前最经典的Hop标签法是2-Hop标签法。最近提出的3-Hop标签法在中间结构运用了一种链式分解的方法来提高2-Hop标签法的性能。尽管如此，这些方法有一个共同的特点就是构造复杂度比较高，直接导致了其可扩展性有一定的问题。为了缩小标签的大小，原始的2-Hop算法依赖于一种依靠于贪婪策略的集合覆盖框架，它不仅涉及到迭代地在一个二分图内寻找最稠密的子图，同时，它也需要去计算出整个传递闭包。这种原始2-Hop的总体时间复杂度达到了O(n3|TC|)。是十分耗费时间的。即使后续研究在一定程度上将其时间复杂度降低了，这种方法能处理的图也远小于一百万个节点。

近期，几种启发式搜索的方法也被提出来减少2-Hop的构造时间，像HOPI算法运用了分治的思想去计算2-Hop标签。其他方法，像是基于几何学的算法和图分割的算法也被用来产生2-Hop标签。虽然这些算法极大的提升了2-Hop的构建时间，但是没有集合覆盖的框架作为保障，他们就不能在他们的标签大小上做任何限制。更何况，他们在大图上的课扩展性也被这些算法所限制。这些算法通常无法在很大的图上进行。

**1．2．2 传递闭包压缩法**

这只方法主要目的是去直接压缩传递闭包（Transitive Closure） 并且制定每个顶点一个压缩的可达性集合。为了决定顶点到的可达性，节点只需要检查。这种大类方法包括链表达式，间隔标签表达，双标签法，比特向量压缩法以及路径树压缩法。使用间隔表达式作为例子，在可达性集合的顶点上，任何连续的顶点分割都可以被压缩成它的起点顶点和结束顶点。例如，如果完整传递闭包是,那么它就能被压缩成两个间隔对：和。

这种大类的方法通常比另外两大类方法要快一些。而在适当中小型的图上，这种类型的方法已经被证实了在查询可达性上时间性能最优（间隔表达式和路径树算法）。此外，这种方法即使在稀疏图上性能依然非常优秀。但是，他们的优点也正是限制其可扩展性的瓶颈：随着顶点的增加，每个节点的被计算出的传递闭包也随之增加，最后在大图上不可避免的超过了内存限制。在一台普通具有8GB内存的计算机上，应用传递闭包压缩方法的极限大约是在一百万个节点。虽然传递闭包可以被持久化到硬盘上，但它的构建和查询性能会被硬盘IO极大拖累。而有一些方法更为甚，像基于树的间隔表达法需要完整的计算所有节点的传递闭包，以至于无法高效持久化这些闭包到硬盘上。

**1．2．3 改良在线搜索法**

第三种类的方法运用了在线搜索去回答可达性查询。这种方法主要引进了辅助标签的信息来极大地进行搜索空间剪枝。最近提出的算法Label+SSPI和GRIPP两种都运用了树覆盖去加速深度优先遍历查询。GRAIL算法赋予了每个节点多个间隔标签，每个标签都是被随机深度优先遍历而生成的。而对于某一个节点所对应的标签间隔是从同样的深度优先遍历而来，那么这样就可以决定一个节点是否有可能去达到另一个节点。如果（中的间隔标签不是间隔标签的子集），那么顶点就不可能到达顶点；但是，当，我们却不能决定到底是否能达到。所以对决定顶点可达性是一个必要但不充分条件；多间隔可以增加拒绝的可能性。GRAIL使用了这种多间隔标签的方法来在深度优先遍历之中剪枝，并且它被认为是最好的改良在线搜索方法之一。同时，对于非常大的图，它也是唯一几种看似能够有良好的可扩展性的算法。

改良在线搜索的方法是他们通常不需要额外的优化步骤和在构建算法时候不需要计算传递闭包，所以这种方法大体上不受图大小限制。尽管如此，它的回答查询时间却是这几种方法里最慢的一类，因为其大多数工作都在真正查询阶段进行，没有过早的提炼出图的许多信息。当图的大小变得十分巨大时候，它的查询效率变得十分慢甚至无法回答可达性查询。

**1.3本课题研究内容**

在这篇文章中，我们会首先详细研究近些年来先进的处理图可达性查询的算法，再到Neo4J图数据库以及算法实现实验与评估。有些图可达性算法已经被提出了很久，但在实际工业界上却很少被应用，本课题目前结合了最成熟最热门的开源图数据库Neo4J，并尝试将这些算法应用到Neo4J自身上，并给出性能分析，评估其工业实际价值。

2 图可达性算法

**2.1 Hop标签：2-Hop标签算法**

**2．1．1 Hop定义**

2-Hop标签法，直观上来说，它尝试去发现原图的一个顶点子集 ，这个子集尝试去捕捉整个DAG中的整体连接信息。然后，对于每个在DAG里面的顶点，我们将在集合里面能达到的顶点的顶点做为一个列表，记为。在另一个列表中存储从出发能到达的集合里面的顶点集合，记为。这两个列表记录了点与点之间的所有必要的信息，以至于从其中可以推断出在顶点和之间的任何可达性查询。也就是说，如果，那么，反之亦然。

一个节点的2-Hop标签存储了部分它的祖先节点和后继节点的信息。这些集合通常比全部的要小很多，所以他们并不需要去囊括一个节点的所有祖先和后继节点。

**定义2.1.1：2-Hop 标签**

假设我们有一个图是一个有向图。对于每一个顶点对, 我们令是从顶点到顶点的一系列路径之和。我们定义一个hop是一对顶点 。其中是一条在的路径，是的双端顶点之一。对于总路径集合我们说关于hop的集合是一个2-Hop，当对任意，如果，那么存在一条路径和两个hop , 。即是h1和h2的中间节点。

**定义2.1.2：2-Hop 覆盖问题**

给定一个图。对于任何，为一系列的从顶点到顶点的路径。我们让。我们定义一个hop是一对。其中是一条在中的路径，是中的两个顶点之一。我们把称作hop的“把手”。对于hop的一个集合，如果是一个对的2-Hop覆盖当且仅当对于每个，有，存在一条路径，并且两个hop有，，也就是。

**2．1．2 2-Hop实现**

为了去表达一个图的传递闭包——让其含有相同的图可达性信息，我们首先的任务是产生一个最小的2-Hop覆盖。但是，最小2Hop覆盖实际上是可以规约成最小集合覆盖问题的——这是一个NP-Hard级别的问题，不能准确在多项式时间内找到最优解。所以我们需要一个近似算法来解决这个NPH难题。Cohen et al. 介绍了一种多项式时间复杂度的算法来计算2-Hop覆盖问题，这种覆盖集合大小最多比最佳大小要大级别。我们接下来介绍这个算法。

