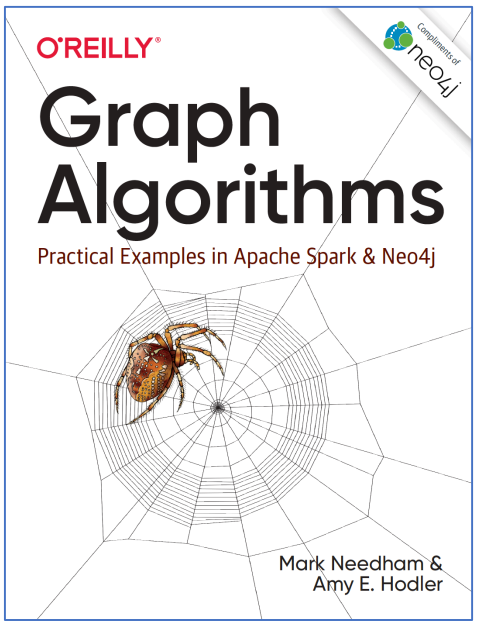


图算法

Apache Spark和Neo4j中实用案例

Graph Algorithm

Practical Examples in Apache Spark & Neo4j

Mark Needham & Amy E. Hodler

杨志宝 冯庆杰 译

本翻译文本，以技术和算法思想的交流为唯一目标。

更多算法和医疗内容
关注微信公众账号
MEDICAL与AI的故事

更多Neo4j内容
请关注
Neo4j中文社区

目录

(数据科学家)-[:热爱]->(Neo4j)

强大 实用 智能

Neo4j是处理数据网络（connected data）排名第一的平台，它同时具备图分析、存储和处理能力。Neo4j中的图算法展示了数据中隐藏的模式，并能增强机器学习的预测能力。

序(Preface)

世界是由连接(connections)驱动的，从金融、通信系统、社会过程到生物过程。揭示这些连接背后的含义，会推动各行业上的突破，比如识别欺诈团伙、对团队能力评估建议进行优化、和预测级联故障等。

因为图算法是基于数学而发展起来，它通过数据间的关系来获得知识见解，随着数据间连接越来越快，大家对图的兴趣也是与日俱增。图分析能够在大的规模上揭示复杂系统和网络的工作原理，这适用于任何组织。

我们对图分析的效用和重要性充满了热情，也体验到揭示复杂场景的内部工作原理所带来的乐趣。直到最近，应用图分析仍然需要大量的专业知识和决心，因为工具和集成并不容易，很少有人知道如何应用图算法到他们的困境中去。我们的目标是帮助改变这种状况。我们写这本书是为了帮助各组织更好地利用图分析做出新的发现，并更快地开发智能解决方案。

本书的内容

本书是给那些使用Apache Spark和Neo4j的开发人员和数据科学家用的图算法实用指南。虽然我们的算法示例是利用 Spark和Neo4j平台，但无论您选择什么图形技术，本书也将有助于理解更通用的图概念。

本书前两章介绍了图分析、算法和理论。第三章简要介绍了本书中使用的平台。之后的三章，专注于经典图形算法：路径查找、中心性、和社区检测。最后的

两章说明如何在工作流程中应用图形算法，一章是一般性的分析，另一章是机器学习。

在每个算法类别的开头,都有一个参考表,可帮助您快速索引到相关算法。对于每种算法，都会有:

- 关于算法作用的说明
- 算法的用例，并告诉你何时使用该算法
- 在Apache Spark或Neo4j上的示例代码

本书中使用的约定

本书使用了以下排版约定:

Italic 指示新术语、URL、电子邮件地址、文件名和文件扩展名。

`Constant width` 用于程序列表，以及段落中引用程序元素。例如变量或函数名称、数据库、数据类型、环境变量、语句和关键字。

`Constant Width Bold` 显示应由用户逐字键入的命令或其他文本。

`Constant Width Italic` 显示应替换为用户提供的值或由上下文确定的值。

意味着，这里是提示或建议。

意味着，这里是常规注释。

意味着，这里是警告，需要特别注意。

使用示例代码

补充材料(代码示例、练习等)可在以下网址下载:https://bit.ly/2FPgGVV。

这本书是帮助你来完成工作的。通常，如果提供了书中的示例代码，您可以在程序和文档中使用它。您不需要联系我们获得许可，除非您正在复制代码中很重要的部分。例如，编写一个程序，该程序使用此书中的代码块，不需要许可。销售或分发O’ Reilly书中的示例光盘是需要得到许可的。引用此书籍和示例代码来回答问题，不需要许可。将本书中的重要部分的代码纳入到你的产品文档中，需要许可。

如果你在使用示例代码时指明了引用，我们会很欢迎，但是我们不做强制性的规定。

引用通常包括标题、作者、出版商和 ISBN。

例如:"Graph Algorithms by Amy E. Hodler and Mark Needham (O’Reilly). Copyright 2019 Amy E. Hodler and Mark Needham, 978-1-492-05781-9."

如果您觉得使用代码示例超出了以上合理使用或授予的权限，请随时联系我们permissions@oreilly.com。

O’Reilly在线学习

O’Reilly在40年来提供技术和业务的培训、知识和智慧，帮助许多公司取得成功。

我们独一无二的专家和创新者的网络，通过书籍、文章、会议和在线学习平台来分享他们的知识和专业技能。O’Reilly的在线学习平台提供随需所取的在线培训课程、深度学习路径和交互式代码环境，以及O’Reilly和其他200多出版商提供的大量的文字和音视频内容。请访问http://oreilly.com，获得更多的信息。

如何与我们联系

请将关于这本书的评论和疑问发给出版商：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

我们有一个关于本书的网页，你可以访问http://bit.ly/graph-algorithms 来获得勘误、示例和任何额外的信息。

关于技术问题的评论和询问，请给bookquestions@oreilly.com这个地址发送邮件。

希望获得更多的关于我们的书籍、课程、会议和新闻，请访问我们的网站 http://www.oreilly.com.

我们在facebook上的位置 http://facebook.com/oreilly

我们在tweeter上的账号 http://twitter.com/oreillymedia

在Youtube上的内容 http://www.youtube.com/oreillymedia

致谢

我们非常喜欢把这本书的内容整理在一起，并感谢所有那些协助的人。我们特别要感谢Michael Hunger的指导，Jim Webber的宝贵编辑，Tomaz Bratanic的热心研究。最后，我们非常感谢Yelp允许我们将其丰富的数据集用于功能强大的示例。

前言(Foreword)

营销因素分析、反洗钱分析(AML)、客户旅程建模、安全事件因果因素分析、文献发现、欺诈网络检测、互联网搜索节点分析、地图应用创建、疾病聚类分析、威廉莎士比亚戏剧表演分析，所有的这些都共同点。你可能已经猜到，这些事物的共同点是使用了图。这证明莎士比亚是正确的，他曾经宣布，"整个世界都是一个图(graph)!

好吧，开个玩笑，这个爱芬河的游吟诗人（指的是莎士比亚）其实并没有在这句话上写上图（graph），他用的词是舞台（stage）。但是，请注意，上面列出的示例都涉及到实体和它们之间的关系，包括直接的和间接的关系。实体是图中的节点，可以是人员、事件、对象、概念或者地方。节点之间的关系是图中的边。因此，莎士比亚的戏剧的精髓，难道不是实体（the Nodes，节点）和它们之间的关系（the edges，边）的写照吗？因此，也许莎士比亚可以在他的注明宣言中写下“图”（graph）这个字。

使得图算法(graph algorithm)和图数据库(graph database)变得有趣而且强大的，不是两个实体之间的简单关系：A 与 B 相关。毕竟，标准的关系型数据库（RDBMS）在几十年前，在实体关系图（Entity-Relationship Diagram, ERD）中已经能够实例化这些类型的关系。所以，使得图（graph）如此重要的是有方向的关系和可传递的关系。在定向关系中，A 可能导致 B，但反过来就不行。在可传递关系中，A 可以直接(directly)与 B 相关，而 B 可与 C 直接相关，A和C并不直接相关，因此A与C间接（transitively）相关。

通过这些可传递的关系，图模型揭示了实体之间的关系，尤其是当它们数量众多、多种多样，且具有许多可能的关系/网络模式/实体间的离散程度的时候。而在关系型数据库中，这种情况就会被看成断开、不相关，或者干脆就无法被发现的。因此，图模型可以有效地应用于许多网络分析用例。

考虑这个营销归因用例：一个人A看到了某个市场营销活动，A在社交媒体中谈论它；B这个人和A之间有关系，并且B看到了这个评论。之后，B购买了市场营销活动所推荐的产品。从市场营销活动经理的角度来看，标准关系模型无法识别归因，因为B看不到市场营销活动，而A未对市场活动做出直接响应。市场活动看起来像是失败的，但实际上它是成功的，因为ROI（投入产出比）是正的！它是通过

市场营销活动和最终客户购买之间的非直接（transitive）关系,也就是通过一种中介(中间的实体)来完成的。

接下来,考虑反洗钱分析案件：两个人A和C都涉嫌非法贩运。有关部门将标记两者之间的任何互动(例如，金融数据库中的金融交易)，并严格审查（scrutinize）。

但是,如果A和C从未一起交易业务,而是通过安全、受尊重和未被标记的财务机构B进行金融交易,那么该交易又能获得什么结果呢？这要看图分析算法！图引擎通过中间人B发现A和C之间的可传递关系。

在互联网搜索中，主流搜索引擎使用基于图的超链接网络的算法来查找任何给定搜索词集在整个互联网上的中央权威节点。在这种情况下，关系（边）的方向性至关重要，因为网络中的权威节点是许多其他节点指向的节点。

文献发现（Literature-based discovery, LBD)：基于图的知识网络应用程序支持在数千(甚至数百万)研究期刊文章的知识库中进行重大发现，” 隐性知识（hidden knowledge）"只能通过已发表的研究成果之间的连接来发现，其间可能有许多的非直接关系。LBD正被应用于癌症研究，其中大量的语义医学知识库的症状、诊断、治疗、药物相互作用、遗传标记、短期结果和长期后果，可能是” 隐藏的” 而且在之前未知的治愈手段，或对疑难杂症病例的有益治疗。知识可能已经在网络中，但我们需要连接各个 “点” 来找到它。

以上通过图算法进行网络分析，对于其他上面列举到的用例，可以给出类似的图功能描述。每个案例都深深涉及实体(人、对象、事件、行动、概念和地点)及其关系(接触点、因果关联和简单关联)。

在考虑图的强大功能时,我们应该记住,对于实际用例,图形模型中最强大的节点可能是"上下文"。上下文可能包括时间、位置、相关事件、邻近实体等。因此,将上下文合并到图形中，作为节点或作为边，就会产生令人印象深刻的预测性分析和规范分析的能力。

Mark Needham 和 Amy Hodler的《Graph Algorithm - Practical Examples in Apache Spark & Neo4j》(本书)旨在围绕这些重要的图分析类型，包括算法、概念、算法在机器学习上的实际应用，来扩展我们的知识和能力。从基本概念到基本算法，从处理平台和实际用例，作者为图的精彩世界编制了一份具有启发性和说明性的指南。

— Kirk Borne博士
首席数据科学家，常务顾问

Booz Allen Hamilton公司
2019年3月

第一章 图的基本介绍

图是计算机科学的统一主题下的分支之一，这是一种抽象的表示形式，它描述了运输系统、人机交互和电信网络的组织。这么多不同的结构可以用单一的形式来建模，这给图程序员赋予了强大力量。

—The Algorithm Design Manual, by Steven S. Skiena (Springer),
Distinguished Teaching Professor of Computer Science
Stony Brook University

当今最紧迫的数据挑战，都集中在关系上，而不仅仅把离散的数据进行表格化（tabulating discrete data）。图技术和分析为用于研究、社交行动和业务解决方案提供了强大的工具，比如:

- 用于从金融市场到IT服务的各类动态环境的建模
- 预测流行病的蔓延以及服务延误和中断
- 找到机器学习的预测性特征，以打击金融犯罪
- 发现个性化体验和推荐模式

随着数据日益互联,系统日益复杂,能否利用数据中丰富且不断发展的关系至关重要。本章介绍图分析和图算法。首先,我们将简要复习图的起源,然后引入图算法并解释图数据库和图处理之间的区别。我们将探讨现代数据本身的本质，也就是：数据网络的分析结果能够比基本统计方法的结果要精巧得多。本章最后将介绍可以使用图算法的一些用例。

什么是图？

图的历史可以追溯到1736年,当时Leonhard Euler（著名数学家欧拉）解决了"哥尼斯堡七桥问题”。这个问题是这样的，一个城市的所有四个区域由七座桥梁连接，问是否有路径能够访问整个城市，但只跨越每座桥梁一次。答案是没有这样的路径。随着思考的深入，欧拉发现只有连接本身是相关的，他因此建立了图理论及

其数学的基础。图1-1描绘了Euler的进展与他的原始草图之一,这些图来源于论文“Solutio problematis ad geometriam situs pertinentis”。

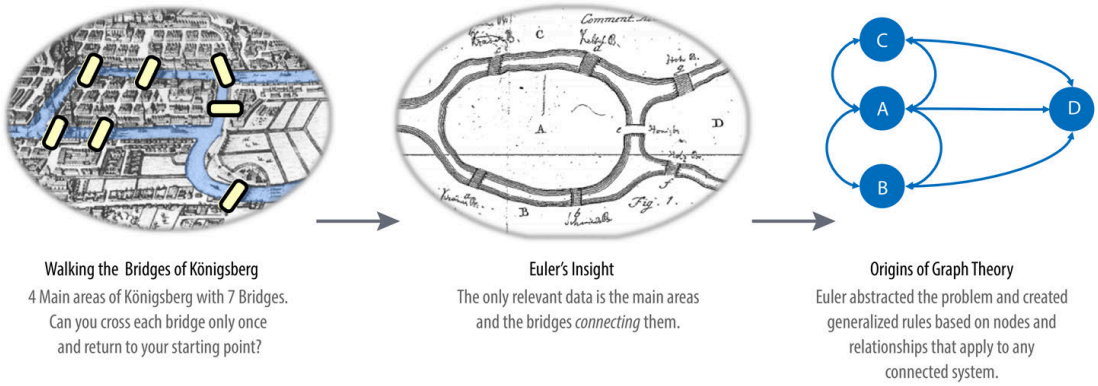


图 1-1。图理论的起源。哥尼斯堡市包括两个大岛，通过七座桥梁连接彼此，并和城市的两个大陆部分相连接。题目是创造一次连续步行穿过城市,跨越每座桥一次,而且只有跨越一次。

虽然图起源于数学,但它们也是一种实用性和高保真度（high fidelity）的建模和分析方法。组成图的对象称为节点(node)或顶点(vertice)，它们之间的链接称为关系(relationship)、链接(link)或边(edge)。我们使用本书中的术语节点(node)和关系(relationship)：你可以想到节点作为句子中的名词,将关系作为动词,为节点提供上下文。为了避免任何混淆,我们在这本书中谈论的图与方程绘图(graphing equation)或表格图表(chart)没有任何的关系,如图 1-2 所示。

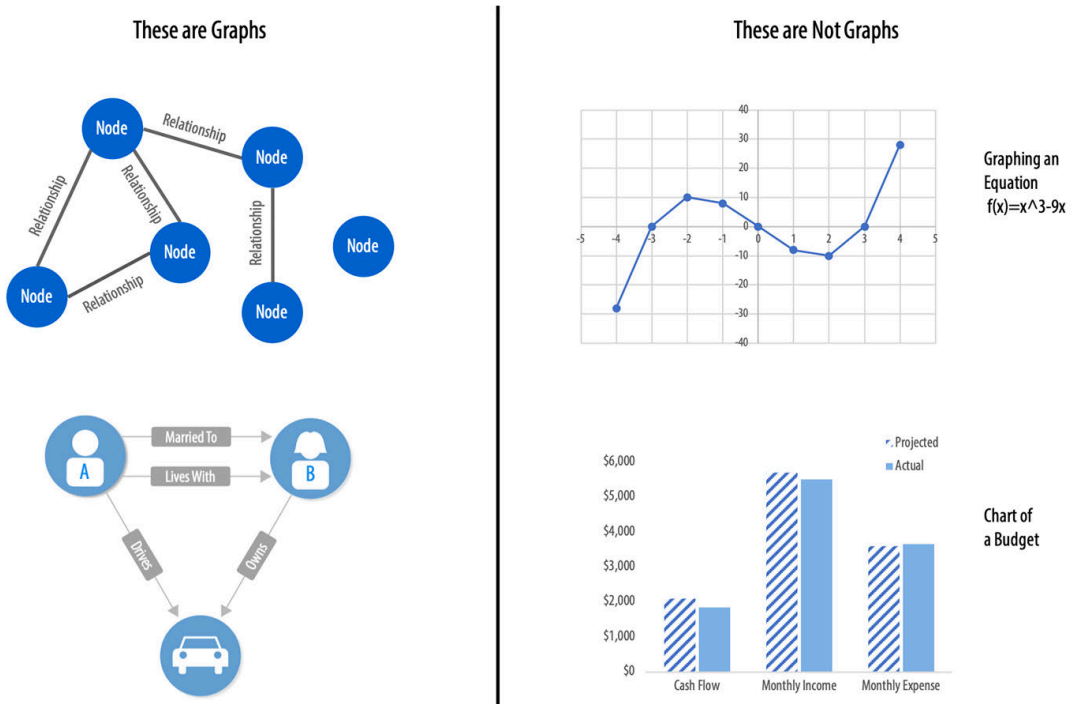


图 1-2。图是网络的表示形式,通常用圆圈表示我们调用节点的实体,以及表示关系的线路。左侧是Graph（图），右侧不是Graph，它们是方程绘图(graphing equation)或者是图表(chart)。

查看图 1-2 中的人员关系图,我们可以轻松地构造几个句子描述它。例如,A与拥有汽车的人B住在一起,A驾驶B拥有的汽车。这种建模方法极具吸引力，因为它很容易映射到现实世界,并且非常"白板友好" （意思是，能够很轻松地在白板上画出数据的模型）。这非常有助于数据建模和分析。

但建模图只是开始，我们可能还希望将它们处理到形成洞察力的水平。那就到了图算法的领域。

什么是图分析和算法？

图算法是图分析工具的子集。（图分析是一个比图算法更宽的概念，它包含图算法。）

图分析我们是这样做的，它是使用任何基于图的方法来分析数据网络。我们可以使用各种方法:可以查询图数据,使用基本统计,直观地探索图,或将图纳入我们的机器学习任务。基于图的模式查询通常用于本地数据分析,而图计算算法通常引用更全局和迭代分析。尽管在如何使用这些类型的分析方面存在重叠,但我们使用术

语图算法（graph algorithm）来表示后者,它用于更密集的计算分析和数据科学用途。

图算法提供了分析数据网络的最有效方法之一,因为它们的数学基础是专门为处理关系而构建的。它们描述了处理图发现一般或特定品质的步骤。基于图理论的数学（图论，graph theory）,图算法用节点之间的关系推断出复杂系统的组织和动力学特性。网络科学家使用这些算法来发现隐藏的信息、测试假设，并预测行为。

网络科学

网络科学是一个学术领域,它深深植根于图论,它涉及到对象之间关系的数学模型。因为他们的数据量大小、连接性和复杂性的不同，网络科学家需要依赖于图算法和数据库管理系统。

关于复杂性和网络科学，有许多奇妙的资源。下面是一些可供您探索的参考。

- Network Science,由Albert-László Barabási著,是一个入门级的电子书
- Complexity Explorer提供在线课程
- New England Complex Systems Institute提供各种资源和论文

图算法具有广泛的潜力,比如从防止欺诈和优化呼叫路由到预测流感的传播。例如,我们希望对电力系统中的可能过载的特定节点进行评分。或者,我们希望在一个运输系统中发现阻塞的区域。

事实上,在 2010 年,美国航空旅行系统经历了两起严重的事件,涉及多个拥堵的机场。后来，网络科学家P.Fleurquin、J.J.Ramasco和V. M. Eguíluz使用图算法来确认这两个事件作为系统级联延迟（多级延迟，cascading delay）的一部分,并利用这些信息进行改进性建议。这些成果发表在"Systemic Delay Propagation in the US Airport Network" 这个论文中。

为了可视化支撑航空运输的网络，图1-3是Martin Grandjean为他的文章 "Connected World: Untangling the Air Traffic Network" 创作的配图。该图清楚地显示了航空运输集群的高度连接结构。许多运输系统都表现出了集中的分布,有明确的枢纽和辐条模式,容易会产生延误。

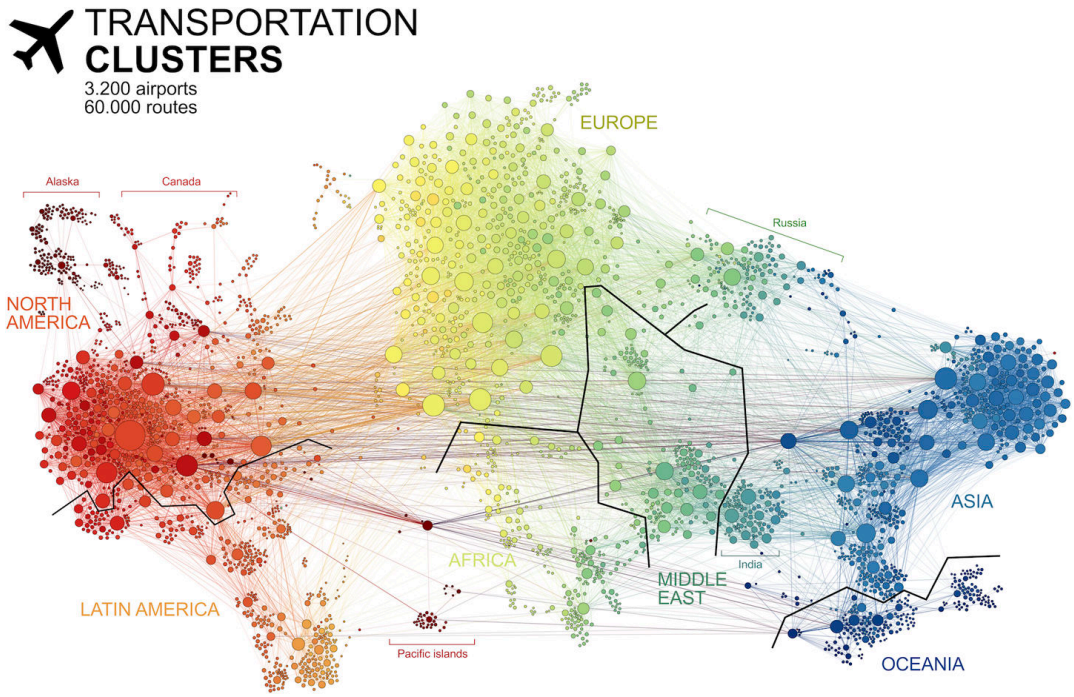


图 1-3。航空运输网络展示了在多个尺度上演变的枢纽和辐条结构。这些结构解释了乘客流动的模式。

图还有助于揭示导致全球突变中非常小的相互作用和动力学。比如通过精确地表示全球结构内交互的事物，将微观和宏观尺度结合在一起。这些关联用于预测行为并确定缺失的连接。图1-4是草原物种的食物链网络，该图被用来评估分层组织和物种的相互作用，然后预测缺失的关系。该成果由A. Clauset, C. Moore,和M. E. J. Newman在论文 “Hierarchical Structure and the Prediction of Missing Links in Network” 中发表。

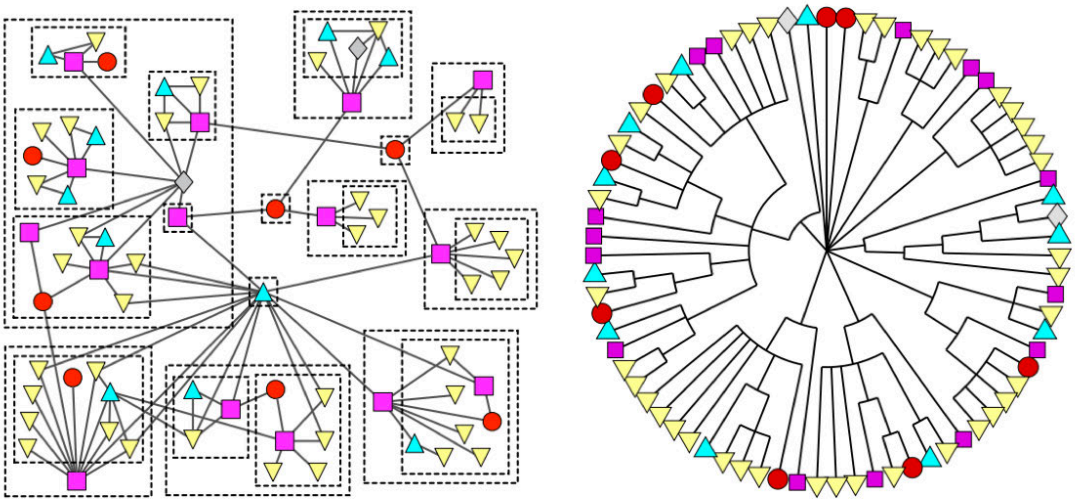


图1-4.这个草原物种的食物链网络，使用图将小规模的相互作用与大的结构形成联系起来。

图处理、数据库、查询和算法

图处理（graph processing）包含了那些承载了图相关工作负载和任务的方法。大多数图查询都考察图的特定部分（例如，起始节点），工作通常集中在周围的子图形中。我们将这种类型的工作称为图局部(graph local)，图局部意味着声明式地（用一种声明式的语言）查询一个图的结构，正如Ian Robinson, Jim Webber, 和Emil Eifrem在《Graph Databases》（O'Reilly）这本书中所解释的那样。这种图局部处理通常用于实时事务和基于模式的查询。

在谈到图算法时，我们通常会寻找全局模式和结构。算法的输入通常是整个图，输出可以是一个图中被增加的属性，或一些聚合值，如分数。我们将这种处理归类为图全局（graph global），它意味着使用计算算法（通常是迭代的）处理图形的结构。这种方法通过连接来揭示了网络的整体性质。组织倾向于使用图算法来建模系统，并根据事物的传播方式、重要组件、组标识和系统的整体鲁棒性来预测行为。

在图局部和图全局的定义中，可能有些重叠，有时我们可以使用算法的处理来回答本地查询，反之亦然。但通常意义上，可以简单的认为，全图的操作是由计算算法处理的，子图的操作是在数据库中查询的。

传统上，事务处理和分析都是孤立的。这种人为的隔阂是因为技术的限制造成的。我们的观点是，图分析推动更智能的交易，为进一步分析创造新的数据和机会。最近出现了一种将事务处理和分析集成在一起的趋势，以便实现更实时的决策。

OLTP和OLAP

在线交易处理（OLTP，Online transaction processing）通常是短时间内的行为和操作，如预订机票、贷记账户、预订销售等。OLTP的主要任务是实现大量的低延迟查询处理和高数据完整性。OLTP的每个事务可能只涉及少量记录，为了应对业务，OLTP会并行处理许多事务。

在线分析处理（OLAP，Online analytical processing）有助于对历史数据进行更复杂的查询和分析。这些分析可能包括多个数据源、格式和类型。检测趋势、执行假设中的场景、进行预测和发现结构模式是典型的OLAP的用例。与OLTP相比，OLAP系统在大规模记录上运行长时间的分析任务，但和不一定都是事务。

OLAP系统倾向于快速读取，而尽量减少在OLTP中经常出现的事务（如更新）。在OLAP中，面向批处理的操作是常见的。

然而，最近，OLTP和OLAP之间的界限开始变得模糊。现代数据密集型应用程序现在将实时事务操作与分析相结合。这种处理的集成，是由软件方面的一些进步，如更可扩展的事务管理和增量流处理，以及成本更低的大内存硬件所推动的。将分析和事务结合在一起，可以使连续分析成为常规操作的自然组成部分。随着数据从销售点（POS）机器、制造系统或物联网（IOT）设备中不断收集，现代的数据分析支持在处理过程中进行实时推荐和决策的能力。这一趋势是在几年前观察到的，转义（translytics）和混合事务和分析处理（HTAP，hybrid transactional and analytical processing）这些术语就是用来描述这些进步的。图1-5说明了只读副本是如何被用于这些不同类型的处理的（指的是OLTP和OLAP）。

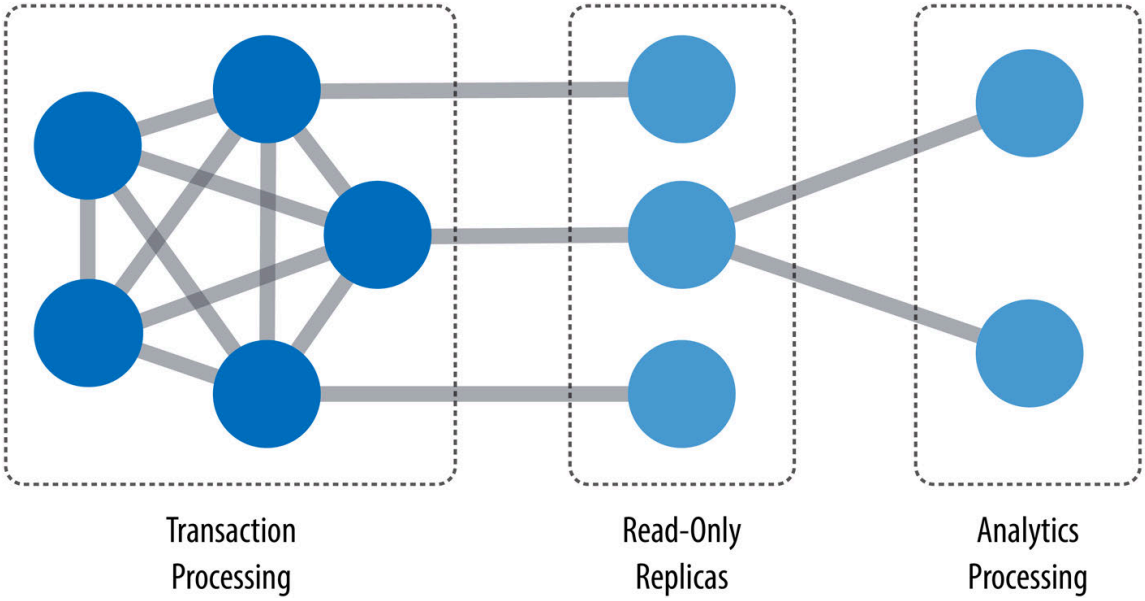


图1-5.混合平台，支持了事务处理所需的低延迟查询处理和高数据完整性，又同时集成对大量数据的复杂分析。

Gartner：

因为实时高级分析（例如：计划、预测和假设分析）成为流程本身的一个组成部分，而不是事后执行的单独活动，HTAP会重新定义一些业务流程的执行方式。这种改进将使实时业务驱动决策过程成为可能，最终，HTAP将成为智能业务运营的关键支持架构。

随着OLTP和OLAP变得更加集成，并开始支持以前仅在单独的服务中提供的功能，这使得我们不再需要使用不同的数据产品或系统。我们可以通过使用相同的平台来简化架构，这意味着我们的分析查询可以利用实时数据，这可以简化分析的迭代过程。

为什么我们要关心图算法？

图算法被用于帮助理解数据网络。我们可以看到现实系统中的关系，比如从蛋白质之间交互到社交网络，从通信系统到电网，从零售体验到火星任务规划。了解网络及其内部连接为洞察和创新提供了难以置信的潜力。

图算法特别适合在高度连接的数据中发现出集中结构和展现出模式。没有什么地方比大数据有更显著的连通性和交互性了。在大数据中，有大量的汇集、混合和动态更新的信息量。大数据中，图算法可以帮助理解我们的数据总体，我们也可以通过更复杂的分析，用关系来为人工智能提供上下文信息和特征。

随着我们的数据变得越来越紧密，了解它的关系和相互依赖性变得越来越重要。研究网络增长的科学家们注意到，连接性随着时间的推移而增加，但并不一致。择优附着（preferential attachment）是研究生长动力学对结构影响的理论之一。如图1-6所示，这个想法描述了一个节点链接到已经有很多连接的其他节点的趋势。

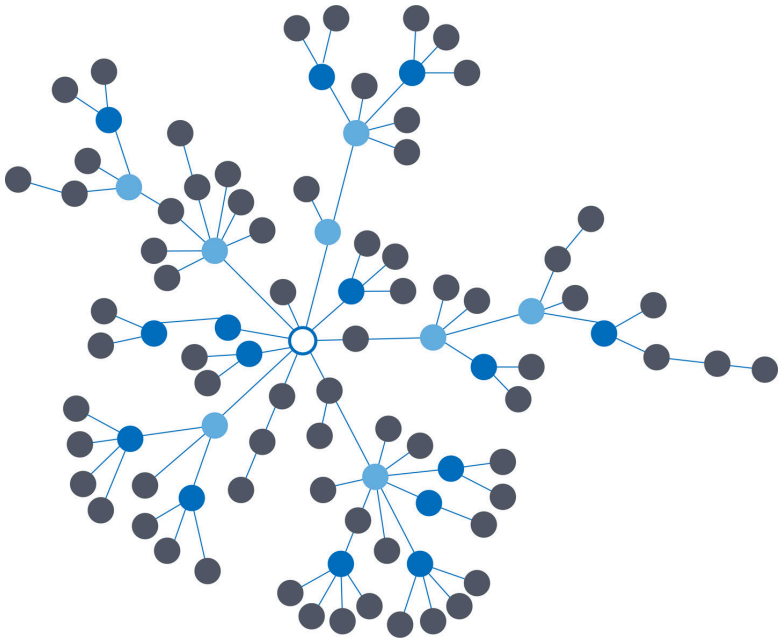


图1-6.择优附着指的是一个节点连接越多，接收新链接的可能性就越大的现象。这会导致不均匀的浓度和中心化。

在Steven Strogatz的书《Sync: How Order Emerges from Chaos in the Universe, Nature, and Daily Life》（Hachette）中，他用例子解释了现实生活系统进行自我组织的不同方式。不管潜在的原因是什么，许多研究人员认为，网络的发展与其产生的形状和层次结构密不可分。高度密集的群体和块状数据网络往往会被发展起来，随着数据规模的增加，复杂性也在增加。我们今天看到了大多数现实世界网络中的这种关系集群，从互联网到社交网络，如图1-7所示的游戏社区。

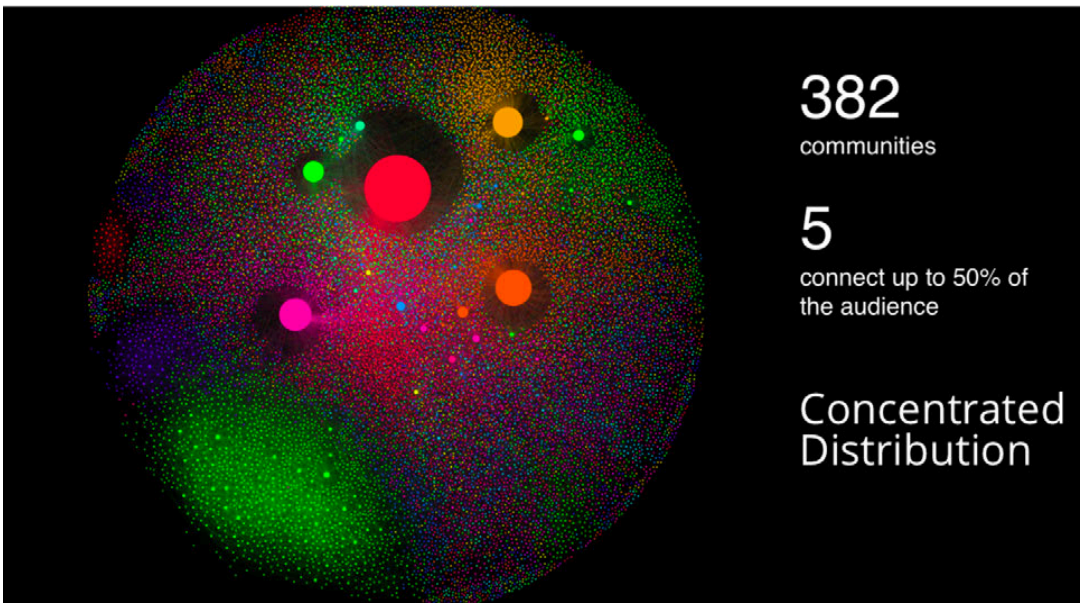


图1-7.这个游戏社区分析显示，382个社区中只有5个社区的有密集的连接。

图1-7所示的网络分析是由Pulsar的Francesco D'Orazio创建的，用于帮助预测内容的重要性和分发信息的策略。D'Orazio发现了一个社区的分布集中度和一段内容的传播速度之间的相关性。

以上这些（择优附着、中心化、不均匀）与平均分布模型（average distribution model）有很大的不同。在平均分布模型中，大多数节点的连接数都相同。例如，如果万维网拥有平均分布的连接，那么所有页面进出的链接数将大致相同，因为平均分布模型的假设就是：大多数节点是相等连接的，但许多类型的图和许多实际网络表现出集中特性。网络、旅游和社交网络等图形一样，具有幂次法则分布，其中一些节点高度连接，大多数节点适度连接。

幂次法则（power law）

幂次法则（也称为比例规律）描述了两个量之间的关系，其中一个量随另一个量的幂而变化。例如，一个立方体的面积与其边长的幂为3有关。一个著名的例子是帕累托分布或“80/20规则”，最初用来描述20%的人口控制了80%的财富。我们在自然世界和网络中也能看到各种幂次法则。

试图“平均”一个网络通常不会很好地用于调查关系或预测，因为现实世界中的网络具有不均匀的节点分布和关系。我们可以很容易地在图1-8中看到，对不均匀的数据使用平均特征将导致不正确的结果。

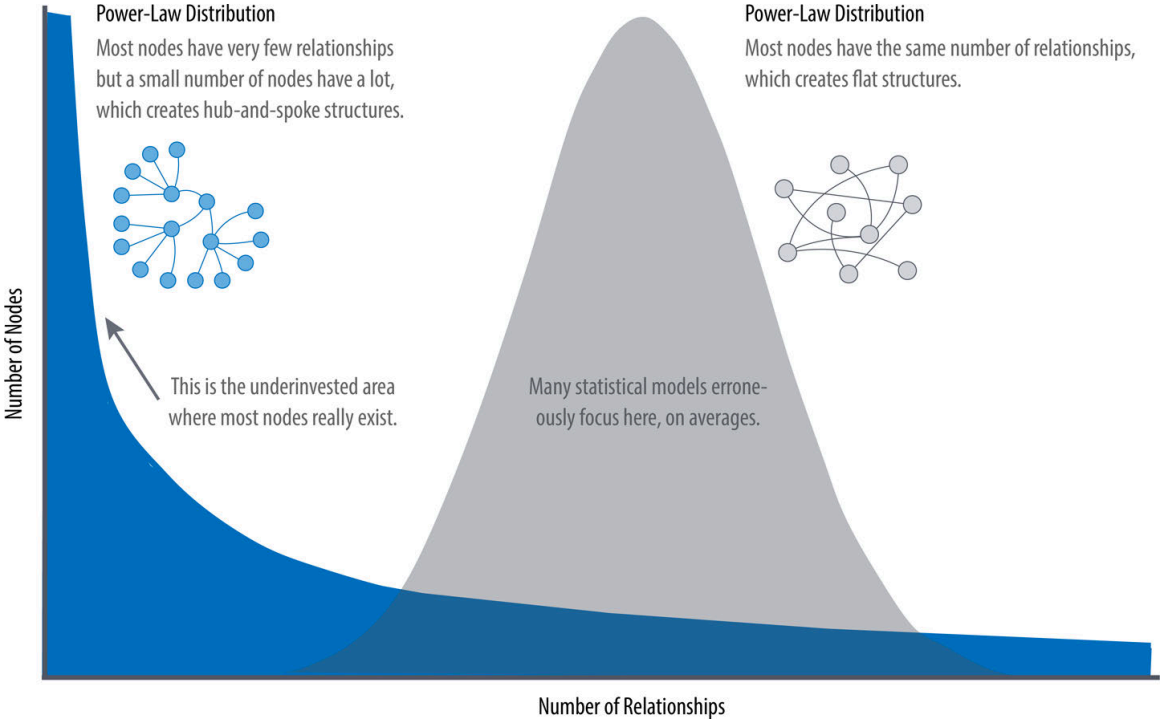


图1-8.现实世界中的网络节点和关系分布不均匀，其极端表现为幂次法则分布。平均分布假定大多数节点具有相同数量的关系，并在随机网络中生成结果。

由于高度连接的数据不遵循平均分布，网络科学家使用图分析来搜索和解释现实世界中结构和关系分布。

> *我们所知道的自然界中没有一个网络可以用随机网络模型来描述。*
> ***Albert-László Barabási, Director, Center for Complex Network Research, Northeastern University,***
> *众多网络科学图书的作者*

大多数用户面临的挑战是，用传统的分析工具分析数据时，连接紧密且不均匀是很麻烦的。那里可能有一个结构，但很难找到。对杂乱的数据采用平均方法是很有诱惑力的，但这样做会隐藏很多模式，而且我们的结果不代表任何真实的群体。例如，如果你对所有客户的人口统计信息进行平均，并且仅仅根据平均值提供一种体验服务，那么你肯定会错过大多数社区：社区倾向于围绕年龄、职业或婚姻状况和位置等相关因素聚集。

此外，动态行为，特别是在突发事件和突发事件周围，是不能用平均化的快照看到的。举例来说，你想象一个社会团体的人际关系不断增加，你也会期待更多的交流。这可能导致协作的拐点，随后的联盟或小团体的形成或极化。所以，需要复杂的方法来预测一个网络随着时间的发展，如果我们了解数据中的结构和交互，我们可以推断出行为。由于各种关注关系的存在，图分析被用于预测群体的弹性。

图分析用例

在最抽象的层次上，图分析应用于动态群体的行为预测和行动规划，这需要对团队的关系和结构进行了解。图算法通过检查网络的连接来实现这一点。通过这种方法，你也可以了解相互连接的系统群的拓扑结构，并对其流程建模。

有三大类问题，表明图分析和算法是否有必要，如图1-9所示。

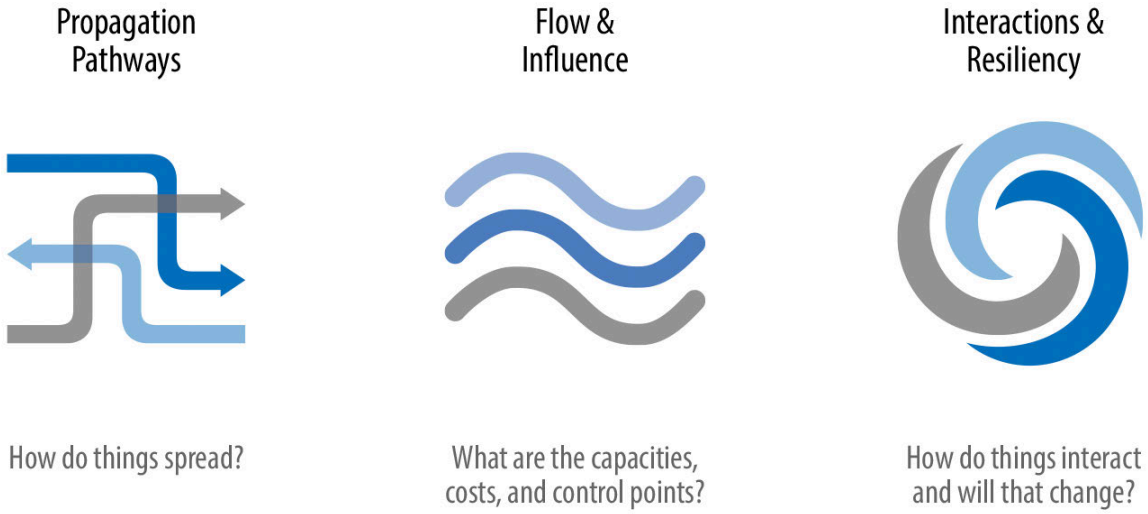


图1-9.图分析能够回答的问题类型

这些都是图算法能够解决的问题。你的挑战相似吗？

- 调查疾病或级联传输失败的路径。
- 发现网络攻击中最易受攻击或损坏的组件。
- 找出最便宜或最快的信息或资源路由方法。
- 预测数据中丢失的链接。
- 定位复杂系统中的直接和间接影响。
- 发现看不见的层次结构和依赖关系。
- 预测团队是合并还是分离。

- 找出瓶颈或谁有能力拒绝/提供更多资源。
- 展示基于行为的社区，提供个性化建议。
- 减少欺诈和异常检测中的误报。
- 提取更多机器学习的预测特征。

结论

在本章中，我们研究了今天的大数据是如何紧密相连的，以及这一点给我们的启示。在分析群体动态和关系方面存在着强大的科学实践，但这些工具在企业中并不总是常见的。在高层级的分析中，我们应该考虑数据的品质，了解社区的属性，预测复杂行为。如果我们的数据是一个网络，我们应该抵制用平均值考虑问题的诱惑。相反，我们应该使用与我们的数据和我们正在寻求的见解相匹配的工具。在下一章中，我们将介绍图概念和术语。

第二章 图论和概念

在本章中，我们阐述了图算法的框架和术语。介绍图论的基本原理时，重点介绍与实践最相关的概念。

首先，我们将描述如何表示图，然后解释不同类型的图及其属性。这在以后的章节中很重要，因为我们的图的特性将指引我们的算法选择和解释结果。在本章的最后，我们将对本书后面章节的图算法的类型进行概览。

术语(Terminology)

带有标签属性(labled property)的图是最常用的图数据建模方法之一。一个标签（label）将节点（node）标记为某个组的成员之一。在图2-1中，我们有两组节点：人和车。（尽管在经典的图理论中，标签应用于单个具体的节点，但在本书的场景下，它通常用于表示节点组。）

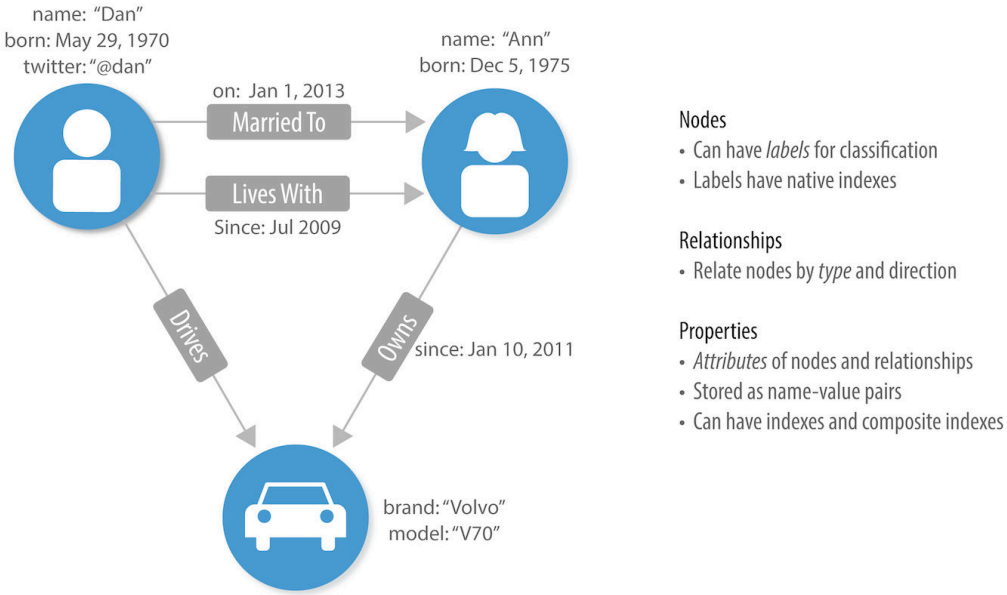


图2-1.带有标签属性的图模型是一种简单灵活的数据网络表示方法。

根据类型，我们可以把关系（Relationship）进行分类。比如，DRIVES、OWNS、LIVES_WITH以及MARRIED_TO关系类型。

属性(Properties, attribute)可以包含各种数据类型，从数字和字符串到空间和时间数据。在图2-1中，我们将属性表示为“属性名-值”（name-value pairs），其中属性的名称首先出现，然后是它的值。例如，左侧的Person节点有一个属性name：” Dan"，而MARRIED_TO关系上有一个属性on:"2013年1月1日"。

一个相对较大图的一部分，可以叫做子图（subgraph）。比如当我们需要一个具有特定特性的子集来进行聚焦分析时，子图就很有用。

路径（Path）是一组节点加上它们之间的关系。图2-1中，比如，简单的路径可以包含节点Dan、Ann和Car以及DRIVES和OWNS关系。

图与图之间在类型、形状和大小以及可用于分析的属性各方面均不相同。接下来，我们将描述最适合图算法的图的分类法。请记住，这些名词解释不但适用于图，也适用于一个图的子图。

图的型和结构

在经典图论中，术语“图”等同于一个简单图(simple graph)或者严格图（strict graph），在这种图中，节点之间只有一个关系，如图2-2的左边第一个图所示。然而，大多数实际图在节点之间有许多关系，甚至有自引用关系。现在，“图”这个术语可以用于图2-2中的所有三种图形类型，因此也包含本书中提到的图。

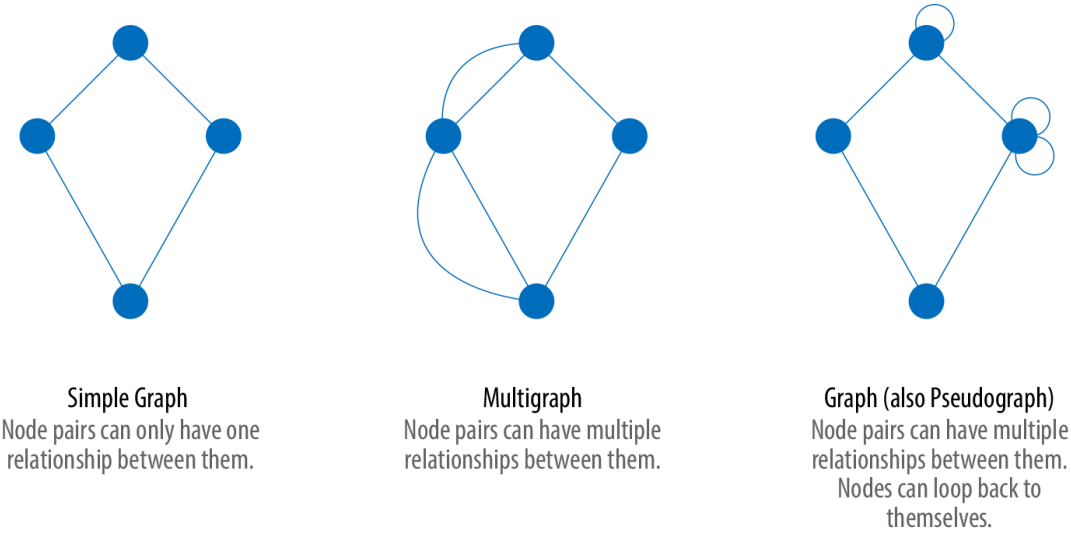


图2-2.在本书中，我们使用术语图(graph)来包含所有这些经典类型的图。

随机结构、小世界结构、自由缩放结构

图有许多形状。图2-3显示了三种具有代表性的网络类型：

- 随机网络(random network)
- 小世界网络（small-world network）
- 自由缩放网络(scale-free network)

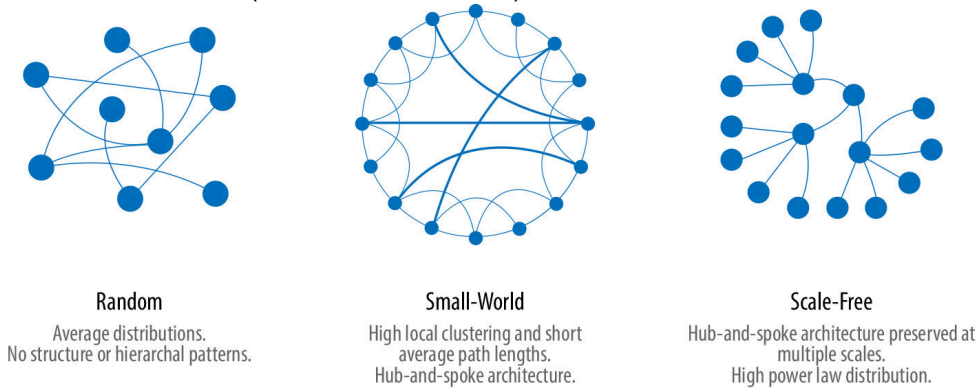


图2-3.三种具有独特图和行为的网络结构

- 随机网络是没有层级结构的，它的节点之间的连接完全平均分布。这种无形状的图形是“扁平的”，没有可辨别的模式。所有节点都具有与任何其他节点相同的连接概率。
- 小世界网络在社交网络中极为常见；这种网络由局部的连接组成，并呈现出了一些“中心-辐射”的模式。“六度凯文培根”（Six Degrees of Kevin Bacon）游戏可能是小世界效应中最著名的例子。虽然你主要和一小群朋友交往，即使是著名的演员或是在地球的另一边的人，你与他们的距离都不会太远。
- 当幂次法则存在的时候，且无论规模大小，都保留了“中心-辐射”架构时，这个网络就是一个自由缩放网络。比如万维网（world wide web）就是这样的例子。

这些网络类型产生了具有独特结构、分布和行为的图。当我们使用图算法时，我们将在我们的结果中认识到类似的模式。

图的风格(flavor)

为了充分利用图算法，我们必须熟悉将会遇到的图的最主要特性。表2-1总结了常见的图属性。在下面的部分中，我们将更详细地介绍不同的风格（或者说是图的属性，这和节点或关系的属性不同）。

表2-1 图的常用属性

图特性	关键因素	算法考量
连接(connected) vs 断开(disconnected)	在无论多远的任意两个节点之间是否存在路径	孤立的节点会导致不可预期的行为，比如在无连接的区域出现停顿或者错误
有权重(weighted) vs 无权重(unweighted)	是否在关系或者节点上有业务相关的赋值	许多算法依赖于权重，当权重被忽略的时候，我们会看到性能和结果上的巨大的差别
有向(directed) vs 无向(undirected)	是否关系被显性定义了起始节点和终止节点	它为推理出意义提供了丰富的上下文。在一些算法中，你可以显性地使用单向、双向或者干脆不设置方向
有环(cyclic) vs 无环(acyclic)	是否有路径的起始节点和终止节点是同一个节点	有环图很常见，但是算法需要特别注意，比如环会导致无法停止。无环图（或者说是生成树）是许多图算法的基础。
稀疏(sparse) vs 稠密(dense)	关系数量和节点数量的比值	特别稀疏或者稠密的图都会导致异常的结果。在数据建模中，要确保业务领域的图避开极端稠密和极端稀疏
单分(mono-partite) vs 二分(bi-partite) vs k-分(k-partite)	是否节点只连接向一种类型的节点或者是多种类型的节点	有助于创建分析中用的关系，也有助于投影出更有用的子图

连通图（Connected graphs）和断开的图（Disconnected graphs）

如果所有节点之间都有可连通的路径，那么这个图就是连通图。如果在图中有孤岛，它就是断开的图。如果这些岛上的节点是相互连接的，则称这个岛为图的组件(component)，或者叫做集群（cluster），如图2-4所示。

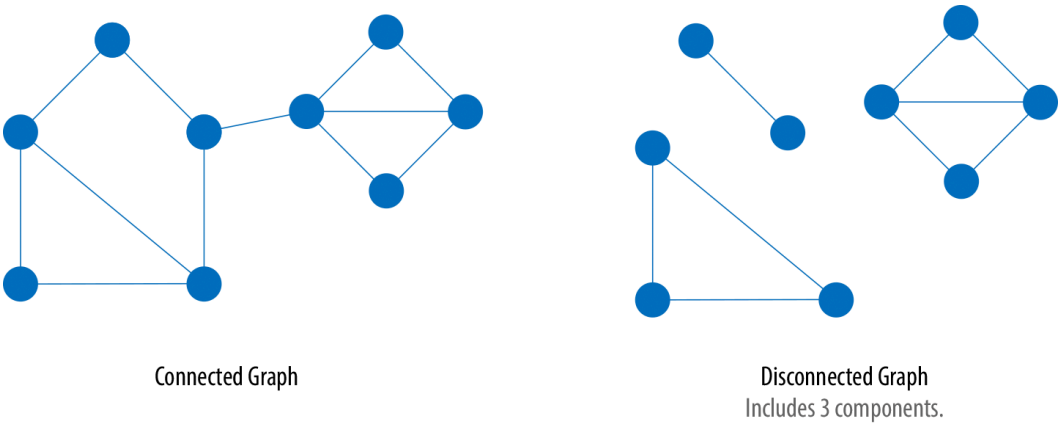


图2-4.如果图中有孤岛，它就是一个断开的图。

有些算法在断开的图上运行，会产生误导性的结果。为了避免意外发生，检查图的连通结构是很好的第一步。

加权图（weighted graphs）和无权图（unweighted graphs）

无权图没有为其节点或关系指定权重值。对于加权图，这些值可以表示各种度量，例如成本、时间、距离、容量，甚至是特定域的优先级。图2-5举了一个加权图和无权图的例子。

图2-5.加权图可以保存关系或节点上的值。

基本的图算法可以使用权重来处理关系的强度或值。许多算法计算出一些指标，然后将其写入权重，以被后续处理使用。有些算法在查找累计总数、最小值或最佳值时会更新权重值。

加权图的一个典型用途是路径查找（pathfinding）算法。我们手机上的地图应用程序用的就是这种算法，比如计算位置之间最短/最便宜/最快的交通路线。图2-6展现了加权图和无权图上最短路径的差别。

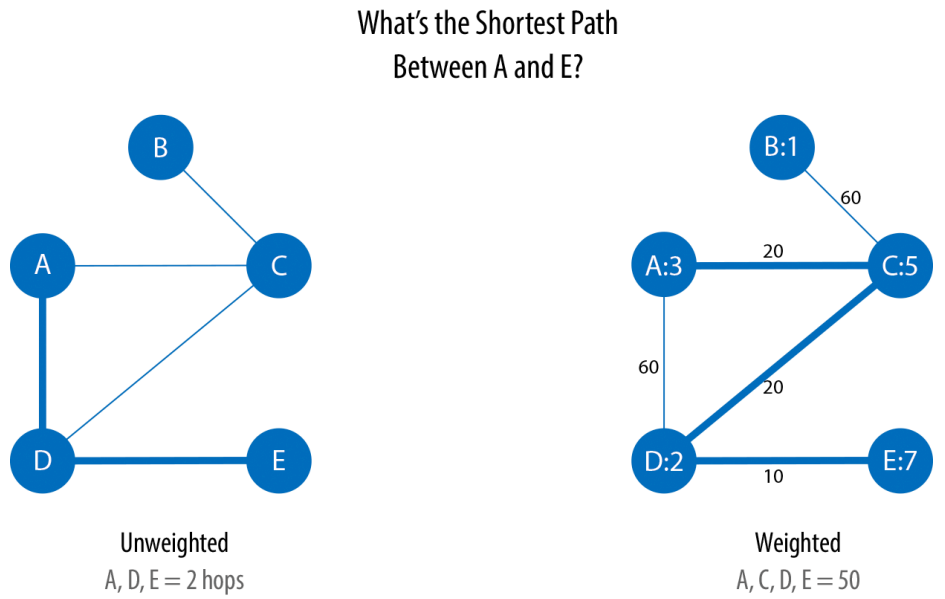


图2-6.对于其他相同的无权图和加权图，最短路径可能有所不同。

在没有权重的情况下，我们的最短路径是根据关系的数量（通常称为跃点，hop）来计算的。A和E有一条最短的路径是两个跃点（two-hop），它们之间只有一个节点（D）。然而，从A到E的最短加权路径是从A到C到D到E。如果权重以公里为单位来表示物理距离，则总距离为50公里。在这种情况下，跃点数最少的路径相当于70公里长的物理路径。

无向图（undirected graph）和有向图（directed graph）

在无向图中，关系被认为是双向的（例如，友谊）。在有向图中，关系有一个特定的方向。指向一个节点的关系被称为这个节点的入链（in-link），同样的，以某节点为起始节点的关系被称为这个节点的出链（out-link）。

方向（direction）增加了一个维度的信息。同一类型但方向相反的关系，代表了不同的语义，比如说，表示依赖或表示流向的关系。方向还可以用作信用或者群体强度的指标。用方向能够很好地表示个人偏好和社会关系。例如，在图2-7的有向图中，假定有向图是一个学生网络，关系是“喜欢”，那么我们将得出A和C更受欢迎。

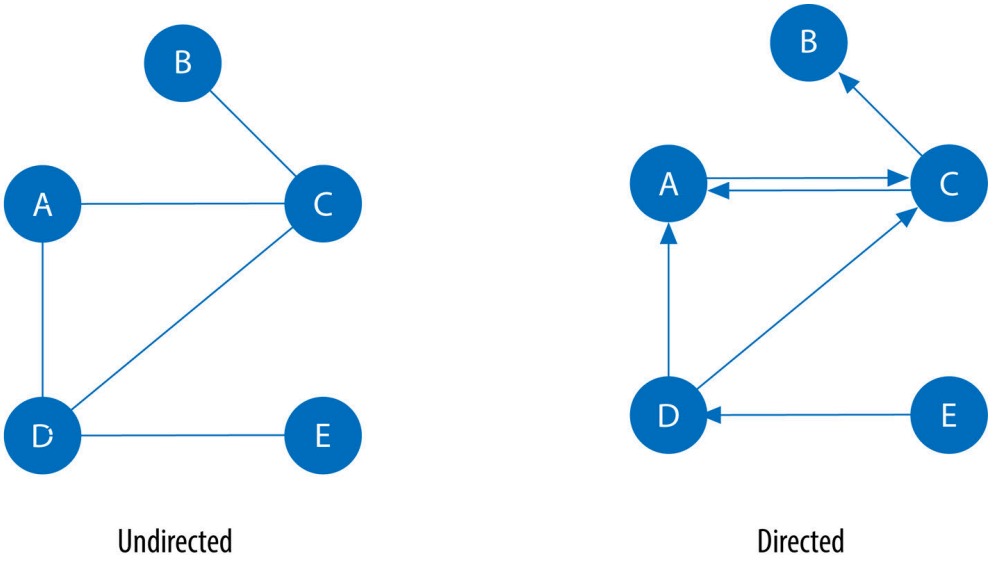


图2-7.许多算法在是否有方向上做了规定，只允许我们在入链或出链、两个方向或没有方向进行计算。

道路网络说明了我们为什么要同时使用这两种类型的图。例如，城市之间的公路经常是双向行驶。然而在城市里，有些道路是单行道。（某些信息流向也是如此！）

算法在有向图和无向图中运行，我们会得到不同的结果。在无向图中，例如高速公路或朋友间的友谊，我们假设所有的关系总是双向的。

如果将图2-7重新看作是一个定向道路网络的话，你可以从C和D行驶到A，但是之后只能通过C离开A。如果将A到C之关系断开，那么意味着A是一个死胡同。当然，这对于单向道路网络来说不太可能，但对于流程或网页来说却有可能发生。

有环图（cyclic graphs）和无环图（acyclic graphs）

在图论中，环是在从一个节点开始和并在同一个节点结束的路径。无环图中就没有这样的路径。如图2-8所示，有向图和无向图都可以有环。在有向图中，路径必须遵循关系的方向。graph 1是一个有向无环图（directed acyclic graph，DAG），它一定会有死胡同终端（dead end，也称为叶节点，leaf node）。

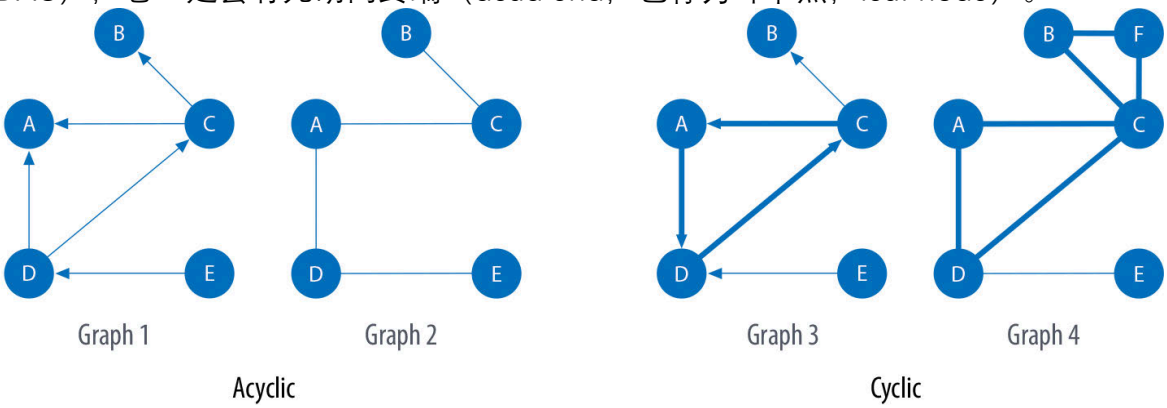


图2-8.在无环图中，如果不倒退，是不可能形成在同一节点开始并结束的路径。

如果不重复走某一个关系，在graph 1和graph 2上都无法在同一节点上开始和结束，因此它们都是无环的。你可能记得在第一章中，不能重复的关系是在哥尼斯堡问题提出来的，这是整个图论的开始！图2-8中的graph 3显示了A-D-C-A之后的一个简单循环，而且路径中间没有重复节点。在graph 4中，通过添加一个节点和关系，使无向有环图变得更有趣。现在有一个闭合的环，环中还有一个重复的节点（C），遵循B-F-C-D-A-C-B。实际上，在graph 4中，有多个环存在。

环是常见的，我们有时需要通过切割关系，将有环图转换为无环图，以解决环处理的问题。有向无环图很自然地大量出现在调度、系谱和版本历史等场景中。

树（tree）

在经典图论中，无向无环图(undirected acyclic graph, UAG)称为树。但在计算机科学中，树也包含有向无环图（DAG）。一种更具包容性的树的定义是这样说的，在一个图中，其中任意两个节点仅有一条路径相连接，这样的图就称之为树。树对于理解图结构和许多算法都具有重要意义。它们在设计网络、数据结构和搜索优化以改进分类或组织层次结构方面发挥着关键作用。

关于树及其变种的报道很多。图2-9说明了我们可能遇到的常见树。

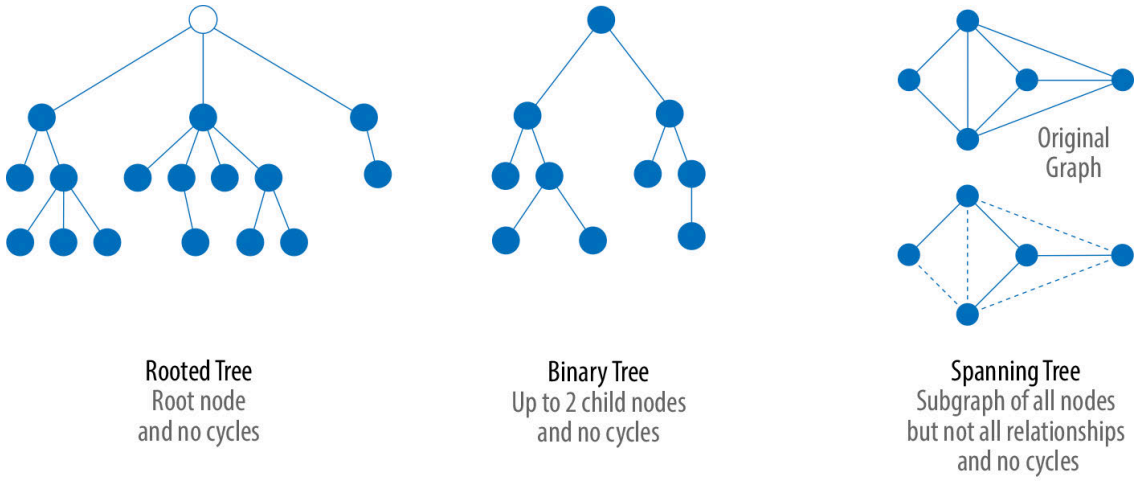


图2-9.在这些典型的树图中，生成树最常用于图算法。

在这些变体中，生成树（spanning tree）是最适合本书的。生成树是某个较大的无环图的子图，它包含了父图的全部节点，但并不包含父图的所有关系。最小生成树指的是所有生成树中，用最少跃点数或最少路径权重就将全部节点连接起来的生成树。

稀疏图（sparse graphs）和稠密图（dense graphs）

图的稀疏度是基于它的关系数与最大可能的关系数（如果每对节点之间都有关系的话）进行比较的结果。每个节点都与其他节点有关系的图称为完全图（complete graph），或者称为组件集体图(a clique of component)。例如，如果我所有的朋友都认识，那就是一个小集体图。

图的最大密度是完全图中可能存在的关系数。用公式 Max D = n(n-1)/2计算，其中n是节点数。为了测量实际密度，我们使用公式d=2(R)/(n(n-1))，其中R是关系数。在图2-10中，我们可以看到三个无向图的实际密度。

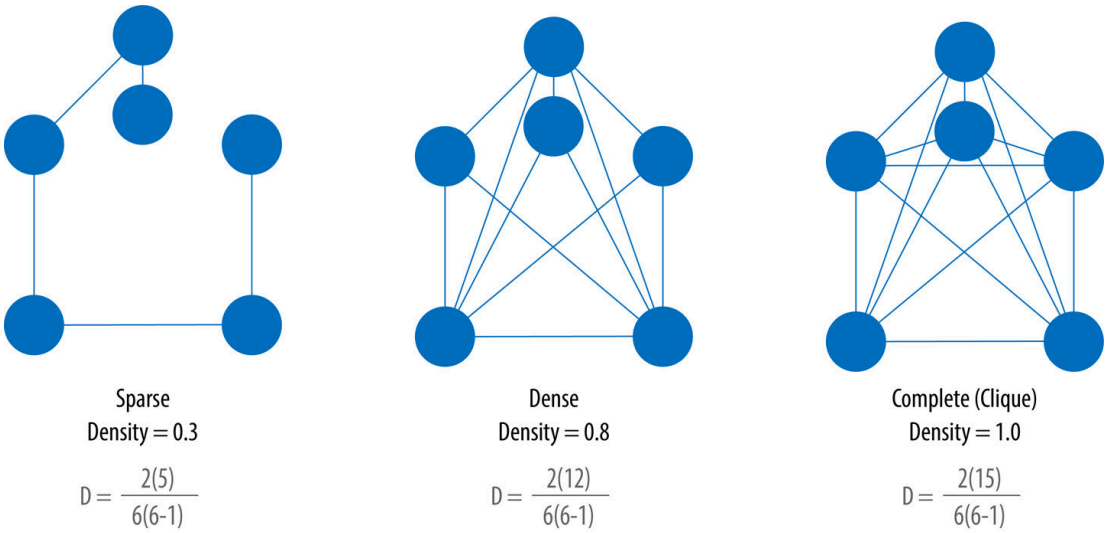


图2-10。检查图的密度可以帮助你评估意外结果。

虽然没有严格的分界线，但任何实际密度接近最大密度的图都被认为是稠密的。大多数基于真实网络的图趋向于稀疏，总节点与总关系的近似线性相关。尤其是在物理元素发挥作用的情况下，例如在一个点上可以连接多少电线、管道、道路或友谊的实际限制。

有些算法在极稀疏或极密集的图上执行时会返回无意义的结果。如果图太稀疏，则可能没有足够的关系让算法计算有用的结果。或者，由于节点之间的连接非常紧密，因此它们不会给出太多附加信息。高密度也会扭曲某些结果或增加计算的复杂性。在这种情况下，筛选出业务相关的子图再进行分析和计算，是一种实用的方法。

单分图（mono-partite graphs）、二分图（bi-partite graphs）和K-分图（k-partite graphs）

（这了的K-分，指的是有K种成分）

大多数网络都包含具有多个节点和关系类型的数据。然而，图算法通常只考虑一种节点类型和一种关系类型。具有一个节点类型和关系类型的图有时被称为单分图。

二分图是这样的一种图，它的节点可以分为两组，任何一个关系只能在两组之间进行连接。图2-11显示了这样一个图的例子。它有两个节点集：一类是观

众，一类是电视节目。只有在两个组之间有关系，组内部没有关系。换句话说，在graph 1中，电视节目只与观众相关，而与其他电视节目无关，而且观众也不直接与其他观众相关。

从观众和电视节目的二分图开始，我们创建了两个单分投影：基于共同节目的观众连接（图2），以及基于共同观众的电视节目（图3）。我们还可以根据关系类型进行筛选，如监视、分级或审阅。

用推理出来的关系构建的投影单分图是图论分析的重要组成部分。这些类型的投影有助于揭示间接关系和品质。例如，在图2-11中的图2，Bev和Ann只看到一个共同的电视节目，而Bev和Evan有两个共同的电视节目。在图3中，我们通过共同的观众的观看次数来聚合电视节目之间的关系。这类指标，或者其他类似指标，可以用来推断观看Battlestar Galactica和Firefly等行为之间的意义。这可以得出我们给类似Evan这样的人的建议，他在图2-11中刚刚看完最后一集Firefly。

K-分图意味着我们的数据的节点类型是K。例如，如果我们有三种节点类型，我们会得到一个三分图（tripartite graph）。这只是扩展了二分图和单分图的概念来考虑更多的节点类型。许多现实世界的图，特别是知识图，有一个很大的K值，因为它们结合了许多不同的概念和信息类型。有许多使用大量节点类型的例子，比如，通过将一个配方集映射到一个成分集到一个化学化合物，然后推断出连接流行偏好的新混合，来创建新的配方。因此，我们可以通过泛化来减少节点类型的数量，例如，将许多形式的节点（如菠菜或羽衣甘蓝）视为“绿叶蔬菜”。

现在我们回顾了我们最可能使用的图的类型（或者说是风格），让我们来了解我们将在这些图上执行的图算法的类型。

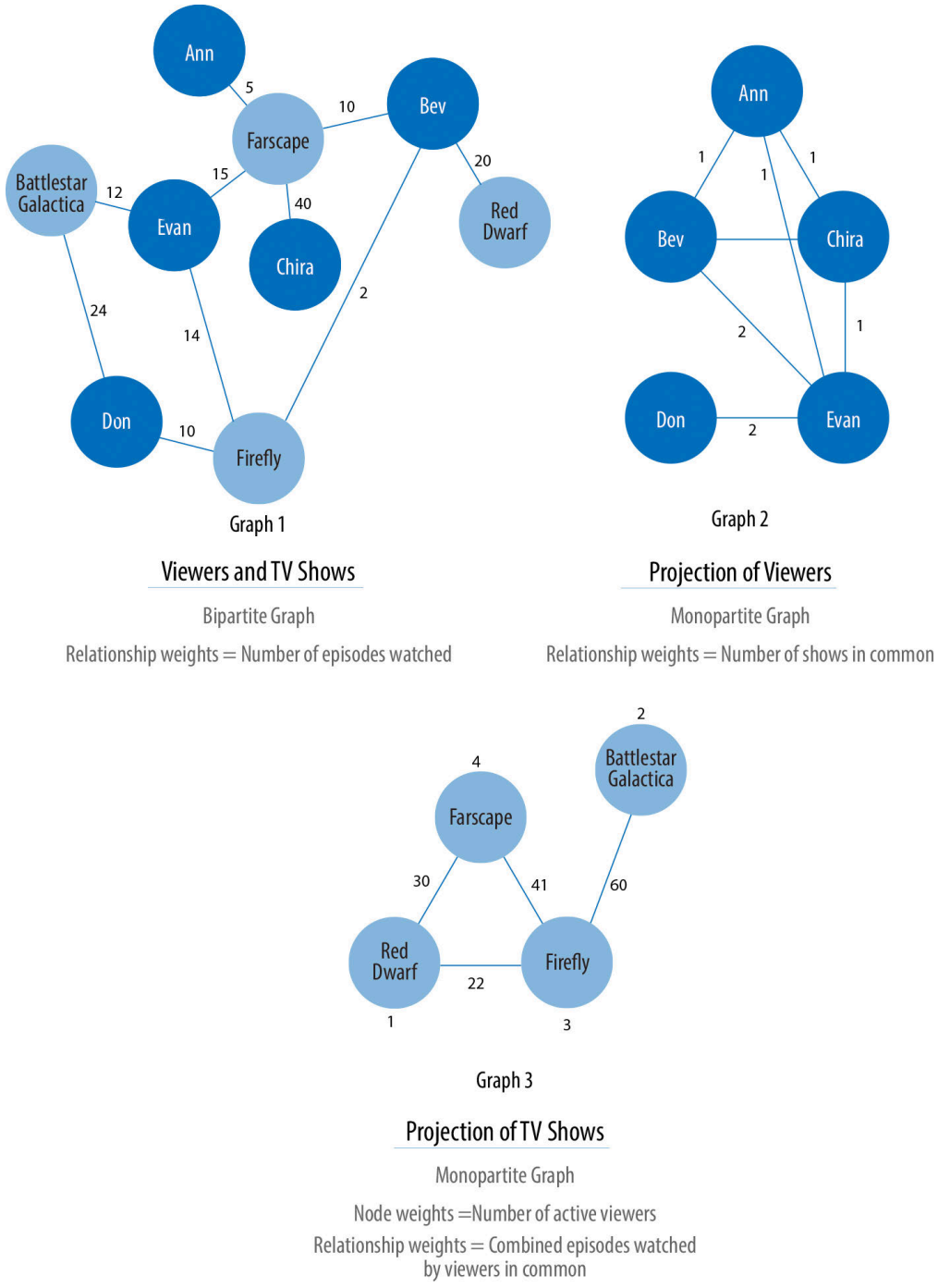


图2-11.二分图通常被投影到单分图上进行更具体的分析。

图算法类型

让我们来看看图算法的核心的三个分析领域。分别是，路径查找（Pathfinding）和搜索（searching）、中心性计算（centrality computation）和社区检测(community detection)算法的章节。

路径查找（pathfinding）

路径是图分析和算法的基础，因此我们将从这里开始我们的算法章节，并给出具体的算法示例。寻找最短路径可能是图算法执行的最频繁的任务，也是几种后续分析的先决条件。最短路径是跃点最少或权重最小的遍历路径。如果图是有向的，那么它是关系方向允许的两个节点之间的最短路径。

路径类型（Path Type）

平均最短路径（average shortest path）用于考虑网络的整体效率和弹性，例如理解地铁站之间的平均距离。

有时，我们可能还想了解最长路线的的优化结果。例如，在两个最远的地铁站之间，即使已经选择了最佳路线，还要确定哪个地铁站间相距较远，最多设置多少站等问题。

在这个场景下，我们将使用图的直径（diameter）来查找所有节点对之间最长的最短路径。

中心性（centrality）

中心性就是网络中哪些节点更重要。但我们所说的重要性是什么呢？有不同类型的中心性算法中，有不同的度量，例如，度量之一是在被连接的各个群体之间快速传播信息的能力。在本书中，我们将重点讨论节点和关系的结构。

社区检测（community detection）

连通性（connectedness）是图论的一个核心概念，它支持复杂的网络分析，如寻找社区。大多数现实网络都显示出或多或少独立子图的子结构（通常是准分形）。

连接特性（connectivity）被用于寻找社区和量化分组的品质。在图中评估不同类型的社区可以发现结构，如中心和层次结构，以及群体吸引或排斥他人的倾向。这些技术用于研究紧急现象，如导致回声室和过滤气泡效应。

总结

图是直观的。它们与我们思考和绘制系统的方式一致。一旦我们了解了术语和概念分层，图的使用原则就会很快被掌握。在本章中，我们解释了后面章节会用到的思想和定义，并描述了你会遇到的图的风格。

图论参考书

如果你很希望了解更多关于图论本身的知识，我们可以推荐如下的材料：

- 由Richard J.Trudeau（Dover）撰写的《Introduction to Graph Theory》是一篇行文优美很耐心的导论。
- 由Robin J.Wilson（Pearson）撰写的《Introduction to Graph Theory》（第五版）是一篇扎实的导论，有很好的插图。
- Graph Theory and Its Applications，第三版，Jonathan L. Gross, Jay Yellen，和Mark Anderson (Chapman and Hall)，需要更多的数学背景，并提供了更多的细节和练习。

接下来，在深入研究如何在Apache Spark和Neo4j中使用图算法之前，我们先研究图处理和分析的类型。

第三章 图平台与处理

在本章中，我们将快速介绍图处理的不同方法和最常见的平台。我们将更仔细地研究本书中使用的两个平台，Apache Spark和Neo4j，以及它们何时适合不同的需求。包括平台安装指南，这样为接下来的几章做好准备。

图平台和处理注意事项

图分析处理具有结构驱动、全局聚焦、难以解析等独特的计算特性。在本节中，我们将讨论图平台和处理的一般注意事项。

平台注意事项

垂直扩展（scale up）和水平扩展(scale out)之间，一直是有争论的。应该使用功能强大的多核、大内存机器，并专注于高效的数据结构和多线程算法吗？或者，投资在分布式处理框架和相关算法上？

一种有用的评估方法是COST（Configuration that Outperforms a Single Thread），由F. McSherry、M. Isard和D. Murray在一篇文献《Scalability! But at What COST?》中提出。COST给我们提供了一种将系统的可伸缩性与系统引入的开销进行比较的方法。它核心概念是，经过算法优化和数据结构优化的良好配置系统，会优于一般性目标的水平扩展解决方案。这是一种测量性能得分的办法，不需要为低配机器增加并行。将可扩展性和有效利用资源的这两点区分开，将有助于我们建立一个明确为我们需求而配置的平台。

在这样的配置方法下，图平台的的一些高度集成的解决方案，如优化算法、处理和内存，可以更紧密地协调工作。

处理过程的考虑

有多种表示方法来表达数据处理过程，比如，流处理或批处理，或者是以记录存储的数据上Map-Reduce范式。不过，对于图数据，也有一些方法将图结构中内在的数据依赖性整合到它们的处理中：

处理方法	说明
以节点为中心（Node-centric）	这种方法使用节点作为处理单元，让它们累积并计算状态，并通过消息传递机制将状态更改传递给它们的附近的节点。这个模型使用已有的转换函数来更简单地实现每个算法。
以关系为中心（relationship-centric）	这种方法与以节点为中心的模型相似，但在子图分析和顺序分析中可能表现得更好。
以图为中心（graph-centric）	这些模型在一个子图中处理，而且该子图独立于其他的子图。子图之间用消息进行通信。
以遍历为中心（traversal-centric）	遍历器（traverser）在遍历中累积了数据，作为计算的基础。
以算法为中心（algorithm-centric）	这些方法使用各种方法来优化每个算法的实现，可以看成是前面几种模型的混合。

Pregel

Pregel是一个以节点为中心、容错的并行处理框架，由Google创建，用于对大型图进行性能分析。pregel基于批量同步并行（bulk synchronous parallel, BSP）模型。BSP包含不同的计算和通信阶段，这一点简化了并行编程。

Pregel在BSP上添加了一个以节点为中心的抽象层，算法通过该抽象层计算来自每个节点的附近节点的传入消息的值。这些计算在每次迭代中执行一次，可以更新节点值并向其他节点发送消息。这些节点还可以在通信阶段将消息打包进行传输，这有助于减少网络通信的数量。当再没有新消息或达到设置的限制时，算法完成。

这些应用于图的方法中，大多数都需要整个图才能进行有效的交叉拓扑运算。这是因为分离的或者分布的图数据会导致工作实例之间大量的数据传输和重组（reshuffling）。这对于许多算法来说，因为需要迭代处理全局的图结构，这会带来一些麻烦。

有代表性的平台

有几种平台满足图处理的要求。传统上，图计算引擎和图数据库之间是分离的，这要求用户根据其流程需要操纵数据：

图计算引擎

图计算引擎是只读的、非事务性的引擎，专注于高效执行迭代图分析和整个图的查询。图计算引擎支持图算法的不同定义和处理范例，例如以节点为中心（例如，Pregel、Gather-Apply-Scatter）或基于MapReduce的方法（例如，PACT）。这些引擎的例子有Giraph、GraphLab、Graph-Engine和Apache Spark。

图数据库

从事务性的背景来看，图数据库关注点集中在使用较小的查询进行快速的写入和读取，这些查询通常接触到图的一小部分。它们的优势在于操作健壮性和对许多用户的高并发和可扩展性。

选择我们所需的平台

选择一个生产平台需要考虑很多因素，例如要运行的分析类型、性能需求、现有环境和团队偏好。我们在本书中使用Apache Spark和Neo4j来展示图算法，因为它们都提供了独特的优势。

Spark是一个扩展和以节点为中心的图计算引擎的例子。它的计算框架和库支持各种数据科学工作流，因而很受欢迎。当如下情形满足的时候，Spark可能是一个正确的选择：

- 算法基本上是可并行或可分的。
- 算法工作流需要使用多种工具和语言进行“多语言”操作。
- 分析可以在批处理模式下脱机运行。
- 图表分析是针对未转换为图表格式的数据。
- 团队需要并具备编码和实现自己算法的专业知识。
- 团队很少使用图算法。

- 团队倾向于将所有数据和分析保存在Hadoop生态系统中。

Neo4j图平台紧密集成了图数据库和以算法为中心的处理，并针对图进行了优化。它在构建基于图的应用程序方面很受欢迎，并且包含一个针对其本地图数据库进行优化的图算法库。当如下情形满足的时候，Neo4j可能是一个正确的选择：

- 算法更具迭代性，需要良好的内存配置。
- 算法和结果对性能敏感。
- 图分析针对复杂的图数据和/或需要深路径遍历。
- 分析/结果与事务性工作负载集成。
- 用于为已有的图增加更多的属性。
- 团队需要与基于图的可视化工具集成。
- 团队喜欢预先打包和支持的算法。

最后，一些组织同时使用Neo4j和Spark进行图处理：Spark用于大规模数据集和数据集成的高级过滤和预处理，Neo4j用于更具体的处理和与基于图的应用程序的集成。