在有向图中。令是图的一个传递闭包。对于，，其中 且在中有一条从到的路径。类似的，对于，是一个集合，在中有一条从到的路径。

对于一个节点。令

节点我们称作为集合的**中心。**

对于一个给定的还不是2-Hop覆盖覆盖的2-Hop标签，定义是还没有被覆盖的连接集合。所以集合 包含了图G中包含和没有覆盖的所有的连接。有比例

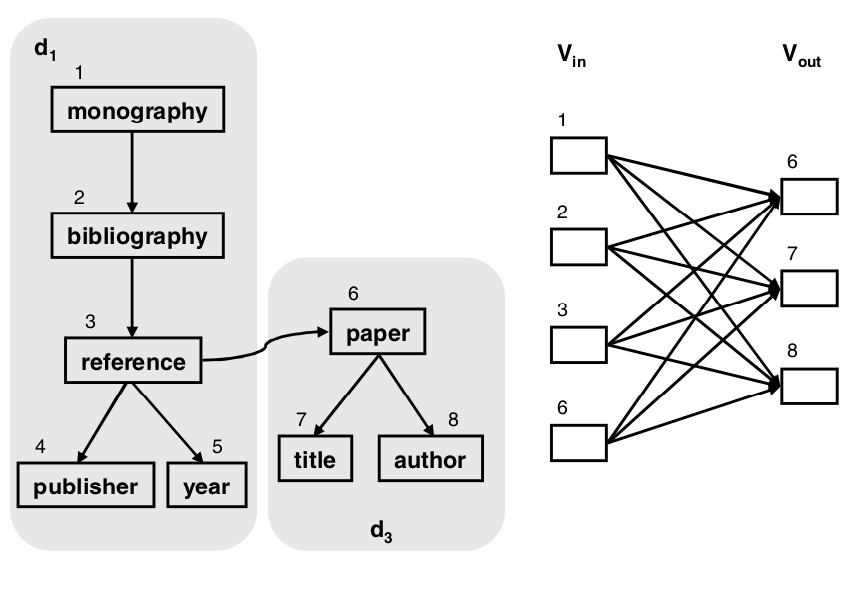
描述了还没有被覆盖的通过节点的连接数量与依赖此连接的总数节点的关系。

这个计算接近最佳的2-Hop覆盖的算法是从，并且在中的每个节点都是没有2-Hop标签。 在每个阶段，集合包含还没有被覆盖的连接。我们通过一个贪婪算法来选择最佳的节点所以它能尽其所能覆盖更多的连接。这样我们的集合在大体上是接近使用最少的节点来达到全覆盖。如果我们选择了拥有最高比例的节点，我们就可以得到一个目前来说能覆盖尽可能多连接的集合，并且其不会让目前的2-Hop标签大小增长很多。在有最大的的w节点选择后，它的相关节点是用来更新2-Hop标签：

然后 就会被从中移除。这个算法当空时终止。在为空集时，在中的所有连接都已经被之前的2-Hop覆盖。

对于一个节点，在有一个指数级别大小的子集，然而我们在一个计算循环的时候只需要选择单一个。所以，以上的算法需要指数级别的时间去计算一个具体的2-Hop覆盖。这个算法需要更多改进来实现多项式级别的运行时间。

对于一个给定的节点，发现使其有最高的的问题恰好就是寻找最稠密子图问题。我们构建关于节点w的辅助无向二分中心图如下。集合对于每个节点包含两个节点和。有一条无向边当且仅当仍然还没有被覆盖并且 并且 。最后，所有被分离的节点都可以从中移除，图2.1展示了以做成的中心图



图**2.1 w=6的中心图**

**定义2.1.3：中心图**

是一个有向图。对于一个给定的2-Hop标签，假设 是一个在中还没有被覆盖的连接。若，则关于的中心图是一个无向二分图，其中有顶点集合和边集合。顶点的集合是，其中：

存在一个无向边当且仅当且 且。

一个子图的密度是其节点的平均度数（进入节点的边和出节点的边的数目）。而对于一个给定的中心图的最稠密的子图可以通过一个线性时间的2-近似算法计算而得。这个算法迭代地从图中移除最小度数的节点。这样就生成了一序列的子图以及他们的序列。这个算法返回一个拥有最高稠密度的子图，也就是给定中心图最稠密的子图，稠密度的定义是边的数量比上节点的数量。我们把这个子图的稠密度记为。

**定义2.1.4：最稠密子图**

是一个无向图。最稠密子图问题是去发现一个子集，这个子集代表的子图拥有最大的节点平均度数。

在这里是的子集，且能连接在中的两个节点。

改进的计算2-Hop覆盖问题的算法在子图中剩余的节点中以密度降序选取了最佳的节点。其中图是原图的子图。所以我们可以很有效的获得集合, ,其中是有最高比值。

所以依据这个思想，对于给定图中的连接集合，我们有一个多项式时间的算法来计算一个2-hop覆盖。

构建2-Hop覆盖的时间复杂度为因为对于给定图使用Floyd-Warshall算法计算传递闭包需要时间，然后从传递闭包中计算2-Hop覆盖需要时间。（最初步骤计算最稠密组图的个节点，第二次计算个节点，以此类推，总共要次计算，且每次计算最坏时间复杂度为。

**2.2 传递闭包压缩法：Path-Tree**

**2．2．1 Tree Cover**

在介绍Path-Tree算法之前先来介绍Tree Cover方法。Agrawal et al. 使用了一个生成树来覆盖整个图并且压缩传递闭包矩阵。这个算法使用的就是最佳树覆盖能够最大化的压缩传递闭包。树覆盖的方法主要是依靠于间隔标签（Interval Labeling）。对于给定的一颗树，我们首先给每个顶点指定一对数字（一个间隔对）。如果顶点能到达顶点，那么的间隔对就包含在的间隔对里面。间隔可以通过实现一个对树的后续遍历而生成。每个顶点与一对间隔相关联，其中是节点的子节点的后续遍历数字，是其后续遍历中的最小数字。（每个节点自身都是其自己的子节点）。

假定我们对于给定有向无环图，使用间隔标签在中的节点做成索引。那么，对于任何顶点，我们只仅仅需要去记住那些它能到达的节点，但是可达性查询并不是被具体蕴含在间隔标签里。用另一种话说，如果顶点能到达一个子树的树根，那么我们仅仅需要去记录其树根节点，而不是去记录在那个子树里面其他节点（因为其已经被树根的间隔囊括了）。所以为了去回答是否能到达，我们将检查的间隔是否被在中记录的间隔有关系。

**2．2．2 Path-Tree简介**

Path-Tree是从Tree Cover方法改进而来。首先对于Tree Cover的方法，计算一个好的Tree Cover方法非常耗时间。对于给定图。构建一个Tree Cover的平均构建时间为。其次，树覆盖不能推广成一些常见形式的DAG，举个例子，比如网格模式的DAG，其中每个节点像图一样和它的上方和右方节点相连接。对于一个 大小的网格，树覆盖能最大覆盖一半边，并且压缩的传递闭包会变得甚至与原来的图相同大小。在这里困难点就是严格的树结构在日常生活中的图中比较少见。从另一个方面，大多数使用树覆盖的方法很容易被没有被覆盖的边所影响性能。

而Path-Tree算法是被提出来解决这些难题的。它创造了一种树形结构，其中每个在树中的节点代表着在原图中的一条路径。这种方法潜在性地给与了双倍的可能性去覆盖有向无环图。 在通常情况下，我们的现实遇到的图一般都是十分稀疏的图，像图边不会比节点数的两倍要大。Path-Tree算法提供了一个更好的覆盖有向无环图。除此之外，Path-Tree使用了一种只含三个元素的标签去回答可达性问题。Path-Tree算法能被在*)* 时间内构建。理论上，Path-Tree算法总是能比Optimal Tree Cover 算法表现的更好。

Path-Tree覆盖是一个在DAG G中树形状的生成子图。在Path-Tree算法下，每个节点拥有一个三元组索引，理论上能在*)* 时间回答可达性查询。一个好的Path-Tree 覆盖可以从中被提取出来用来大幅度减少的传递闭包的大小。

**2．2．3 Path-Tree基本定义**

对于有向无环图 ，其中是一个顶点集合，并且 是边的集合。我们使用来表示从顶点到顶点的边，并且我们使用来表示从到的一条路径。由于是无环的图，所以在一条路径的节点必须是不相同的。如节点可以从节点达到（用 表示），则有一条从出发以结束的路径。

对于一个节点，我们称所有从出发的边为的出边，所有在结束的为的进边。节点v的祖先集合是能到达节点的所有节点集合，记做。同理，后继节点集合记做。的传递闭包是一个这样的有向图：对于每个节点都有一条通向任何后继节点的边。

除此之外，我们说是的子图如果 并且。我们把表达成的一个生成子图，如果它能覆盖的所有节点，也就是。一个树是一个特殊的DAG（每个节点只有一条入边，且除了根节点，其没有任何入边）。一个森林是树的集合。只要加上虚拟顶点指向每一个独立的树，一个森林就能被转换成一个树。为了简化论述，我们把森林和树统称为树。

一个对于图G的Path-Tree覆盖，我们记为，是一个的一个生成子图并且有一个像树一样的形状，其可被记为一个树：每个在图的节点可以被唯一的映射到在中的节点，我们用*表*示这个。对于每条在中的边也可以被唯一的映射到中的单个边，也就是，也可以被映射成一个单独的在*T*中的节点。

**2．2．4 有向无环图的路径分解**

设，是的两条路径。我们使用表示两条路径重合的节点，使用表示至少出现在两边中一边的节点。接下来来定义图分割：

**定义2.2.1：宽度**

在的有向无环图，我们说一个V的分割 是一个图G的路径分解，当且仅当如果，并且 我们同样把称为分解的宽度。

通过这样的分解，我们就可以通过一对ID: 来鉴别出每个顶点。是该节点属于的分割的ID。则是在该路径中的相对距离。对于在一条路径中的两个顶点和，我们使用 来表示在路径中领先于。用公式表达就是:

一个简单的路径分解算法是：首先我们对DAG进行一个拓扑排序。然后我们依次从DAG中抽取路径，如下。我们找到一个节点，在拓扑排序中最小的节点，并把它加入到路径之中。然后我们去寻找，是在图中最小的有到的边的节点。以此类推，我们重复地添加最小的节点到最近加入节点上直到这条路径无法被拓展。然后我们就可以把整个路径从DAG中移除，然后开始提取新的路径。当DAG为空时，整个路径分解完毕。如图2.2所示。

**2．2．5 路径子图和最小相等边集**

考虑边路的两个关系，我们使用 来表示的一个子图，其中包括在和的节点和 （在 中节点有关的边）。我们说 是相连接的，如果不为空。

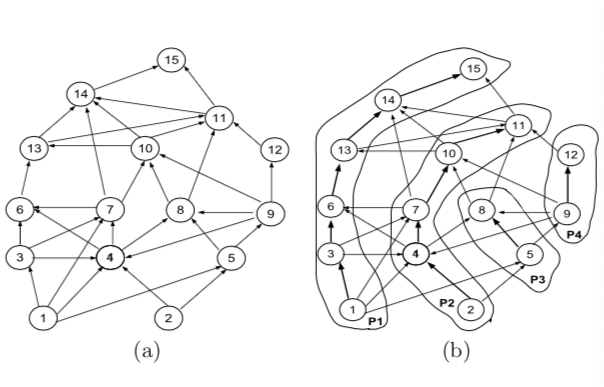
现在对于在 中的一个节点，如果我们想要发现所有在中能到达的节点。（只通过的边）。我们只需要去知道中序列最小的节点就可以。我们把这个最小的的记为。

更简单一些的表示：

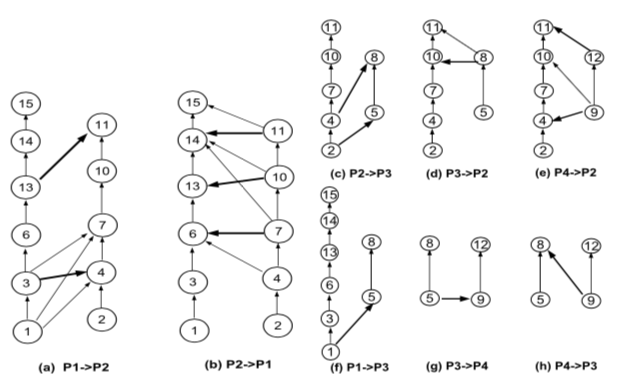
而在中的边可以在不改变可达性的前提下被移除，可以被以下定义：

**定义2.2.2：最小相等集合**

边的集合被叫做的最小相等集合，如果移除在中的任何边就会改变节点集合 的可达性。



图**2.2 一个有向无环图的路径分解**



图**2.3 有向无环图里面的路径关系**

在图2.3(a),对于，是最小相等边集 。在图2.3(b),

**定义2.2.3：平行**

令和是在的两条边。其中且 。我们说这两条边是交叉的如果且。

举个例子和是相交的。给定一个边的集合，如果集合里面没有两条边是相交的，那么这个集合就是平行的。接着我们很自然的可以得出以下两个结论。

* 的边都是平行的
* 对于子图，是唯一的。

下面描述了一个得到最小相等集合 的算法。首先我们把到的所有边依据它们在集合中的结束顶点排序。并让 成为在中的第一个到达的节点。让成为最后一个在的能到达的及诶单。然后，我们添加到 中，并且把所有在中的穿过边的边移除。重复这个操作直到集合变为空为止。

**算法2.2.1：最小集合覆盖**

1：=∅

2：**while** **do**

3： ) {在中第一个能到达中的节点}

4：

5:

6: {把所有穿过的边移除}

7：**end while**

8：**return**

**2．2．5 路径图和路径生成树**

我们对有向无环图创建的一个路径图有如下描述。每个在路径图中的节点对应在图中的一条路径。如果路径与路径相连接，我们就在路径图里创建一条边。让是这个路径图的一个有向生成树（或者是森林）。设是图的子图。首先这个子图包含图的所有路径。其次这个子图拥有最小的边集合，也就是如果包含在中。