Apache Spark

Apache Spark（或简写成为Spark）是一个用于大规模数据处理的分析引擎。它使用一种叫做DataFrame的抽象表来表示和处理行和列数据，其中行有标识，列有类型。该平台集成了各种数据源，并支持Scala、Python和R等语言。Spark支持各种分析库，如图3-1所示。它基于内存的系统能使用高效的分布式图计算。

GraphFrames是Spark的一个图处理库，它在2016年继承了GraphX，但它与核心Apache Spark是分离的。GraphFrames基于GraphX，但使用DataFrame作为其基础数据结构。GraphFrames支持Java、Scala和Python作为编程语言。2019年春天，“Spark Graph: Property Graphs, Cypher Queries, and Algorithms”提案被通过（见“Spark图的演变”）。我们希望这能将使用DataFrame框架和Cypher查询语言的大量特征引入到核心Spark项目中。不过，在本书中，我们的示例将基于Python API（PySpark），因为它目前在Spark数据科学家中很受欢迎。

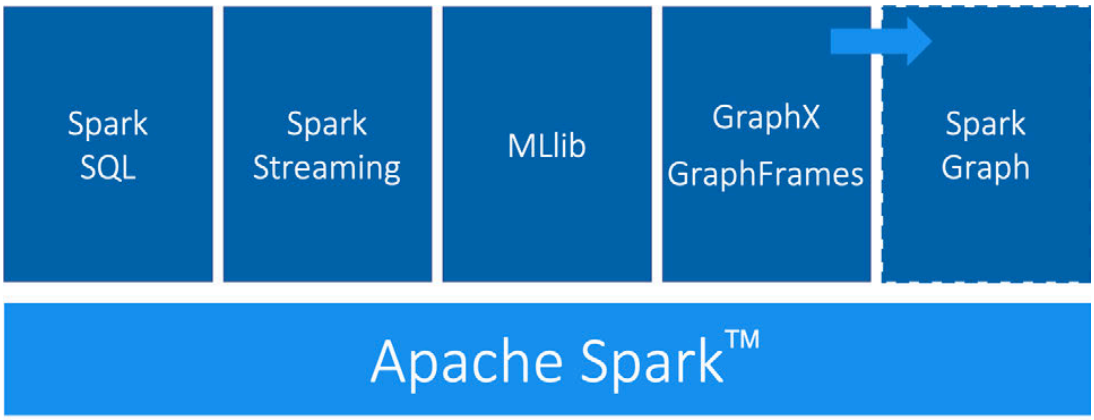


图3-1.Spark是一个开源的分布式通用集群计算框架。它包括几个用于不同工作负载的模块。

Spark图的演变

Spark Graph Project是由Databricks和Neo4j的Apache项目贡献者发起的一项联合倡议，旨在将对DataFrames、Cypher和对DataFrames算法的支持作为3.0版本的一部分引入核心Apache Spark项目。

Cypher最初是在Neo4j中实现的一种声明性图查询语言，但通过OpenCypher项目，它现在被多个数据库供应商和一个开源项目（Cypher for ApacheSark，CAPS）使用。

在不久的将来，我们期望使用CAPS加载和投影图数据作为Spark平台的一个集成部分。我们将在Spark图项目实现之后发布Cypher示例。

这一发展不会影响本书所涵盖的算法，但可能会为如何调用过程添加新的功能。图算法的基础数据模型、概念和计算将保持不变。

节点和关系表示为DataFrame，每个DataFrame包含节点唯一ID和每个关系的源节点和目标节点。我们可以看到表3-1中的节点DataFrame和表3-2中的关系DataFrame的示例。基于这些DataFrame的GraphFrame将有两个节点，即JFK和SEA，以及一个从JFK到SEA的关系。

表3-1 节点DataFrame

id	City	state
JFK	New York	
SEA	Seattle	WA

表3-2 关系DataFrame

src	dst	delay	tripId
JFK	SEA	45	1058923

节点DataFrame必须具有ID列。此列中的值用于唯一标识每个节点。关系DataFrame必须有SRC和DST列。这些列中的值描述连接的节点，并应引用出现在节点数据框的ID列中的条目。

节点和关系DataFrame可以使用形式数据源（包括Parquet、JSON和CSV）来加载。查询的话，可以一同使用PySpark API和Spark SQL。

GraphFrame还为用户提供了一个扩展点来那些非实现开箱即用(not out of the box)的算法。

安装Spark

你可以从Apache Spark网站下载Spark。下载后，需要安装以下库才能从Python执行Spark任务：

```
pip3 install pyspark graphframes
```

运行pyspark，来启动pyspark REPL（交互解释器，Read-Eval-Print Loop）：

```
pyspark --driver-memory 2g --executor-memory 6g --packages graphframes:graphframes:0.7.0-spark2.4-s_2.11
```

其中--package是必须要带上的，否则会出现错误。它代表了这个程序还要应用graphframes:0.7.0-spark2.4-s_2.11所对应的jar包。对于这个参数不熟悉的读者，请用pyspark —help来了解更详细的信息。

虽然Spark作业应该在一个机器集群上执行，但是为了演示目的，我们只在一台机器上执行这些作业。在由Bill Chambers和Matei Zaharia（O'Reilly）撰写的《Spark: The Definitive Guide》中，你可以了解更多关于在生产环境中运行Spark的信息。

现在你已经准备好学习如何在Spark上运行图算法了。

Neo4j图平台

Neo4j图平台支持图数据的事务处理和分析处理。它包括图存储和使用数据管理和分析工具进行计算。集成工具集位于通用协议、API和查询语言Cypher之上，为不同的用途提供有效的访问，如图3-2所示。

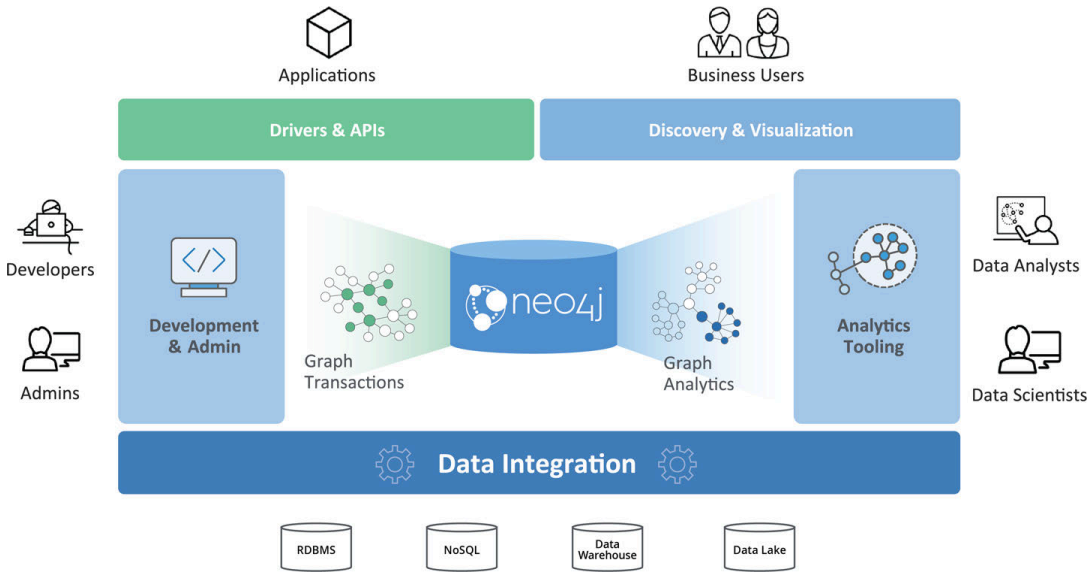


图3-2.Neo4j图平台是围绕支持事务应用程序和图分析的本机图数据库构建的。

在这本书中，我们将使用Neo4j图算法库。库作为插件安装在数据库中，并提供一组用户定义的过程（User-defined procedures），这些Procedure可以通过Cypher查询语言执行。

图算法库包括支持图分析和机器学习工作流的并行算法版本。算法在基于任务的并行计算框架之上执行，并针对Neo4j平台进行了优化。对于不同的图大小，内部实现可以扩展到数百亿个节点和关系。

结果可以作为元组流传输到客户机，表格结果可以用作进一步处理的驱动表。结果还可以作为节点属性或关系类型有效地写回数据库。

在这本书中，我们还将使用Neo4J AwesomeProcedures on Cypher（APOC）库。APOC由450多个过程和函数组成，用于帮助完成数据集成、数据转换和模型重构等常见任务。

安装NEO4j

开发人员可以便利地使用Neo4j Desktop，它包含一个本地的NEO4j数据库。Neo4j Desktop可以从Neo4j网站上下载。一旦安装并启动了Neo4j Desktop，图算法和APOC库就可以作为插件（plugins）安装。在左侧菜单中，创建一个项目并选择它。然后在要安装插件的数据库上单击“管理”。在插件选项卡上，你将看到几个插件的选项。单击图算法和APOC的安装按钮。

见图3-3和3-4。

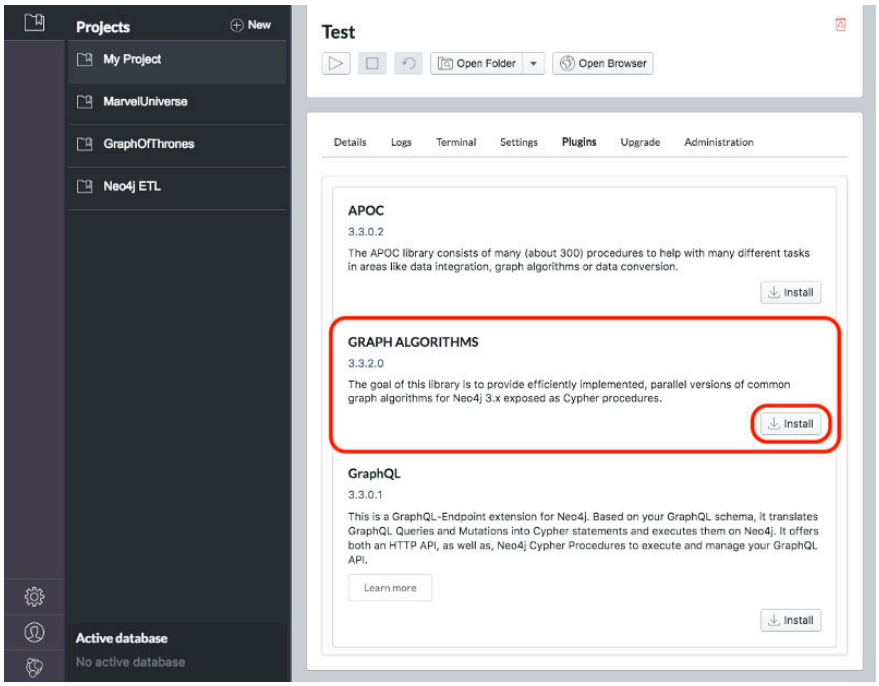


图3-3.安装图算法库

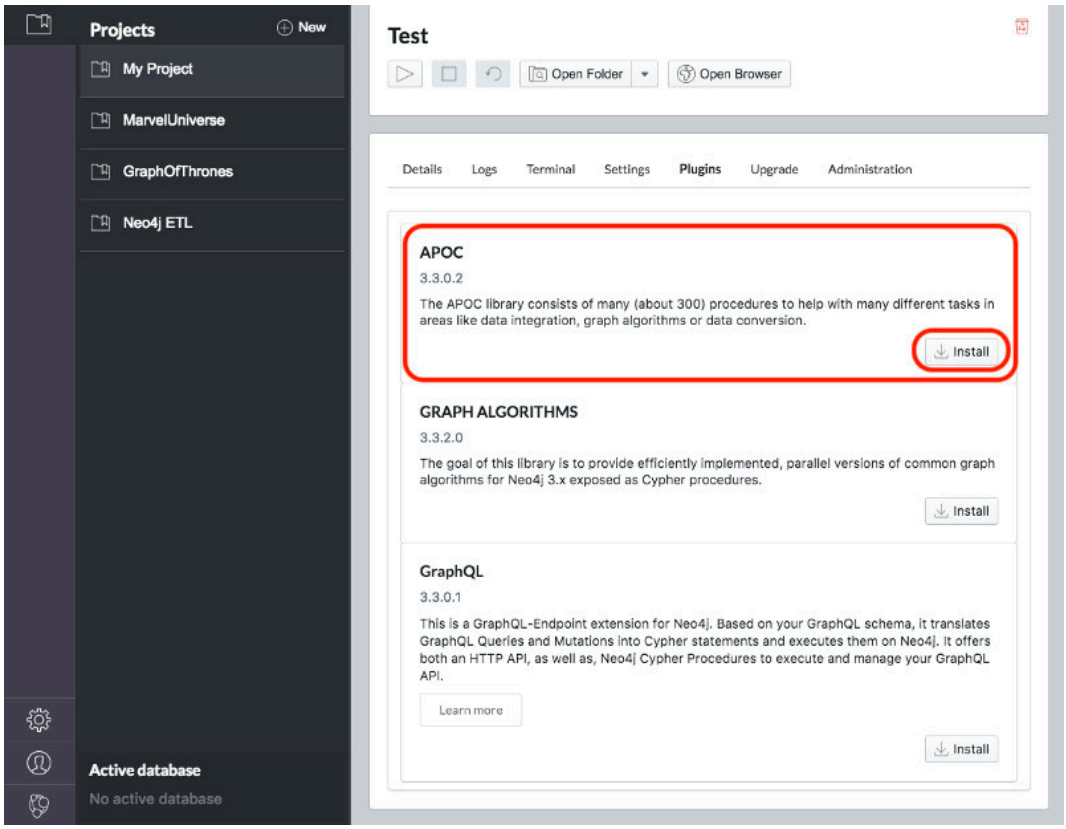


图3-4.安装APOC库

Jennifer Reif在她的博客文章“Explore New Worlds—Adding Plugins to Neo4”中更详细地解释了安装过程。现在你已经准备好学习如何在Neo4j中运行图算法了。

总结

在前面的章节中，我们已经描述了为什么图分析对于研究现实网络很重要，并研究了基本的图概念、分析和处理。这为我们了解如何应用图算法打下了坚实的基础。在下一章中，我们将开始了解如何使用Spark和Neo4j中的示例运行图算法。

第四章 路径查找和图搜索算法

图搜索（Graph Search）算法是用于在图上进行一般性发现或显式地搜索的算法。这些算法在图上找到出路径，但没有期望这些路径是在计算意义上是最优的。我们将涵盖广度优先搜索（Breadth First Search，BFS）和深度优先搜索（Deep First Search，DFS），因为它们是遍历一个图的基础算法，通常也是许多其他进一步分析的先决条件。

路径查找算法（Pathfinding）是建立在图搜索算法的基础上，它探索节点之间的路径，从一个节点开始，遍历关系，直到到达目的节点。这些算法用于识别图中的最优路由，算法可以用于诸如物流规划、最低成本呼叫或IP路由以及游戏模拟等用途。

具体来说，我们将介绍的路径查找算法包括：

- 最短路径（shortest path），以及它的两种变体（A*和Yen's）：找到两个指定节点之间的最短路径
- 所有结对最短路径（All Pairs Shortest Path，APSP）和单源最短路径（Single Source Shortest Path，SSSP）：用于查找所有节点对之间的最短路径，或从选定节点到所有其他所有节点的最短路径
- 最小生成树（Minimum Spanning Tree，MST）：用于找到一个连接树结构，在这个结构中，从指定节点访问所有节点的成本最小
- 随机行走（Random Walk）：它是机器学习工作流或其他图算法的一个有用的预处理/采样步骤。

在本章中，我们将解释这些算法是如何工作的，并在Spark和Neo4j中上运行示例。如果一个算法只在一个平台中可用，我们将仅提供一个示例，或者说明如何优化我们的实现。

图4-1显示了这些算法类型之间的主要区别，表4-1是对每个算法具体做什么给出了例子。

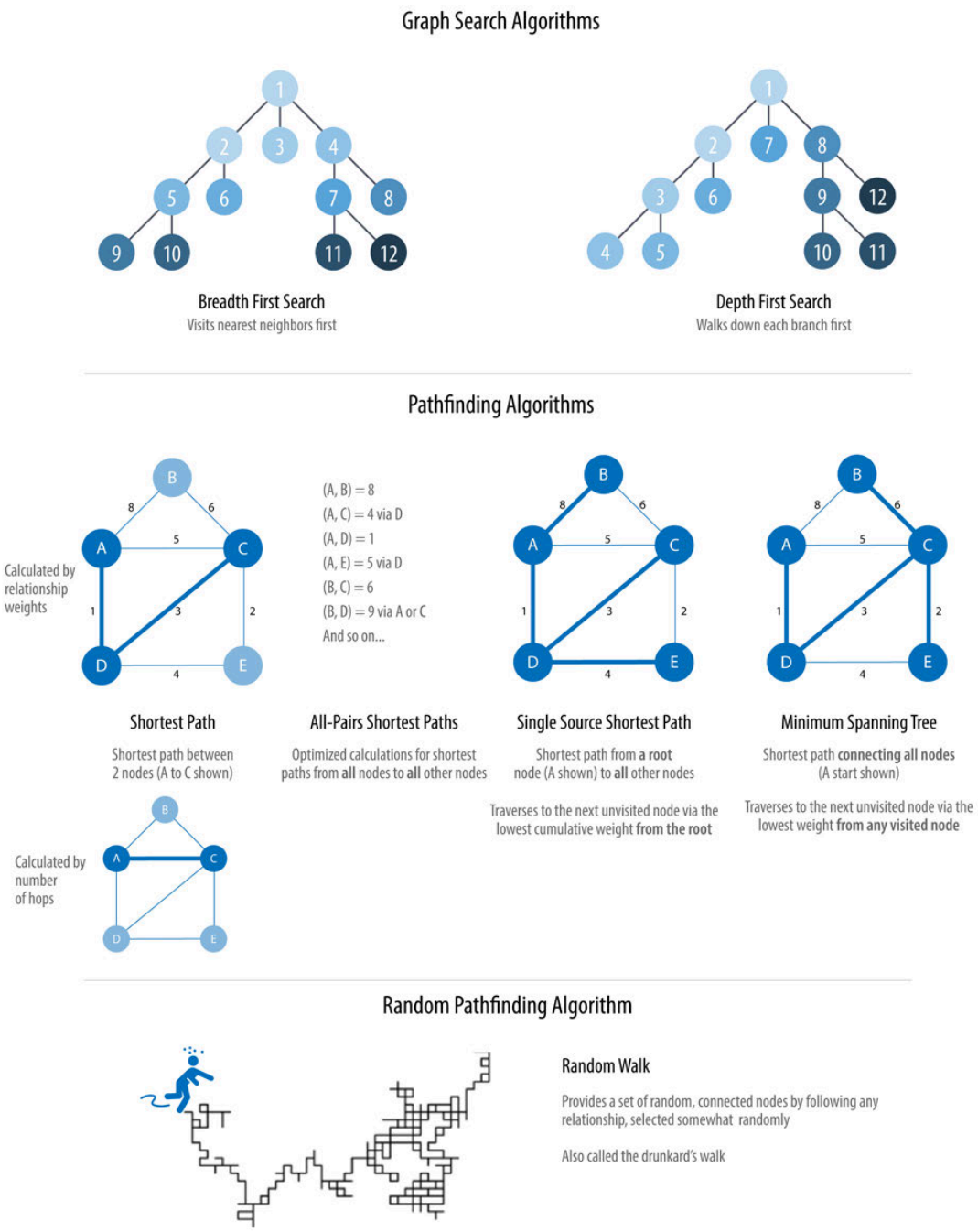


图4-1.路径查找和图搜索算法

表4-1.路径查找和图搜索算法概述

算法类型	它怎么工作	使用举例	Spark示例	Neo4j示例
广度优先搜索	遍历一个树结构，优先探索最近的邻近节点，然后是这些邻近节点的邻近节点	为了发现附近的有趣位置，在GPS系统上找到最近的节点，	Yes	No

深度优先搜索	遍历一个树结构，在回溯之前尽可能地在每个分支上深入探索	在多级选择的游戏模拟中找到一条更优的路线	No	No
最短路径及其变体A*，Yen's	计算在一个节点对之间的最短路径	找到两个位置之间的驾驶方向	Yes	Yes
所有结对最短路径	计算一个图上所有节点对的最短路径	评估在交通堵塞附近的路线变更	Yes	Yes
单源最短路径	计算一个根节点和其他所有的节点的最短路径	电话呼叫路由的最低成本路线	Yes	Yes
最小生成树	在一个连接的树结构中，计算出一个对于访问所有节点代价最小的路径	优化已连接的路线，比如埋设电缆或者垃圾搜集	No	Yes
随机行走	给定具体的步数，随机选择关系去遍历，返回来一系列的节点集合	增强机机器学习的训练，或者增强图算法的数据	Yes	Yes

首先，让我们查看一下示例中的数据集，并介绍如何将数据导入Apache Spark和Neo4j中。对于每种算法，我们将从对算法的简短描述以及有关它如何操作的相关信息开始。大多数章节还包括有关何时使用相关算法的指导。最后，我们在每个算法部分的最后，将提供用示例代码和数据集运行的结果。

让我们开始吧！

示例数据：交通图

所有的数据网络都包含节点之间的路径，因此，图搜索和路径查找是图分析的起点。交通数据集以直观和可访问的方式说明了这些关系。本章中的示例是欧洲道路网子集的图对应的数据。你可以从本书的Github下载节点和关系文件。

表 4-2. transport-nodes.csv

id	latitude	longitude	population
Amsterdam	52.379189	4.899431	821752
Utrecht	52.092876	5.10448	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

表 4-3. transport-relationships.csv

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

图4-2显示了我们想要构建的目标图。

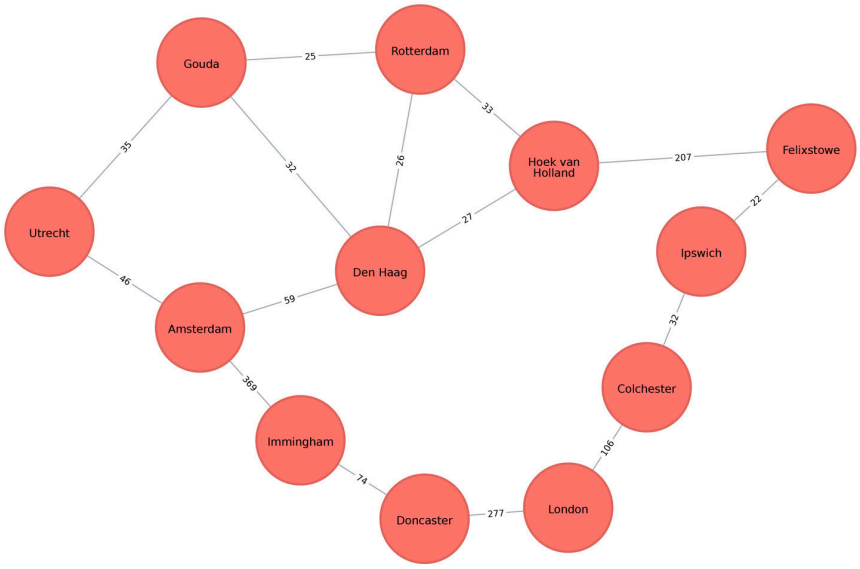


图4-2.交通图

为了简单起见，我们认为图4-2中的图表是无向的，因为城市之间的大多数道路是双向的。如果我们把图当做有向的，会得到稍微不同的结果，因为会有少量的单向街，但是总体方法仍然是相似的。但是，Spark和Neo4j的算法都在有向图上操作。在这种情况下，如果我们想使用无向图（例如双向道路），有一种简单的方法可以做到：

- 对于Spark，我们将在transport-relationships.csv的每一行创建两个关系，一个从dst到src，另一个从src到dst。
- 对于Neo4j，我们将创建单个有向的关系，并在运行算法时忽略关系方向。

了解了这些基本的建模解决方法后，我们现在可以开始从示例csv文件将图加载到Spark和Neo4j中。

将数据导入Apache Spark

从Spark开始，我们将首先从Spark和GraphFrames包中导入所需的包：

```
from graphframes import *
from pyspark.sql.types import *
```

以下的函数在示例CSV文件基础上创建出一个GraphFrame

```
def create_transport_graph():

    base = "file:///home/retire2053/source/graph_algorithms_resources/"
    node_fields = [
        StructField("id", StringType(), True),
        StructField("latitude", FloatType(), True),
        StructField("longitude", FloatType(), True),
        StructField("population", IntegerType(), True)
    ]
    nodes = spark.read.csv(base+"data/transport-nodes.csv", header=True,
                           schema=StructType(node_fields))

    rels = spark.read.csv(base+"data/transport-relationships.csv", header=True)
    reversed_rels = (rels.withColumn("newSrc", rels.dst)
                     .withColumn("newDst", rels.src)
                     .drop("dst", "src")
                     .withColumnRenamed("newSrc", "src")
                     .withColumnRenamed("newDst", "dst")
                     .select("src", "dst", "relationship", "cost"))
    relationships = rels.union(reversed_rels)
    return GraphFrame(nodes, relationships)
```

以上代码上的base变量，需要替换成为具体的路径，以便于pyspark找到csv文件。

装载这些节点是很容易的，但是对于关系，我们需要一些预处理才能把每个关系创建两遍（因为要表示双向交通）。让我们调用这个函数。

```
g = create_transport_graph()
```

将数据导入Neo4j

现在是将数据导入Neo4j的时候了，用如下的Cypher语句，先导入节点。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data" AS base
WITH base + "/transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
```

```
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
place.longitude = toFloat(row.latitude),
place.population = toInteger(row.population);
```

以下是导入关系的语句。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "/transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination);
```

虽然我们保存的是有向的关系，但是我们在本章后面内容中，将会以忽略方向的模式来执行算法。

广度优先搜索

广度优先搜索（Breadth First Search, BFS）是一种基本的图遍历算法。它从一个选定的节点开始，在一个跃点（one hop）距离处探索附近节点，然后在两个跃点（two hops）距离处访问所有节点，依此类推。

该算法最早于1959年由Edward F. Moore出版，他用它来寻找迷宫中最短的路径。然后，C.Y.Lee于1961年将其开发成一种布线算法，发表在文章An Algorithm for Path Connections and Its Applications中。

BFS是其他目标更明确的算法的基础。比如，最短路径（Shortest Path）、连接组件（Conneced Component）和紧密中心性（Closeness Centrality）都使用BFS算法。它还可以用来寻找节点之间的最短路径。

图4-3显示了如果我们执行从荷兰城市Den Haag（海牙）开始的广度优先搜索，我们访问传输图节点的顺序。城市名称旁边的数字表示访问每个节点的顺序。

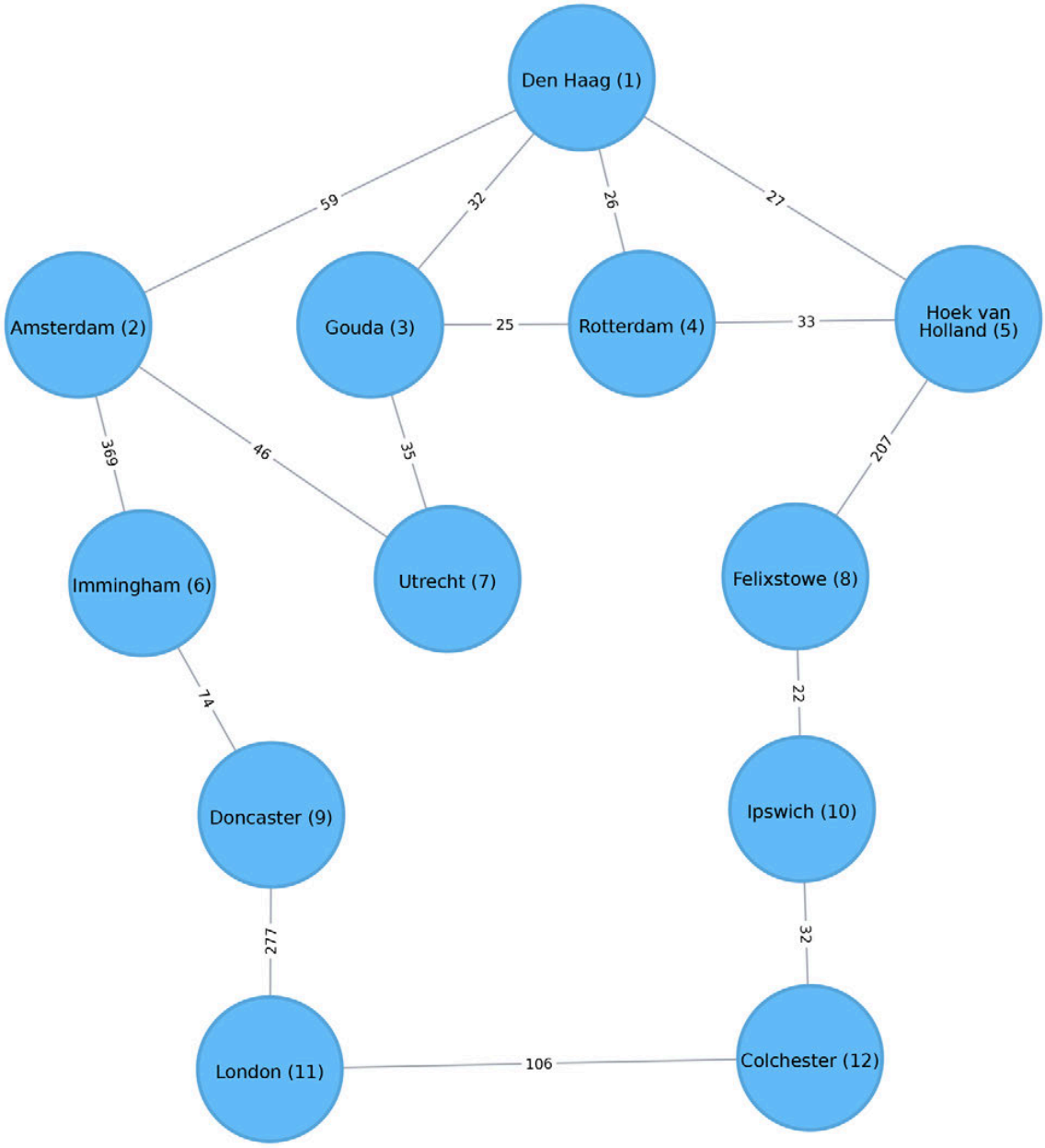


图4-3.从Den Haag开始的广度优先搜索。节点号表示遍历的顺序。

我们先去拜访所有Den Haag的最近邻近节点，然后再去拜访邻近节点的邻近节点，直到我们跑遍了所有的关系。

Apache Spark上的广度优先搜索

Spark实现的广度优先搜索算法通过两个节点之间跃点数来寻找到两个节点之间的最短路径。你可以显式（explicitly）指明目标节点或添加要满足的条件。

例如，我们可以使用BFS函数找到第一个中等城市（按照欧洲标准），人口在10万到30万之间。

让我们首先检查哪些地方的人口符合这些标准：

```
(g.vertices
.filter("population > 100000 and population < 300000")
.sort("population")
.show())
```

我们将会得到如下结果

```
+----------+--------+---------+----------+
|        id|latitude|longitude|population|
+----------+--------+---------+----------+
|Colchester|51.88921|  0.90421|    104390|
|   Ipswich|52.05917|  1.15545|    133384|
+----------+--------+---------+----------+
```

只有两个地方符合我们的标准，我们在广度优先搜索中会率先到达Ipswich。

以下代码用来查找从Den Haag到中等城市的最短路径：

```
from_expr = "id='Den Haag'"
to_expr = "population > 100000 and population < 300000 and id <> 'Den Haag'"
result = g.bfs(from_expr, to_expr)
```

结果包含描述两个城市之间的节点和关系的列。我们可以运行以下代码来查看返回的列列表：

```
print(result.columns)
```

这是我们将看到的输出：

```
['from', 'e0', 'v1', 'e1', 'v2', 'e2', 'to']
```

以e开头的列表示关系（边），以v开头的列表示节点（顶点）。我们只对节点感兴趣，所以我们过滤掉从生成的DataFrame中以e开头的任何列：

```
columns = [column for column in result.columns if not column.startswith("e")]
result.select(columns).show()
```

如果我们在PySpark中运行代码，我们将看到这个输出：

```
+--------------------+--------------------+--------------------+--------------------+
|                from|                  v1|                  v2|                  to|
+--------------------+--------------------+--------------------+--------------------+
|[Den Haag, 52.078...|[Hoek van Holland...|[Felixstowe, 51.9...|[Ipswich, 52.0591...|
+--------------------+--------------------+--------------------+--------------------+
```

如预期，bfs算法返回Ipswich！记住，当找到第一个匹配时，这个函数是满足的，如图4-3所示，ipswich在Colchester之前被遍历到。

深度优先搜索

深度优先搜索（Deep First Search，DFS）是另一种基本的图遍历算法。它从一个选定的节点开始，选择它的一个邻近节点，然后在回溯之前沿着该路径尽可能地进行遍历。

DFS最初是由法国数学家Charles Pierre Tremaux发明的用来一种解决迷宫的策略。这是对分析模拟场景建模中所有可能路径非常有用的工具。

如果我们执行从Den Haag开始的DFS，图4-4显示了访问交通图节点的顺序。

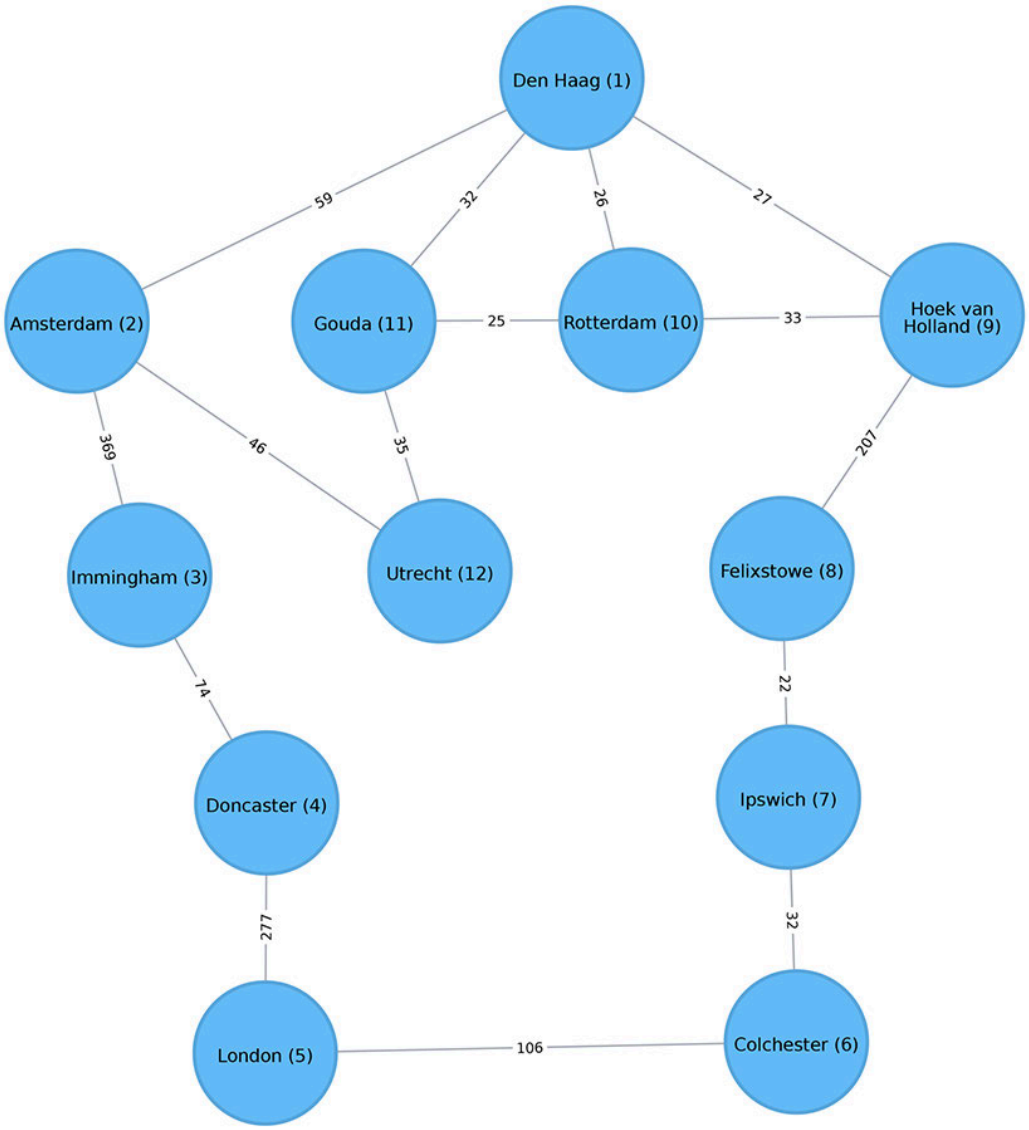


图4-4.深度优先搜索从Den Haag开始。节点号表示遍历的顺序。

注意节点顺序与BFS的区别。对于这个DFS，我们首先从Den Haag遍历到Amsterdam，然后能够到达图中的每个其他节点，而不需要回溯！

我们可以看到搜索算法是如何在图中进行移动的。现在，让我们看看路径查找的算法，它们根据跃点数或权重找到最经济的路径。权重可以是任何度量的量，例如时间、距离、容量或成本。

注意，在Spark和Neo4j上均没有深度优先搜索的代码示例。

两种特殊路径/环

图分析有两种值得注意的特殊路径。欧拉路径（Eulerian path）是每个关系访问一次的路径，另一个是汉密尔顿路径（Hamiltonian path），它是每个节点都访问一次的特殊路径。如果你在同一节点开始和结束，它被认为是一个环形旅行，这种路径可能同时是欧拉路径和汉密尔顿路径。图4-5展示了一些例子。

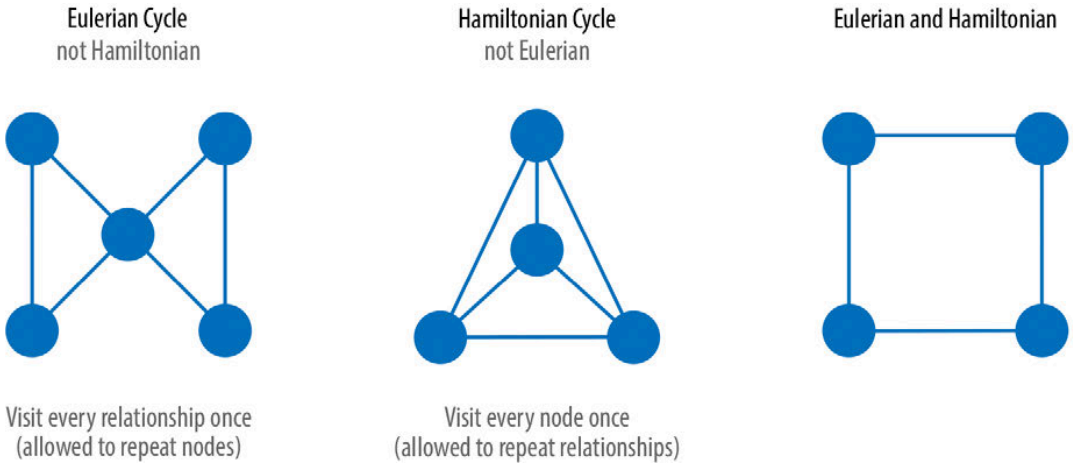


图4-5.欧拉环和汉密尔顿环是具有特殊的历史意义的路径。

第一章的哥尼斯堡七桥问题是在寻找欧拉环。很容易想到，欧拉环可以适用于如定向除雪或邮件传递等其他场景。不仅仅如此，欧拉路径也被其他算法用于处理树结构中的数据，并且在数学上比其他路径更容易研究。

汉密尔顿环最广为人知的是它被用于商旅问题（Traveling Salesman Problem，TSP）。TSP是这样的：对于一个销售员来说，访问他们指定的每个城市并返回原来的城市，最短路线是什么？尽管看起来与欧拉巡演相似，但在近似替换的情况下，TSP的计算量更大。它被用于各种各样的计划、物流和优化问题中。

最短路径

最短路径（Shortest Path）算法计算一对节点之间的最短（加权）路径。它在和用户产生交互的动态工作流很有用，因为它可以实时工作。

路径查找算法的历史可以追溯到19世纪，被认为是一个经典的图论问题。它在20世纪50年代早期的可替换路由问题中体现突出的作用；也就是说，如果最短的路由被阻塞，则找到第二条最短的路由。1956年，Edsger Dijkstra发明了这些最著名的算法。

Dijkstra的最短路径算法首先找到从起始节点到直接连接节点的最小权重关系。它跟踪这些权重并移动到“最近”节点。然后，它执行相同的计算，只不过权重的累积是从初始节点开始算的。算法继续持续迭代，就可以评估从初始节点的一个累积权重的“波浪”，并始终选择要前进的最小加权累积路径，直到到达目标节点。

在图分析中，当描述关系和路径时，你会注意到“权重”（weight）、“成本”（cost）、“距离”（distance）和“跃点数”（hop）等术语的使用。“权重”是关系的特定属性的数值。“成本”（cost）的用法与此类似，但在考虑路径的总权重时，我们会更经常看到它。

“距离”（distance）通常在算法中用作关系属性的名称，表示一对节点之间的遍历成本。并不一定是非要在实际物理测量中才这么用。跃点数通常用于表示两个节点之间经历的关系数。你可能会看到其中一些术语组合在一起，如“到伦敦有五个跃点的距离”或“这是距离的最低成本”。

最短路径算法的使用场景

利用最短路径算法可以在一对节点之间找到最佳路径，衡量的指标是跃点数或者任何权重值。例如，它可以提供关于分离程度、点之间短距离或最小扩展路径的实际答案。你也可以使用这个算法简单地探索特别节点之间的连接。

使用场景可以包括：

- 确定位置之间的方向。在手机地图软件（比如谷歌地图）用最短路径算法或者某些变体算法来提供驾驶导航。
- 发现社交网络中人之间的离散距离。比如，当你看到某人在社交网站上的简介时，它会显示出图中你离他之间有多少人，也会显示你们之间的共同联系人。
- 找到一个演员和Kevin Bacon所出现的电影之间的逻辑距离。在Oracle of Bacon website. The Erd.s Number Project 提供了一个“和Paul Erdos的合

作网络”的类似功能，其中Paul Erdos是20世纪最有影响力的数学家之一。

Dijkstra的算法不支持负的权重。该算法假定，在路径上增加一个关系，只会让路径变得更长。如果有负权重，这个假设会被破坏掉。

Neo4j上的最短路径算法

Neo4j图算法库有一个内置的存储过程，我们可以使用它来计算无权重和有权重的最短路径。让我们先学习如何计算无权重的最短路径。

所有Neo4j的最短路径算法都假定基础图是无向的。你可以通过传递参数 direction: "OUTGOING" 或者direction: "INCOMING"来更改默认设置。Outgoing代表了出链，Incoming代表了入链。

为了让Neo4j的最短路径算法忽略权重，我们需要将空值作为过程的第三个参数传递，这表明在执行算法时我们不想考虑权重属性。然后，算法将假定每个关系的默认权重为1.0：

```
MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

此查询返回以下输出：

```
+--------------------+
| place       | cost |
+--------------------+
| "Amsterdam" | 0.0  |
| "Immingham" | 1.0  |
| "Doncaster" | 2.0  |
| "London"    | 3.0  |
+--------------------+
```

这里的成本是累积的关系数量（或跃点数量）。这与我们在Spark中使用广度优先搜索看到的相同。

我们甚至可以通过编写一些带后置处理的Cypher来计算遵循这条路径的总距离。以下过程计算最短的无权重路径，然后计算出该路径的实际成本：

```
MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
WITH collect(algo.getNodeById(nodeId)) AS path
UNWIND range(0, size(path)-1) AS index
WITH path[index] AS current, path[index+1] AS next
WITH current, next, [(current)-[r:EROAD]-(next) | r.distance][0] AS distance
WITH collect({current: current, next:next, distance: distance}) AS stops
UNWIND range(0, size(stops)-1) AS index
WITH stops[index] AS location, stops, index
RETURN location.current.id AS place,
reduce(acc=0.0,
distance in [stop in stops[0..index] | stop.distance] |
acc + distance) AS cost;
```

如果前面的代码感觉有点笨拙，那么请注意，最复杂的部分是如何处理数据，以包括整个遍历的成本。这有助于我们在需要累积路径成本时记住这一点。

查询返回以下结果：

```
+---------------------+
| place       | cost  |
+---------------------+
| "Amsterdam" | 0.0   |
| "Immingham" | 369.0 |
| "Doncaster" | 443.0 |
| "London"    | 720.0 |
+---------------------+
```

图4-6显示了从Amsterdam到London的无权重的最短路径，使我们通过最少的城市。总成本是720公里。

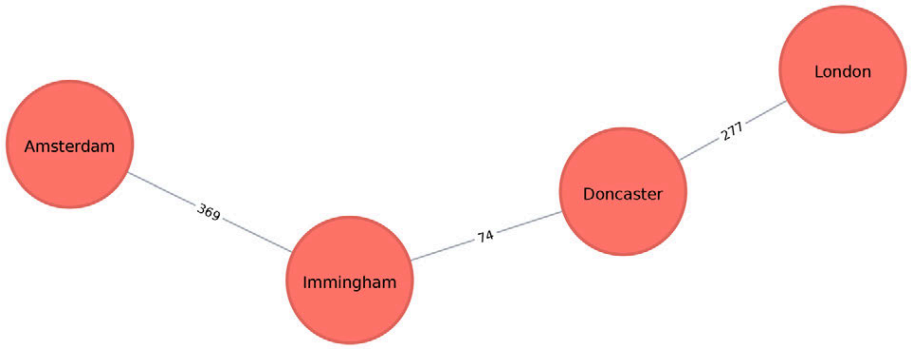


图4-6.Amsterdam和London之间的无权重最短路径，会选择访问节点数最少的路线，在地铁系统等需要较少站点的情况下可能非常有用。然而，在驾驶场景中，我们可能更感兴趣的是使用最短权重路径的总成本。

Neo4j上的加权重最短路径

我们可以执行有权重的最短路径算法，找到Amsterdam和London之间的最短路径，如下所示：

```
MATCH (source:Place {id: "Amsterdam"}),
(destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, "distance")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

传递给此算法的参数为：

参数	意义
source	最短路径搜索开始的节点
destination	最短路径搜索结束的节点
distance	关系属性的名称，指示一对节点之间的遍历成本。成本是两个地点之间的公里数

查询返回以下结果：

```
+---------------------------+
| place              | cost |
+---------------------------+
| "Amsterdam"        | 0.0  |
| "Den Haag"         | 59.0 |
| "Hoek van Holland" | 86.0 |
| "Felixstowe"       | 293.0|
| "Ipswich"          | 315.0|
| "Colchester"       | 347.0|
| "London"           | 453.0|
+---------------------------+
```

最快的路线是通过Den Haag，Hoek van Holland，Felixstowe，Ipswich和Colchester！显示的成本是我们在城市中前进时的累积的距离总量。首先，我们从Amsterdam去Den Haag，成本是59km。然后我们从Den Haag到Hoek van Holland，累计花费86km，以此类推。最后，我们从Colchester到达London，总共花费453km。

请记住，无权重的最短路径的总成本为720公里，因此在计算最短路径时，我们可以考虑权重，从而节省267公里。

Apache Spark上的加权最短路径

在Apache Spark的广度优先搜索中，我们学习了如何找到两个节点之间的最短路径。这个最短路径是基于跃点数的，因此与加权最短路径有所不同，加权最短路径这将告诉我们城市之间最短的总距离。

如果我们想要找到最短的权重路径（在本例中是距离），我们需要使用cost这个属性，它用于各种类型的权重计算。这个选项不能与GraphFrame一起使用，因此我们需要使用它的aggregateMessages框架编写我们自己的加权最短路径版本。虽然Spark的大多数算法示例都使用了从库中调用算法的简单过程，但是我们可以选择编写自己的函数。有关AggregateMessages的更多信息，请参阅Message passing via AggregateMessages的AggregateMessages部分。

如果可行的话，我们建议利用现有的、经过测试的库。编写我们自己的函数，特别是对于更复杂的算法，需要对数据和计算有更深的理解。下面的示例应被视为参考实现，并需要在运行于更大的数据集之前进行优化。那些对编写自己的函数不感兴趣的人可以跳过这个例子。

在创建函数之前，我们将导入一些将要使用的库：

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

Aggregate_Message模块是GraphFrames库的一部分，包含一些有用的helper函数。

现在我们来编写函数。我们首先创建一个用户定义函数，用于构建源和目标之间的路径：

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

现在，对于主函数，它计算从原点开始的最短路径，并在访问目的地后立即返回：

```
def shortest_path(g, origin, destination, column_name="cost"):
```

```
    if g.vertices.filter(g.vertices.id == destination).count() == 0:
        return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
                .withColumn("path", F.array()))
    vertices = (g.vertices.withColumn("visited", F.lit(False))
                .withColumn("distance", F.when(g.vertices["id"] == origin, 0)
                           .otherwise(float("inf")))
                .withColumn("path", F.array()))
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
        current_node_id = g2.vertices.filter('visited == False').sort("distance").first().id
        msg_distance = AM.edge[column_name] + AM.src['distance']
         msg_path = add_path_udf(AM.src["path"], AM.src["id"])
        msg_for_dst = F.when(AM.src['id'] == current_node_id,
                             F.struct(msg_distance, msg_path))
        new_distances = g2.aggregateMessages(F.min(AM.msg).alias("aggMess"),
                                             sendToDst=msg_for_dst)
        new_visited_col = F.when(g2.vertices.visited | (g2.vertices.id == current_node_id),True).otherwise(False)
        new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                                  (new_distances.aggMess["col1"]
                                   < g2.vertices.distance),
                                  new_distances.aggMess["col1"]).otherwise(g2.vertices.distance)

        new_path_col = F.when(new_distances["aggMess"].isNotNull() & (new_distances.aggMess["col1"]<
g2.vertices.distance), new_distances.aggMess["col2"]
                              .cast("array<string>")).otherwise(g2.vertices.path)
        new_vertices = (g2.vertices.join(new_distances, on="id",
        how="left_outer")
                                       .drop(new_distances["id"])
                                       .withColumn("visited", new_visited_col)
                                       .withColumn("newDistance", new_distance_col)
                                       .withColumn("newPath", new_path_col)
                                       .drop("aggMess", "distance", "path")
                                       .withColumnRenamed('newDistance', 'distance')
                                       .withColumnRenamed('newPath', 'path'))
        cached_new_vertices = AM.getCachedDataFrame(new_vertices)
        g2 = GraphFrame(cached_new_vertices, g2.edges)
        if g2.vertices.filter(g2.vertices.id == destination).first().visited:
            return (g2.vertices.filter(g2.vertices.id == destination)
                    .withColumn("newPath", add_path_udf("path", "id"))
                    .drop("visited", "path")
                    .withColumnRenamed("newPath", "path"))
    return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
.withColumn("path", F.array()))
```

如果我们在函数中存储对任何DataFrame的引用，我们需要使用AM.getCachedDataFrame函数来缓存它们，否则在执行过程中会遇到内存泄漏。在最短路径函数中，我们使用这个函数来缓存顶点和新的顶点DataFrame。

如果我们想找到Amsterdam和Colchester之间最短的路径，我们可以这样称呼这个函数：

```
result = shortest_path(g, "Amsterdam", "Colchester", "cost")
result.select("id", "distance", "path").show(truncate=False)
```

确实，经过一段时间的执行（示例代码的运行效率比较低），结果如下：

```
+----------+--------+-----------------------------------------------------------------------+
|id        |distance|path                                                                   |
+----------+--------+-----------------------------------------------------------------------+
|Colchester|347.0   |[Amsterdam, Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]|
+----------+--------+-----------------------------------------------------------------------+
```

Amsterdam与Colchester之间最短路径的总距离为347公里，途经Den Haag、Hoek van Holland、Felixstowe和Ipswich。相比之下，我们使用广度优先搜索算法（参见图4-4）计算出的位置之间关系数量方面的最短路径则会经过Immingham、Doncaster和London。

最短路径变体：A*

A*最短路径算法改进Dijkstra的算法，它更快一些，因为它在确定下一个探索路径时可用的额外信息都包含进来，将这些额外信息作为启发式函数的一部分。

该算法由Peter Hart、Nils Nilsson和Bertram Raphael发明，并在1968年的论文“A Formal Basis for the Heuristic Determination of Minimum Cost Paths”中进行了描述。

在其核心循环的每次迭代中，A*算法都要决定哪个子路径要展开往下探索。这样做是基于对到达目标节点的成本的启发式估计。

在应用了估计路径成本的启发式方法中要考虑周全。如果启发算法低估了路径成本，可能多余地包括了一些可能应当被消除的路径，但结果仍然是准确的。但是，如果启发式方法高估了路径成本，它可能会跳过实际的较短路径（错误地估计为较长路径），而这些路径实际上应当被评估和遍历，这就可能导致不准确的结果。

A*选择能够最小化以下函数的路径：

f(n) = g(n) + h(n)

在这个函数中，

- g(n) 是从起点到节点n的路径成本。
- h(n) 是从节点n到目标节点的路径的估计成本，是在启发计算的结果。

在Neo4j的实现中，地理空间距离（中间点到目标点的物理空间距离）被用作启发。在我们的示例交通数据集中，我们使用每个位置的纬度和经度作为启发式函数的输入。

Neo4j上的A*算法

以下查询执行A*算法，以查找Den Haag和London之间的最短路径：

```
MATCH (source:Place {id: "Den Haag"}),
(destination:Place {id: "London"})
CALL algo.shortestPath.astar.stream(source,
destination, "distance", "latitude", "longitude")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

传递给此算法的参数为：

参数	意义
source	最短路径搜索开始的节点
destination	最短路径搜索结束的节点
distance	关系属性的名称，指示一对节点之间的遍历成本。成本是两个地点之间的公里数
latitude	节点属性的名称，用于表示每个节点的纬度，作为地理空间启发式计算的一部分
longitude	节点属性的名称，用于表示每个节点的经度，作为地理空间启发式计算的一部分

结果如下：

```
+---------------------------+
| place                | cost   |
```

```
+----------------------------+
| "Den Haag"          | 0.0  |
| "Hoek van Holland"  | 27.0 |
| "Felixstowe"        | 234.0|
| "Ipswich"           | 256.0|
| "Colchester"        | 288.0|
| "London"            | 394.0|
+----------------------------+
```

使用最短路径算法（无变体）将得到相同的结果，但在更复杂的数据集上，A*算法将更快，因为它评估的路径更少。

最短路径变体：Yen's K-最短路径

Yen’ s k-最短路径算法与最短路径算法相似，但它不只是在两对节点之间找到最短路径，而是计算最短路径的第二最短路径、第三最短路径等，最多可得到k-1种不同的路径。

Jin Y. Yen于1971年发明了该算法，并将其描述为 “Finding the K Shortest Loopless Paths in a Network” 。该算法在寻找绝对的最短路径并非我们唯一目标时，会有助于找到替代的路径。当我们需要多个备份计划时，它会特别有用！

Neo4j上的Yen's算法

以下查询执行Yen's算法，以查找Gouda和Felixstowe之间的最短路径：

```
MATCH (start:Place {id:"Gouda"}),
(end:Place {id:"Felixstowe"})
CALL algo.kShortestPaths.stream(start, end, 5, "distance")
YIELD index, nodeIds, path, costs
RETURN index,
[node in algo.getNodesById(nodeIds[1..-1]) | node.id] AS via,
reduce(acc=0.0, cost in costs | acc + cost) AS totalCost
```

传递给此算法的参数为：

参数	意义
start	最短路径搜索开始的节点
end	最短路径搜索结束的节点
5	要查找的最短路径的最大数目

distance	关系属性的名称，指示一对节点之间的遍历成本。成本是两个地点之间的公里数

在返回最短路径之后，我们查找每个节点ID的关联节点，然后从集合中筛选出开始和结束节点。

结果如下：

```
+-----------------------------------------------------------------------------+
| index | via                                                       | totalCost |
+-----------------------------------------------------------------------------+
| 0     | ["Rotterdam", "Hoek van Holland"]                         | 265.0     |
| 1     | ["Den Haag", "Hoek van Holland"]                          | 266.0     |
| 2     | ["Rotterdam", "Den Haag", "Hoek van Holland"]             | 285.0     |
| 3     | ["Den Haag", "Rotterdam", "Hoek van Holland"]             | 298.0     |
| 4     | ["Utrecht", "Amsterdam", "Den Haag", "Hoek van Holland"] | 374.0     |
+-----------------------------------------------------------------------------+
```

图4-7显示了Gouda和Felixstowe之间的最短路径。

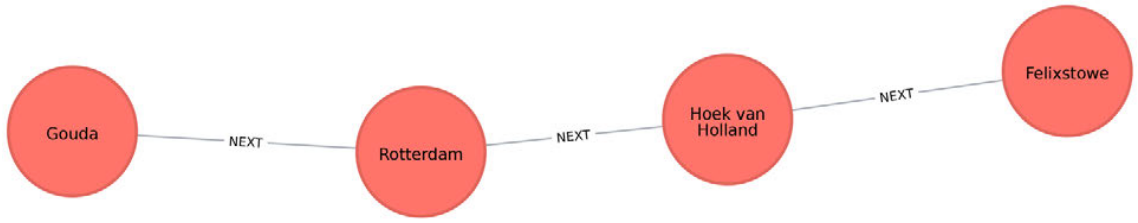


图4-7.Gouda和Felixstowe之间的最短路线

图4-7中的最短路径与其他最短路径结果相比一下，会感觉很有趣。它说明，有时你可能需要考虑几个最短路径或其他选择。在这个例子中，第二条最短的路线只比最短的路线长1公里。如果我们喜欢风景，我们可以选择稍长一点的路线。

所有结对最短路径

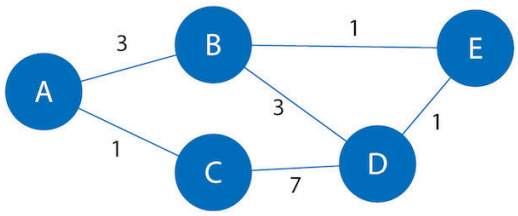
所有结对最短路径（All Pairs Shortest Path, APSP）算法计算所有对节点之间的最短（加权）路径。相比于对图中的每一对节点运行单源最短路径算法（Single Source Shortest Path, SSSP），APSP的效率更高。

APSP是通过跟踪迭代过程中已计算的距离并在节点上进行并行优化。在计算到未遍历节点的最短路径时，可以复用这些已知距离。你可以按照下一节中的示例，更好地了解算法的工作原理。

有些节点之间可能无法相互的连接，这意味着这些节点之间没有最短路径。算法不会返回这些节点对的距离。

APSP的原理

当你按照如下的顺序来考察时，就会很容易理解APSP。图4-8中的表将从节点A开始运行。



All nodes start with a ∞ distance and then the start node is set to a 0 distance			Each Step Keeps or Updates to the Lowest Value Calculated so Far Only steps for node A to all nodes shown				
			1st from A	2nd from A to C to Next	3rd from A to B to Next	4th from A to E to Next	5th from A to D to Next
A	∞	0	0	0	0	0	0
B	∞	∞	3	3	3	3	3
C	∞	∞	1	1	1	1	1
D	∞	∞	∞	8	6	5	5
E	∞	∞	∞	∞	4	4	4

图4-8.从节点A到所有其他节点的最短路径的步骤，其中更新的数据用蓝色的底色显示

一开始，APSP算法假定到所有节点之间距离为无限远（∞）。选择开始节点后，到该节点的距离设置为0。之后的计算过程如下：

1) 从开始节点A，我们评估移动到可到达的节点的成本，并更新成本的值。在B（成本为3）或C（成本为1），我们选择最小的成本。结果是，我们选择C继续下一阶段的遍历。
2) 现在，从节点C作为中间节点，算法更新从A到所有C能直接到达节点的累积距离。当只有找到的距离成本时，才会更新值：A=0，B=3，C=1，D=8，E=∞，此时D被更新。

3) B还没有被选择成为中间节点，这次选择B作为中间节点，节点B与节点A、D和E有关系。算法通过将从A到B的距离与从B到邻近节点（A、E、E）的距离相加来计算到这些节点的距离。请注意，从开始节点A到当前节点的最低成本始终保留为当前的成本（sunk cost，已投入的成本）。距离（用小d表示）计算结果：

   a) d(A,A) = d(A,B) + d(B,A) = 3 + 3 = 6
   b) d(A,D) = d(A,B) + d(B,D) = 3 + 3 = 6
   c) d(A,E) = d(A,B) + d(B,E) = 3 + 1 = 4

ii. 在此步骤中，从节点A到B再回到A的距离，d(A,A) = d(A,B) + d(B,A) = 3 + 3 = 6，它大于已计算的最短距离0，因此其值不会更新。

iii. 节点D的计算值6和E的计算值4，小于先前计算的距离，因此它们被更新。

接下来选择节点E为中间节点。到D节点的累积总数5低于此前计算，因此它是唯一更新的值。

当最终计算中间节点D时，没有新的最小路径权重；如果没有更新任何内容，则算法终止。

尽管APSP算法都经过了优化，可以为每个节点并行运行计算，但对单个节点出发，已经是一个非常大的图。如果只需要计算某类节点之间的路径，请考虑使用子图。

APSP算法的使用场景

当最短路由被阻塞或变得不理想时，APSP通常用于找到所有备用路由。例如，该算法用于逻辑路由规划，以确保多样性路由的最佳多路径。当需要考虑所有或大部分节点之间的所有可能路由时，请使用APSP。

示例用例包括：

- 优化城市设施的位置和货物分配。其中一个例子是确定运输网格中不同路段上预期的交通负荷。有关更多信息，请参阅R. C. Larson和A. R. Odoni的著作, Urban Operations Research (Prentice-Hall)。
- 作为数据中心设计算法的一部分，查找具有最大带宽和最小延迟的网络。在A.R.Curtis等人的论文REWIRE: An Optimization-Based Framework for Data Center Network Design中，有更多关于这种方法的细节。

Apache Spark上的APSP

Spark的shortestPaths函数用于查找从所有节点到一组称为地标的节点的最短路径。如果我们想找到从每个地点到 Colchester, Immingham，和Hoek van Holland的最短路径，我们将编写以下查询：

```
result = g.shortestPaths(["Colchester", "Immingham", "Hoek van Holland"])
result.sort(["id"]).select("id", "distances").show(truncate=False)
```

在PySpark中运行该代码，我们将看到这个输出：

```
+----------------+------------------------------------------------------------+
|id              |distances                                                   |
+----------------+------------------------------------------------------------+
|Amsterdam       |[Immingham -> 1, Hoek van Holland -> 2, Colchester -> 4]|
|Colchester      |[Colchester -> 0, Immingham -> 3, Hoek van Holland -> 3]|
|Den Haag        |[Hoek van Holland -> 1, Immingham -> 2, Colchester -> 4]|
|Doncaster       |[Immingham -> 1, Colchester -> 2, Hoek van Holland -> 4]|
|Felixstowe      |[Hoek van Holland -> 1, Colchester -> 2, Immingham -> 4]|
|Gouda           |[Hoek van Holland -> 2, Immingham -> 3, Colchester -> 5]|
|Hoek van Holland|[Hoek van Holland -> 0, Immingham -> 3, Colchester -> 3]|
|Immingham       |[Immingham -> 0, Colchester -> 3, Hoek van Holland -> 3]|
|Ipswich         |[Colchester -> 1, Hoek van Holland -> 2, Immingham -> 4]|
|London          |[Colchester -> 1, Immingham -> 2, Hoek van Holland -> 4]|
|Rotterdam       |[Hoek van Holland -> 1, Immingham -> 3, Colchester -> 4]|
|Utrecht         |[Immingham -> 2, Hoek van Holland -> 3, Colchester -> 5]|
+----------------+------------------------------------------------------------+
```

在 “distance” 列中每个地点旁边的数字是从源节点到该位置需要穿过的城市之间的关系（道路）数。在我们的例子中，Colchester是我们的目的地城市之一，你可以看到它有0个节点需要穿过才能到达它自己，但是从Immingham和Hoek van Holland那里要经历三个跃点。如果我们计划旅行，我们可以利用这些信息来帮助我们最大限度地利用时间。

Neo4j上的APSP

Neo4j实现了APSP的并行算法，该算法返回每对节点之间的距离。

该Procedure的第一个参数是用于计算最短权重路径的属性。如果我们将其设置为空，那么算法将计算所有节点对之间的无权重最短路径。

运行如下查询：

```
CALL algo.allShortestPaths.stream(null)
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
algo.getNodeById(targetNodeId).id AS target,
distance
ORDER BY distance DESC
LIMIT 10
```

此算法返回每对节点之间的最短路径两次，每次以其中的一个节点为源节点。如果你评估单向街道的有向图，这将很有帮助。但是，我们不需要看到每个路径两次，因此我们使用sourceNodeId < targetNodeId 这个谓词来筛选，保留其中一个。

结果如下：

```
+-----------------------------------------+
| source       | target       | distance  |
+-----------------------------------------+
| "London"     | "Gouda"      | 5.0       |
| "Utrecht"    | "Ipswich"    | 5.0       |
| "London"     | "Rotterdam"  | 5.0       |
| "Colchester" | "Gouda"      | 5.0       |
| "Utrecht"    | "Colchester" | 5.0       |
| "Amsterdam"  | "Colchester" | 4.0       |
| "Immingham"  | "Ipswich"    | 4.0       |
| "Den Haag"   | "Colchester" | 4.0       |
| "Doncaster"  | "Felixstowe" | 4.0       |
| "Utrecht"    | "Felixstowe" | 4.0       |
+-----------------------------------------+
```

这个输出显示了10对位置，它们之间的经历的关系最多，因为我们要求按降序排列结果（desc）。

如果要计算最短的加权路径，那就不将空值作为第一个参数传递，而是传递包含最短路径计算中使用的成本的属性名。这样，算法对该属性进行评估，得出每对节点之间的最短加权路径。

运行如下查询：

```
CALL algo.allShortestPaths.stream("distance")
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
algo.getNodeById(targetNodeId).id AS target,
distance
ORDER BY distance DESC
LIMIT 10
```

结果如下：

```
+-----------------------------------------+
| source      | target             | distance |
+-----------------------------------------+
| "Doncaster" | "Hoek van Holland" | 529.0    |
| "Doncaster" | "Rotterdam"        | 528.0    |
| "Doncaster" | "Gouda"            | 524.0    |
| "Immingham" | "Felixstowe"       | 511.0    |
| "Den Haag"  | "Doncaster"        | 502.0    |
| "Immingham" | "Ipswich"          | 489.0    |
| "Utrecht"   | "Doncaster"        | 489.0    |
| "Utrecht"   | "London"           | 460.0    |
| "Immingham" | "Colchester"       | 457.0    |
| "Immingham" | "Hoek van Holland" | 455.0    |
+-----------------------------------------+
```

现在我们看到的是所有最短距离节点对中最远的10个节点对。注意，Doncaster和其他荷兰的城市一起出现。如果我们想在这这些地区进行一次公路旅行的话，看起来要开很长的路。

单源最短路径

单源最短路径（Single Source Shortest Path，SSSP）算法在Dijkstra的最短路径算法的相同时间前后出现，作为单一最短路径和单源所有最短路径问题的实现。

SSSP算法计算了从根节点到图中所有其他节点的最短（权重）路径，如图4-9所示。

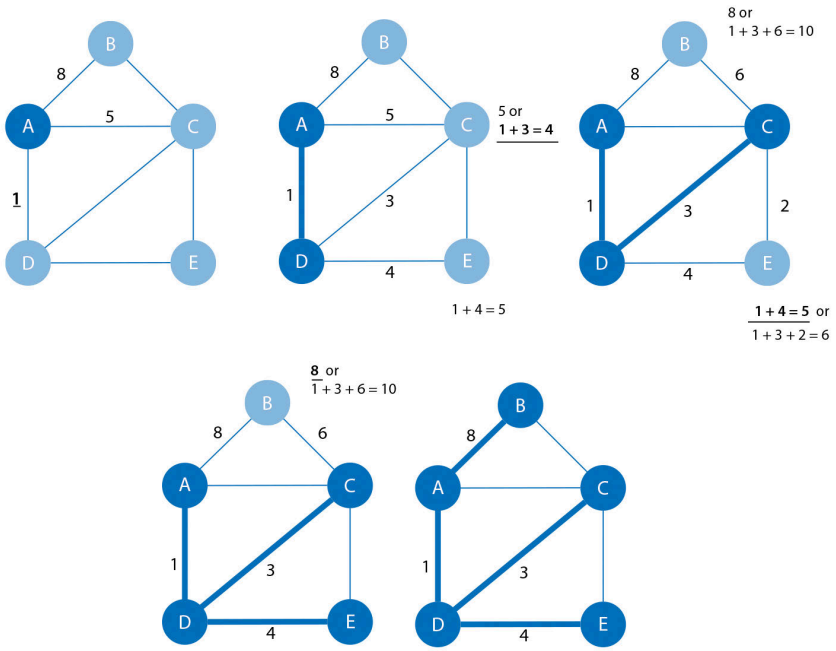


图4-9单源最短路径算法的步骤

它是按照如下的步骤工作的

1) 它始于一个根节点，与这个根节点相关所有的路径都将被测量。在图4-9中，我们选择了一个节点A作为根。
2) 从根节点计算出权重最小的附近节点，并将这个节点选择出来，加入到树中，与此一同被加入树上的还有和这个节点相连的其他节点。在这个例子中，d(A,D)=1。
3) 任何未访问节点中，从根节点到它的累积权重最小的节点被选择出来，被添加到树上。我们在图4-9中的选择是d(A,B)=8,d(A,C)=5或者是A-D-C这个路径的4，d(A,E)=5，因此，A-D-C被选中，C被添加到树上。
4) 持续按照这个进行下去，直到没有新的节点被添加进来，我们就得到了SSSP的最终结果。

SSSP算法的使用场景

当你需要评估从一个固定的起始点到所有其它单个节点的最佳路径时，就适合使用SSSP。因为路由是根据一个节点到根节点的总路径权重来选择的，所以SSSP算法被用来确定到每个节点的最佳路径，但在所有节点都需要被访问的遍历中（比如，汉密尔顿路径），这个算法是不适用的。

例如，SSSP有助于确定用于紧急服务的主要路线，因为在每次一次事故中，你并不需要到每个地点，你只需要全部计算并评估它。在垃圾回收的场景中，SSSP并不适用，因为垃圾回收的场景中，你必须要到达每个地点（在后者这个场景中，就可以使用最小生成树的算法，这将在后面讲到）。

示例用法包括

- 探测拓扑变化，如链接失败，并建议在第二个路径结构中采用新的路径结构。
- 在像局域网（LAN）这样的自主系统中，使用Dijkstra的IP路由协议场景。

Apache Spark上的SSSP

我们可以改编shortest_path函数来计算一个节点到所有其他节点的距离。注意，我们将再次使用Spark的aggregateMessages框架来定制我们的函数。

我们将首先导入与以前相同的库：

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

我们使用相同的用户定义函数来构造路径：

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

以下是主函数，它计算从源节点开始的最短路径：

```
def sssp(g, origin, column_name="cost"):
    vertices = g.vertices.withColumn("visited", F.lit(False)).withColumn("distance",F.when(g.vertices["id"] == origin,
0).otherwise(float("inf"))).withColumn("path", F.array())
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
        current_node_id = g2.vertices.filter('visited == False').sort("distance").first().id
        msg_distance = AM.edge[column_name] + AM.src['distance']
        msg_path = add_path_udf(AM.src["path"], AM.src["id"])
        msg_for_dst = F.when(AM.src['id'] == current_node_id, F.struct(msg_distance, msg_path))
        new_distances = g2.aggregateMessages(F.min(AM.msg).alias("aggMess"), sendToDst=msg_for_dst)
       new_visited_col = F.when(g2.vertices.visited | (g2.vertices.id == current_node_id), True).otherwise(False)
        new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                                  (new_distances.aggMess["col1"] < g2.vertices.distance),
                                  new_distances.aggMess["col1"]) \
                            .otherwise(g2.vertices.distance)
        new_path_col = F.when(new_distances["aggMess"].isNotNull() &
```

```
                            (new_distances.aggMess["col1"] < g2.vertices.distance),
                            new_distances.aggMess["col2"].cast("array<string>")) \
                    .otherwise(g2.vertices.path)
        new_vertices = g2.vertices.join(new_distances, on="id", how="left_outer") \
            .drop(new_distances["id"]) \
            .withColumn("visited", new_visited_col) \
            .withColumn("newDistance", new_distance_col) \
            .withColumn("newPath", new_path_col) \
            .drop("aggMess", "distance", "path") \
            .withColumnRenamed('newDistance', 'distance') \
            .withColumnRenamed('newPath', 'path')
        cached_new_vertices = AM.getCachedDataFrame(new_vertices)
        g2 = GraphFrame(cached_new_vertices, g2.edges)
    return g2.vertices \
                .withColumn("newPath", add_path_udf("path", "id")) \
                .drop("visited", "path") \
                .withColumnRenamed("newPath", "path")
```

如果我们想找到从Amsterdam到所有其他地点的最短路径，我们可以这样调用函数：

```
via_udf = F.udf(lambda path: path[1:-1], ArrayType(StringType()))
result = sssp(g, "Amsterdam", "cost")
(result
.withColumn("via", via_udf("path"))
.select("id", "distance", "via")
.sort("distance")
.show(truncate=False))
```

我们定义了这个函数，从结果路径中过滤出开始和结束节点。

如果运行该代码，结果如下：

```
+----------------+--------+-----------------------------------------------------------------+
|id              |distance|via                                                              |
+----------------+--------+-----------------------------------------------------------------+
|Amsterdam       |0.0     |[]                                                               |
|Utrecht         |46.0    |[]                                                               |
|Den Haag        |59.0    |[]                                                               |
|Gouda           |81.0    |[Utrecht]                                                        |
|Rotterdam       |85.0    |[Den Haag]                                                       |
|Hoek van Holland|86.0    |[Den Haag]                                                       |
|Felixstowe      |293.0   |[Den Haag, Hoek van Holland]                                     |
|Ipswich         |315.0   |[Den Haag, Hoek van Holland, Felixstowe]                         |
|Colchester      |347.0   |[Den Haag, Hoek van Holland, Felixstowe, Ipswich]                |
|Immingham       |369.0   |[]                                                               |
|Doncaster       |443.0   |[Immingham]                                                      |
|London          |453.0   |[Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]|
+----------------+--------+-----------------------------------------------------------------+
```

在这些结果中，我们可以看到从Amsterdam这个根根节点到图中所有其他城市的物理距离（以公里为单位），由短到长排序。

Neo4j上SSSP

Neo4j实现了SSSP的变体，被称之为增量步进算法（Delta-Stepping Algorithm），它将Dijkstra的算法分为多个可以并行执行的阶段。

执行以下的查询：

```
MATCH (n:Place {id:"London"})
CALL algo.shortestPath.deltaStepping.stream(n, "distance", 1.0)
YIELD nodeId, distance
WHERE algo.isFinite(distance)
RETURN algo.getNodeById(nodeId).id AS destination, distance
ORDER BY distance
```

结果如下：

```
+-------------------------------+
| destination         | distance |
+-------------------------------+
| "London"            | 0.0      |
| "Colchester"        | 106.0    |
| "Ipswich"           | 138.0    |
| "Felixstowe"        | 160.0    |
| "Doncaster"         | 277.0    |
| "Immingham"         | 351.0    |
| "Hoek van Holland"  | 367.0    |
| "Den Haag"          | 394.0    |
| "Rotterdam"         | 400.0    |
| "Gouda"             | 425.0    |
| "Amsterdam"         | 453.0    |
| "Utrecht"           | 460.0    |
+-------------------------------+
```

在这些结果中，我们可以看到图中从根节点London到所有其他城市的物理距离（以公里为单位），由短到长排序。

最小生成树

最小（加权）生成树算法（Minimum Spanning Tree, MST）是从一个给定的节点开始，到其所有可到达的节点并使得经过这些节点的权重尽可能地小。它从已被访问的节点出发，根据最小权重访问下一个节点，而且不产生环。

捷克科学家Otakar Borů vka于1926年开发了第一个已知的最小（权重）生成树算法。Prim的算法，发明于1957年，是最简单和最著名的。Prim的算法类似于Dijkstra的最短路径算法，但它不是最小化每个关系的路径总长度，而是单独优化每个关系的长度。与Dijkstra的算法不同，它允许权重为负。

最小生成树算法的操作如图4-10所示。

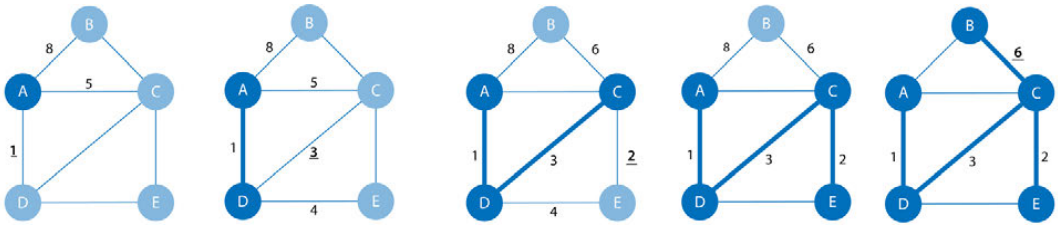


图4-10.最小生成树算法的步骤

步骤如下：

1) 起初，树只包含一个节点。在图4-10中，我们从节点A开始。
2) 将选择来自该节点的最小权重的关系并将其添加到树（及其连接的节点）。在当前的案例中，是A-D。
3) 这个过程是重复的，总是选择连接树中任何节点的最小权重关系。

如果将这里的示例与图4-9中的SSSP示例进行比较，你会注意到在第四个图中，路径会有所不同。这是因为SSSP根据从根开始的累计总数来计算最短路径，而最小生成树只考虑下一步的成本。

当没有更多要添加的节点时，树是最小生成树。这种算法还有一些变体，可以找到最大权重生成树（最高成本树）和K生成树（树大小有限）。

MST算法的使用场景

当需要访问所有节点的最佳路由时，请使用最小生成树。因为路由是根据下一步的成本来选择的，所以当你必须在一次行走中访问所有节点时，它非常有用。

你可以使用此算法优化连接系统（如水管和电路设计）的路径。它还用于近似一些计算时间未知的问题，如商旅问题和某些类型的路线问题。虽然该算法不一定总能找到绝对最优解，但它使得实际上相当复杂和密集的计算更加容易接近。

示例用例包括：

- 尽可能降低探索一个国家的旅行成本。 “An Application of Minimum Spanning Trees to Travel Planning” 描述了算法如何分析航空和海上航线来实现这一点。
- 使得货币回报之间的相关性。这在 “Minimum Spanning Tree Application in the Currency Market” 中进行了描述。
- 追踪疫情中感染传播的历史。有关更多信息，请参阅 “ “Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection” 。

最小生成树算法只在关系具有不同权重的图上运行时给出有意义的结果。如果图没有权重，或者所有关系都有相同的权重，那么任何生成树都是最小生成树。

Neo4j上的最小生成树

让我们看看MST算法的作用。以下查询查找从Amsterdam开始的生成树：

```
MATCH (n:Place {id:"Amsterdam"})
CALL algo.spanningTree.minimum("Place", "EROAD", "distance", id(n), {write:true, writeProperty:"mst"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

传递给此算法的参数为：

参数	意义
Place	计算生成树时要考虑的节点标签
EROAD	计算生成树时要考虑的关系类型
distance	表示一对节点之间遍历成本的关系属性的名称
id(n)	生成树应该从中开始的节点的内部节点ID

此查询将其结果存储在图中。如果要返回最小权重生成树，再运行如下查询：

```
MATCH path = (n:Place {id:"Amsterdam"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination,
rel.distance AS cost
```

结果如下：（Neo4j上的MST算法有问题）

source	destination	cost
"Amsterdam"	"Utrecht"	46.0
"Utrecht"	"Gouda"	35.0
"Gouda"	"vRotterdam"	25.0
"Rotterdam"	"Den Haag"	26.0
"Den Haag"	"Hoek van Holland"	27.0
"Hoek van Holland"	"Felixstowe"	207.0
"Felixstowe"	"Ipswich"	22.0
"Ipswich"	"Colchester"	32.0
"Colchester"	"London	106.0
"London"	"Doncaster"	277.0
"Doncaster"	"Immingham"	74.0

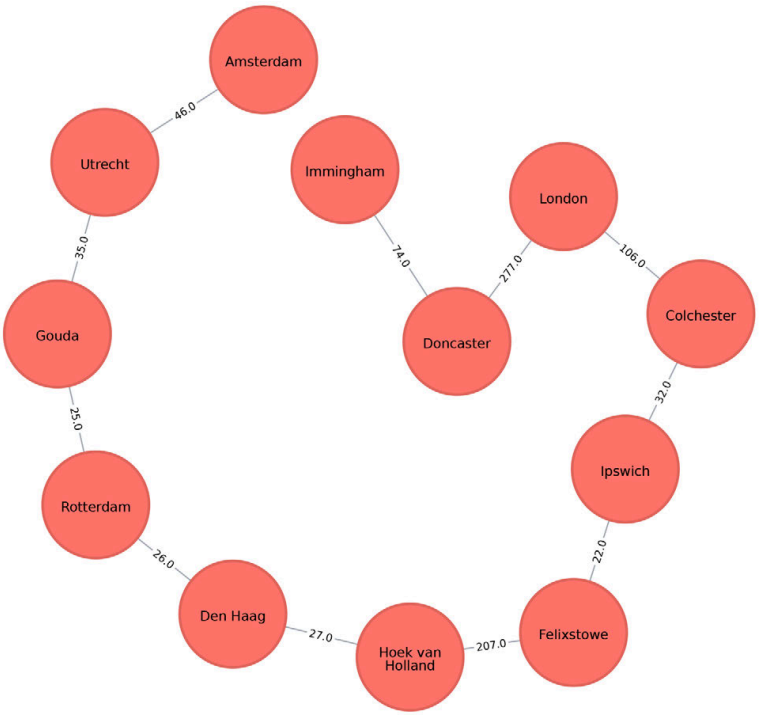


图4-11.Amsterdam的最小权重生成树

如果我们在Amsterdam，并且想在同一次旅行中访问示例数据集当中的其他地方，图4-11显示了最短的连续路线。

随机行走

随机行走算法(Random Walk，RW)会在图中的随机路径上给我们提供一组节点。Karl Pearson在1905年给《Nature》杂志的一封题为 “The Problem of the Random Walk” 的信中首次提到了这个词。尽管这一概念可以追溯到更远的地方，但直到最近，随机行走才被应用到网络科学中。

一般来说，一次随机行走有时被描述为类似于醉汉如何穿越城市。他们知道他们想要到达的方向或终点，但可能会走一条非常迂回的路线到达那里。

该算法从一个节点开始，在某种程度上随机地跟踪一个关系向前或向后到邻居节点。然后，它从该节点执行相同的操作，依此类推，直到达到设置的路径长度。（我们说有点随机，因为一个节点和它的邻居之间的关系数量会影响一个节点被遍历的概率。）

RW的使用场景

当需要生成一组大部分随机连接的节点时，可以将随机行走算法用作其他算法或数据管道的一部分。

示例用例包括：

- 作为node2vec和graph2vec算法的一部分，创建节点嵌入。这些节点嵌入可以用作神经网络的输入。
- 作为Walktrap和Infomap社区检测的一部分。如果随机行走重复返回一小组节点，则表明节点集可能具有社区结构。
- 作为机器学习模型训练过程的一部分。这一点在David Mack的文章 Review Prediction with Neo4j and TensorFlow中有进一步的描述。

你可以在N. Masuda, M. A. Porter，和 R. Lambiotte的一篇论文中阅读更多的用例，Random Walks and Diffusion on Networks。

Neo4j上的RW算法

Neo4j实现了随机游走算法。它支持在算法的每个阶段选择下一个要遵循的关系的两种模式：

- random，随机选择要跟踪的关系
- node2vec，根据计算前一个邻居的概率分布选择要跟踪的关系

执行以下查询：

```
MATCH (source:Place {id: "London"})
CALL algo.randomWalk.stream(id(source), 5, 1)
YIELD nodeIds
UNWIND algo.getNodesById(nodeIds) AS place
RETURN place.id AS place
```

传递给此算法的参数为：

参数	意义
id(source)	随机行走起点的内部节点ID
5	随机行走的跃点数
1	我们要计算的随机行走数

返回结果如下：

```
+--------------+
| place        |
+--------------+
| "London"     |
| "Colchester" |
| "Ipswich"    |
| "Felixstowe" |
| "Ipswich"    |
| "Felixstowe" |
+--------------+
```

在随机行走的每个阶段，算法随机选择下一个关系。这意味着，如果我们重新运行算法，即使使用相同的参数，我们可能也不会得到相同的结果。步行也有可能自行返回，如图4-12所示，从Amsterdam到Den Hagg再返回。

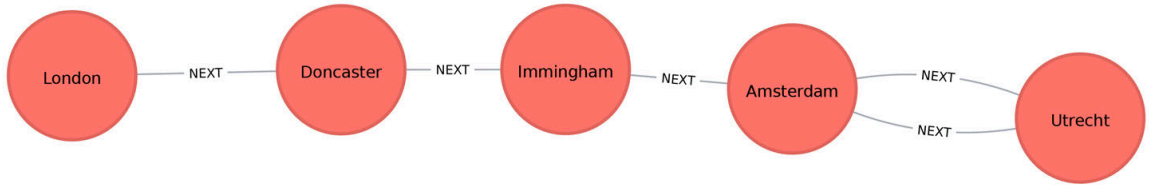


图4-12.从London开始的随机行走

总结

路径查找算法（Pathfinding algorithms）对于理解数据的连接方式很有用。在本章中，我们首先介绍了基本的广度和深度优先算法（BFS & DFS），然后再介绍Dijkstra和其他最短路径算法（Shortest Path）。我们还研究了最短路径算法的变体（A* & Yen's），这些算法优化后可以找到从一个节点到所有其他节点（SSSP）或图中所有节点对之间的最短路径（APSP）。我们完成了随机游走算法（Random Walk），可以用来寻找任意路径集。