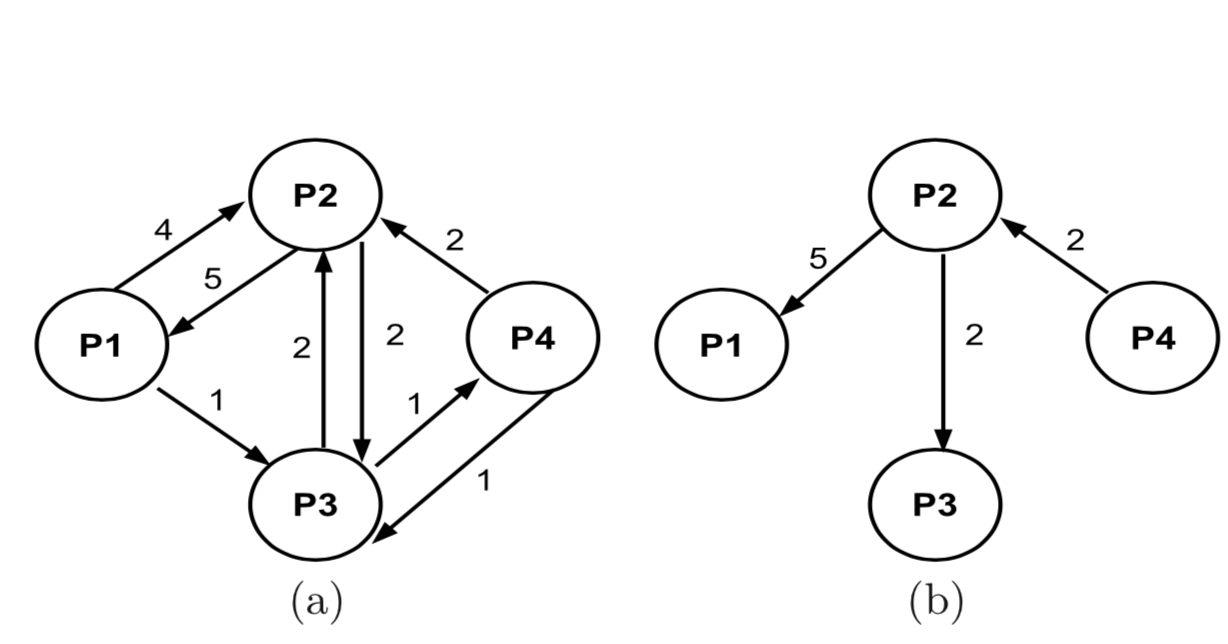
这个被称为图的Path-Tree覆盖，我们可以依靠对做一个向量标签，可以在固定时间内回答可达性查询。

为了去实施Path-Tree覆盖，我们首先需要“记住“还没有被Path-Tree覆盖的边。理想状况下，我们期望去让我们的索引最小化，也就意味着需要去把没有覆盖的边的数量最小化。与此同时，不像Tree Cover一样，我们需要避免去计算先辈集合（计算每个节点的先辈集合与计算整个传递闭包的花费是相等的。）在这一节我们介绍在不知先辈集合的情况下来减少索引大小的两种方法。

第一种标准是被称作。主要的思想是使用Path-Tree去中覆盖尽可能多的边。设为在中的剩余边（没有被Path-Tree覆盖的边）。为的压缩专递闭包提供了一个上界，也就是说每条节点至多要去记录个节点来回答可达性查询。考虑到这个，可以简单的将计做边的花费。

第二种标准是称作，稍微更复杂一些。对于给定的两条路径和它们的连接集合。设 是以节点为结束的的后继。举个例子，节点是中能到达的最后一个节点，并且。我们把权值赋值为的大小。在这个例子中，。

如果我们想要具体获得从到的具体可达信信息的话，这些权重就是我们标签的代价。对于和中的顶点，为了在的顶点和我们只需要去记录在的顶点，为了的顶点和，只需要去记录的顶点和。接着，我们就能回答如果任意在的顶点是否能到达的任意顶点。所以，这样一个带权有向路径图中寻找一个最大生成树就等同于使用路径标签规划把索引最小化。图2.4(a)就是使用规则的带权有向路径图。



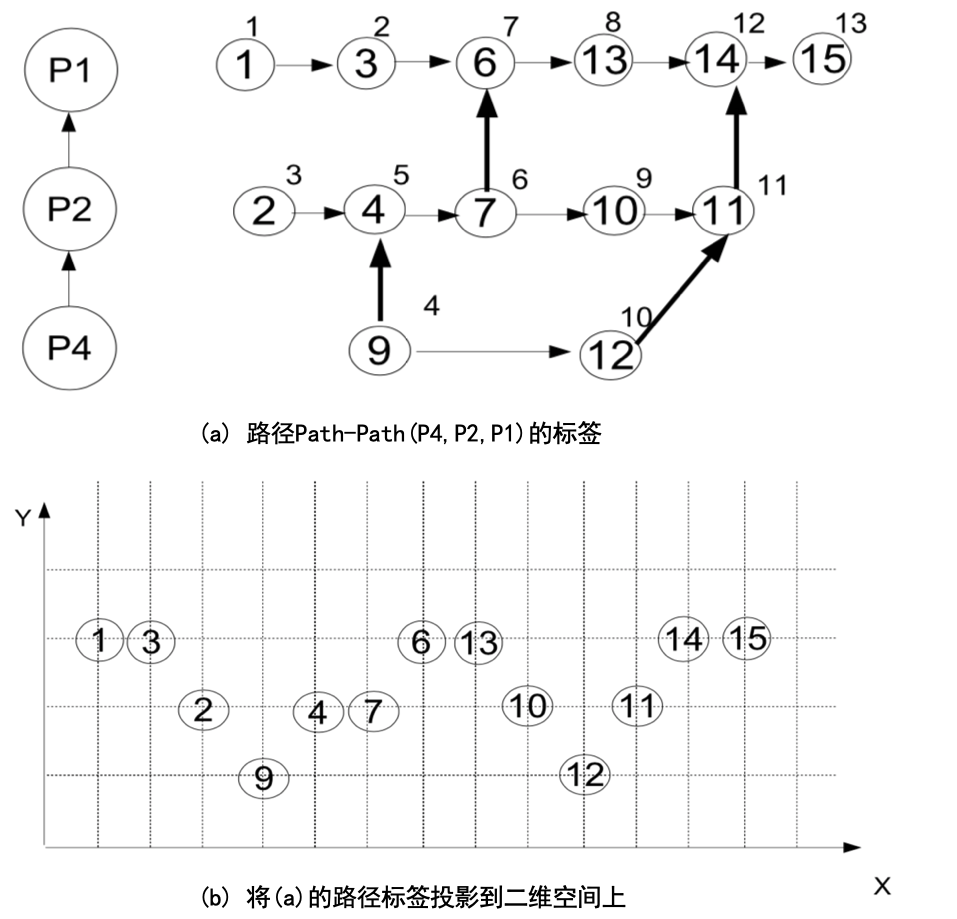
图**2.4 （a）带权有向路径图（b）最大有向生成树**

为了减少路径树覆盖的索引的大小，我们提取出一个最大有向生成树（或者森林）。图2.4(b)是从带权有向路径图 (a) 中提取出的一个最大有向生成树。用Chu-Liu/Edmonds 算法可以来解决这个问题。

**2．2．6 Path-Tree覆盖的可达性标签**

Path-Tree是在最小相等集合和最大有向生成树建立后才构建的。在这一节，我们介绍一种允许我们在常数时间内回答可达性查询的标签策略。我们把Path-Tree分成路径，然后我们叫其path-path覆盖，记做。

从一个简单的方案做起。举个例子，在图2.5（a）中，我们有一个path-path：。我们把在path-path中的每一个节点映射成一个二维空间。首先所有在每条单边的及诶单拥有相同的path ID，我们把这个记为Y。例如顶点P4,P2和P1各自拥有path ID 1,2 和3。然后我们实施一个深度优先算法去给每个节点指定X标签。



**图2.5 单路径Path-Tree（a）path-path的标签（b）映射到二维空间结果**

**算法2.2.2：深度遍历标签**

： 是图G的路径分解

：是被表示成链表。对于 记录了中所有立即可达的邻居节点。对于，如果不是路径的最后一个节点，在链表中的最开始的节点就是在路径中的下一个节点。

1：

2：**for** i = 1 to k **do**

3： { 是路径的第一个节点}

4： **if** 没有被访问过 **then**

5：

6： **end if**

7：**end for**

1：

2：**for each** **do**

3： **if**  没有被访问过 **then**

4：

5： **end if**

6：**end for**

7： {把节点标记为}

8：

对于标签步骤，在每个节点的链表中，我们总是把它在同一个路径的邻居比其他路径的邻居要前面。这样允许我们去优先访问在同一路径的邻居，而不是其他路径的邻居。

在DFS搜索中，我们将维护一个计数器，它用来记录在图中最初有多少个顶点。（例如在上图，）。DFS开始于在第一条路径的第一个节点。在图中这个例子中，是从路径的节点开始的。从这个节点开始， 我们的DFS总是尝试去访问它的右邻居然后去访问上邻居。对于一个给定的节点，当我们完成访问它的所有邻居后，我们就把这个节点赋予标签并且把减少一。当我们访问完所有的能从的节点，我们就开始在第二条路径的第一个节点开始访问（如果他没有被访问过。）重复这个步骤直到所有的节点被访问过。图4(a)显示了根据DFS遍历后的X标签，4(b)显示了依据X，Y标签的二维空间。

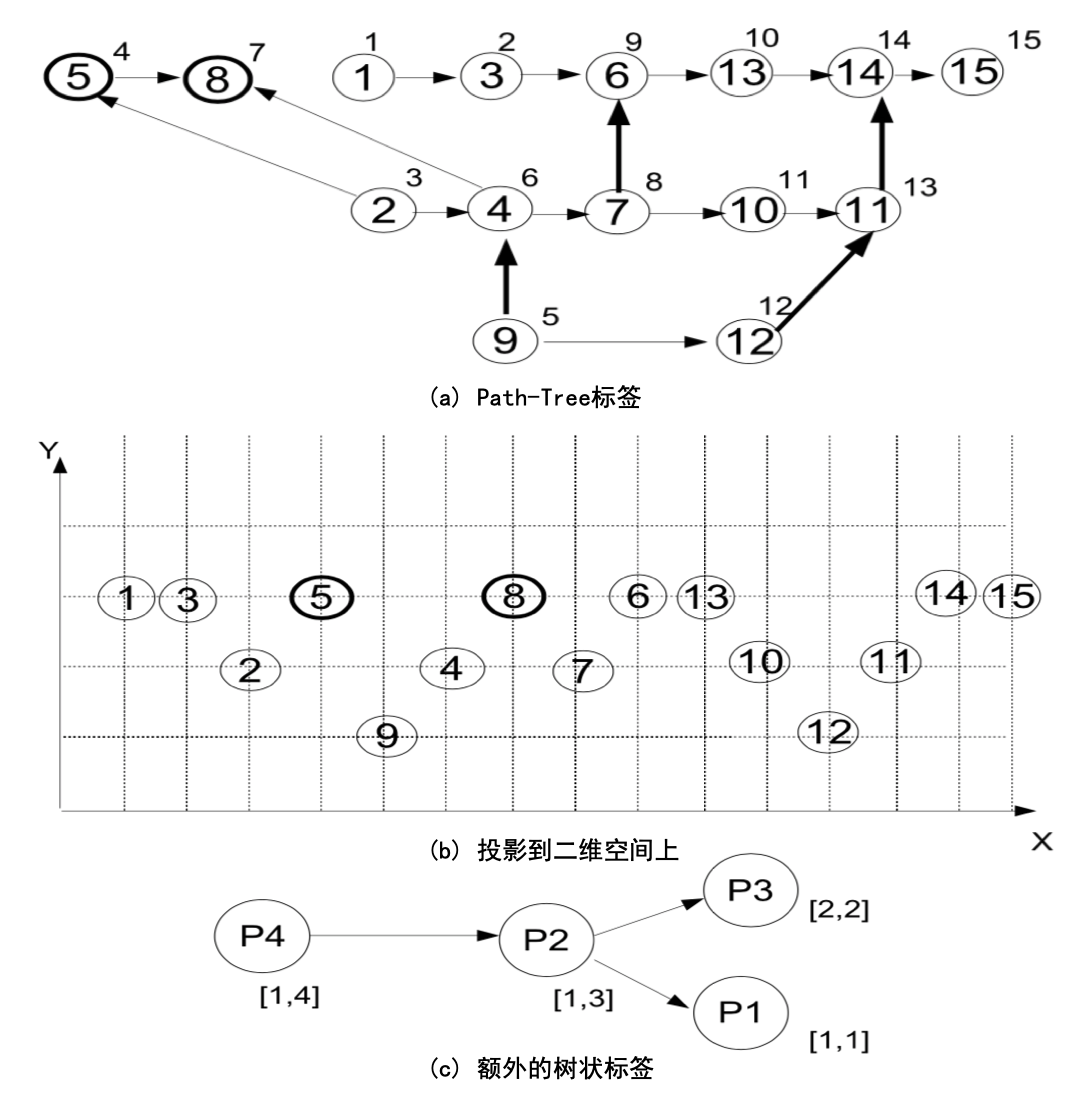
于是我们自然地可以得出以下两个结论:

* 对于在path-path中给定的两个节点和，能到达当且仅当
* 对于在path-tree中给定两个节点和，能到达当且仅当 1) 且 。 2) ，和是和对应的内部标签。

对于第一个结论证明如下：我们首先证明 。显然，当能到达的时候我们有（path-path的性质）。然后DFS访问会比首先访问，只有其在访问完所有的邻居才会访问。 的证明类似，在此省略。

对于更加通常的情况：我们在树中有多条路径，而不是只有一个路径。在这种情况下，每个节点就需要有一个依据于树结构的额外间隔标签对。图5 展示了在中的path-tree的间隔标签对。一条路径的所有节点在同一条路径拥有同一个间隔对。除此之外，对于每个节点的Y标签被概括成在树路径下的路径层级（从根节点到路径的距离，我们假设有一个虚拟根节点连接着所有树的根节点。）X标签和简单path-path标签相似。唯一的区别就是每个节点有一个上界限邻居。除此之外，我们要依据路径在path-tree中的层级进行遍历。图2.6(a) 显示了所有在path-tree中的所有节点的X标签，图(b)2.7显示了映射到二维空间上的结果。

对于第二个结论的证明如下：首先当能到达的时候，有。这是一句DFS步骤得到的。假设能到达，然后就在树种有一条从的路径到的路径的路径。所以我们会得出（依据树的标签）和*。*除此之外，如果有，就说明有一条从的路径通向的路径。



图**2.6 完整Path-Tree（a）path-path的标签（b）映射到二维空间结果（c）额外的树标签**

假设间隔以格式表达。我们有下面这样的结论，对于Path-Tree中的任意一个节点，一个三维向量可以去回答在path-tree中的任何可达性查询。

证明如下：首先，当，那么。于是我们就可以把*Y*标签给丢弃，因为其与间隔标签携带的信息重复了。对于任何节点，我们有它的(第一维)和(最后两维度的间隔对)。

Path-Tree的间隔标签算法和path-path很相似，有两步：

从带权有向路径图中为最大有向生成树创建一个树标签（通过Edmond’s 算法）

设,是在最大有向生成树种层级i的节点的集合（路径），其有层。用调用算法。

Path-tree覆盖总体的构造时间如下。第一步路径分解是,同时也包括了拓扑排序的时间。第二步构建带权有向路径图的时间是。第三步，提取最大生成树的时间为*)*，其中。第四步标记，主要是用DFS的方法，花费为，其中是在path-tree中边的数目。所以综上总共构建的path-tree覆盖的时间为*)*。