接下来我们将学习中心性算法，它可以用来在图中找到有影响的节点。

算法资源

有许多算法书籍，但有一本是最突出的，它涵盖了基本概念和图算法：《Algorithm Design Manual》，由Steven S.Skiena（Springer）撰写。我们强烈推荐这本教科书给那些寻求关于经典算法和设计技术的综合资源的人，或者那些只想深入了解各种算法如何操作的人。

第五章 中心性算法

中心性算法（centrality algorithm）用于理解图中特定节点的角色及其对网络的影响。之所以有用，是因为这些算法能够识别最重要的节点，并帮助我们了解群体动态，例如可信度、可访问性、事物传播的速度以及群体之间的桥梁。尽管这些算法中有许多是为社交网路分析而发明的，但它们已经在各种行业和领域中得到了应用。

我们将介绍以下算法：

- 度中心性(Degree Centrality)作为连通性的基线度量
- 紧密中心性（Closeness Centrality）用于测量节点对群体的意义，包括对断开连接群体的两个变体
- 中介中心性（Betweenness Centrality）用于查找控制点，包括一个近似的估计算法
- 网页排名（PageRank）用来了解整体性的影响，包括一个个性化的网页排名变体

不同的中心性算法所用的度量值均不同，因而会在算法之间差生显著的差异结果。当你看到次优答案的时候，最好检查一下你使用的算法是否符合它的预期目的。我们将解释这些算法如何工作，并在Spark和Neo4j中运行示例。如果一个算法在一个平台上不可用，或者平台之间的差异并不重要，我们将只提供一个平台上的示例。

图5-1显示了中心性算法可以回答的问题类型之间的差异，表5-1是每个算法使用示例计算的快速参考。

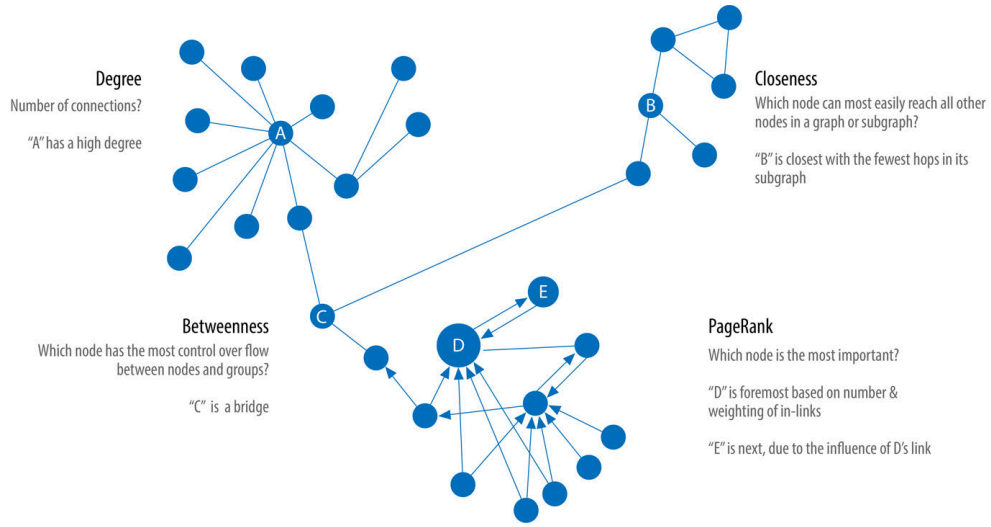


图5-1.有代表性的中心性算法及其回答的问题类型

表5-1.中心性算法概述

算法类型	它怎么工作	使用举例	Spark示例	Neo4j示例
度中心性	测量每个节点上关系的数量	根据一个人接受的关系（入链）来判断他的受欢迎程度，根据他对外建立的关系（出链）评估合群性	Yes	No
紧密中心性（变体：Wasserman & Faust, Harmonic中心性）	计算哪个节点有到其他节点的距离最短	为一个新的公共服务找到一个最优化的位置，使得它有最大的便利性	Yes	Yes
中介中心性（RA-Brandes）	测量经过一个节点的最短路径的数量	找到特定疾病的控制基因，以改进药品的靶点	No	Yes
网页排名（变体：个人化网页排名）	根据节点上链接的邻近节点和链接到这些邻近节点的节点，来估算一个节点的重要性	在机器学习中找到最有影响力的特征，在自然语言处理中根据实体相关性来将文本进行排序	Yes	Yes

一些中心性算法计算每对节点之间的最短路径。这对中小型图很有效，但是对于大型图来说，计算上是不允许的。为了避免大型图上的长运行时间，一些算法（例如，中间中心性）有估算版本。

首先，我们将为我们的示例描述数据集，并将数据导入Apache Sark和Neo4j中，按表5-1中列出的顺序介绍各个算法。我们将从对算法的简短描述开始，并在有必要时提供有关它如何操作的原理性信息。对于涉及到的变种算法，将会只做简要描述。大多数章节还包括有关相关算法的使用场景指导。我们在每个部分的末尾，在示例数据集上演示示例代码。

让我们开始吧！

图数据示例：社交图

中心性算法与所有领域的图都相关，但是社交网络提供了一种非常相关的方式来考虑动态影响和信息流向。本章中的例子是在一个类似Twitter的小规模图上运行的。您可以从Github上下载节点和关系文件，我们将用它们来创建图。

表5-2 social-node.csv

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

表5-3 social-relationships.csv

src	dst	relationship
Alice	Bridget	FOLLOWS
Alice	Charles	FOLLOWS
Mark	Doug	FOLLOWS
Bridget	Michael	FOLLOWS
Doug	Mark	FOLLOWS
Michael	Alice	FOLLOWS
Alice	Michael	FOLLOWS
Bridget	Alice	FOLLOWS
Michael	Bridget	FOLLOWS
Charles	Doug	FOLLOWS
Bridget	Doug	FOLLOWS
Michael	Doug	FOLLOWS
Alice	Doug	FOLLOWS
Mark	Alice	FOLLOWS
David	Amy	FOLLOWS
James	David	FOLLOWS

图5-2展现了我们想要构建的图。

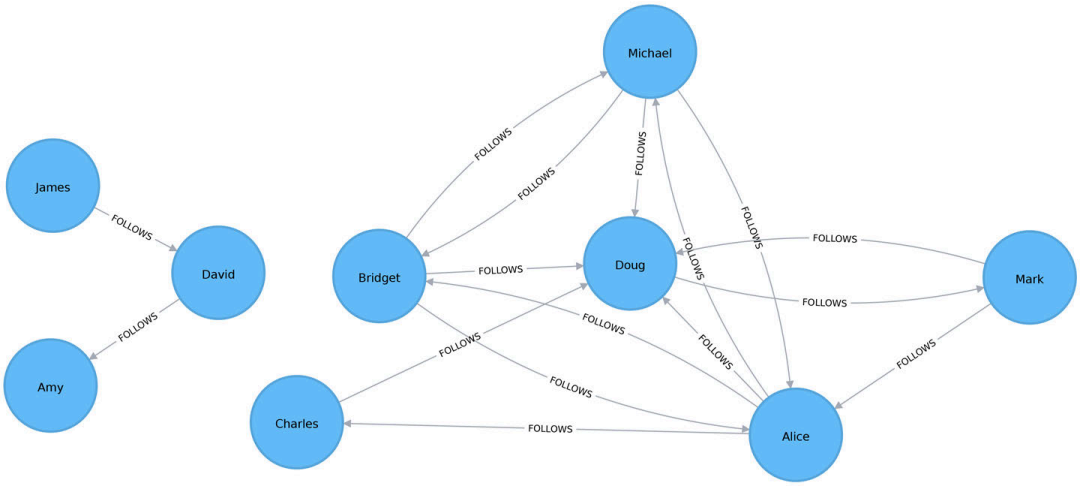


图5-2.图模型

图中有一个较大的用户集，它们之间有连接，而另外一部分较小的用户组没有连接到较大的用户组上，它们相互之间是断开的。基于这些csv文件的内容，我们将在在Spark和Neo4j中创建图。

将数据导入Apache Spark

首先，我们将需要的包从Spark和GraphFrames中导入

```
from graphframes import *
from pyspark import SparkContext
```

我们可以用如下的代码来根据csv文件创建一个GraphFrame

```
base = "file:///home/retire2053/source/graph_algorithms_resources/"
v = spark.read.csv(base+"data/social-nodes.csv", header=True)
e = spark.read.csv(base+"data/social-relationships.csv", header=True)
g = GraphFrame(v, e)
```

将数据导入Neo4j

接下来，我们将数据装载到Neo4j，以下查询语句用来导入节点。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:User {id: row.id})
```

然后是导入关系

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```

现在，我们的图已经被装载了，是开始使用算法的时候了。

度中心性

度中心性（Degress Centrality）是我们将在本书中讨论的最简单的算法。它计算节点上传入和传出关系的数量，该算法可以用于在图中查找”热''（poplular）的节点。度中心性是Linton C. Freeman在1979年的论文“Centrality in Social Networks: Conceptual Clarification” 中提出的。

到达范围（Reach）

节点的到达范围（reach）是衡量重要性的合理标准。一个节点能接触到多少个其他节点？一个节点的度（Degree）是它所具有的直接关系的数目，包括入度数和出度数。你可以将度视为该节点的直接到达节点。例如，一个在活跃的社交网络中很高节点度数的人会有很多直接的接触，因此而通常更容易得感冒一些。

网络的平均度（average degree）只是关系总数除以节点总数；平均度数容易被高度节点严重扭曲，也就是低度节点容易被平均。度分布（degree distribution）是被随机选择的节点具有某个度数的概率。

图5-3说明了Subreddit上，主题之间连接的实际分布的差异。如果你简单地取平均值，你会假设大多数主题有10个连接，而实际上大多数主题只有2个连接。

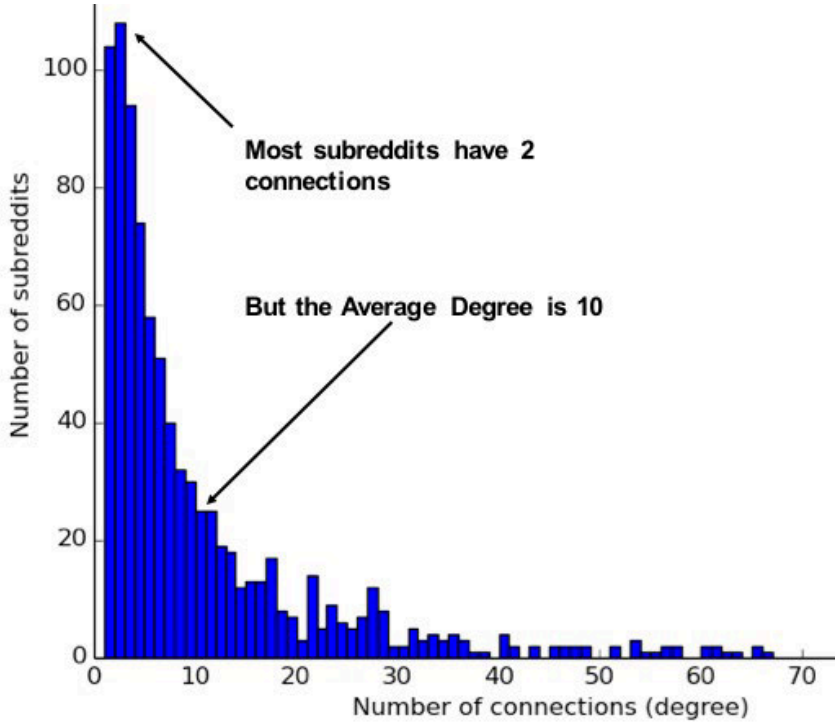


图5-3.Jacob Silterrapa绘制的这张Subreddit度分布图提供了一个例子，说明平均值通常不反映网络中的实际分布。CC BY-SA 3.0。

这些评估用于对网络类型进行分类，如第2章讨论的自由缩放（Scale-free）或小世界（small world）网络。它们还提供了一个快速的措施来帮助估计事物在网络中传播或波动的可能性。

度中心性算法的使用场景

如果您试图通过查看传入和传出关系的数量来分析影响，或者找到单个节点的“流行度”，请使用“度中心性”。当你关注即时连通性时，它会很好地工作。不仅仅于此，当您想要评估整个图的最小程度、最大程度、平均程度和标准偏差时，度中心性也应用于全局分析。

示例用例包括：

- 通过人际关系识别有权势的个人，例如社交网络中的人际关系。例如，在BrandWatch的“Most Influential Men and Women on Twitter 2017”中，每个类别的前5名都有超过4000万名粉丝。
- 将欺诈者与在线拍卖网站的合法用户分开。欺诈者的加权中心性倾向于明显更高，因为他们的目的是人为地提高价格。阅读P. Bangcharoensap等人的论文Two Step Graph-Based Semi-Supervised Learning for Online Auction Fraud Detection。

在Apache Spark上的度中心性算法

现在，我们用如下的代码来执行度中心性算法

```
total_degree = g.degrees
in_degree = g.inDegrees
out_degree = g.outDegrees
(total_degree.join(in_degree, "id", how="left")
.join(out_degree, "id", how="left")
.fillna(0)
.sort("inDegree", ascending=False)
.show())
```

我们首先计算总度数，进度数和出度数。然后我们将这些DataFrame连接在一起，使用左连接来保留没有传入或传出关系的任何节点。如果节点没有关系，则使用fillna函数将该值设置为0。

下面是在pyspark中运行代码的结果：

```
+-------+------+--------+---------+
|     id|degree|inDegree|outDegree|
+-------+------+--------+---------+
|   Doug|     6|       5|        1|
|  Alice|     7|       3|        4|
|Michael|     5|       2|        3|
```

```
|Bridget|      5|        2|         3|
|Charles|      2|        1|         1|
|    Amy|      1|        1|         0|
|  David|      2|        1|         1|
|   Mark|      3|        1|         2|
|  James|      1|        0|         1|
+-------+-------+---------+----------+
```

我们可以在图5-4中看到，Doug是我们Twitter图中最受欢迎的用户，有五个追随者（在链接中）。图中该部分的所有其他用户都关注他，而他只关注一个人。在真实的Twitter网络中，名人有很高的追随者数量，但往往关注很少的人。因此我们可以认为Doug是名人！

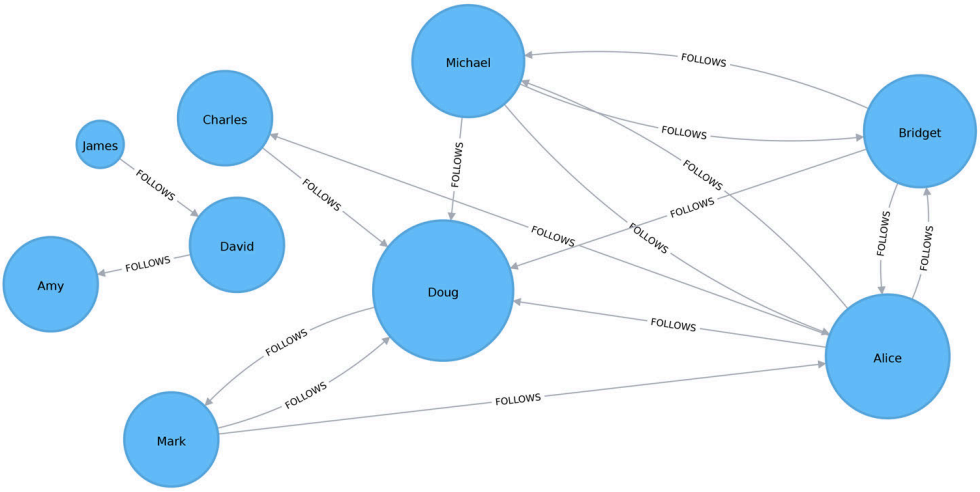


图5-4.度中心性的可视化

如果我们正在创建一个显示被最多关注的用户的页面，或者希望建议用户来跟踪，我们可以使用此算法来识别这些用户。

有些数据可能包含关系很多的非常密集的节点。这些节点不会增加太多额外的信息，并且会扭曲一些结果或增加计算复杂性。你可能希望使用子图过滤掉这些密集的节点，或者使用投影方法将关系汇总成为一个权重。

紧密中心性

紧密性中心性(Closeness Centrality)是一种检测节点通过子图传播信息有效性的方法。该方法度量是节点与所有其他节点的距离近的程度。高紧密中心性的节点与所有其他节点的距离最短。

在所有节点对的最短路径计算的基础上，紧密中心性算法，计算一个节点到所有其他节点的距离之和。然后将得到的和求倒数，以确定该节点的紧密性中心性得分。节点的紧密中心性用以下的公式来计算：

$$C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$$

在这个公式中

- u是一个节点。
- n是图中的节点数。
- d(u,v)是另一个节点V和U之间的最短路径距离。

更常见的是将该分数归一化，使该得分代表最短路径的平均长度，而不是它们的总和。这种调整允许比较不同大小图节点的紧密性中心性。

归一化后的紧密中心性公式如下：

$$C_{norm}(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

紧密中心性算法的使用场景

当你需要知道哪个节点传播的东西最快时，应使用紧密中心性。在紧密中心性算法中，使用加权关系的话，会特别有助于评估交流和行为分析中的交互速度。

示例用例包括：

- 发现对控制和获取组织内重要信息和资源处于非常有利位置的个人。这方面的一项研究是V. E. Krebs的 “Mapping Networks of Terrorist Cells” 。
- 作为一种启发式方法，用于估计电信和包裹交付中的到达时间，在交付过程中，内容通过最短路径流向预先定义的目标。它还被用来揭示通过所有最短路径同时传播的情况，例如通过当地社区传播的感染。在S. P. Borgatti的 “Centrality and Network Flow” 中找到更多细节。

- 可以应用在基于基于图的关键字提取过程，比如评估文档中单词的重要性。这一过程由F. Boudin在 “A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction” 中有描述。

紧密性中心性在连通图上效果最好。当该公式应用于一个不连通图时，两个节点之间的距离是无限的，两个节点之间没有路径。这意味着，当我们计算所有节点到那个节点的距离并求和时，我们将得到一个无限大的紧密中心性分数。在下一个示例之后将给出原始公式的变体，用来避免这种情况。

Apache Spark上的紧密中心性算法

Apache Spark没有提供一个紧密中心性的内置算法，但是我们可以用aggregateMessages框架来实现，这个框架在最短路径算法部分已经介绍过了。

在我们创建函数之间，我们需要导入将会用到的包。

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
from pyspark.sql.types import *
from operator import itemgetter
```

我们创建几个稍后会被引用到的自定义函数（user-defined function)

```
def collect_paths(paths):
    return F.collect_set(paths)

collect_paths_udf = F.udf(collect_paths, ArrayType(StringType()))

paths_type = ArrayType(
    StructType([StructField("id", StringType()), StructField("distance", IntegerType())]))

def flatten(ids):
    flat_list = [item for sublist in ids for item in sublist]
    return list(dict(sorted(flat_list, key=itemgetter(0))).items())

flatten_udf = F.udf(flatten, paths_type)

def new_paths(paths, id):
    paths = [{"id": col1, "distance": col2 + 1} for col1,
    col2 in paths if col1 != id]
    paths.append({"id": id, "distance": 1})
    return paths

new_paths_udf = F.udf(new_paths, paths_type)

def merge_paths(ids, new_ids, id):
    joined_ids = ids + (new_ids if new_ids else [])
```

```
    merged_ids = [(col1, col2) for col1, col2 in joined_ids if col1 != id]
    best_ids = dict(sorted(merged_ids, key=itemgetter(1), reverse=True))
    return [{"id": col1, "distance": col2} for col1, col2 in best_ids.items()]

merge_paths_udf = F.udf(merge_paths, paths_type)

def calculate_closeness(ids):
    nodes = len(ids)
    total_distance = sum([col2 for col1, col2 in ids])
    return 0 if total_distance == 0 else nodes * 1.0 / total_distance
    closeness_udf = F.udf(calculate_closeness, DoubleType())

closeness_udf = F.udf(calculate_closeness, DoubleType())
```

以下代码是计算每个节点的紧密中心性

```
vertices = g.vertices.withColumn("ids", F.array())
cached_vertices = AM.getCachedDataFrame(vertices)
g2 = GraphFrame(cached_vertices, g.edges)

for i in range(0, g2.vertices.count()):
    msg_dst = new_paths_udf(AM.src["ids"], AM.src["id"])
    msg_src = new_paths_udf(AM.dst["ids"], AM.dst["id"])
    agg = g2.aggregateMessages(F.collect_set(AM.msg).alias("agg"),
                sendToSrc=msg_src, sendToDst=msg_dst)
    res = agg.withColumn("newIds", flatten_udf("agg")).drop("agg")
    new_vertices = (g2.vertices.join(res, on="id", how="left_outer")
            .withColumn("mergedIds", merge_paths_udf("ids", "newIds",
            "id")).drop("ids", "newIds")
            .withColumnRenamed("mergedIds", "ids"))
    cached_new_vertices = AM.getCachedDataFrame(new_vertices)
    g2 = GraphFrame(cached_new_vertices, g2.edges)

(g2.vertices
.withColumn("closeness", closeness_udf("ids"))
.sort("closeness", ascending=False)
.show(truncate=False))
```

结果如下。Alice、Doug和David是图中连接最紧密的节点，得分为1.0，这意味着每个节点都直接连接到图中其部分的所有节点。

```
+-------+-----------------------------------------------------------------+------------------+
|id     |ids                                                              |closeness         |
+-------+-----------------------------------------------------------------+------------------+
|Alice  |[[Bridget, 1], [Doug, 1], [Charles, 1], [Michael, 1], [Mark, 1]] |1.0               |
|Doug   |[[Bridget, 1], [Charles, 1], [Michael, 1], [Mark, 1], [Alice, 1]]|1.0               |
|David  |[[James, 1], [Amy, 1]]                                           |1.0               |
|Bridget|[[Doug, 1], [Charles, 2], [Michael, 1], [Mark, 2], [Alice, 1]]   |0.7142857142857143|
|Michael|[[Bridget, 1], [Doug, 1], [Charles, 2], [Mark, 2], [Alice, 1]]   |0.7142857142857143|
|James  |[[David, 1], [Amy, 2]]                                           |0.6666666666666666|
|Amy    |[[James, 2], [David, 1]]                                         |0.6666666666666666|
|Charles|[[Bridget, 2], [Doug, 1], [Michael, 2], [Mark, 2], [Alice, 1]]   |0.625             |
|Mark   |[[Bridget, 2], [Doug, 1], [Charles, 2], [Michael, 2], [Alice, 1]]|0.625             |
+-------+-----------------------------------------------------------------+------------------+
```

图5-5说明了即使David在密友群中只有几个联系，但他在他的群体中，也是重要的。换句话说，这个分数表示每个用户在其子图（能连通的到的子图）中与其他用户之间的亲密程度，而不一定全图。

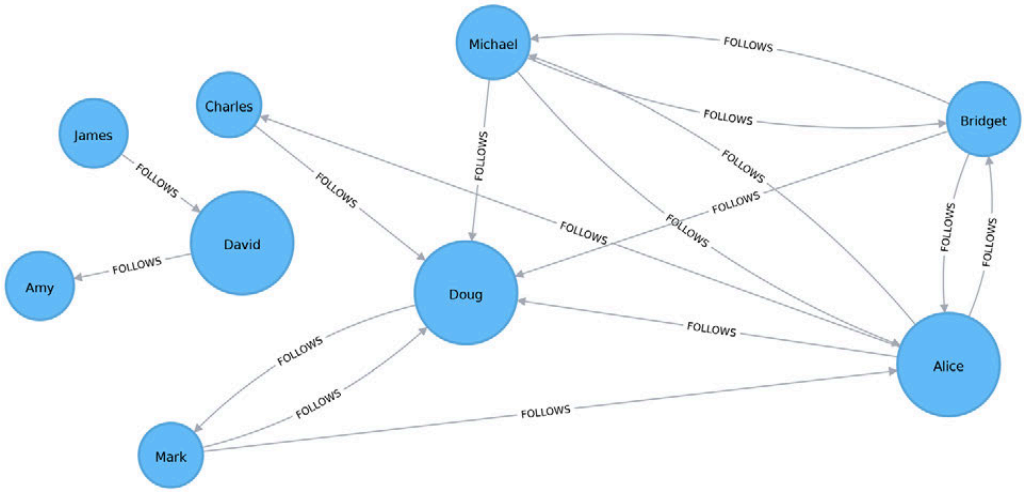


图5-5.紧密中心性的可视化

Neo4j上的紧密中心性

Neo4j上用如下的公式来实现紧密中心性算法。

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

在这个公式中，

- u是一个节点
- n是在当前组件（component）内节点数量
- d(u,v)是其他节点v和u之间的最短路径。

运行如下查询，将会计算图中每个节点的紧密中心性。

```
CALL algo.closeness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id, centrality
ORDER BY centrality DESC
```

以下是运行的结果

```
+-------------------------------------------------+
| algo.getNodeById(nodeId).id | centrality        |
```

```
+-----------------------------------------------+
| "Alice"                       | 1.0           |
| "Doug"                        | 1.0           |
| "David"                       | 1.0           |
| "Bridget"                     | 0.7142857142857143 |
| "Michael"                     | 0.7142857142857143 |
| "Amy"                         | 0.6666666666666666 |
| "James"                       | 0.6666666666666666 |
| "Charles"                     | 0.625         |
| "Mark"                        | 0.625         |
+-----------------------------------------------+
```

我们得到了与Spark算法相同的结果，但是，和以前一样，分数代表了他们在子图中与其他人的紧密程度，而不是整个图。

在严格意义上的紧密性中心性算法中，我们图中的所有节点都会得到无穷大的分数，因为每个节点至少有一个它无法到达的其他节点。不过，实现每个断开的组件（component，局部连通的图）的得分通常会更有用。

理想情况下，我们希望在整个图中得到一个中心性的整体认识，在接下来的两个部分中，我们将学习实现紧密中心性算法的一些变体，它们将实现这一点。

紧密中心性的变体：Wasserman & Faust

Stanley Wasserman和Katherine Faust提出了一个改进的公式，用于计算具有多个无相互连接的子图的紧密中心性。他们的书Social Network Analysis: Methods and Applications中详细介绍了他们的公式。此公式的结果是可到达组中节点的分数与可到达节点的平均距离之比。

公式如下：

$$C_{WF}(u) = \frac{n-1}{N-1}\left(\frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}\right)$$

在这个公式中

- u是一个节点

- N是所有节点的数量
- n是u所在的组件上的节点数量
- d(u,v)是在其他节点v和u之间的最短路径

为了能够使用这个公式，我们可以通过传递参数{improved:true}给紧密中心性算法。

以下查询使用Wasserman & Faust紧密中心性算法：

```
CALL algo.closeness.stream("User", "FOLLOWS", {improved: true})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

运行的结果如下：

```
+---------------------------------+
| user      | centrality          |
+---------------------------------+
| "Alice"   | 0.5                 |
| "Doug"    | 0.5                 |
| "Bridget" | 0.35714285714285715 |
| "Michael" | 0.35714285714285715 |
| "Charles" | 0.3125              |
| "Mark"    | 0.3125              |
| "David"   | 0.125               |
| "Amy"     | 0.08333333333333333 |
| "James"   | 0.08333333333333333 |
+---------------------------------+
```

如图5-6所示，结果现在更能代表节点与整个图的紧密性。较小子图（David、Amy和James）成员的得分已被降低，现在他们的得分是所有用户中最低的。这很有意义，因为它们是最孤立的节点。这个公式对于检测节点在整个图中的重要性更有用，而不是在它自己的子图中。

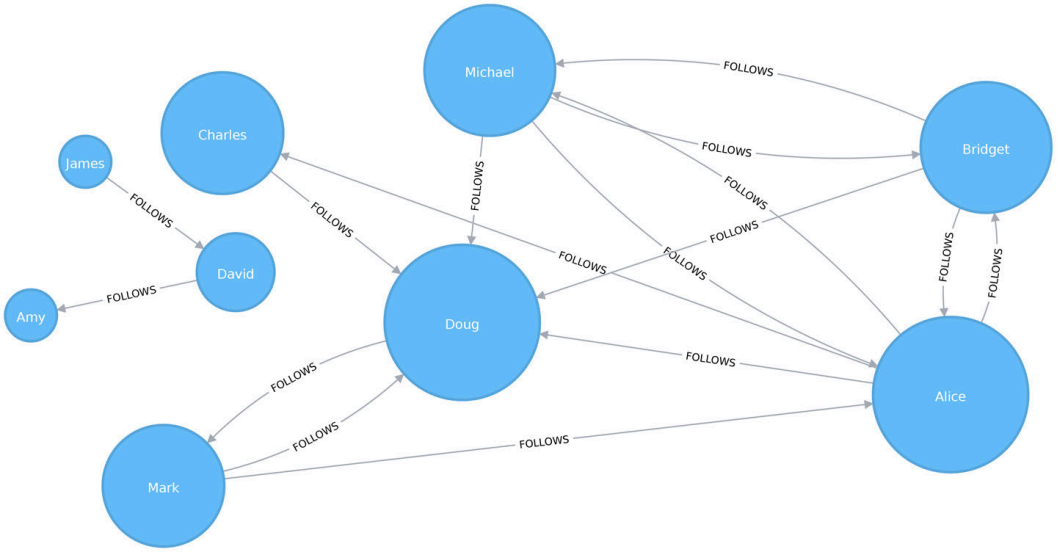


图5-6.紧密性中心性的可视化

在下一节中，我们将学习和谐中心性算法，它使用另一个公式来计算紧密性，从而获得类似的结果。

紧密中心性变体：和谐中心性

和谐中心性（也称为值化中心性）是紧密中心性的一种变体，被发明来解决不连通图的问题。在《Harmony in a Small World》一书中，M. Marchiori和 V. Latora提出了这个概念，作为一个平均最短路径的实用表示。

在计算每个节点的接近度得分时，它不是求一个节点到所有其他节点的距离之和，而是求这些距离的倒数。这意味着无穷大的值变得无关紧要。基础的和谐中心性算法使用以下公式：

$$H(u) = \sum_{v=1}^{n-1} \frac{1}{d(u,v)}$$

在公式中

- u是一个节点
- n是图中的节点数
- d(u,v)是在其他节点v和u之间的最短路径

与紧密中心性一样，我们也可以用以下公式计算归一化和谐中心性：

$$H_{norm}(u) = \frac{\sum_{v=1}^{n-1} \frac{1}{d(u,v)}}{n-1}$$

在这个公式中，无穷大被很干净利索地处理掉了。

Neo4j上的和谐中心性

用如下的查询来执行和谐中心性算法。

```
CALL algo.closeness.harmonic.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

该Procedure的运行结果如下：

```
+-------------------------+
| user       | centrality |
+-------------------------+
| "Alice"    | 0.625      |
| "Doug"     | 0.625      |
| "Bridget"  | 0.5        |
| "Michael"  | 0.5        |
| "Charles"  | 0.4375     |
| "Mark"     | 0.4375     |
| "David"    | 0.25       |
| "Amy"      | 0.1875     |
| "James"    | 0.1875     |
+-------------------------+
```

该算法的结果与原始的紧密性中心性算法不同，但与Wasserman和Faust改进算法的结果相似。当处理具有多个连接组件的图时，可以使用任一算法。

中介中心性

有时，系统中最重要的齿轮不是最明显的动力最高的齿轮。有时反而是一些中间环节把对资源或信息流控制最大的群体或经纪人联系起来。中介中心性是一种检测节点对图中信息或资源流的影响程度的方法。它通常用于查找充当从图的一部分到另一部分的桥梁型节点。

中介中心性(Betweenness Centrality)算法首先计算连接图中每对节点之间的最短（权重）路径。每个节点都会根据这些通过该节点的最短路径的数量得到一个分数。经过节点的最短路径越多，该节点的得分越高。

当Linton C. Freeman在1971年的论文A Set of Measures of Centrality Based on Betweenness中介绍这个概念后，中介中心性被认为是“三个截然不同的直观的中心性概念”之一。

桥和控制点

网络中的桥可以是节点或关系。在一个非常简单的图中，您可以通过查找节点或关系来找到它们，如果删除这些节点或关系，将导致图的一部分断开连接。然而，由于这种简单情形在典型的图并不是实际发生，实际情况会更复杂，我们需要使用中介中心性算法来评估在一个群体中某个节点的中介特性。

如果一个节点位于这些节点之间的每一条最短路径上，则它被认为是其他两个节点的关键，如图5-7所示。

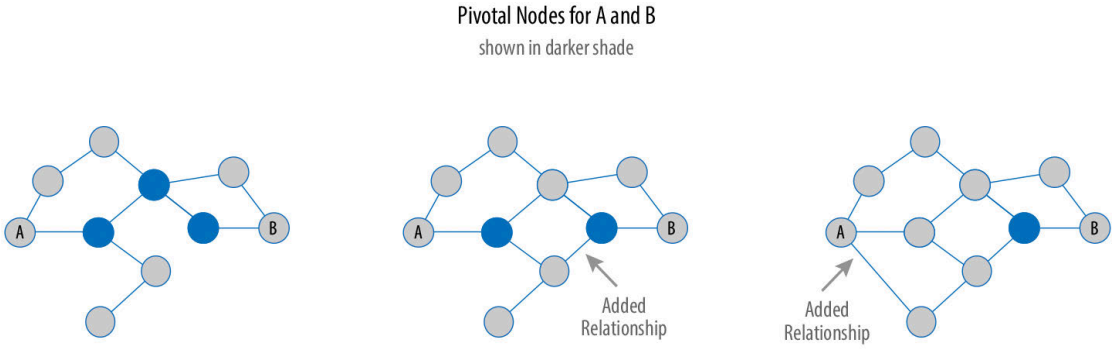


图5-7.关键节点位于两个节点之间的每一条最短路径上。所以，创建更多的最短路径，会减少这个关键节点的数量，并减少风险。

关键节点（pivotal node）在连接其他节点时起着重要作用。如果删除关键节点，则各节点对的新的最短路径将更长或更昂贵。这可以作为评估单一脆弱性点的考虑因素。

计算中介中心性

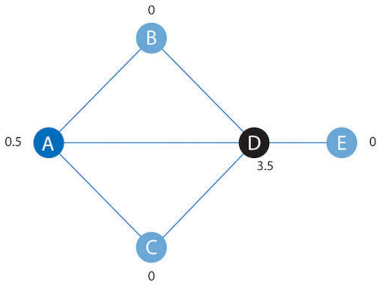
中介中心性是将最短路径通过如下公式计算后累加的结果。

$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

在公式中

- u是一个节点
- p是节点s和t之间最短路径的数量
- p(u)是s和t之间通过u的最短路径的数量

图5-8显示了计算出中介中心性的步骤



Node D Calculation

Pairs with Shortest Paths Through D	Total Possible Shortest Paths for That Pair	% of Total Through D (1/Total)
A, E	1	1
B, E	1	1
C, E	1	1
B, C	2 (through D & A)	0.5
Betweenness Score		3.5

图5-8. 计算中介中心性的基本概念

这是计算过程：

1) 对于每个节点，找到通过它的所有最短路径。
   a) B、C、E没有最短路径，并被赋值为0。
2) 对于步骤1中的每个最短路径，计算其在该对可能最短路径总数中的百分比。（两点间可能有多条最短路径）
3) 将步骤2中的所有值相加，以找到节点的中间中心性得分。图5-8中的表说明了节点D的步骤2和3。
4) 对每个节点重复该过程。

中介中心性算法的使用场景

中介中心性适用于现实网络中的各种问题。我们使用它来发现瓶颈、控制点和漏洞。

示例用例包括：

- 识别不同组织中的影响因素。有权势的个人不一定在管理岗位上，但可以出现在“中介性岗位”上，这能够通过中介中心性来找到。消除这些影响者会严重破坏组织的稳定。如果该组织是犯罪组织，这可能被认为是一种很有效的执法方式；另一种场景下，但是如果企业失去了被低估

的关键员工，这可能是一场灾难。更多细节见C. Morselli和J. Roy的Brokerage Qualifications in Ringing Operations。

- 发现电网等网络中的关键转移点。与直觉相反，移除特定的桥梁或者关键节点，实际上可以通过使得干扰因素孤岛化而提高整体的鲁棒性。研究细节包含在R. Sol.等人的 “Robustness of the European Power Grids Under Intentional Attack” 中。
- 通过为目标人物提供推荐引擎，帮助微博用户在Twitter上传播他们的影响力。这一方法在S. Wu等人的一篇论文 “Making Recommendations in a Microblog to Improve the Impact of a Focal User” 中进行了描述。

中介中心性假设节点之间的所有通信都是沿着最短路径以相同的概率进行的，这在现实生活中情况并不相同。因此，它并没有给我们一个在全图中最有影响力的节点的概览，而只是提供了一个很好的表示方法。 Mark Newman在《Networks: An Introduction 》(Oxford University Press, p186)中有介绍。

Neo4j上的中介中心性

Spark没有中介中心性的内置算法，因此我们将使用Neo4j来演示此算法。调用以下过程将计算图中每个节点的中介中心性：

```
CALL algo.betweenness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

运行这个procedure可以得到如下的结果：

```
+-------------------------+
| user       | centrality |
+-------------------------+
| "Alice"    | 10.0       |
| "Doug"     | 7.0        |
| "Mark"     | 7.0        |
| "David"    | 1.0        |
| "Bridget"  | 0.0        |
| "Charles"  | 0.0        |
```

```
| "Michael" | 0.0         |
| "Amy"     | 0.0         |
| "James"   | 0.0         |
+-----------------------+
```

如图5-9所示，Alice是这个网络的主要中介节点，但Mark和Doug并不落后。在较小的子图中，所有最短的路径都经过David，因此他对于这些节点之间的信息流很重要。

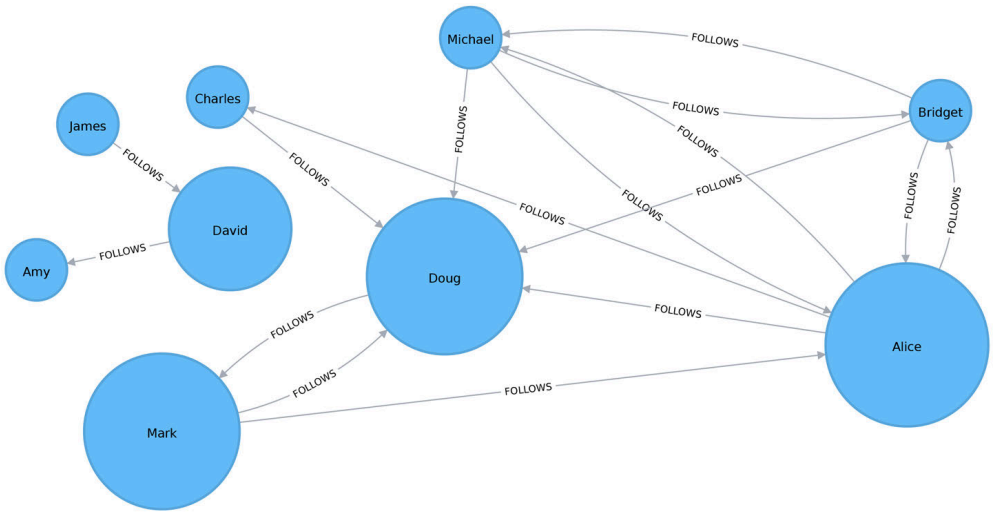


图5-9.中介中心性的可视化

对于更大的图来说，精确的中心性计算是不实际的，因为已知最快的精确计算所有节点中间数的算法的运行时间与节点数和关系数的乘积成正比。我们可能希望先过滤后留下子图，并使用子图来（在下一节中描述）处理所有节点的子集。

我们可以将两个断开连接的图组件连接在一起，方法是引入一个名为Jason的新用户，该用户关注两个组件的用户人员，而且被两个组件的人所关注。

```
WITH ["James", "Michael", "Alice", "Doug", "Amy"] AS existingUsers
MATCH (existing:User) WHERE existing.id IN existingUsers
MERGE (newUser:User {id: "Jason"})
MERGE (newUser)<-[:FOLLOWS]-(existing)
MERGE (newUser)-[:FOLLOWS]->(existing)
```

如果我们重新运行算法，我们将看到这个输出：

```
+--------------------------------+
| user       | centrality        |
+--------------------------------+
| "Jason"    | 44.33333333333333 |
| "Doug"     | 18.333333333333332|
| "Alice"    | 16.666666666666664|
| "Amy"      | 8.0               |
| "James"    | 8.0               |
| "Michael"  | 4.0               |
| "Mark"     | 2.1666666666666665|
| "David"    | 0.5               |
| "Bridget"  | 0.0               |
| "Charles"  | 0.0               |
+--------------------------------+
```

Jason的得分最高，因为这两组用户之间的交流将通过他。可以说Jason是两组用户之间的本地桥梁，如图5-10所示。

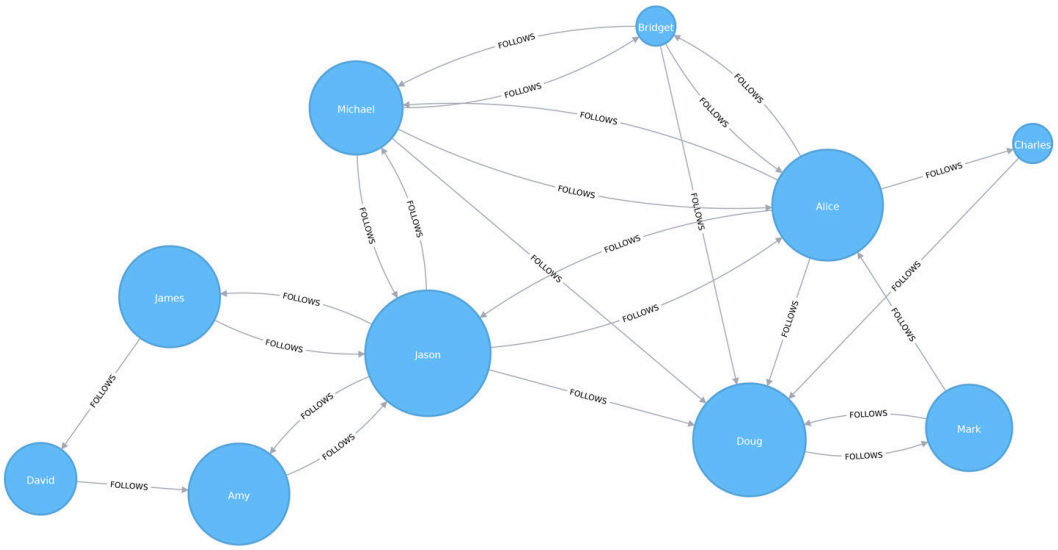


图5-10.有Jason之后的中介中心性的可视化

在我们继续下一节之前，让我们通过删除Jason和他的关系：

```
MATCH (user:User {id: "Jason"}) DETACH DELETE user
```

中介中心性变体：RA-Brandes

回想一下，在大规模图上计算精确的中介中心性是非常昂贵的。因此，我们可以选择使用运行速度更快但仍然提供有用信息（尽管不精确）的近似算法。

Randomized-Approximate Brandes（随机近似 Brandes，简称RA-Brandes）算法是计算中介中心性近似分数的最著名算法。RA-Brandes算法不计算每对节点之间的最短路径，只考虑所有节点的一个子集。而选择节点的子集的两种常见策略是分别是：

- 随机（random）策略：节点的选择是一致的，随机的，具有确定的选择概率。默认概率为：lgN/(e^2)。如果概率是1，意味着所有的节点都被装载，这样RA-Brandes算法和通常的中介中心性是一致的。
- 度（degree）策略：节点是随机选择的，但是那些度低于平均值的节点会被自动排除（即只有具有大量关系的节点才有机会被访问）。

作为进一步的优化，您可以限制最短路径算法使用的深度，这样该算法将会提供所有最短路径的子集。

用Neo4j实现RA-Brandes

用查询来执行随机模式下的RA-Brandes算法：

```
CALL algo.betweenness.sampled.stream("User", "FOLLOWS", {strategy:"degree"})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

这将会得到如下的结果：

```
+------------------------+
| user       | centrality |
+------------------------+
| "Alice"    | 9.0        |
| "Mark"     | 9.0        |
| "Doug"     | 4.5        |
| "David"    | 2.25       |
| "Bridget"  | 0.0        |
| "Charles"  | 0.0        |
| "Michael"  | 0.0        |
| "Amy"      | 0.0        |
| "James"    | 0.0        |
+------------------------+
```

尽管Mark现在的排名比Doug高，我们最有影响力的人和以前差不多。由于此算法的随机性，每次运行它时，我们可能会看到不同的结果。相比于小样本图，在较大的图上，这种随机性的影响要小一些。

页面排名

网页排名（PageRank）是最著名的中心性算法。它测量节点的传递（或方向）影响。我们讨论的所有其他中心性算法都度量节点的直接影响，而网页排名算法则考虑节点的邻居及其邻居的影响。例如，拥有一些非常强大的朋友比拥有很多不那么强大的朋友能让你更有影响力。网页排名算法可以通过迭代地将一个节点的排名分布在其相邻节点上，或者通过随机遍历图并计算在这些遍历过程中每个节点的命中频率来计算。

PageRank是以谷歌创始人之一Larry Page的名字命名的，Larry Page创建这个网站是为了在谷歌的搜索结果中对网站进行排名。其基本假设是，一个有更多传入链接和更具影响力的传入链接的页面更有可能是可靠的来源。PageRank衡量一个节点的传入关系的数量和质量，以确定该节点的重要性。对网络影响更大的节点被认为具有更多来自其他影响节点的传入关系。

影响

在直觉上，对于“影响”(influence)是这样定义的：与不太重要节点的连接相比，与更重要节点的关系对相关节点的影响贡献更大。测量“影响”通常涉及到对节点进行评分，而节点通常连着带权重关系，这些关系在许多迭代中不断更新评分。在一些情况中，所有节点会被评分，在另外的一些情形中，通过随机产生的代表性分布上被评分。

请记住，中心性是一个节点与其他节点相比时候的重要性。中心性是对节点潜在影响的排名，而不是衡量实际的影响。例如，您可以确定两个人在一个网络中具有最高的中心性，但也许因为政策或文化规范也在发挥作用，实际上的影响力被转移到其他人。量化实际影响是制定其他的影响评估指标的一个活跃研究领域，但并不在本书的范围之内。

PageRank公式

$$PR(u) = (1 - d) + d\left(\frac{PR(T1)}{C(T1)} + \ldots + \frac{PR(Tn)}{C(Tn)}\right)$$

PageRank在谷歌的论文中是这样定义的：

- 我们假设一个页面u引用了T1到Tn这n个页面。
- d是一个阻尼系数，设置在0和1之间。通常设置为0.85。您可以将这个值视为用户继续单击的可能性。（这有助于最小化Rank Sink，这将在下一节中解释。）
- 1-d是用户不通过任何关系而直接到达节点的概率。
- C(Tn)定义节点T的对外的度。

图5-11给出了一个小例子，说明PageRank将如何继续更新节点的评分，直到它收敛或满足所设置的迭代次数。

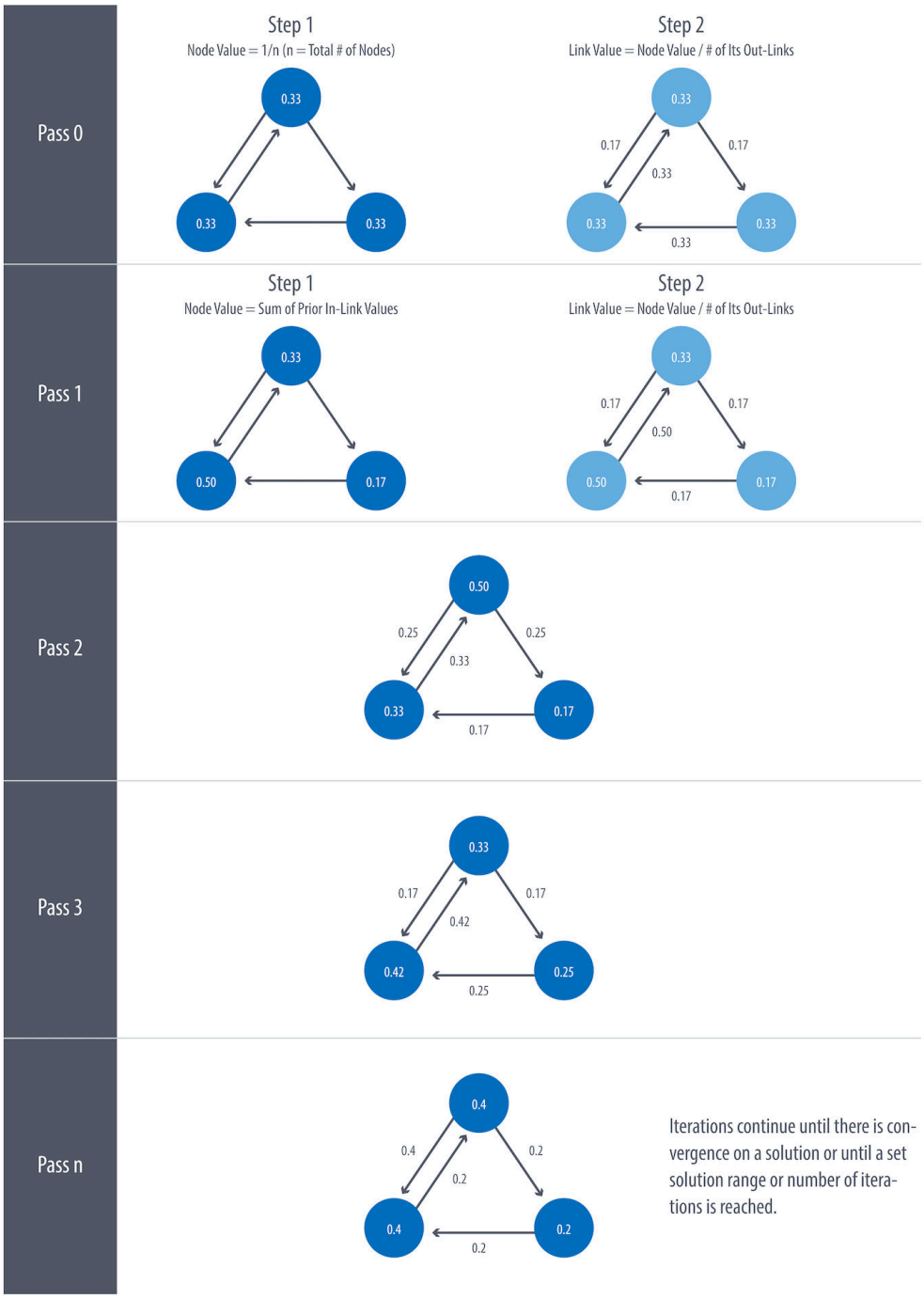


图5-11.PageRank的每次迭代都有两个计算步骤：一个用于更新节点值，另一个用于更新链接值。

迭代、Random Surfers和Rank Sinks

PageRank是一种迭代算法，运行到评分收敛或达到一组迭代次数为止。从概念上讲，PageRank假设有一个网络冲浪者（web surfer）通过链接或者使用随机的

URL来访问网页。阻尼系数（damping factor）_d_ 是指下一次点击链接的概率。你可以把它看作是一个冲浪者可能会变得无聊，然后随机切换到另一个页面。PageRank分数表示通过传入链接而非随机访问页面的可能性。

没有传出关系的节点或节点组（也称为悬空节点, Dangling node），这些节点可以通过拒绝和其他节点连接，来独占PageRank分数。这就是所谓的Rank Sink。你可以把这想象成一个冲浪者被困在一个页面或页面的一个子集上，没有出路。另一个困难是节点只指向一个组中的其他节点（比如类似小世界网络的情况）。当冲浪者在节点之间来回跳跃时，循环引用会导致其等级增加。这些情况如图5-12所示。

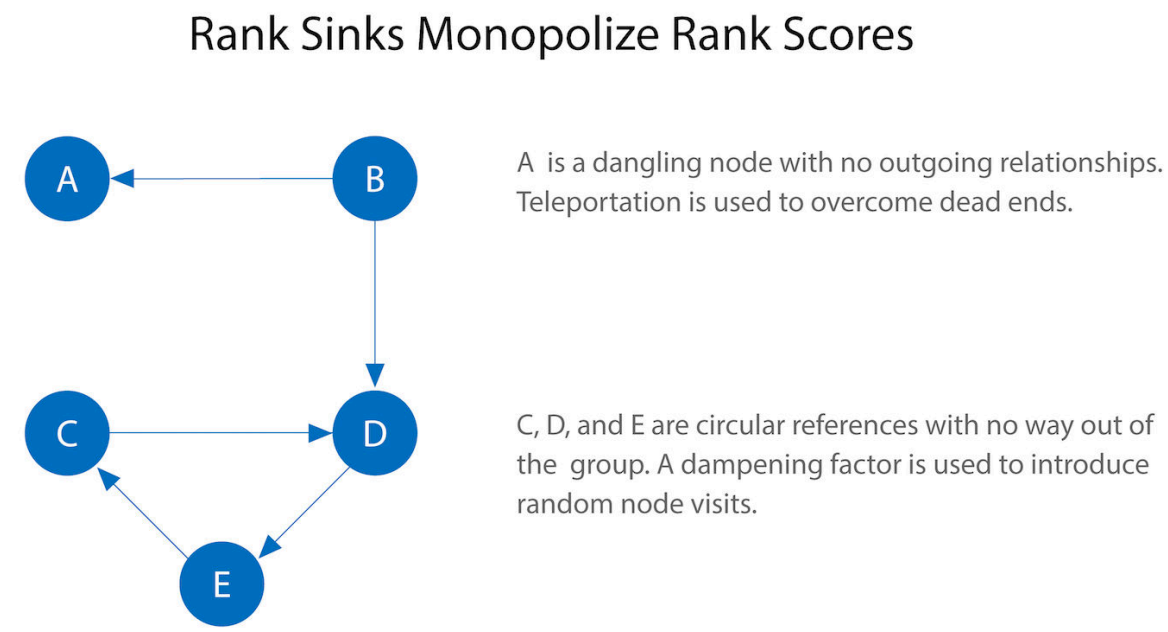


图5-12.Rank Sink是由没有传出关系的节点或节点组引起的。

有两种策略可以避免Rank Sink。首先，当到达没有传出关系的节点时，PageRank假定它对所有节点有传出关系。穿越这些看不见的链接有时被称为心灵传送（teleportation）。第二，阻尼系数（damping factor）提供了另一个避免Rank Sink的机会，通过引入直接链接与随机节点访问的概率。当您将d设置为0.85时，会有15%的可能性访问完全随机的节点。

虽然最初的公式建议的阻尼系数为0.85，但它最初的用途是在符合链接的幂律法则（大多数页面链接很少，少数页面有很多）的万维网上。降低阻尼系数会降

低在进行随机跳跃之前遵循长的关系路径的可能性。反过来，这会增加节点的前置节点对其分数和排名的贡献。

如果您看到PageRank的意外结果，那么值得对该图进行一些探索性分析，以确定这些问题中是否有任何一个是原因。阅读Ian Rogers的文章“The Google PageRank Algorithm and How It Works”，了解更多信息。

PageRank算法的使用场景？

PageRank现在用于Web索引之外的许多域。只要你想在网络上获得广泛影响的计算，就使用这个算法。例如，如果你想寻找一个对生物功能影响最大的基因，它可能不是最相关的算法。事实上，这种影响最大的基因可能是和其他基因其他功能连接最多的那些基因（更多考虑度中心性）。

示例用例包括：

- 向用户提供他们可能希望关注的其他帐户的建议（Twitter为此使用个性化的PageRank）。该算法运行在一个包含共享兴趣和公共的连接的图上。该方法在P. Gupta等人的论文“WTF: The Who to Follow Service at Twitter”中有更详细的描述。
- 预测公共空间或街道的交通流量和人员流动。该算法运行在道路交叉口的图上，其中PageRank分数反映了人们在每条街道上停车或结束行程的趋势。这一点在B. Jiang、S. Zhao和J. Yin的论文“Self-Organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study”中有更详细的描述。
- 作为医疗和保险行业异常和欺诈检测系统的一部分。PageRank有助于揭示医生或服务提供者的异常行为，然后将分数输入机器学习算法。David Gleich在他的论文“PageRank Beyond the Web”中描述了该算法的更多用途。

Apache Spark上的网页排名算法

现在我们准备好执行PageRank算法了。GraphFrames支持PageRank的两种实现：

第一个实现，以固定的迭代次数运行PageRank。这可以通过设置maxIter参数来运行。

第二个实现，运行PageRank直到收敛。这可以通过设置tol参数来运行。

具有固定迭代次数的PageRank

让我们看一个固定迭代的案例。

```
results = g.pageRank(resetProbability=0.15, maxIter=20)
results.vertices.sort("pagerank", ascending=False).show()
```

请注意，在Spark中，阻尼系数(damping factor)更直观地称为重置概率（reset probability），其值为相反的值。也就是说，本例中的重置概率=0.15等于阻尼因子：Neo4j中的0.85。

如果我们在PySpark中运行代码，就可以看到如下的输出：

```
+-------+------------------+
|     id|          pagerank|
+-------+------------------+
|   Doug| 2.2865372087512252|
|   Mark| 2.1424484186137263|
|  Alice|  1.520330830262095|
|Michael| 0.7274429252585624|
|Bridget| 0.7274429252585624|
|Charles| 0.5213852310709753|
|    Amy| 0.5097143486157744|
|  David|0.36655842368870073|
|  James| 0.1981396884803788|
+-------+------------------+
```

正如我们所料，Doug拥有最高的PageRank，因为他的子图中被所有的人所关注。虽然Mark只有一个追随者，但那个追随者是Doug，所以Mark在这个图中也被认为是重要的。重要的不仅是追随者的数量，还有这些追随者的重要性。

在PageRank算法中，图中的关系没有权重，因此每个关系都被认为是相等的。通过在关系DataFrame中指定权重列来添加关系权重。

收敛型PageRank

现在，我们将尝试运行PageRank的收敛型实现，在这个算法中，运行直到在设置的范围之内：

```
results = g.pageRank(resetProbability=0.15, tol=0.01)
results.vertices.sort("pagerank", ascending=False).show()
```

如果我们在PySpark上运行，会看到如下的结果

```
+-------+------------------+
|     id|          pagerank|
+-------+------------------+
|   Doug| 2.2233188859989745|
|   Mark|  2.090451188336932|
|  Alice| 1.5056291439101062|
|Michael|  0.733738785109624|
|Bridget|  0.733738785109624|
|    Amy|  0.559446807245026|
|Charles| 0.5338811076334145|
|  David|0.40233236274180685|
|  James|0.21747203391449021|
+-------+------------------+
```

每个人的PageRank得分与固定迭代次数的变量略有不同，但正如我们所期望的，最后的排名的顺序保持不变。

尽管完美解决方案的收敛听起来很理想，但在某些情况下PageRank无法在数学意义上收敛。对于较大的图，PageRank执行时间可能很长。误差限制

（tolerance limit）有助于为收敛结果设置可接受的范围，但许多人选择使用最大迭代型PageRank或者将最大迭代与收敛进行联合使用。最大迭代型PageRank通常会在性能上更稳定一些。无论您选择哪个选项，您可能需要测试几个不同的限制，以找到适合您的数据集的内容。较大的图通常需要比中等大小的图更多的迭代或更小的公差，以获得更好的精度。

Neo4j上的PageRank

我们也可以在Neo4j中运行PageRank。用以下查询来计算每个节点的PageRank：

```
CALL algo.pageRank.stream('User', 'FOLLOWS', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.getNodeById(nodeId).id AS page, score
ORDER BY score DESC
```

运行该procedure将会得到如下的结果：

```
+---------------------------------+
| page       | score              |
+---------------------------------+
| "Doug"     | 1.671956494869664  |
| "Mark"     | 1.5623059164267037 |
| "Alice"    | 1.1116563910618424 |
| "Bridget"  | 0.5358271526871249 |
| "Michael"  | 0.5358271526871249 |
| "Amy"      | 0.3858750030398369 |
| "Charles"  | 0.38475333093665537|
| "David"    | 0.27750000506639483|
| "James"    | 0.15000000000000002|
+---------------------------------+
```

与Spark示例一样，Doug是最有影响力的用户，Mark紧随其后，是Doug关注的唯一用户。我们可以在图5-13中看到节点相对彼此的重要性。

PageRank有多种不同的实现，因此即使顺序相同，它们也可以产生不同的得分。Neo4j使用1减去阻尼系数来初始化节点，而SPARK使用值1。在这种情况下，最终的相对排名是相同的，但用于达到这些结果的基础得分值是不同的。

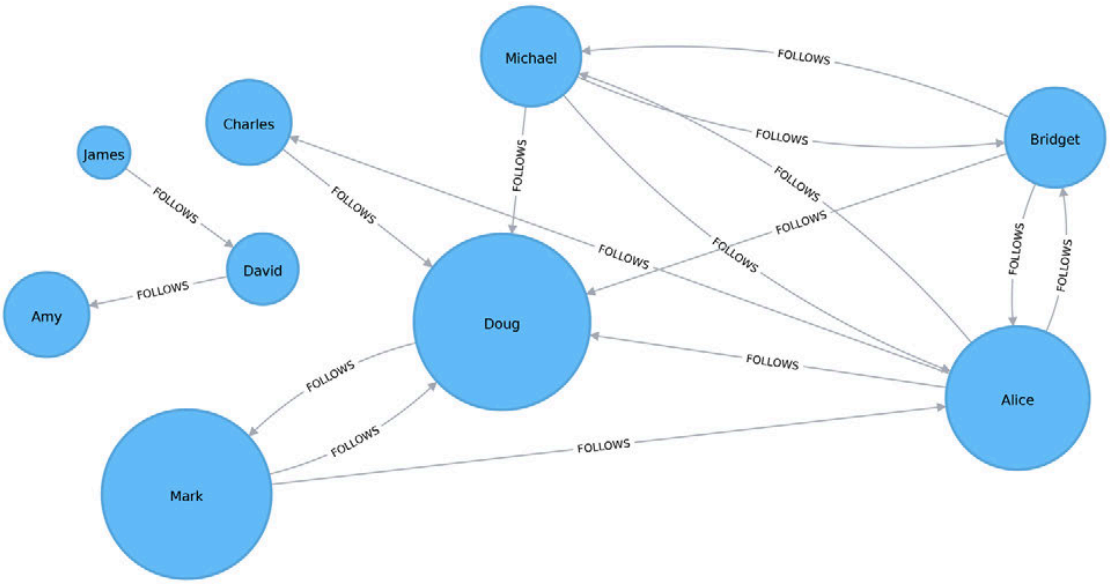


图5-13. 页面排名可视化

与Spark示例一样，运行PageRank算法的图中的关系没有权重，因此每个关系都被认为是相等的。通过在传递给PageRank过程的配置中包含weightProperty属性，可以考虑关系权重。例如，如果关系具有包含权重的属性权重，我们将向过程传递以下配置：weightProperty: "weight"。

PageRank变体：个性化的PageRank

个性化的PageRank（Personalized PageRank, PPR）是PageRank算法的一种变体，它从特定节点的角度计算图形中节点的重要性。对于PPR，随机跳转（PageRank中15%的随机跳转）指的是跳转回来一系列启动节点。这会使结果偏向开始节点，或使其个性化。这种偏差和本地化使得PPR对于高目标的建议很有用。

Apache Spark上的个性化页面排名

我们可以通过传入sourceId参数来计算给定节点的个性化PageRank得分。以下代码计算Doug的PPR：

```
me = "Doug"
results = g.pageRank(resetProbability=0.15, maxIter=20, sourceId=me)
people_to_follow = results.vertices.sort("pagerank", ascending=False)
already_follows = list(g.edges.filter(f"src = '{me}'").toPandas()["dst"])
people_to_exclude = already_follows + [me]
people_to_follow[~people_to_follow.id.isin(people_to_exclude)].show()
```

这个查询的结果可以用来为Doug建议应该关注的人。请注意，我们还确保将Doug已经关注的人以及他自己被排除在我们的最终结果之外。

如果我们在PySpark中运行该代码，我们将看到这个输出：(代码无法运行)

```
+---------+-----------------------+
| id      | pageRank              |
+---------+-----------------------+
| Alice   | 0.1650183746272782    |
| Michael | 0.048842467744891996  |
| Bridget | 0.048842467744891996  |
| Charles | 0.03497796119878669   |
| David   | 0.0                   |
| James   | 0.0                   |
| Amy     | 0.0                   |
+---------+-----------------------+
```

Alice是Doug最应该关注的人，但我们也可以建议关注Michael和Bridget。

总结

中心性算法是识别网络中影响因素的一个很好的工具。在本章中，我们学习了典型的中心性算法：度中心性、紧密中心性、中介中心性和PageRank。我们还讨论了几个变体来处理诸如长运行时间和独立组件等问题，以及其他用途的选项。

中心性算法有许多广泛的用途，我们鼓励各种分析和探索。你可以运用我们学到的知识，找到传播信息的最佳接触点，找到控制资源流动的隐藏经纪人，并发现隐藏在阴影中的间接权力参与者。

接下来，我们将讨论研究组和分区的社区检测算法。

第六章 社区检测算法

形成一个社区在所有类型的网络中都很常见，识别它们对于评估群体行为和突发现象都很重要。通常来说，社区的成员在群体内的关系比在群体外的节点多，这是社区检测的一般原则。识别这些相关集体可以揭示节点群集、独立组和网络结构。此信息有助于推断对等的各组的相似行为和偏好、弹性估算和查找嵌套关系，也可以为其他分析准备数据。社区检测算法也常用于生成用来做一般性检测的网络可视化图。

我们将提供最具代表性的社区检测算法的详细信息，包括：

- 计算整体关系密度的三角形计数(Triangle Count)和聚类系数（Clustering Coefficient）
- 用于查找连接聚类的强连接组件（Strongly Connected Components）和连接组件(Connected Components)
- 基于节点标签快速推断分组的标签传播(Label Propagation)
- 用于查看分组品质和层次结构的Louvain模块化算法（Louvain Modularity）

我们将解释这些算法是如何工作的，并在Apache Spark和Neo4j中运行示例。如果一个算法只在一个平台上可用，我们将只提供一个示例。在这些算法中，我么使用带有权重的关系，因为这些算法通常用于捕获不同关系的重要性。

图6-1概述了这里介绍的社区检测算法之间的差异，表6-1提供了每个算法计算什么，以及示例用法。

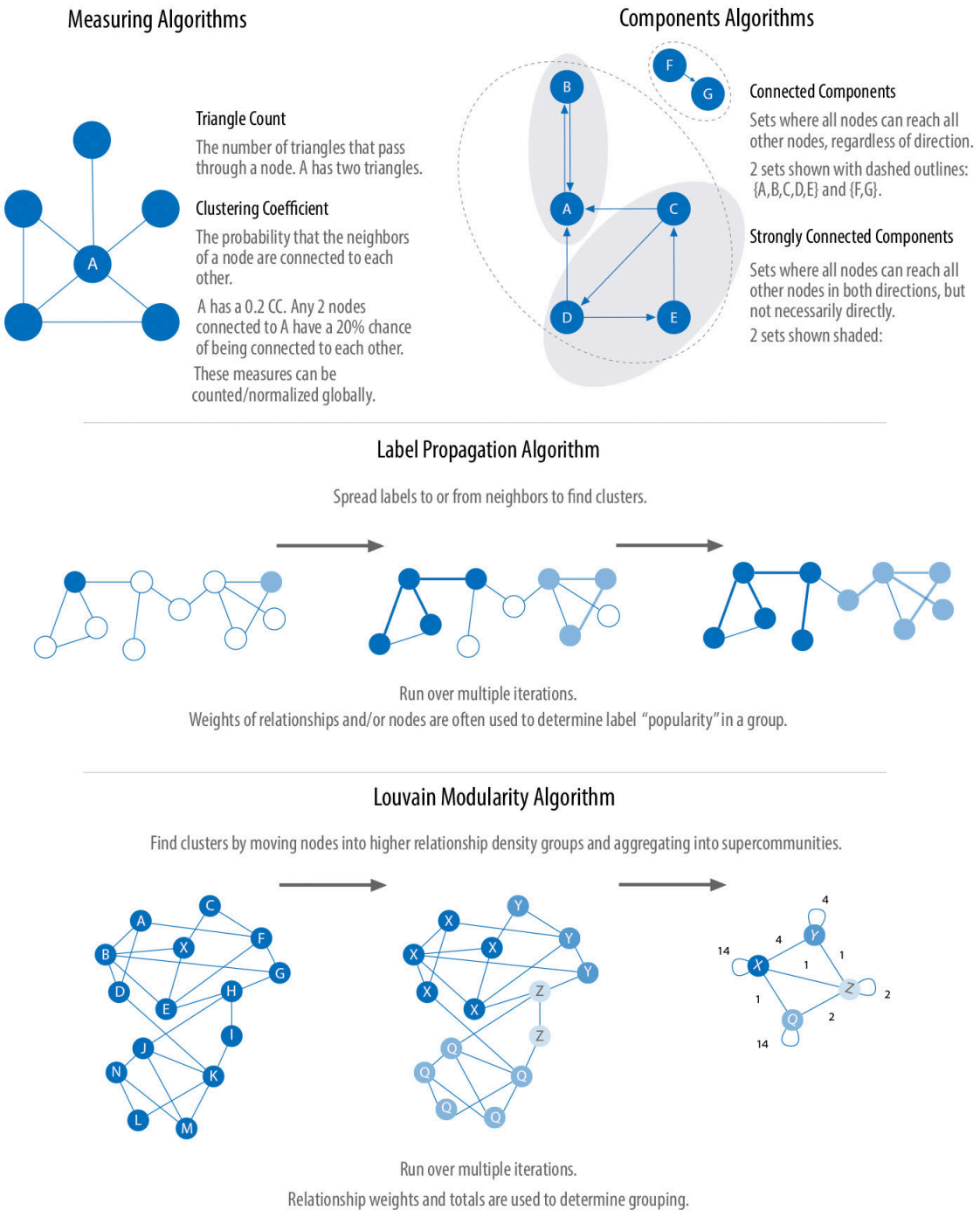


图6-1.典型的社区检测算法

我们会使用以下的可相互替换的术语：集（set）、分区（partition）、集群（cluster）、组（group）和社区（community）。它们表达的都是，节点可以被分组。社区检测（community detection）算法也称为聚类（clustering）和分区（partitioning）算法。在每一部分中，我们使用文献中最突出的那个术语来描述这个算法。

表6-1.社区检测算法综述

算法类型	它怎么工作	使用举例	Spark案例	Neo4j案例
三角计数和聚类系数	测量有多少个节点形成三角形，测量节点形成聚类的程度	评估群组的稳定性，评估网络是否存在小世界行为，它们被看成是紧密聚类	Yes	Yes
强连接组件	找到一个群体，在这个群体内，任何接一个节点都可以从其他节点沿着关系的方向访问到	根据群体的附属关系或者相似商品来形成产品推荐	Yes	Yes
连接组件	找到一个群体，在这个群体内，任何一个节点都可以从其他节点访问到	为其他算法快速形成群组，并识别孤岛	Yes	Yes
标签传播	根据邻近的多数节点来传播标签，来推断聚类	在社交沟通理解舆论，或者在联合处方中找到风险	Yes	Yes
Louvain模块化	通过比较关系权重和与标准相比的密度，来逐渐最大化预设的群体精度	在欺诈分析中，评估一个群体中是否有离散的不良行为或者形成作弊环	No	Yes

首先，我们将描述示例使用的数据，并将数据导入Spark和Neo4j。算法按表6-1中列出的顺序进行介绍。对于每一个算法，你将找到一个简短的描述和建议，关于何时使用它。大多数章节还包括有关何时使用相关算法的指导。我们在每个算法部分的末尾使用示例数据来演示示例代码。

当使用社区检测算法时，要注意关系的密度。如果图非常密集，那么最终可能会导致所有节点聚集在一个或几个集群中。你可以通过度、关系权重或相似性度量的过滤来缓解这一点。

另一方面，如果图太稀疏，连接的节点很少，那么您可能会得到每个聚类只有一个节点。在这种情况下，尝试合并进来更多相关信息和其他关系类型。

图数据示例：软件依赖关系图

依赖关系图特别适合于展示社区检测算法之间的细微的差异，因为它们往往更具关联性和层次性。尽管依赖关系图用于从软件到能源网格的各个领域，本章中的示例是针对包含Python库之间依赖关系的图运行的。开发人员使用这种软件依赖关系图来跟踪软件项目中的可传递依赖关系和冲突。你可以从书的Github存储库下载节点和文件。

表6-2 sw-nodes.csv

id
six
pandas
numpy
python-dateutil
pytz
pyspark
matplotlib
spacy
py4j
jupyter
jpy-console
nbconvert
ipykernel
jpy-client
jpy-core

表6-3 sw-relationships.csv

src	dst	relationship
pandas	numpy	DEPENDS_ON
pandas	pytz	DEPENDS_ON
pandas	python-dateutil	DEPENDS_ON
python-date	six	DEPENDS_ON
pyspark	py4j	DEPENDS_ON
matplotlib	numpy	DEPENDS_ON
matplotlib	python-dateutil	DEPENDS_ON
matplotlib	six	DEPENDS_ON
matplotlib	pytz	DEPENDS_ON
spacy	six	DEPENDS_ON
spacy	numpy	DEPENDS_ON
jupyter	nbconvert	DEPENDS_ON
jupyter	ipykernel	DEPENDS_ON
jupyter	jpy-console	DEPENDS_ON
jpy-console	jpy-client	DEPENDS_ON
jpy-console	ipykernel	DEPENDS_ON
jpy-client	jpy-core	DEPENDS_ON
nbconvert	jpy-core	DEPENDS_ON

图6-2显示了我们想要构建的图。在这个图中，我们看到所有库分成三个组（之间相互断开）。我们可以使用较小数据集上的可视化作为工具来帮助验证由社区检测算法计算出来的聚类。

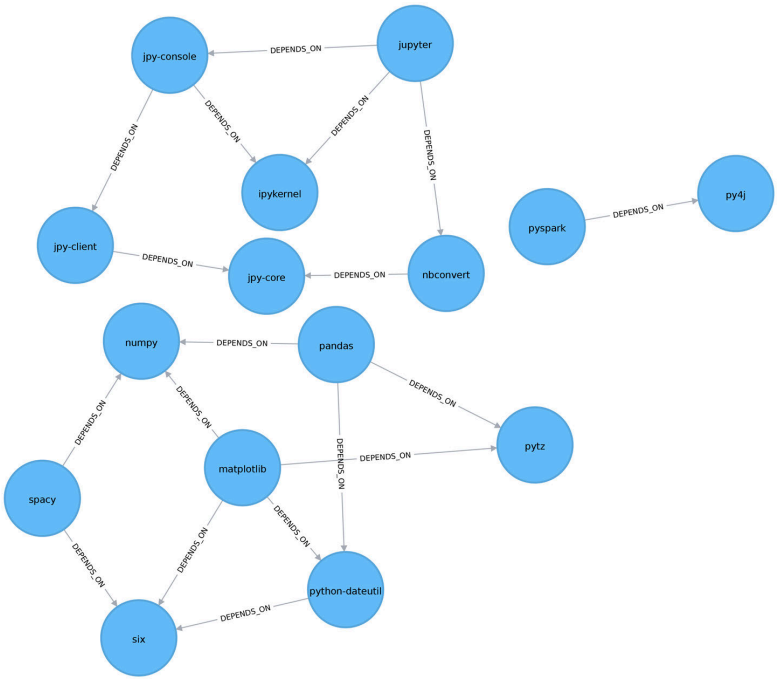


图6-2.图模型

让我们用示例csv文件中的数据，在Spark和Neo4j中创建图。

将数据导入Apache Spark

我们将首先从Apache Spark和graphframes包中导入所需的包：

```
from graphframes import *
```

以下函数从示例csv文件创建一个GraphFrame：

```
def create_software_graph():
    base = "file:///home/retire2053/source/graph_algorithms_resources/"
    nodes = spark.read.csv(base+"data/sw-nodes.csv", header=True)
    relationships = spark.read.csv(base+"data/sw-relationships.csv", header=True)
    return GraphFrame(nodes, relationships)
```

现在来调用这个函数。

```
g = create_software_graph()
```

向Neo4j中导入数据

我们将在Neo4j中做相同的事情。这段查询将导入节点。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:Library {id: row.id})
```

以下查询将会导入关系。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:Library {id: row.src})
MATCH (destination:Library {id: row.dst})
MERGE (source)-[:DEPENDS_ON]->(destination)
```

现在，我们已经加载了我们的图，现在可以开始算法了！

三角形计数和聚类系数

三角形计数(triangle count)算法和聚类系数(clustering coefficient)算法由于经常一起使用，因而常同时出现。三角形计数确定通过图中每个节点的三角形数。三角形是由三个节点组成的集合，三个节点中的每个节点与所有其他节点都有关系。三角形计数也可以全局运行以评估我们的整体数据集。

具有大量三角形的网络更有可能展现出小世界（small-world）的结构和行为。

聚类系数算法的目标是测量一个组的聚类程度（how tightly it is clustered）与它的可能的聚类度(how tightly it could be)之比值。该算法在计算中使用三角形

计数，它计算现有三角形数与可能的关系的比率。最大值为1表示组内的每个节点都连接到其他节点。

聚类系数有两种类型：局部聚类和全局聚类。

局部聚类系数

一个节点的局部聚类系数是其相邻节点也之间被互相连接的可能性。这个分数的计算涉及三角形计数。

一个节点的聚类系数可以这样计算，将通过该节点的三角形数乘以2，然后将其除以组中的最大关系数（始终是该节点的度数减去1）来求得。图6-3描述了具有五个关系的节点的不同三角形和聚类系数的示例。

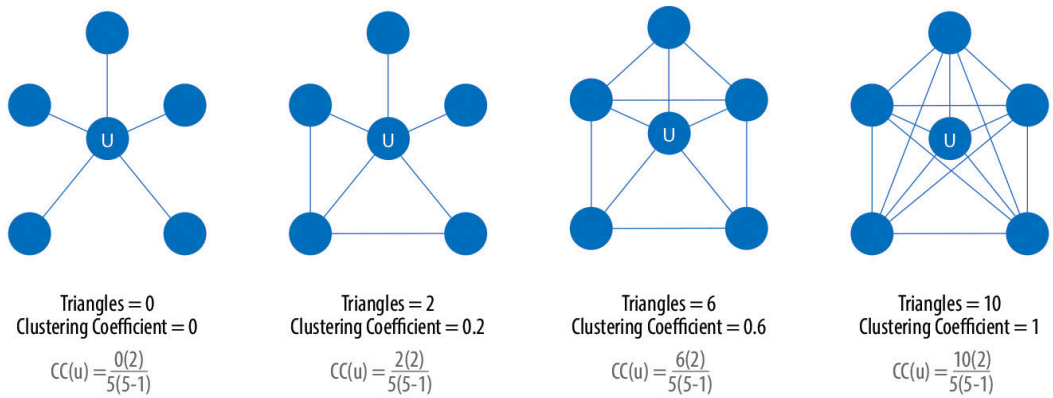


图6-3.节点U的三角形计数和聚类系数

在图6-3中，我们使用一个具有五个关系的节点，这使得聚类系数始终等于三角形数的10%。我们可以看到，当我们改变关系的数量时，情况并非如此。如果我们把第二个例子改为有四个关系（和相同的两个三角形），那么系数是0.33。

节点的聚类系数使用以下公式：

$$CC(u) = \frac{2R_u}{k_u(k_u - 1)}$$

在这个公式中

- u是一个节点
- Ru是通过u的某两个邻居的关系数（这正好就是通过u的三角形数量）。

- Ku是u的度数

全局聚类系数

全局聚类系数是局部聚类系数的归一化之后的和。聚类系数为我们提供了一种有效的方法来寻找明显的群体，如小团体，其中每个节点都与所有其他节点有关系，但我们也可以指定阈值来设置级别（例如，节点有40%的连接）。

三角形计数和聚类系数的使用场景

当你需要确定组的稳定性（相互连接的数量，代表了稳定性）或作为其他网络度量（如聚类系数）的一部分时，请使用三角形计数。三角计数在社交网络分析中很流行，它被用来检测社区。

聚类系数可以提供随机选择的节点被连接的概率。您还可以使用它快速评估特定组或整个网络的内聚性。这些算法一起用于估计弹性和寻找网络结构。

示例用例包括：

- 识别将给定网站分类为垃圾内容的功能。这一点在L. Becchetti等人的论文“Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs”中进行了描述。
- 调查Facebook社交图的社区结构，研究人员在一张原本稀疏的全球图中发现了密集的用户社区。这项研究发表在J.Ugander等人的论文《The Anatomy of the Facebook Social Graph》中。
- 探索网页的主题结构，并基于网页之间的相互链接，检测具有共同主题的网页社区。有关更多信息，请参见J. P. Eckmann和E. Moses的“Curvature of Co-Links Uncovers Hidden Thematic Layers in the World Wide Web”。

Apache Spark上的三角形计数算法

现在我们准备执行三角形计数算法。我们可以使用以下代码：

```
result = g.triangleCount()
(result.sort("count", ascending=False)
```

```
.filter('count > 0')
.show())
```

如果我们在PySpark中运行该代码，我们将看到这个输出：

```
+-----+---------------+
|count|             id|
+-----+---------------+
|    1|            six|
|    1|      ipykernel|
|    1|        jupyter|
|    1|python-dateutil|
|    1|     matplotlib|
|    1|    jpy-console|
+-----+---------------+
```

图中的三角形表示一个节点的两个邻居节点也相互之间是邻居。我们有六个库参与了这种三角关系。

如果我们想知道哪些节点在这些三角形？这就需要一个stream中包含三角形。为此，我们需要Neo4j。

Neo4j的三角形计数

使用Spark无法获取三角形的具体结果，但我们可以使用Neo4j得到这个返回值：

```
CALL algo.triangle.stream("Library","DEPENDS_ON")
YIELD nodeA, nodeB, nodeC
RETURN algo.getNodeById(nodeA).id AS nodeA,
algo.getNodeById(nodeB).id AS nodeB,
algo.getNodeById(nodeC).id AS nodeC
```

运行这个查询，可以得到如下的结果

```
+-----------------------------------------------------------+
| nodeA             | nodeB             | nodeC             |
+-----------------------------------------------------------+
| "six"             | "python-dateutil" | "matplotlib"      |
| "python-dateutil" | "six"             | "matplotlib"      |
| "matplotlib"      | "six"             | "python-dateutil" |
| "jupyter"         | "jpy-console"     | "ipykernel"       |
| "jpy-console"     | "jupyter"         | "ipykernel"       |
| "ipykernel"       | "jupyter"         | "jpy-console"     |
+-----------------------------------------------------------+
```

我们看到的六个库和之前的结果一样，但现在我们知道它们是如何连接的。matplotlib、six和python-dateutil构成一个三角形。jupyter、jpy-console和ipykernel构成另一个。

我们可以在图6-4中看到这些三角形。

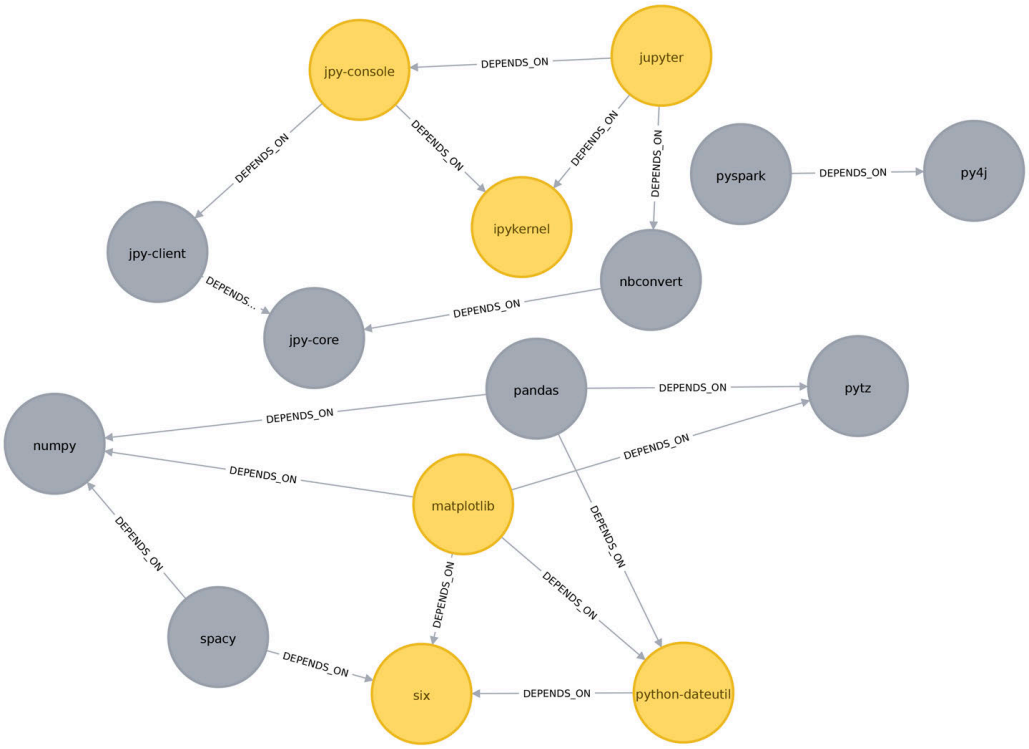


图6-4.软件依赖关系图中的三角形

Neo4j的局部聚类系数

还可以求出局部聚类系数。以下查询将为每个节点计算此值：

```
CALL algo.triangleCount.stream('Library', 'DEPENDS_ON')
YIELD nodeId, triangles, coefficient
WHERE coefficient > 0
RETURN algo.getNodeById(nodeId).id AS library, coefficient
ORDER BY coefficient DESC
```