**2．2．6 传递闭包压缩以及可达性查询回答**

没有被path-tree覆盖的边会导致额外的查询开销。为了对整个DAG做一个可达性查询，一个简单的想法是去为没有被path-tree覆盖的边构建一个传递闭包。依据Dual Labeling，构建时间花费是 ，而索引大小为，其中是非path-tree的边。

此外，如果我们需要一个最大的传递闭包，path-tree结构可以极大地帮助我们减少索引大小和构建时间。设是我们为节点的传递闭包而构建的压缩节点集合。假设是一个在的节点。如果要去回答和之间的可达性关系。我们需要去进行两步测试。1）如果能依据path-tree标签到达的话，就返回否则：2）对于每个在的，在path-tree上测试是否能从中访问得到。在这里我们注意，在最坏情况下，包含了每个路径的至多一个节点，，k是path-tree中的路径数目。所以一个查询最多花费O(k)。

算法3为每个节点构建了一个压缩传递闭包。构建时间是，因为对每个节点，

**算法2.2.3：压缩传递闭包**

1： 对G的逆序拓扑排序 {对G先做一拓扑排序}

2：

3：**for** i=1 to N **do**

4： ;

5： S为在G中的的直接邻居节点集合

6： **for each** **do**

7： **for each** **do**

8： **if** 在中不能到达 **then**

9： 把加入到

10： **end if**

11： **end for**

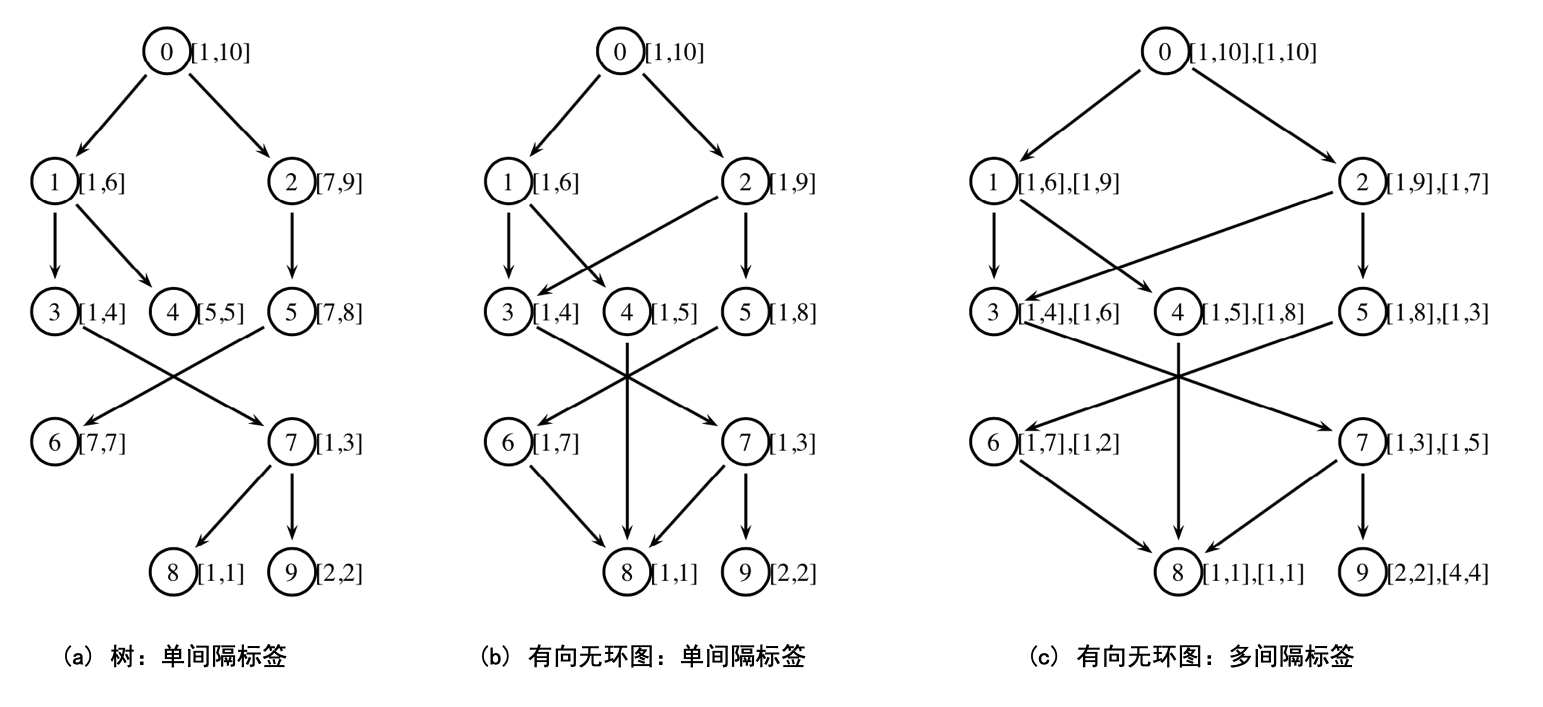
12： **end for**

13：**end for**

**2.3 改良在线搜索法：GRAIL**

GRAIL是一种基于随机间隔标签（randomized interval labeling）标签的随机搜索方法，被用于在非常大的图查询。虽然前面叙述的方法在小图上比GRAIL性能要好，但是GRAIL是这里面唯一能扩展到几百万节点和边的方法。GRAIL使用了多个从随机图遍历获得的标签。

首先来定义一下间隔标签(interval labeling)。这个概念在之前提到过。间隔标签把每个节点都赋予一个范围，表示节点在树的后序遍历的排序。（排序从1开始。）一个节点的所有孩子节点在那次遍历后会被认为是排好序并且不变的。则表示在遍历中遇见的最小的排序顺序（任意以为根节点的树的子树中的节点）。这个方法保证了在树图中间隔标签的方法是和节点之间可达性查询关系是一样的，因为后序遍历会在访问该节点前访问所有它的子节点。于是乎我们有：



**图2.7间隔标签图（a）树；（b）DAG有向无环图，单间隔；（c）多间隔**

举个例子，在图2.7(a)就显示了一个树的间隔标签，假设孩子节点的排序是从左到右。很容易就发现可达性可以被间隔标签所回答。例如，因为，但是，因为。

把间隔标签推广到DAG，我们保持两个条件，首先我们要保证一个节点只被至多访问一次并且一个节点需要保持维持一个后续遍历的排序。但是间隔标签在DAG中并不意味着可达性。在图2.7(b)中，举个例子，但是。但从另一个角度我们可以得到。

而我们这里要介绍的GRAIL算法使用了多个间隔标签来辅助查询。我们使用来表示每个节点拥有的间隔标签数。GRAIL的核心是去对那些不可达的节点对，利用多个标签对来达到一个快速拒绝可达性的效果。也就是，如果，这个能在的时间内完成，我们就可以马上返回。我们索引的空间复杂度是，构建时间是。对于回答可达性查询，时间复杂度从 （这种情况单靠标签就可以进行不可达的判断）到（这种情况下采用普通的DFS）。

**2．3．1 GRAIL方法**

之前介绍了像path-tree的方法，他们都是在第一步识别出DAG的一个子图，然后第二步在剩余没有覆盖的DAG部分结合前面子图进行查询。但是其实大部分的可达性信息已经在第一步已经被挖掘出。GRAIL的思想是多次使用间隔标签（interval labeling）去减少第二步的构建索引和查询的时间。

在GRAIL中，对于一个给定的节点，我们赋予它一个多维标签：

。其中是对DAG进行第次（随机）遍历得到的间隔标签，且，其中是标签的维度。我们定义，当且仅当对于所有的，有。如果，那我们就得出结论说不能到达。

在另一个方面来看，如果，并不意味着一定可以到达，有可能是False Positive，也就是不可到达的情况还是有可能发生的。我们把这种False Positive的情况叫做“意外”（Exception）。举个例子，在图2.7（b）中，总共有15个意外（在表2.8列出）。例如对于节点，节点就是一个意外，因为。但实际上不能到达。GRAIL的动机就是运用多个随机标签让这种不正确的包含关系（就是意外）降到最低。对于图（c）来说，对于同样一个图，15个意外中的12个可以被消灭掉。在这个图中，节点就不是节点的意外了，因为。但也还有是意外的存在，节点依然是节点的意外。。GRAIL的存在显然是可以很大概率降低后者这种意外的。

|  |  |
| --- | --- |
| 节点 | 意外(E) |
| 2 | {1,4} |
| 4 | {3,7,9} |
| 5 | {1,3,4,7,9} |
| 6 | {1,3,4,7,9} |

**表2.8在图2.7（b）的“意外”**

GRAIL我们接下来讨论两个点：1）如何计算一个d维的随机间隔标签。2）当进行可达性查询时如何处理意外。

**2．3．2 索引构建**

GRAIL的索引构建步骤很直接。我们的不同的后序遍历就是简单的随机改变访问孩子节点的顺序。下面展示了一种算法。一个间隔 用以下表示：

虽然可能的标签数目是成指数膨胀的，但是大多数图都能被很少维的标签给覆盖。因为没有任何保证使意外完全被消灭，最好的策略就是在一个不太大的维度标签后就停止标签（例如5）。如果要完全消灭所有的意外会使标签的维度变得太大。

**算法2.3.1：Grail索引-随机间隔**

1：**for** i=1 to d **do**

2: {全局变量，节点的排序。}

3：

4： **for** **do**

5：

6： **end for**

7：**end for**

8：**if** x以前访问过**then return**

9：**for** **do**

10：

11：**end for**

12：

13：

14：

在遍历的策略中，我们的目标是尽可能地是生成与其他标签不同的标签。下面有三种生产策略。

* 完全随机：这是在上面算法展示的步骤，在每维进行随机遍历。
* 随机对：首先把根节点和孩子节点的顺序给随机化，然后修改它。首先生成标签对，然后使用从左到右和从右到左的遍历。这样做的目的是使间隔标签尽可能的不一样；一个在从左到右首先访问的节点在从右到左的时候是最后访问的。
* 自底向上：与之前按照拓扑排序来处理从根节点到底部不同，这个策略我们故意把边“倒置”，并且把节点按照反向拓扑排序处理。通过这个变化，。

显然地，GRAIL的索引构建花费，对应在图中的次遍历。除此之外，空间复杂度为。

**2．3．2 GRAIL可达性查询**

为了去回答在节点和之间的可达性查询。GRAIL采取了一种双交叉的方法。GRAIL首先检查是否。如果是这样，我们就马上得到结论不能到达。在另一个方面，如果，不能得出任何结论，因为我们知道标签可能会有false positives，也就是意外。

我们两种方法来处理意外。第一种是明确的维护一个每个节点的意外列表。给定一个节点x，我们把这个列表记为，定义为：

举个例子，对于在图2.7（b）的DAG，我们注意在这里总共有15个意外。 如果每一个节点有一个明确地意外列表，那么只要我们知道了，我们要做的事情就是检查是否。如果是，那么是一个意外，我们返回不能到达。如果否，那么这个不是意外，我们回答能到达。

但是对于每个节点维护一个明确的意外列表在空间和时间方面代价很大，同时也不能扩大到很大规模的图上。所以GRAIL的原本方法不去维护一个意外列表。GRAIL可以使用一种聪明的DFS策略，一种基于剪枝的递归包含检查。这个方法不需要去计算意外列表所以其构造时间和索引大小都是线性的。

**算法2.3.2：Grail可达性查询**

1：**if** **then**

2： **return False** {u不能到达v}

3：**else if** 使用意外列表==true **then**

4： **if** **then return** False {u不能到达v}

5： **else** **return** True {u能到达v}

6：**else**

7： **for** **do**

8： **if** **then**

9： **return** True {u可达v}

10： **end if**

11： **end for**

12：**return** False

以上是GRAIL的可达性查询的算法。值得注意的是，在第三个分支条件里进行了一个递归的DFS搜索。如果存在节点的孩子节点符合的条件话，我们就检查并查找到的可达性。我们于是就能确定达到，并且GRAIL返回true。否则，如果节点没有孩子能到达，那么我们就断言不能到达。举个例子，我们考虑在图2.7（b）单间隔标签。令并不使用意外列表。因为，我们去用DFS来决定可达性查询。和都是的孩子，但是只有满足。我们接着检查是否能到达。采取DFS递归策略，我们接下来会检查然后最终得到不能到达。所以我们最终返回的结果是False。

关于GRAIL计算复杂度的话，当时候，查询时间是。如果运用意外列表的话，接下来的查询只需要到。最初的选项是做一个DFS，但是注意DFS可能会提前结束，因为我们有剪枝策略。所以最坏的复杂度为。但在实际中，它比这个要快很多，取决于的拓扑逻辑结构和剪枝的效率。所以综上，查询时间在到

3 Neo4j与数据集

**3.1 Neo4j图数据库简介**

Neo4j是一个Java实现的开源的图数据库平台，是基于图论实现的一种新型NoSQL数据库。Neo4j现如今已经被各种行业的数十万家公司和组织采用。Neo4j的使用例涵盖了决策制定、网络管理、科学研究、路由分析、组织和项目管理、社交网络等诸多方面。Neo4j的数据存储结构和数据查询方式都是以图论为基础的（而不是像其他数据以行、列为单位存储的。）图论中图的基本元素为节点和边，在图数据库中对应的就是节点和关系。图数据库使用图中的模型来存储关系和数据。

为什么要使用图数据库？至今，大多数数据以关系的形式存在于不同对象之中。它们中更多的是，关系比对象本身更加重要。传统的关系型数据库存储高度有组织的数据，它们把相同类型的数据存在一起，但是并不存储数据与数据之间的关系。而不像其他数据库一样，图数据把关系与联系当作首要存储载体。

Neo4j图数据库基本的元素与概念与现实图一一对应。在这里，我们主要就用到其节点和关系和路径。

节点（Node）是图数据库中的一个基本元素，用以表示一个实体记录，就像关系数据库中的一条记录一样。在这里我们给每个节点取名有其独特的ID

关系（Relationship）也是图数据最根本的组成成分，其相当于图中的边。在研究可达性查询的方面，我们相当于在两个节点之间用一条单向路径为1的边连接。

路径（Path）。当一个图由节点和关系构成的时候，对于两个节点，之间可能存在由节点和关系组成的路径。

Neo4j现如今已经被各种行业的数十万家公司和组织采用。Neo4j的使用例涵盖了决策制定、网络管理、科学研究、路由分析、组织和项目管理、社交网络等诸多方面。

**3．1．1 Neo4j遍历框架Traversal**

Traversal框架是Neo4j用来遍历节点的框架。是一个基于回调，惰性执行的节点遍历框架。

Traversal框架除了由之前提到的Node和Relation组成，还主要有Relationships，Evaluator，Traverser，Order，BranchSelector，Path和PathExpander组成。