运行该查询，可以得到如下的结果：

```
+-----------------------------------------+
| library           | coefficient         |
+-----------------------------------------+
| "ipykernel"       | 1.0                 |
| "six"             | 0.3333333333333333  |
```

```
| "python-dateutil" | 0.3333333333333333  |
| "jupyter"         | 0.3333333333333333  |
| "jpy-console"     | 0.3333333333333333  |
| "matplotlib"      | 0.16666666666666666 |
+-----------------------------------------+
```

ipykernel的分数是1，这意味着ipykernel的所有邻居节点都是彼此的邻居。我们可以在图6-4中清楚地看到这一点。这告诉我们，直接围绕ipykernel的社区是非常有凝聚力的。

在这个代码示例中，我们过滤了系数得分为0的节点，但是系数较低的节点也可能很有趣。一个低分可以表示一个节点是一个结构孔。结构空可能是一个与其他社区中的节点连接良好的节点，而这些社区之间在其他方面没有相互连接。这是我们在第5章中讨论的一种寻找潜在桥梁的方法。

强连接组件

强连接组件（Strongly Connected Components，SCC）算法是最早的图算法之一。SCC在有向图中查找连接的节点集，其中每个节点都可以从同一集中的任何其他节点的两个方向上访问。它的运行时操作伸缩性很好，与节点数量成比例。在图6-5中，你可以看到SCC组中的节点不需要是直接邻居，但是在集合中的所有节点之间必须有方向路径。

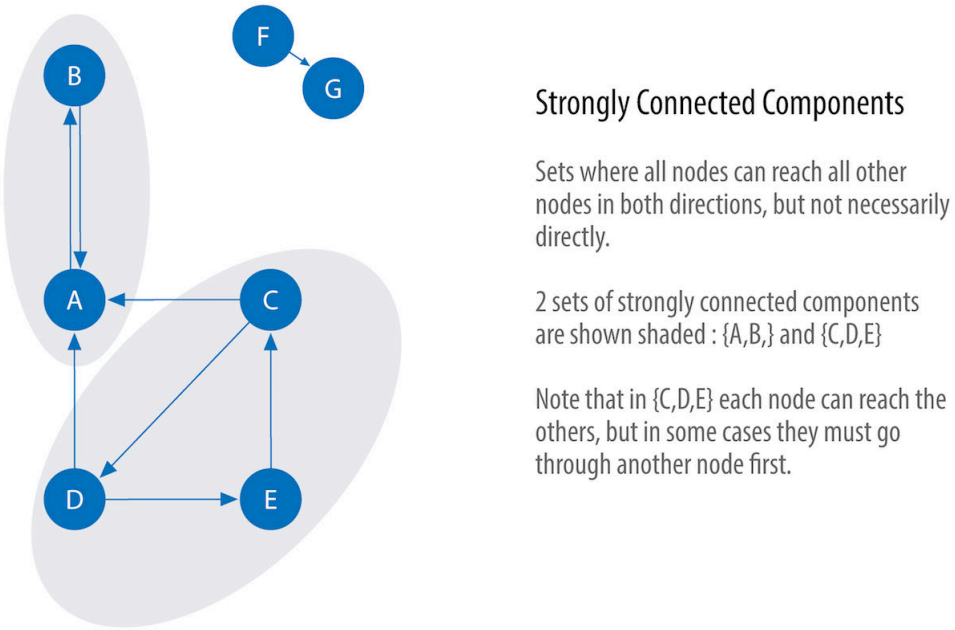


图6-5.强连接组件

将有向图分解为强连接的组件是深度优先搜索算法的经典应用。Neo4j把DFS作为SCC算法的内部实现的一部分。

强连接组件的使用场景

使用强连接组件作为图分析的早期步骤，以了解图的结构，或确定可能需要独立调查的紧密集群。对于推荐引擎等应用程序，可以使用强连接的组件来分析组中的类似行为或关注的重点。

类似SCC的许多社区检测算法被用于查找集群并将其折叠为单个节点，以便进一步进行集群间分析。您还可以使用SCC来可视化分析的周期，例如查找可能死锁的进程，因为每个子进程都在等待另一个成员采取操作。

示例用例包括：

- 如S. Vitali、J. B. Glattfelder和S. Battiston对强大跨国公司的分析 “The Network of Global Corporate Control”，找出每个成员直接和/或间接拥有其他成员股份的一组公司。

- 测量多重调接式无线网络中的路由性能时，计算不同网络配置的连接性。阅读M.K.Marina和S.R.Das的 “Routing Performance in the Presence of Unidirectional Links in Multihop Wireless Networks” 中的更多内容。

- 在许多仅适用于强连接图的图算法中，SCC会充当第一步。在社交网络中，我们发现了许多紧密相连的群体。在这些集合中，人们通常有类似的偏好，SCC算法用于查找此类组，并向尚未购买的组员推荐可能喜欢的页面或可能购买的产品。

有些算法有摆脱无限循环的策略。如果我们正在编写自己的算法或寻找非终止进程，我们可以使用scc来检查循环。

Apache Spark的强连接组件

我们从Apache Spark开始考察算法。我们先导入Spark和GraphFrames所需的包。

```
from graphframes import *
from pyspark.sql import functions as F
```

现在我们已经准备好之行强连接组件算法。我们将会用它来判断是否在图中有环形依赖。

如果在两个节点之间有双向的通路，这两个节点只能在同一个强连接组件中。

我们编写代码如下：

```
result = g.stronglyConnectedComponents(maxIter=10)
(result.sort("component")
.groupby("component")
.agg(F.collect_list("id").alias("libraries"))
.show(truncate=False))
```

在PySpark中运行，得到如下的结果：

```
+-------------+-----------------+
|component    |libraries        |
+-------------+-----------------+
|180388626432 |[jpy-core]       |
|223338299392 |[spacy]          |
|498216206336 |[numpy]          |
|523986010112 |[six]            |
|549755813888 |[pandas]         |
|558345748480 |[nbconvert]      |
|661424963584 |[ipykernel]      |
|721554505728 |[jupyter]        |
|764504178688 |[jpy-client]     |
|833223655424 |[pytz]           |
|910533066752 |[python-dateutil]|
|936302870528 |[pyspark]        |
|944892805120 |[matplotlib]     |
|1099511627776|[jpy-console]    |
|1279900254208|[py4j]           |
+-------------+-----------------+
```

您可能会注意到每个库节点都被分配给一个唯一的组件。这和我们的预期相同，每个节点都在其自己的分区（partition）中。这意味着我们的软件项目在这些库之间没有循环依赖关系。

Neo4j上的强连接组件

让我们使用Neo4j运行相同的算法。执行以下查询以运行算法：

```
CALL algo.scc.stream("Library", "DEPENDS_ON")
YIELD nodeId, partition
RETURN partition, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

传递的参数如下

参数	意义
Library	从图中装载的节点的标签
DEPENDS_ON	从图中装载的关系类型

以下是运行之后的结果

```
+-----------------------------------------------------+
| partition | libraries                               |
```

```
+-------------------------------------------------+
| 0          | [(:Library {id: "six"})]           |
| 1          | [(:Library {id: "pandas"})]        |
| 2          | [(:Library {id: "numpy"})]         |
| 3          | [(:Library {id: "python-dateutil"})] |
| 4          | [(:Library {id: "pytz"})]          |
| 5          | [(:Library {id: "pyspark"})]       |
| 6          | [(:Library {id: "matplotlib"})]    |
| 7          | [(:Library {id: "spacy"})]         |
| 8          | [(:Library {id: "py4j"})]          |
| 9          | [(:Library {id: "jupyter"})]       |
| 10         | [(:Library {id: "jpy-console"})]   |
| 11         | [(:Library {id: "nbconvert"})]     |
| 12         | [(:Library {id: "ipykernel"})]     |
| 13         | [(:Library {id: "jpy-client"})]    |
| 14         | [(:Library {id: "jpy-core"})]      |
+-------------------------------------------------+
```

与Spark示例一样，每个节点都在自己的分区中。

到目前为止，算法仅仅展示了我们的Python库的行为非常好。让我们在图中创建一个循环依赖项，让事情更有趣。这意味着我们将有一些节点在同一个分区中。

以下查询添加了一个额外的库，该库在py4j和pyspark之间创建循环依赖关系：

```
MATCH (py4j:Library {id: "py4j"})
MATCH (pyspark:Library {id: "pyspark"})
MERGE (extra:Library {id: "extra"})
MERGE (py4j)-[:DEPENDS_ON]->(extra)
MERGE (extra)-[:DEPENDS_ON]->(pyspark)
```

我们能够在图6-6中清晰地看到这个循环依赖。

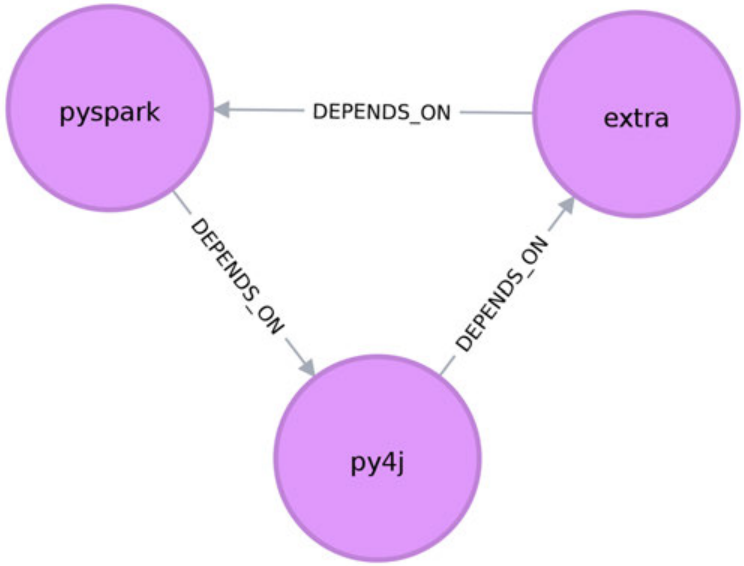


图6-6 在pyspark,py4j和extra之间的循环依赖

我们现在可以运行SCC算法来看是否有不同的结果。

```
+-----------------------------------------------------------------------------------------------------+
| partition | libraries                                                                               |
+-----------------------------------------------------------------------------------------------------+
| 5         | [(:Library {id: "pyspark"}), (:Library {id: "py4j"}), (:Library {id: "extra"})] |
| 0         | [(:Library {id: "six"})]                                                                |
| 1         | [(:Library {id: "pandas"})]                                                             |
| 2         | [(:Library {id: "numpy"})]                                                              |
| 3         | [(:Library {id: "python-dateutil"})]                                                    |
| 4         | [(:Library {id: "pytz"})]                                                               |
| 6         | [(:Library {id: "matplotlib"})]                                                         |
| 7         | [(:Library {id: "spacy"})]                                                              |
| 9         | [(:Library {id: "jupyter"})]                                                            |
| 10        | [(:Library {id: "jpy-console"})]                                                        |
| 11        | [(:Library {id: "nbconvert"})]                                                          |
| 12        | [(:Library {id: "ipykernel"})]                                                          |
| 13        | [(:Library {id: "jpy-client"})]                                                         |
| 14        | [(:Library {id: "jpy-core"})]                                                           |
+-----------------------------------------------------------------------------------------------------+
```

pyspark、py4j和extra都是同一分区的一部分，SCC帮助我们找到了循环依赖关系！

在继续下一个算法之前，我们将从图中删除额外的库及其关系：

```
MATCH (extra:Library {id: "extra"})
DETACH DELETE extra
```

连接组件

连接组件算法（Connected Components，有时称为联合查找或弱连接组件）在无向图中查找连接节点集，其中每个节点都可以从同一集中的任何其他节点访问。它不同于SCC算法，因为它只需要在一个方向上的节点对之间存在路径，而SCC需要在两个方向上都存在路径。BernardA.Galler和MichaelJ.Fischer在1964年的论文“An Improved Equivalence Algorithm”中首次描述了这种算法。

连接组件的使用场景

与SCC一样，连接的组件通常在分析的早期用于理解图的结构。因为它可以有效地伸缩，所以考虑使用此算法来处理需要频繁更新的图。它可以快速显示组之间的新节点，这对于欺诈检测等分析非常有用。

建立起来这样的习惯：事先运行连接组件来测试图是否连接，作为一般图分析的准备步骤。执行这个快速测试可以避免在图的一个孤岛组件上运行算法，那最终会得到不正确的结果。

示例用例包括：

- 跟踪数据库记录的集群，作为删除重复数据过程的一部分。重复数据删除是主数据管理应用程序中的一项重要任务；该方法在A.Monge和C.Elkan的“An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records”中有更详细的描述。
- 分析引文网络。一项研究使用连接的组件来计算一个网络连接的好坏，然后看看如果“集线器”或“权威”节点从图中移走，连接是否仍然存在。本用例将在Y. AN、J.C.M. Janssen和E. E. Milios的论文“Characterizing and Mining Citation Graph of Computer Science Literature”中进一步解释。

Apache Spark上的连接组件

从Apache Spark上先开始，我们将首先从Spark和GraphFrames包中导入所需的包：

```
from pyspark.sql import functions as F
```

现在我们准备执行连接组件算法。

如果两个节点之间有一条路径，则两个节点可以位于同一连接组件中。

为此，我们编写以下代码：

```
spark.sparkContext.setCheckpointDir('/home/retire2053/checkpoint')
result = g.connectedComponents()

(result.sort("component")
.groupby("component")
.agg(F.collect_list("id").alias("libraries"))
.show(truncate=False))
```

运行连接组件算法需要预先提供一个checkpoint目录，并使用setCheckpiontDir方法来实现。运行代码，将会得到如下的结果：

```
+------------+---------------------------------------------------------------------+
|component   |libraries                                                            |
+------------+---------------------------------------------------------------------+
|180388626432|[jpy-core, nbconvert, ipykernel, jupyter, jpy-client, jpy-console]|
|223338299392|[spacy, numpy, six, pandas, pytz, python-dateutil, matplotlib]    |
|936302870528|[pyspark, py4j]                                                      |
+------------+---------------------------------------------------------------------+
```

这个结果显示了三个节点的聚类，在图6-7中可以看到

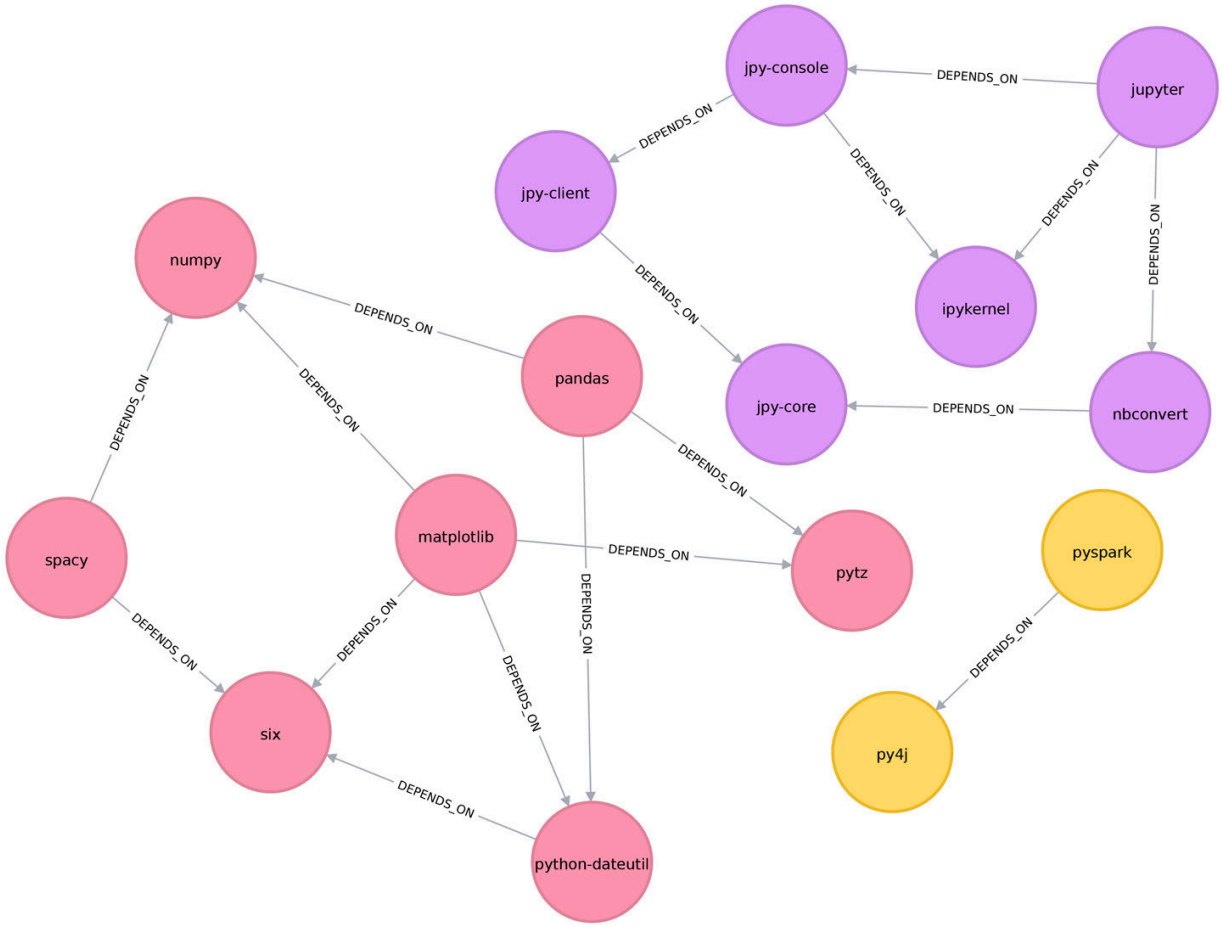


图6-7.通过连接组件算法找到的群集

在这个例子中，很容易看到仅仅通过目视检查就有三个组件。该算法在较大的图上显示出更多的价值，在这些图中，视觉检查不可能或非常耗时。

Neo4j上的连接组件

我们还可以通过运行以下查询在Neo4j中执行此算法：

```
CALL algo.unionFind.stream("Library", "DEPENDS_ON")
YIELD nodeId,setId
RETURN setId, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

传入该算法的参数如下：

参数	意义

Library	从图中装载的节点的标签
DEPENDS_ON	从图中装载的关系类型

以下是输出的结果：

```
+-------+-----------------------------------------------------------------------------------------+
| setId | libraries                                                                               |
+-------+-----------------------------------------------------------------------------------------+
| 1     | [ "six",  "pandas",  "numpy",  "python-dateutil",  "pytz",  "matplotlib",  "spacy"]  |
| 9     | [ "jupyter",  "jpy-console",  "nbconvert",  "ipykernel",  "jpy-client",  "jpy-core"] |
| 5     | [ "pyspark",  "py4j"]                                                                  |
+-------+-----------------------------------------------------------------------------------------+
```

正如预期的那样，我们得到的结果与Spark完全相同。

到目前为止，我们所讨论的两种社区检测算法都是确定性的：每次运行它们时，它们返回相同的结果。接下来的两个算法是非确定性算法的例子，如果我们多次运行它们，即使在相同的数据上，也可能会看到不同的结果。

标签传播

标签传播算法（ Label Propagation algorithm，LPA）是一种在图中快速查找社区的算法。在LPA中，节点根据其直接邻居选择其组。这个过程非常适合在分组不太清晰，但权重可以用来帮助节点确定将自己放在哪个社区中的那些网络。它也适合半监督学习，因为您可以用预先分配的、指示性的节点标签为进程种子。

该算法的灵感是，在一组密集连接的节点中，单个标签可以很快占据主导地位，但在穿越稀疏连接区域时会遇到困难。标签被困在一组密集连接的节点中，当算法完成时，最终具有相同标签的节点被视为同一社区的一部分。该算法通过将成员资格分配给具有最高组合关系和节点权重的标签邻域来解决重叠，其中节点可能是多个集群的一部分。

LPA是U. N. Raghavan, R. Albert，和 S. Kumara于2007年在题为 “Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks” 的论文中提出的一种相对较新的算法。

图6-8描述了标签传播的两种变化，一种是简单的push方法，另一种是依赖关系权重的更典型的pull方法。pull方法很适合并行化。

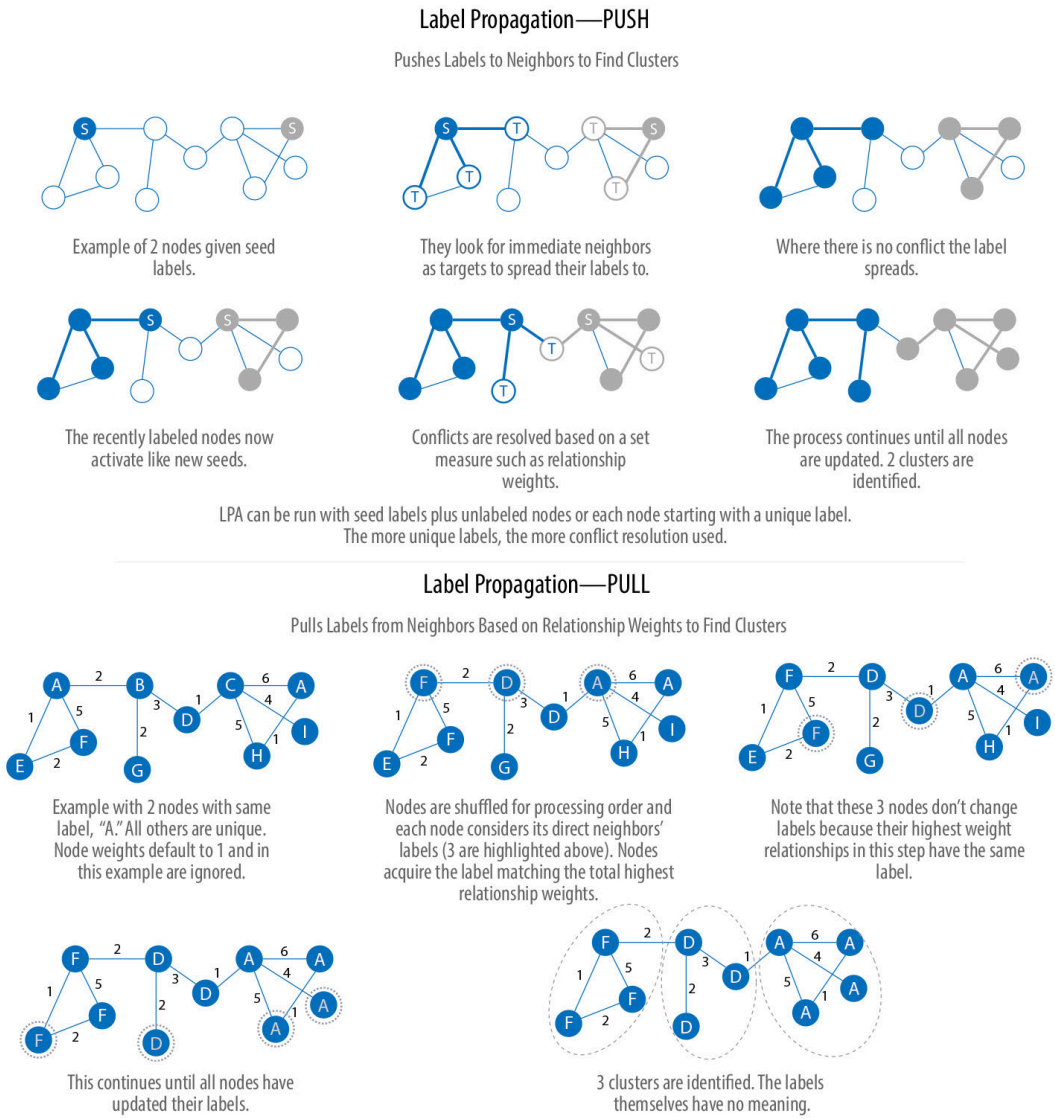


图6-8.标签传播的两种变体

标签传播pull方法常用的步骤有：

1) 每个节点都用一个唯一的标签（标识符）初始化，并且可以使用可选的初步的“种子”标签。
2) 这些标签通过网络传播。
3) 在每次传播迭代中，每个节点更新其标签以匹配具有最大权重的节点，该权重是根据相邻节点的权重及其关系计算得出的。
4) 当每个节点都有其相邻节点的大多数标签时，LPA就会达到收敛。

随着标签的传播，密集连接的节点群很快就在一个独特的标签上达成共识。在传播结束时，只保留少数标签，具有相同标签的节点属于同一个社区。

半监督学习和种子标签

与其他算法不同，标签传播可以在同一个图上多次运行时返回不同的社区结构。LPA评估节点的顺序可能会影响最终的社区形成结果。

当某些节点被赋予初始标签（即种子标签），而其他节点则没有标记，最终方案的范围就在缩小。未标记的节点更可能采用初步标记。

标签传播的这种使用可以被认为是一种半监督学习方法来寻找社区。半监督学习是一类机器学习任务和技术，它对少量标记数据以及大量未标记数据进行操作。我们还可以在图演化时在图上重复运行该算法。

最后，LPA有时不收敛于单个解决方案。在这种情况下，我们的社区结果将不断地在几个非常相似的社区之间切换，并且算法永远不会完成。种子标签有助于引导它走向解决方案。

Spark和Neo4j使用一组最大迭代次数来避免无休止的执行。你应该测试数据的迭代设置，以平衡准确性和执行时间。

标签传播的使用场景

在大型网络中使用标签传播进行初始社区检测，特别是在有权重的情况下。该算法可以并行化，因此在图划分方面速度非常快。

示例用例包括：

- 将Tweet的极性指定为语义分析的一部分。在这个场景中，来自分类器的正种子标签和负种子标签与Twitter关注图相结合使用。有关更多信息，请参见M. Speriosu等人的 “Twitter Polarity Classification with Label Propagation over Lexical Links and the Follower Graph” 。
- 根据化学相似性和副作用特征，找出可能的联合处方药物的潜在危险组合。参见P.Zhang等人的论文 “Label Propagation Prediction of Drug–Drug Interactions Based on Clinical Side Effects” 。
- 推断机器学习模型的对话特征和用户意图。有关更多信息，请参阅Y. Murase等人的论文 “Feature Inference Based on Label Propagation on Wikidata Graph for DST” 。

在Apache Spark上使用标记传播

让我们从Apache Spark开始，首先从Spark和GraphFrames包中导入所需的包：

```
from pyspark.sql import functions as F
```

我们开始运行LPA，以下是代码：

```
result = g.labelPropagation(maxIter=10)
(result
.sort("label")
.groupby("label")
.agg(F.collect_list("id"))
.show(truncate=False))
```

运行代码，将会得到如下的结果：

```
+-------------+----------------------------------+
|label        |collect_list(id)                  |
+-------------+----------------------------------+
|180388626432 |[jpy-core, jpy-console, jupyter]  |
|223338299392 |[matplotlib, spacy]               |
|498216206336 |[python-dateutil, numpy, six, pytz]|
|549755813888 |[pandas]                          |
|558345748480 |[nbconvert, ipykernel, jpy-client]|
|936302870528 |[pyspark]                         |
|1279900254208|[py4j]                            |
+-------------+----------------------------------+
```

与连接组件算法相比，本例中有更多的库的集群。就如何确定集群而言，LPA比连接组件的标准更松散。使用标签传播可以发现两个相互邻近的节点（直接连接的节点）位于不同的集群中。但是，使用连接组件，节点将始终与它的相邻节点处于同一个集群中，因为该算法严格基于关系进行分组。

在我们的示例中，最明显的区别是jupyter库分处于两个社区，一个包含库的核心部分，另一个包含面向客户机的工具。

Neo4j上的LPA

现在让我们用Neo4j上尝试同样的算法。我们可以通过运行以下查询来执行LPA：

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
{ iterations: 10 })
YIELD nodeId, label
RETURN label,
collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

传给该查询的参数如下：

参数	意义
Library	从图中装载的节点的标签
DEPENDS_ON	从图中装载的关系类型
iterations: 10	最大运行的迭代次数

得到的结果如下：

```
+-------------------------------------------------------------------------------+
| label | libraries                                                             |
+-------------------------------------------------------------------------------+
| 14    | ["jupyter", "jpy-console", "nbconvert", "jpy-client", "jpy-core"] |
| 0     | ["six", "python-dateutil", "matplotlib"]                              |
| 2     | ["pandas", "numpy", "spacy"]                                          |
| 8     | ["pyspark", "py4j"]                                                   |
| 4     | ["pytz"]                                                              |
| 12    | ["ipykernel"]                                                         |
+-------------------------------------------------------------------------------+
```

在图6-9中可以看到这个结果，和再Apache Spark中得到的结果类似。

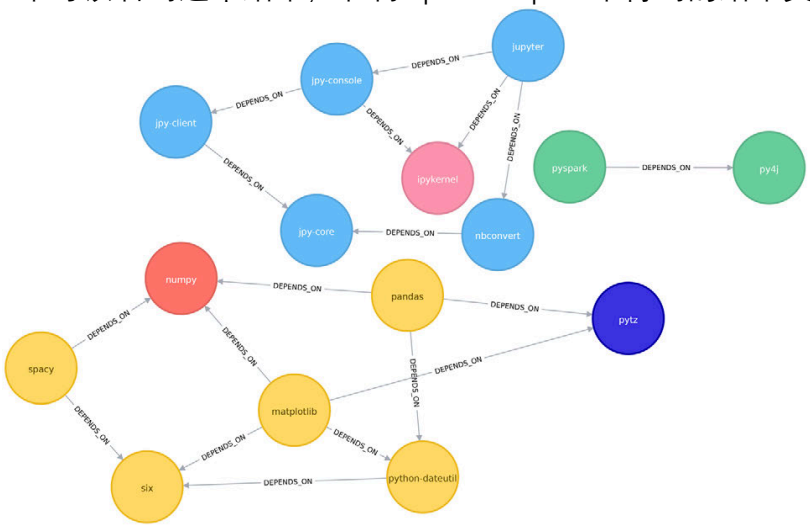


图6-9.标签传播算法发现的聚类

我们也可以在假设图是无向的情况下运行该算法，这意味着节点将尝试采用它们所依赖的库以及依赖这些库的库中的标签。

为此，我们将 DIRECTION:BOTH参数都传递给算法：

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
{ iterations: 10, direction: "BOTH" })
YIELD nodeId, label
RETURN label,
collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

如果运行程序，得到如下的结果：

```
+-------------------------------------------------------------------------------------------+
| label | libraries                                                                         |
+-------------------------------------------------------------------------------------------+
| 2     | ["six", "pandas", "numpy", "python-dateutil", "pytz", "matplotlib", "spacy"]      |
| 14    | ["jupyter", "jpy-console", "nbconvert", "ipykernel", "jpy-client", "jpy-core"]    |
| 5     | ["pyspark", "py4j"]                                                               |
+-------------------------------------------------------------------------------------------+
```

聚类的数量从6个减少到了4个，图中matplotlib部分的所有节点现在都分组在一起。这可以在图6-10中更清楚地看到。

图6-10。忽略关系方向时，标签传播算法找到的聚类

虽然在这些数据上运行标签传播的结果对于无向和有向计算是相似的，但是在复杂的图上，您会看到更显著的差异。这是因为忽略方向会导致节点尝试采用更多的标签，而不考虑关系的方向。

Louvain模块化

Louvain模块化算法在将节点分配给不同的组时，通过比较社区密度来查找集群。您可以将其视为一种“假设”分析，尝试各种分组，以达到全球最佳。

Louvain算法于2008年提出，是最快的模块化算法之一。除了检测社区之外，它还揭示了不同规模的社区层次。这对于理解不同粒度级别的网络结构很有用。

与平均或随机样本相比，Louvain通过查看聚类中连接的密度来量化节点分配给一个组的最佳程度。这种分配社区的测量称为模块化。

基于品质分组的模块化

模块化是一种通过将一个图划分为更粗粒度的模块（或聚类），然后测量分组强度来发现社区的技术。与仅仅观察一聚类内连接的浓度不同，该方法将给定聚类中的关系密度与聚类间的密度进行比较。这些分组的品质的度量方法称为模块化。

模块化算法先构成局部社区之后在全局范围内优化，使用多个迭代测试不同的分组并增加粒度。该策略确定了社区层次结构，并提供了对整体结构的广泛理解。然而，所有模块化算法都有两个缺点：

他们会将较小的社区合并为较大的社区。

当存在具有类似模块性的多个候选分区时，可能会出现平台，形成局部最大值并阻止进展。

有关更多信息，请参阅B.H.Good、Y.A.De Montjoye和A.Clauset的论文“The Performance of Modularity Maximization in Practical Contexts”。

计算模块化

简化的模块化计算是基于给定组中关系的比率（fraction）减去期望比率（如果关系在所有节点之间随机分布）。该值始终介于1和-1之间，正值表示的关系密度比您预期的几率更大，负值则相反。图6-11说明了基于节点分组的几个不同模块化得分。

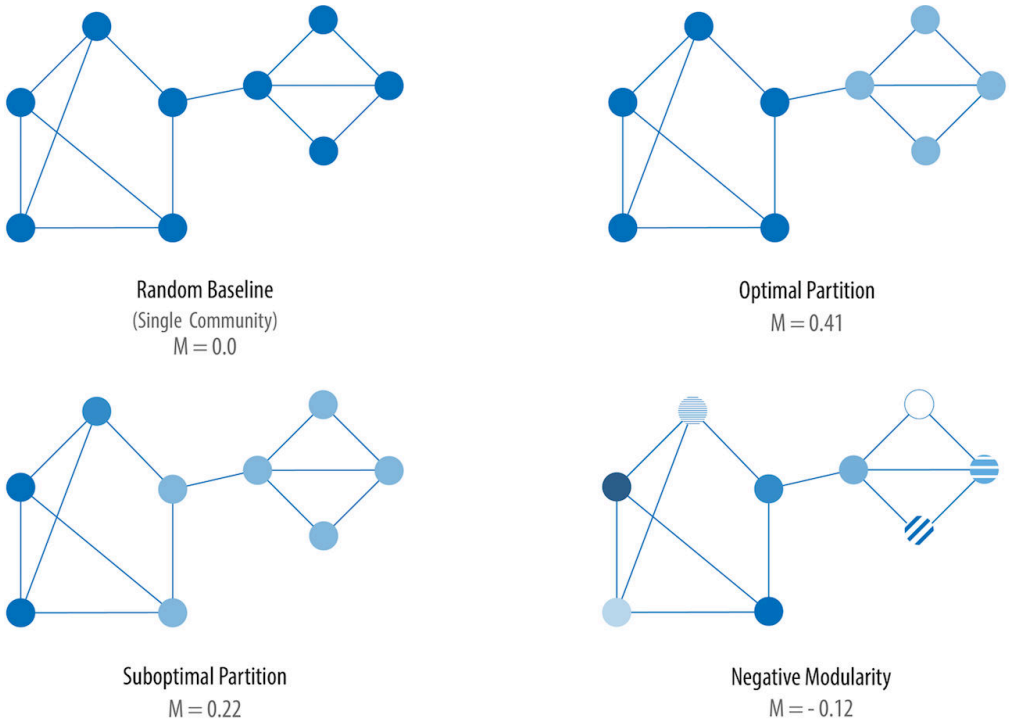


图6-11 基于不同分区选择的四个模块化得分

分组的模块化公式为：

$$M = \sum_{c=1}^{n_c} \left[\frac{L_c}{L} - \left(\frac{k_c}{2L} \right)^2 \right]$$

在这个公式中

- L是整个组(group)中的关系数。
- Lc是一个分区（partition）中的关系数。
- Kc是一个分区中节点的总度数。

图6-11的顶部的最佳分区（Optimal Partition）计算如下：

- 暗分区是7/13-(15/(2*13))^2 = 0.205

- 亮分区是5/13-(11/(2*13))^2 = 0.206
- 加在一起M = 0.205 + 0206 = 0.41

最初，Louvani模块化算法在所有节点上局部优化模块化，找到小的社区；然后将每个小社区被合并到一个较大的联合节点，并重复第一步，直到达到全局最优。

该算法包括两个步骤的重复应用，如图6-12所示。

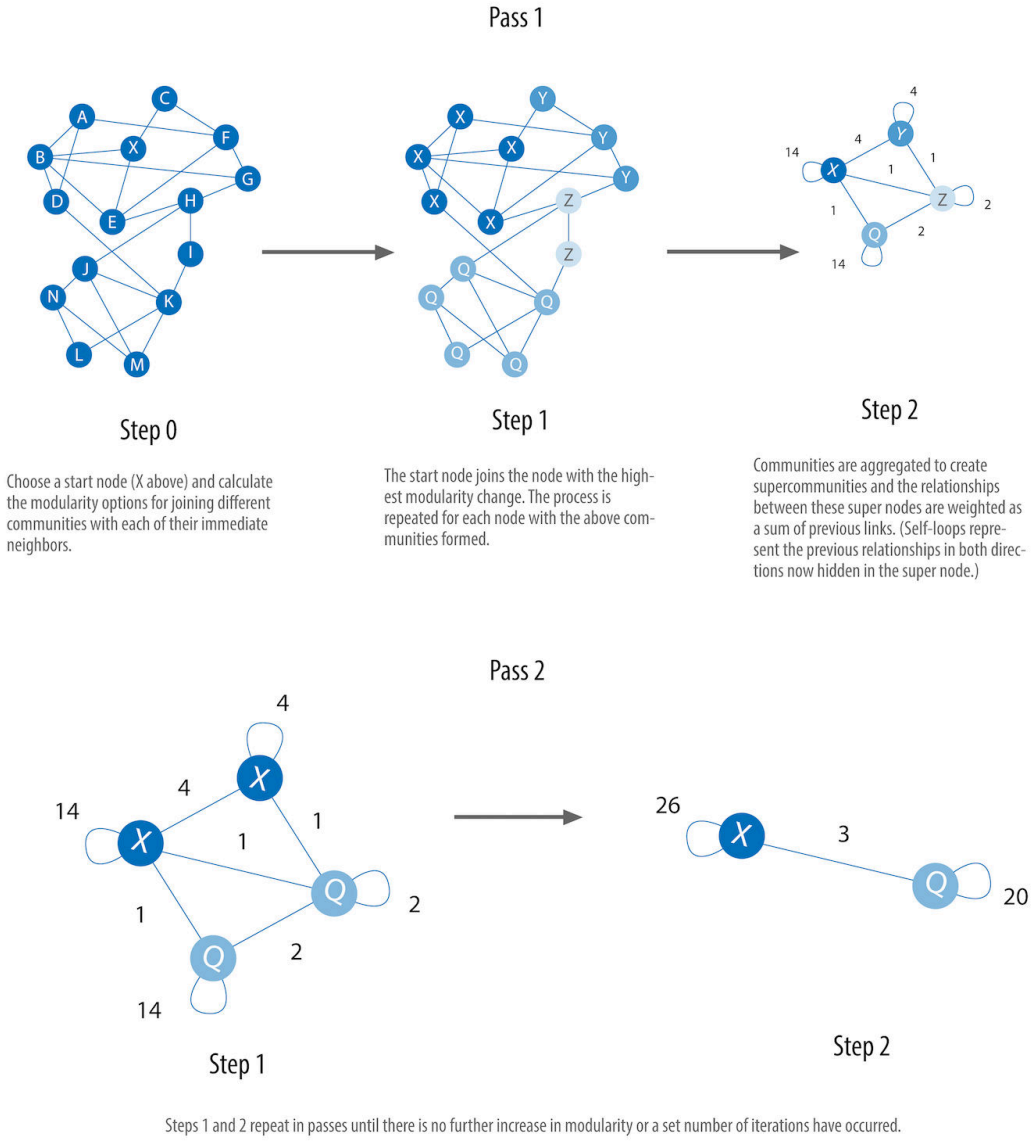


图6-12。Louvain算法过程

Louvain算法的步骤包括：

1) 将节点“贪婪”地分配给社区，有利于模块化的局部优化。

2) 基于第一步中发现的社区定义更粗粒度的网络。这种粗粒度的网络将在算法的下一次迭代中使用。

重复这两个步骤，直到无法进一步模块化，也就是无法进一步进行增加社区的重新分配为止。

Louvain模块化是模块化的优化算法。第一步的优化步骤是评估组的模块性。Louvain使用以下公式来实现这一点：

$$Q = \frac{1}{2m}\Sigma_{u,v}\left[A_{uv} - \frac{k_u k_v}{2m}\right]\delta(c_u, c_v)$$

在这个公式里

- U和V是节点。
- m是整个图的总关系权重（在模块化公式中2m是常见的归一化的值）。
- $A_{u,v} - \frac{k_u k_v}{2m}$ 是u和v之间关系的强度和网络中随机分配的结果的相比的结果
  - ✓ Auv是u和v之间的关系权重
  - ✓ Ku是u的所有关系权重的总和
  - ✓ Kv是v的所有关系权重的总和
- δ (cu, cv) ：当u和v被分配到同一社区时为1，当它们处于不同社区时为0。

第一步的另一部分评估了如果将一个节点移动到另一个组中，模块性的变化。Louvain使用了这个公式更复杂的变体，最后确定了最佳的组分配。

Louvain模块化的使用场景

使用Louvain模块化在庞大的网络中查找社区。该算法采用启发式算法，而不是昂贵而精确的模块化算法。因此，当标准的模块化算法在这些图上可能会遇到困难的时候，Louvain可以被派上用场。

Louvain对于评估复杂网络的结构也非常有帮助，尤其是发现许多层次结构——就像你可能在犯罪组织中发现的层级结构一样。该算法可以提供结果，您可以在其中放大不同级别的粒度，并在子社区中查找子社区中的子社区。

示例用例包括：

- ✓ 检测网络攻击。Louvain算法被S. V. Shanbhaq用于2016年的一项研究，该研究是关于大规模网络安全网络中快速社区检测的应用。一旦这些社区被发现，它们就可以用来检测网络攻击。
- ✓ 从Twitter和YouTube等在线社交平台中提取主题，基于文档中共同出现的术语，作为主题建模过程的一部分。这一方法在G. S. Kido、R. A. Igawa和S. Barbon Jr.， “Topic Modeling Based on Louvain Method in Online Social Networks” 一文中进行了描述。
- ✓ 如D. Meunier等人的 “Hierarchical Modularity in Human Brain Functional Networks” 所述，在大脑功能网络中找到层次化的社区结构。

模块化优化算法（包括Louvain）面临两个问题。首先，这些算法可以忽略大型网络中的小型社区。您可以通过评估中间合并步骤来克服这个问题。其次，在具有重叠社区的大型图中，模块化优化器可能无法正确地确定全局最大值。在后一种情况下，我们建议使用任意的模块化算法作为粗略估计的指导，但不完全准确。