Relationships是指遍历的不同关系。默认的Relationships列表是空的，所以会遍历所有关系。我们可以指定需要遍历的节点Relationship类型到列表之中。

Evaluator用于在遍历节点的时候，是否要继续遍历以及是否要把节点假如到结果集中。

Traverser对象是调用traverse()具体动作之后遍历之后的结果，其中包含目前遍历在图中的具体位置。

Order和BranchSelector是用来决定在遍历的时候有多个条分支到底先走哪条分支。像Neo4j自带的选择规则有先序深度优先、后序深度优先、先序宽度优先、后序宽度优先。

Path是Traversal框架接口的一部分。Traversers返回的结果就可以是一个Path，同时Path也是在遍历时候重要的参考对象。

PathExpander则是Neo4j用来规划扩展路径的一个抽象接口。

Traversal框架高度集成在Neo4j内，是其用于遍历节点的主要方法。但是在我们这里的算法并不适用于用Traversal框架实现。原因如下。首先Traversal框架节点遍历方法比较局限，基本只在深度遍历和宽度遍历之间，定制一些独特的遍历逻辑比较实现比较困难。其次，我们的算法需要在原图上构建有一定体积的索引，如果用Traversal框架实现的类要重复构建若干个全新的子图——Neo4j里面节点和关系的实现有很多多余的类成员，而我们的索引子图并不需要那么多，会大大加大额外的内存开销。另外Traversal框架在遍历时候会进行许多范围检查，比较笨重，不如直接遍历效率高。我们选择直接在Neo4j原图的Node和Relation上构建我们自定义的新子图。

**3.2 实现环境与数据集**

**3．2．1 实现环境**

实现环境为处理器2.7GHZ 4核Intel Core i7，内存16GB。Java为JDK8，IDE为Intellij idea。Neo4j版本为3.4.6 Community版本，操作系统为macOS Mojave 10.14.3。我们主要对比四种算法，2Hop，Grail，PathTree以及Neo4j自带的PathFinder算法。

**3．2．2 数据集**

所有的查询时间是100K次查询之和。我们生成100K随机查询对，并对所有查询使用相同查询对。

我们使用多种来源于真实数据集的图，有偏小与偏大的数据集。

**稀疏图：**这些图比较小，节点的平均度数小于1.2。从[15]来，并且在表3.1列出。Xmark和nasa是XML文档，amaze和kegg来自于生物中的代谢网络。其余的是从BioCyc（biocyc.org），一个关于基因路径的数据库网站，收集的。Amaze和kegg在结构上与其他有一些不同，因为他们都有一个有非常大入度和出度的中心节点。

**稠密图：**这些图比较稠密，也是取自于真实世界于arxiv，citeceer和pubmed。都是有关于应用的图数据集。GO是GeneOntology图的子图。Yago是语义网络数据库YAGO的子图。

|  |  |  |  |
| --- | --- | --- | --- |
| 数据集 | 节点数 | 边数 | 平均度数 |
| agrocyc | 12684 | 13657 | 1.07 |
| amaze | 3710 | 3946 | 1.06 |
| anthra | 12499 | 13327 | 1.07 |
| ecoo | 12620 | 13575 | 1.08 |
| human | 38811 | 39816 | 1.01 |
| kegg | 3617 | 4395 | 1.22 |
| mtbrv | 9602 | 10438 | 1.09 |
| nasa | 5605 | 6538 | 1.17 |
| vchocyc | 9491 | 10345 | 1.09 |
| xmark | 6080 | 7051 | 1.16 |

**表3.1稀疏图数据集**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据集 | 节点数 | 边数 | 平均度数 |
| arxiv | 6000 | 66707 | 11.12 |
| citeseer | 10720 | 44258 | 4.13 |
| go | 6793 | 13361 | 1.97 |
| pubmed | 9000 | 40028 | 4.45 |
| yago | 6642 | 42392 | 6.38 |

**表3.2稠密图数据集**

**3．2．3 衡量标准**

我们将从三个方面衡量我们的算法。首先就是索引构建时间。过于长的索引构建时间会让算法在更大节点的图上性能更差。接着，就是我们非常关心的查询时间，查询时间是我们研究可达性查询的最关注的一点。其会占我们研究可达性查询的非常大一部分比值。最后，索引大小也是我们关心的重要一点，索引的大小必须足够小让我们能把之完整的装进内存之中。

关于Grail的参数，由于作者在原Paper中提及Grail在随机标签下表现的足够好，其余方法提升作用不大。所以我们所有关于Grail的实验都使用随机标签。除此之外，对图维护一个意外列表的开销实在太大，所以我们默认使用Grail的深度优先遍历加上剪枝策略。

而关于Path-Tree我们使用MinPathIndex标准来构建。

首先我们将数据集通过Neo4j native java api导入进Neo4j数据库。在maven之中配好neo4j的drive之后，我们编写代码将其导入进Neo4j。

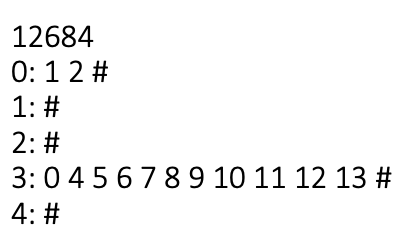
基本思路是将Neo4j的Node再从外面抽象出一层具体算法的Node，用于不同算法的具体Node。

4 实验

**4．1 Neo4j 数据导入**

Neo4j基于Cypher声明式查询语言来具体进行图数据信息查询的。而Cypher语言具有通过模式匹配图数据库中节点和关系，来获得信息以及更新数据。Cypher语言的对于Neo4j就像SQL语句对于关系式数据库一样重要。然而在我们的实验中，由于从最底层增加Cypher来抽象可达性函数带来的工作量太大，我们直接从Neo4j提供的最基本的关系与节点API入手。

在用Java依赖仓库管理工具maven导入Neo4j的dependencies后我们便可以开始导入数据图。我们的数据源图的格式如图4.1。



**图4.1数据部分样本**

第一行为图节点数，接下来每一行第一个数为节点的编号ID（从0开始）冒号后面是其连接的节点编号ID。例如0:1 2#表示着我们从节点0到节点1有一条边，从节点0到节点2也有一条边。

Neo4j用原生Java Api构建数据十分方便，在使用数据库新建类GraphDatabaseFactory的工厂类，选择方法工厂类的newEmbeddedDatabase即可在指定路径新建一个Neo4j数据库的实例。

由于Neo4j是支持ACID的，我们要在数据库里插入数据也是相当于一个事物。我们在Java代码里将我们插入数据的过程用Neo4j的Transaction类包裹起来。

具体插入过程是调用我们刚刚用newEmbeddedDatabase方法创造出来的GraphDatabaseService类。这个类相当于我们管理的数据库的实例，我们可以通过这个类访问到图的所有信息。