在Neo4j上使用Louvain模块化

让我们看一下Louvain模块化如何应用。我们可以在图中执行如下的查询。

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
RETURN algo.getNodeById(nodeId).id AS libraries, communities
```

查询中传入的参数如下：

参数	意义
Library	从图中装载的节点的标签
DEPENDS_ON	从图中装载的关系的类型

如下是运行的结果(Neo4j Louvain失败)

```
+------------------+--------------+
| libraries        | communities  |
```

```
+------------------+--------------+
| pytz             | [0, 0]       |
| pyspark          | [1, 1]       |
| matplotlib       | [2, 0]       |
| spacy            | [2, 0]       |
| py4j             | [1, 1]       |
| jupyter          | [3, 2]       |
| jpy-console      | [3, 2]       |
| nbconvert        | [4, 2]       |
| ipykernel        | [3, 2]       |
| jpy-client       | [4, 2]       |
| jpy-core         | [4, 2]       |
| six              | [2, 0]       |
| pandas           | [0, 0]       |
| numpy            | [2, 0]       |
| python-dateutil  | [2, 0]       |
+------------------+--------------+
```

“社区”列描述节点分为两个级别的社区。数组中的最后一个值是最终社区，另一个是中间社区。

分配给中间社区和最终社区的数字只是标签，没有可测量的意义。将它们视为指示哪些社区节点属于的标签，例如“属于标记为0的社区”、“标记为4的社区”等等。

例如，matplotlib的结果是[2,0]。这意味着Matplotlib的最终社区标记为0，中间社区标记为2。

如果我们使用该算法的写入版本存储这些社区，然后再对其进行查询，那么更容易看到这是如何工作的。以下查询将运行Louvain算法，并将结果存储在每个节点的communities属性中：

```
CALL algo.louvain("Library", "DEPENDS_ON")
```

我们还可以使用该算法的流版本存储结果社区，然后调用set子句来存储结果。下面的查询显示了我们如何执行此操作：

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
WITH algo.getNodeById(nodeId) AS node, communities
SET node.communities = communities
```

一旦我们运行以上任何的查询，我们可以用如下的查询来找到最终的聚类:

```
MATCH (l:Library)
RETURN l.communities[-1] AS community, collect(l.id) AS libraries
```

```
ORDER BY size(libraries) DESC
```

l.communities[-1] 从存储的数组中返回最后一个元素。运行查询会得到如下的输出：

```
+------------+-----------------------------------------------------------------------+
| community  | libraries                                                             |
+------------+-----------------------------------------------------------------------+
| 0          | [pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]        |
| 2          | [jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]    |
| 1          | [pyspark, py4j]                                                       |
+------------+-----------------------------------------------------------------------+
```

这个聚类和我们在连接组件算法中看到的一样。matplotlib与pytz、spacy、six、panda、numpy和pythondateutil在一个社区中。我们可以在图6-13中更清楚地看到这一点。

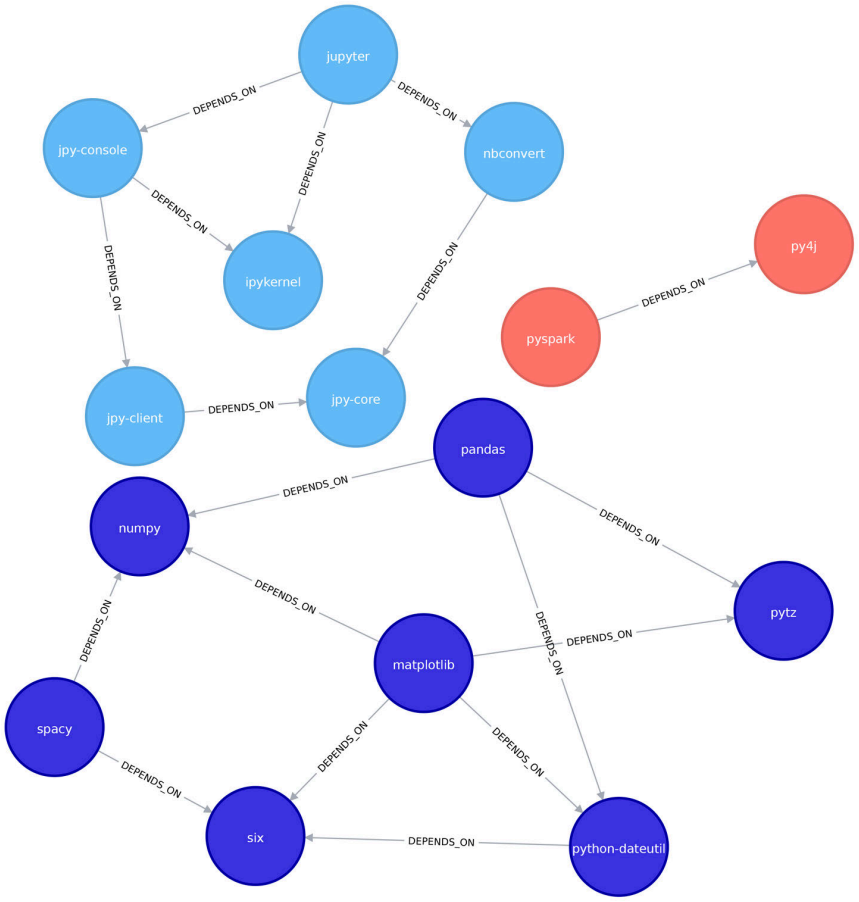


图6-13。Louvain算法发现的簇

Louvain算法的另一个特点是我们也可以看到中间聚类。这将向我们展示比最终层更细粒度的集群：

```
MATCH (l:Library)
RETURN l.communities[0] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

运行该查询可以得到如下的结果：

```
+----------+-------------------------------------------+
| community| libraries                                 |
+----------+-------------------------------------------+
| 2        | [matplotlib, spacy, six, python-dateutil] |
| 4        | [nbconvert, jpy-client, jpy-core]         |
| 3        | [jupyter, jpy-console, ipykernel]         |
| 1        | [pyspark, py4j]                           |
| 0        | [pytz, pandas]                            |
| 5        | [numpy]                                   |
+----------+-------------------------------------------+
```

matplotlib社区被分割成为三个小社区：

- matplotlib, spacy, six, and python-dateutil
- pytz和pandas
- numpy

我们可以在图6-14中看到这个分解。

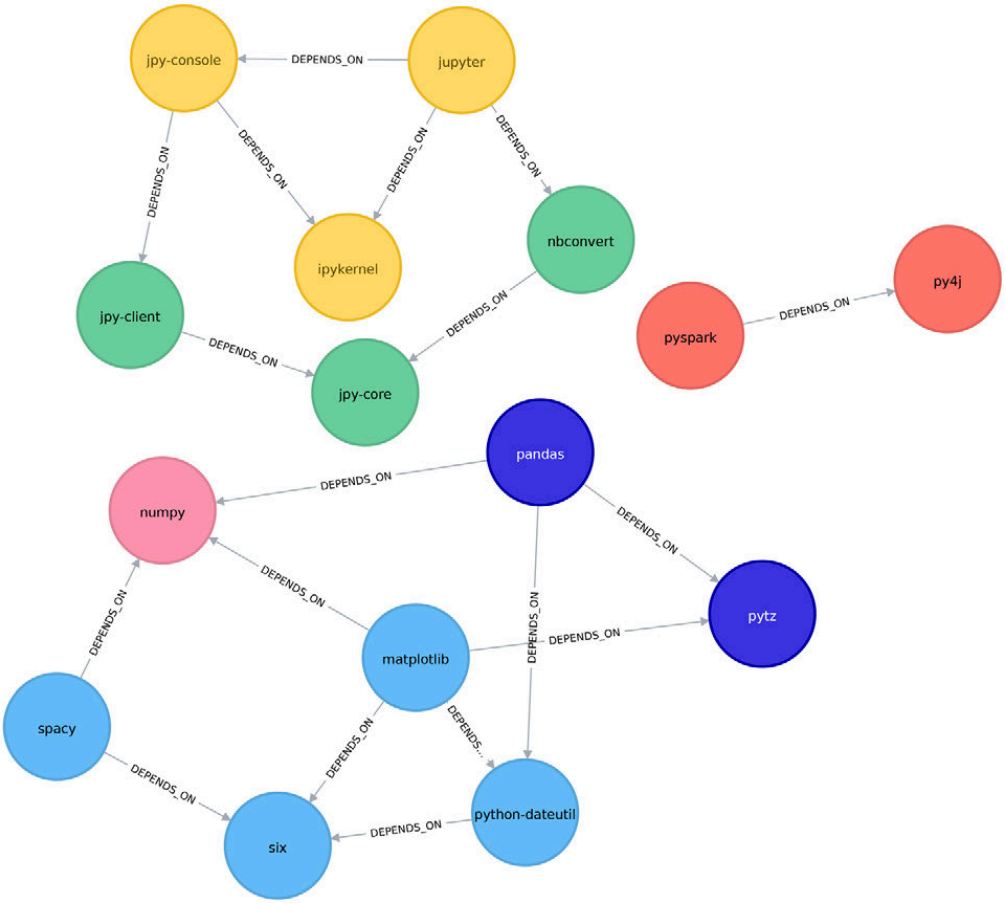


图6-14.Louvain算法发现的中间聚类

虽然这个图只显示了两层层次结构，但是如果我们在一个更大的图上运行这个算法，我们会看到一个更复杂的层次结构。Louvain揭示的中间集群对于检测可能无法被其他社区检测算法检测到的细粒度社区非常有用。

验证社区

社区检测算法通常有相同的目标：识别群体。然而，由于不同的算法从不同的假设开始，它们可能会发现不同的社区。这使得为一个特定的问题选择正确的算法变得更具挑战性，并且有点探索性。

与周围环境相比，当群体内的关系密度较高时，大多数社区检测算法都表现得相当好，但现实世界中的网络往往不那么明显。通过将我们的结果与基于已知社区数据的基准进行比较，我们可以验证所发现社区的准确性。

其中两个最著名的基准是Girvan-Newman (GN) 和Lancichinetti–Fortunato–Radicchi (LFR) 算法。这些算法生成的参考网络是完全不同的：GN生成一个更为均匀的随机网络，而LFR创建一个更为异构的图，其中节点度和社区大小根据幂律分布。

由于我们测试的准确性取决于所使用的基准，因此将我们的基准与数据集匹配是很重要的。尽可能多地寻找相似的密度、关系分布、社区定义和相关领域。

总结

社区检测算法对于理解节点在图中的分组方式很有用。

在本章中，我们首先学习三角形计数和聚类系数算法。然后我们继续讨论两种确定性社区检测算法：强连接组件和连接组件。这些算法对社区的构成有严格的定义，对于在图分析管道的早期了解图结构非常有用。

我们完成了标签传播和Louvain模块化，这两个非确定性算法，它们能够更好地检测更细粒度的社区。Louvain还向我们展示了不同等级的社区。

在下一章中，我们将学习一个更大的数据集，并学习如何将这些算法组合在一起，以获得对所连接数据的更深入了解。

第七章 图算法实践

随着我们越来越熟悉特定数据集上不同算法的行为，我们对图分析采用的方法也在不断发展。在本章中，我们将通过几个示例来帮助你更好地了解如何使用Yelp和美国运输部的数据集来处理大规模图数据分析。我们将在Neo4j中进行Yelp数据分析，其中包括数据的一般概述，组合算法以制定旅行建议，以及挖掘用户和业务数据以进行咨询。在Spark中，我们将查看美国航空公司数据，以了解不同航空公司的交通模式和延误以及机场如何连接。

由于寻路算法很简单，我们的示例将主要使用这些中心和社区检测算法：

- 用网页排名（PageRank）算法找到有影响力的Yelp评论员，然后关联他们对特定酒店的评分
- 用中介中心性(Betweenness Centrality)算法发现连接到多个组的审阅者，然后提取他们的偏好
- 用带有投影的标签传播（Label Propagation）算法来创建类似Yelp业务的超类别
- 用度中心性(Degree Centrality)算法快速识别美国运输数据集中的机场枢纽
- 用强连接组件（Strong Connected Component）查看美国机场路线集群

用Neo4j分析Yelp数据

Yelp帮助人们根据评论、偏好和建议找到当地企业。截至2018年底，已有超过1.8亿条评论发表在平台上。自2013年以来，Yelp开展了Yelp数据集挑战赛，这项竞赛鼓励人们探索和研究Yelp的开放数据集。

截至挑战的第12轮（2018年进行），开放数据集包括：

- 超过700万条评论和贴士
- 超过150万用户和28万张图片
- 超过188,000家企业拥有140万个属性
- 10个大都市地区

自从发布以来，这个数据集已经变得流行起来，数百篇学术论文都是用这种材料写的。Yelp数据集表示结构良好且高度互联的真实数据。这是一个很好的图算法展示，你也可以下载和探索。

Yelp社交网络

除了撰写和阅读商业评论，Yelp的用户还形成了一个社交网络。用户可以向浏览yelp.com时遇到的其他用户发送好友请求，也可以连接他们的通讯录或Facebook图结构。

Yelp数据集还包括一个社交网络。图7-1是Mark Yelp简介的Friends部分的屏幕截图。

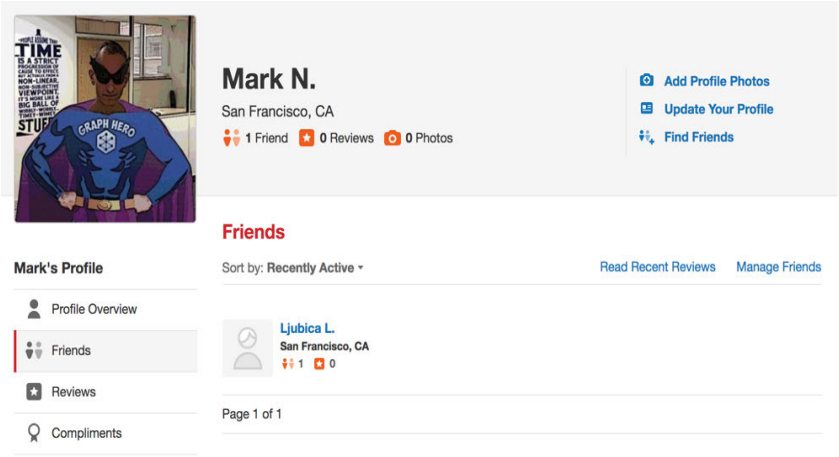


图7-1.Mark的Yelp截图

很显然Mark需要更多的朋友，但是我们的重点是我们已经准备好开始了。为了说明我们如何分析Neo4j中的Yelp数据，我们将使用我们为旅游信息业务工作的场景。我们将首先研究Yelp数据，然后看看如何帮助人们使用我们的应用程序来计划旅行。我们将在Las Vegas等主要城市寻找好的住宿地点和工作建议。

我们业务场景的另一部分将涉及到旅游目的地业务的咨询。在一个例子中，我们将帮助酒店识别有影响力的访客，然后确定他们应该以交叉促销计划为目标的业务。

数据导入

有许多不同的方法可以将数据导入Neo4j，包括导入工具，我们在前面章节中看到的加载csv命令，以及neo4j驱动程序。

对于Yelp数据集，我们需要一次性导入大量数据，因此导入工具是最佳选择。更多详情请参见附录的 “Neo4j批量数据导入和Yelp” 。

图模型

Yelp数据以图模型表示，如图7-2所示。

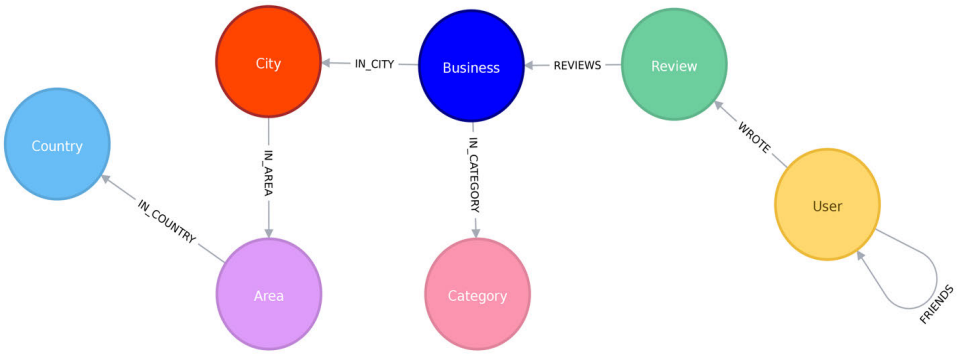


图7-2.Yelp图模型

我们的图包含标签为User的节点，这些节点与其他用户有FRIENDS关系。用户还可以撰写关于Business的Reviews和提示。所有元数据都存储为节点的属性，而业务类别除外。业务类别由单独的Category节点表示。对于位置数据，我们已经将City、Area和Country属性提取到子图中。在其他用例中，将其他属性提取到节点（如日期）或将节点折叠到关系（如审阅）可能是有意义的。

Yelp数据集还包括用户提示和照片，但我们不会在示例中使用这些提示和照片。

Yelp数据的快速概述

一旦我们在Neo4j中加载了数据，我们就会执行一些探索性查询。 我们将询问每个类别中有多少节点或存在哪种类型的关系，以了解Yelp数据。 以前我们已经为我们的Neo4j示例展示了Cypher查询，但我们可能正在从另一种编程语言执行这些查询。由于Python是数据科学家的首选语言，当我们想要将结果连接到

Python生态系统中的其他库时，我们将在本节中使用Neo4j的Python驱动程序。如果我们只想显示查询结果，我们将直接使用Cypher。

我们还将展示如何将Neo4j与流行的pandas库相结合，这对于数据库之外的数据争论是有效的。我们将看到如何使用制表库来美化我们从pandas那里得到的结果，以及如何使用matplotlib创建数据的可视化表示。

我们还将使用Neo4j的APOC程序库来帮助我们编写更强大的Cypher查询。在附录的“APOC和其他Neo4j工具”中有关于APOC的更多信息。

让我们首先安装Python库：

```
pip install neo4j-driver tabulate pandas matplotlib
```

完成后我们将导入这些库：

```
from neo4j.v1 import GraphDatabase
import pandas as pd
from tabulate import tabulate
```

在macOS上导入matplotlib可能很繁琐，但以下几行应该可以解决问题：

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

如果你在另一个操作系统上运行，则可能不需要中间的那一行。 现在让我们创建一个指向本地Neo4j数据库的Neo4j驱动程序的实例：

```
driver = GraphDatabase.driver("bolt://localhost", auth=("neo4j", "neo"))
```

你需要用你自己的主机和凭据来修改上面这一行。

首先，让我们看一些节点和关系的一般数字。 以下代码计算数据库中节点标签的基数（即，计算每个标签的节点数）：

```
result = {"label": [], "count": []}
with driver.session() as session:
    labels = [row["label"] for row in session.run("CALL db.labels()")]
```

```
        for label in labels:
            query = f"MATCH (:`{label}`) RETURN count(*) as count"
            count = session.run(query).single()["count"]
            result["label"].append(label)
            result["count"].append(count)
df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',tablefmt='psql', showindex=False))
```

如果运行该代码，我们将看到每个标签有多少个节点：

```
+----------+---------+
| label    | count   |
+----------+---------+
| Country  | 17      |
| Area     | 54      |
| City     | 1093    |
| Category | 1293    |
| Business | 174567  |
| User     | 1326101 |
| Review   | 5261669 |
+----------+---------+
```

我们还可以使用以下代码创建基数的可视化表示：

```
plt.style.use('fivethirtyeight')
ax = df.plot(kind='bar', x='label', y='count', legend=None)
ax.xaxis.set_label_text("")
plt.yscale("log")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

我们可以在图7-3中看到这个代码生成的图表。请注意，此图表使用的是对数刻度。

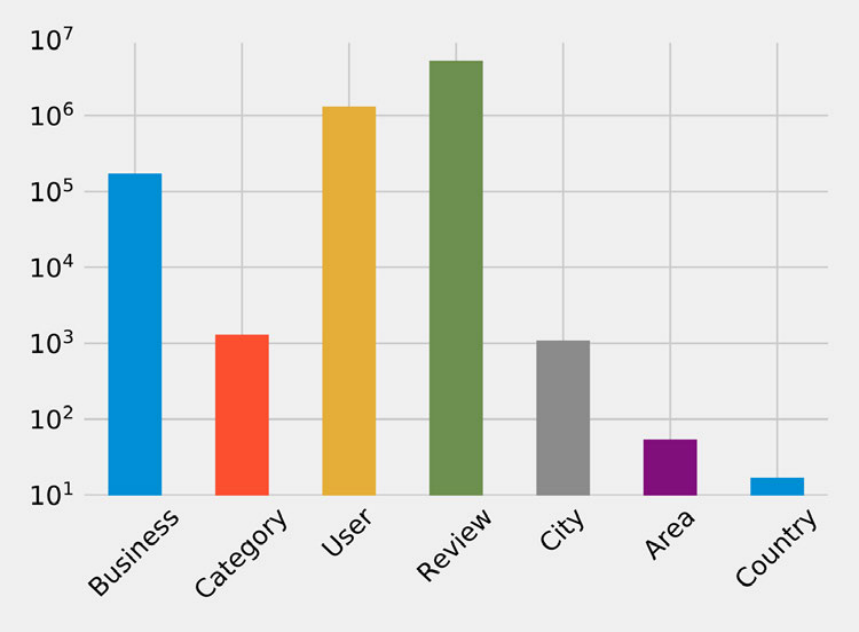


图7-3.每个标签类别的节点数

同样，我们可以计算关系的基数：

```
result = {"relType": [], "count": []}
with driver.session() as session:
    rel_types = [row["relationshipType"] for row in session.run ("CALL db.relationshipTypes()")]
    for rel_type in rel_types:
        query = f"MATCH ()-[:`{rel_type}`]->() RETURN count(*) as count"
        count = session.run(query).single()["count"]
        result["relType"].append(rel_type)
        result["count"].append(count)
df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',tablefmt='psql', showindex=False))
```

如果运行该代码，我们将看到每种关系类型的数量：

```
+-------------+----------+
| relType     | count    |
+-------------+----------+
| IN_COUNTRY  | 54       |
| IN_AREA     | 1154     |
| IN_CITY     | 174566   |
| IN_CATEGORY | 667527   |
| WROTE       | 5261669  |
| REVIEWS     | 5261669  |
| FRIENDS     | 10645356 |
+-------------+----------+
```

我们可以看到图7-4中的基数图表。与节点基数图表一样，此图表使用的是对数比例。

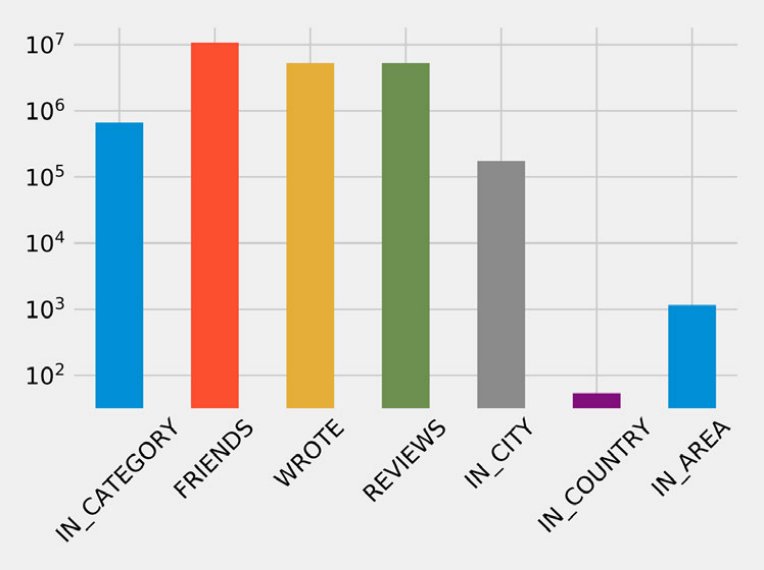


图7-4.按关系类型划分分组计算出来的关系数

这些查询不应该显示任何令人惊讶的信息，但是它们对于了解数据中的内容很有用。这还可以快速检查数据是否正确导入。

我们假设Yelp有很多酒店评论，但是在我们关注这个行业之前检查是有意义的。通过运行以下查询，我们可以了解该数据中有多少酒店企业以及它们有多少评论：

```
MATCH (category:Category {name: "Hotels"})
RETURN size((category)<-[:IN_CATEGORY]-()) AS businesses,size((:Review)-[:REVIEWS]->(:Business)-[:IN_CATEGORY]->
(category)) AS reviews
```

结果如下：

```
+------------+---------+
| businesses | reviews |
+------------+---------+
| 2683       | 183759  |
+------------+---------+
```

我们有很多业务要做，以为有好多的评论！在下一节中，我们将进一步探讨我们的业务场景中的数据。

旅行规划应用

为了在我们的应用程序中添加受欢迎的推荐，我们首先找到最受欢迎的酒店作为预订热门选择的启发式方法。我们可以补充一下，他们是如何被评价了解实际经验的。为了查看10家最受关注的酒店并绘制其评级分布，我们使用以下代码：

```
    # Find the 10 hotels with the most reviews
    query = """
    MATCH (review:Review)-[:REVIEWS]->(business:Business),
          (business)-[:IN_CATEGORY]->(category:Category {name: $category}),
          (business)-[:IN_CITY]->(:City {name: $city})
    RETURN business.name AS business, collect(review.stars) AS allReviews
    ORDER BY size(allReviews) DESC
    LIMIT 10
    “””

    fig = plt.figure()
    fig.set_size_inches(10.5, 14.5)
    fig.subplots_adjust(hspace=0.4, wspace=0.4)

    with driver.session() as session:
        params = { "city": "Las Vegas", "category": "Hotels”}
        result = session.run(query, params)
        for index, row in enumerate(result):
            business = row["business"]
            stars = pd.Series(row["allReviews”])

            total = stars.count()
            average_stars = stars.mean().round(2)

            # Calculate the star distribution
            stars_histogram = stars.value_counts().sort_index()
            stars_histogram /= float(stars_histogram.sum())

            # Plot a bar chart showing the distribution of star ratings
            ax = fig.add_subplot(5, 2, index+1)
            stars_histogram.plot(kind="bar", legend=None, color="darkblue",title=f"{business}\nAve: {average_stars},
Total:
{total}")
    plt.tight_layout()
    plt.show()
```

我们受城市和类别的限制，只能专注于Las Vegas的酒店。我们运行代码，得到图7-5中的图表。请注意，X轴代表酒店的星级，Y轴代表每个等级的总百分比。

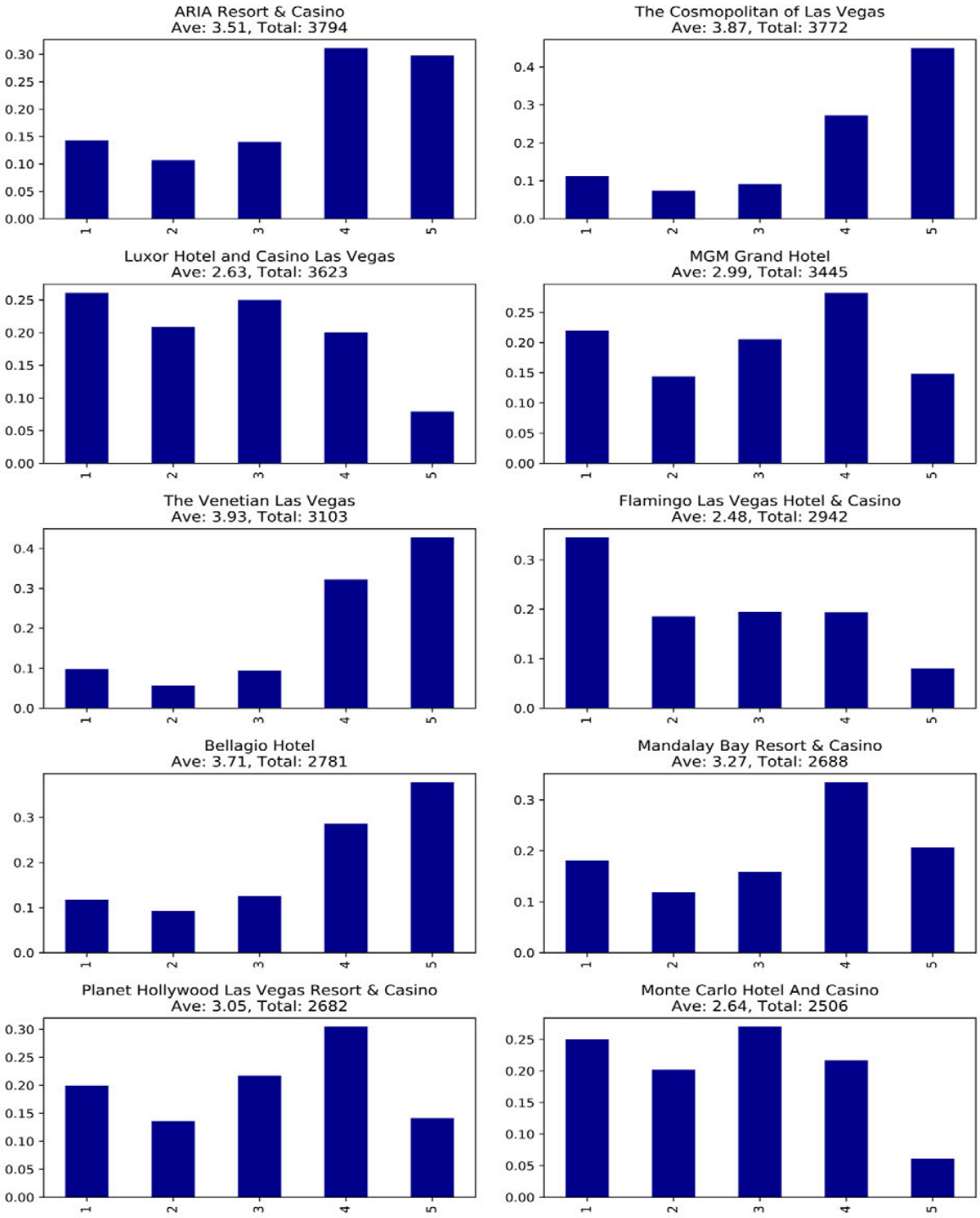


图7-5.最受欢迎的10家酒店，X轴上的星级数和Y轴上的总体评分百分比

这些酒店有很多评论，远远超过任何人可能读到的。最好是向我们的用户展示最相关评论的内容，并使它们在我们的应用程序中更加突出。为了进行这个分析，我们将从基本的图探索转向使用图算法。

寻找有影响力的酒店评论

我们如何决定发布哪些评论呢？其中的一种方法是根据评论人对Yelp的影响来排序评论。我们将运行PageRank算法，对所有浏览过至少三家酒店的用户的投影图进行搜索。请记住，在前面的章节中，投影可以帮助过滤不必要的信息，并添加关系数据（有时是推断出来的关系）。我们将使用Yelp的朋友关系图作为用户之间的关系。PageRank算法将发现那些对更多用户有更大影响力的评论者，即使评论者和用户之间不是直接的朋友。

如果两个人是朋友，他们之间有两种FRIENDS关系。例如，如果A和B是朋友，那么A和B之间会有FRIENDS关系，B和A之间会有FRIENDS关系。

我们需要编写一个查询，用三个以上的评论来投影用户的子图，然后在投影的子图上执行pagerank算法。

用一个小例子更容易理解子图投影是如何工作的。图7-6显示了三个共同的朋友Mark、Arya和Pravena的关系图。Mark和Pravena都对三家酒店进行了审查，并将成为投影图的一部分。另一方面，Arya只审查了一家酒店，因此将被排除在预测之外。

图7-6.一个Yelp图的采样子图

我们的投影图只包括Mark和Pravena，如图7-7所示。

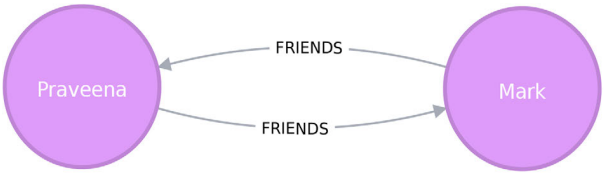


图7-7.我们的示例投影图

既然我们已经了解了图投影是如何工作的，那么让我们继续前进。下面的查询对投影图执行pagerank算法，并将结果存储在每个节点的hotelPageRank属性中：

```
    CALL algo.pageRank(
   'MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->(:Category {name: $category})
    WITH u, count(*) AS reviews
    WHERE reviews >= $cutOff
    RETURN id(u) AS id',
   'MATCH (u1:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->(:Category {name: $category})
    MATCH (u1)-[:FRIENDS]->(u2)
    RETURN id(u1) AS source, id(u2) AS target',{graph: "cypher", write: true, writeProperty: "hotelPageRank",
    params: {category: "Hotels", cutOff: 3}}
)
```

你可能已经注意到，我们没有设置第5章中讨论的阻尼因子或最大迭代限制。如果没有明确设置，Neo4j默认为0.85阻尼系数，maxIterations设置为20。

现在让我们看看pagerank值的分布，这样我们就知道如何过滤数据：

```
MATCH (u:User)
WHERE exists(u.hotelPageRank)
RETURN count(u.hotelPageRank) AS count,
    avg(u.hotelPageRank) AS ave,
    percentileDisc(u.hotelPageRank, 0.5) AS `50%`,
    percentileDisc(u.hotelPageRank, 0.75) AS `75%`,
    percentileDisc(u.hotelPageRank, 0.90) AS `90%`,
    percentileDisc(u.hotelPageRank, 0.95) AS `95%`,
    percentileDisc(u.hotelPageRank, 0.99) AS `99%`,
    percentileDisc(u.hotelPageRank, 0.999) AS `99.9%`,
    percentileDisc(u.hotelPageRank, 0.9999) AS `99.99%`,
    percentileDisc(u.hotelPageRank, 0.99999) AS `99.999%`,
    percentileDisc(u.hotelPageRank, 1) AS `100%`
```

如果我们运行这个查询，我们将得到这个输出：

```
+---------+-----------+------+------+----------+----------+----------+----------+----------+----------+----------+
| count   | ave       | 50%  | 75%  | 90%      | 95%      | 99%      | 99.9%    | 99.99%   | 99.999%  | 100%     |
+---------+-----------+------+------+----------+----------+----------+----------+----------+----------+----------+
| 1326101 | 0.1614898 | 0.15 | 0.15 | 0.157497 | 0.181875 | 0.330081 | 1.649511 | 6.825738 | 15.27376 | 22.98046 |
+---------+-----------+------+------+----------+----------+----------+----------+----------+----------+----------+
```

为了解释这个百分比表，90%的值0.157497意味着90%的用户的pagerank得分比这个得分更低。99.99%反映了前0.0001%评审员的影响等级，100%仅仅表示的是最高的pagerank分数。

有趣的是，我们90%的用户的得分低于0.16，接近总体平均值，仅略高于通过pagerank算法初始化的0.15。这一数据似乎反映了幂律法则分布，其中有几个非常有影响力的评论家。

因为我们只想找到最有影响力的用户，所以我们将编写一个查询，它只查找pagerank分数在所有用户中排名前0.001%的用户。接下来的查询将查找pagerank分数高于1.64951的审阅者（请注意，这是99.9%的组）：

```
// Only find users that have a hotelPageRank score in the top 0.001% of users
MATCH (u:User)
WHERE u.hotelPageRank > 1.64951
// Find the top 10 of those users
WITH u ORDER BY u.hotelPageRank DESC
LIMIT 10
RETURN u.name AS name,
    u.hotelPageRank AS pageRank,
    size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]-> (:Category {name: "Hotels"})) AS hotelReviews,
    size((u)-[:WROTE]->()) AS totalReviews, size((u)-[:FRIENDS]-()) AS friends
```

如果运行该查询，我们将在此处看到结果：

```
+---------+--------------------+--------------+--------------+---------+
```

name	pageRank	hotelReviews	totalReviews	friends
Phil	17.361242	15	134	8154
Philip	16.871013	21	620	9634
Carol	12.416060999999997	6	119	6218
Misti	12.239516000000004	19	730	6230
Joseph	12.003887499999998	5	32	6596
Michael	11.460049	13	51	6572
J	11.431505999999997	103	1322	6498
Abby	11.376136999999998	9	82	7922
Erica	10.993773	6	15	7071
Randy	10.748785999999999	21	125	7846

这些结果表明Phil是最可信的评论人，尽管他没有评论过很多酒店。他可能和一些很有影响力的人有关系，但如果我们想要一系列新的评论，他的个人资料就不是最好的选择。Philip的分数稍低，但朋友最多，写评论的次数比Phil多五倍。虽然J写的评论最多，而且有相当数量的朋友，但J的pagerank分数并不是最高的，但仍在前10名。对于我们的应用程序，我们选择突出显示来自Phil、Philip和J的酒店评论，为我们提供适当的影响者和评论数量组合。

既然我们已经通过相关的评论改进了我们的应用内建议，那么让我们转到业务的另一个方面：咨询。

旅游商务咨询

作为我们咨询服务的一部分，当有影响力的客人写下他们的住宿情况时，酒店会收到通知，以便他们采取任何必要的行动。首先，我们来看看Bellagio的收视率，由最有影响力的评论家排序：

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank)
RETURN user.name AS name,
user.hotelPageRank AS pageRank,review.stars AS stars
"""
with driver.session() as session:
    params = { "hotel": "Bellagio Hotel" }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)]) df = df.round(2)
    df = df[["name", "pageRank", "stars"]]
top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

如果我们运行这个代码，我们将得到这个输出：

```
+-----------+--------------------+-------+
| name      | pageRank           | stars |
+-----------+--------------------+-------+
| Misti     | 12.239516000000004 | 5     |
| Michael   | 11.460049          | 4     |
| J         | 11.431505999999997 | 5     |
| Erica     | 10.993773          | 4     |
| Christine | 10.740770499999998 | 4     |
| Jeremy    | 9.576763499999998  | 5     |
| Connie    | 9.118103499999998  | 5     |
| Joyce     | 7.621449000000001  | 4     |
| Henry     | 7.299146           | 5     |
| Flora     | 6.7570075          | 4     |
+-----------+--------------------+-------+
```

请注意，这些结果与我们之前的最佳酒店评估表不同。这是因为在这里，我们只关注那些评价Bellagio的评论家。

Bellagio酒店的客户服务团队情况良好。前10位有影响力的人都给出了酒店的良好排名。他们可能希望鼓励这些人再次访问并分享他们的经历。

有没有什么有影响力的客人体验很差？我们可以运行以下代码来查找页面排名最高的客人，他们的消费级别低于四星级：

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank) AND review.stars < $goodRating
RETURN user.name AS name,user.hotelPageRank AS pageRank,review.stars AS stars
"""
with driver.session() as session:
    params = { "hotel": "Bellagio Hotel", "goodRating": 4 }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)])
    df = df.round(2)
    df = df[["name", "pageRank", "stars”]]

top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

如果我们运行该代码，将得到以下结果：

```
+----------+----------+-------+
| name     | pageRank | stars |
+----------+----------+-------+
| Chris    | 5.84     | 3     |
| Lorrie   | 4.95     | 2     |
| Dani     | 3.47     | 1     |
| Victor   | 3.35     | 3     |
| Francine | 2.93     | 3     |
| Rex      | 2.79     | 2     |
| Jon      | 2.55     | 3     |
```

```
| Rachel   | 2.47     | 3     |
| Leslie   | 2.46     | 2     |
| Benay    | 2.46     | 3     |
+----------+----------+-------+
```

我们排名最高的Bellagio用户，Chris和Lorrie，是排名前1000位最有影响力的用户（根据我们之前的查询结果），所以可能有必要进行个人接触。此外，由于许多评论人在逗留期间都会写文章，因此实时提醒有影响力的人可能会促进更积极的互动。

Bellagio交叉推广

在我们帮助他们找到有影响力的评论员之后，Bellagio现在要求我们在关系良好的客户的帮助下，帮助确定其他企业进行交叉推广。在我们的场景中，我们建议他们通过吸引来自不同类型社区的新客人来增加客户群。我们可以使用之前讨论过的中介中心性算法来计算出哪些Bellagio评论员不仅在整个Yelp网络中有很好的联系，而且还可以作为不同组之间的桥梁。

我们只对在Las Vegas寻找有影响力的人感兴趣，所以我们首先要标记这些用户：

```
MATCH (u:User)
WHERE exists((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CITY]->(:City {name: "Las Vegas"}))

SET u:LasVegas
```

在我们的Las Vegas用户上运行中介中心性算法需要很长时间，因此我们将使用RA-Brandes变体。该算法通过采样节点计算中介性得分，并且仅探索到某个深度的最短路径。

经过一些实验，我们改进了一些参数设置不同于默认值的结果。 我们将使用最多4个跳点的最短路径（maxDepth=4）并对20％的节点进行采样（概率为0.2）。请注意，增加跳数和节点数通常会提高准确性，但需要花费更多时间来计算结果。 对于任何特定问题，最佳参数通常需要测试以识别收益递减点。

以下查询将执行算法并将结果存储在*between*属性中：

```
CALL algo.betweenness.sampled('LasVegas', 'FRIENDS',
      {write: true, writeProperty: "between", maxDepth: 4, probability: 0.2}
)
```

在我们在查询中使用这些分数之前，让我们编写一个快速的探索性查询来查看分数的分布方式：

```
MATCH (u:User)
WHERE exists(u.between)
RETURN count(u.between) AS count,
    avg(u.between) AS ave,
    toInteger(percentileDisc(u.between, 0.5)) AS `50%`,
    toInteger(percentileDisc(u.between, 0.75)) AS `75%`,
    toInteger(percentileDisc(u.between, 0.90)) AS `90%`,
    toInteger(percentileDisc(u.between, 0.95)) AS `95%`,
    toInteger(percentileDisc(u.between, 0.99)) AS `99%`,
    toInteger(percentileDisc(u.between, 0.999)) AS `99.9%`,
    toInteger(percentileDisc(u.between, 0.9999)) AS `99.99%`,
    toInteger(percentileDisc(u.between, 0.99999)) AS `99.999%`,
    toInteger(percentileDisc(u.between, 1)) AS p100
```

如果我们运行该代码，将得到以下结果：

```
+--------+-------------+-----+-------+--------+---------+---------+----------+-----------+-----------+------------+
| count  | ave         | 50% | 75%   | 90%    | 95%     | 99%     | 99.9%    | 99.99%    | 99.999%   | 100%       |
+--------+-------------+-----+-------+--------+---------+---------+----------+-----------+-----------+------------+
| 506028 | 320538.6014 | 0   | 10005 | 318944 | 1001655 | 4436409 | 34854988 | 214080923 | 621434012 | 1998032952 |
+--------+-------------+-----+-------+--------+---------+---------+----------+-----------+-----------+------------+
```

我们有一半的用户得分为0，这意味着他们根本没有很好的联系。前1个百分位数（99%列）位于我们500000个用户组之间至少400万条最短路径上。综上所述，我们知道我们的大多数用户连接不良，但有一些用户对信息施加了很大的控制；这是小世界网络的典型行为。

我们可以通过运行以下查询来了解我们的超级连接者是谁：

```
MATCH(u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"})
WHERE exists(u.between)
RETURN u.name AS user,toInteger(u.between) AS betweenness,u.hotelPageRank AS pageRank, size((u)-[:WROTE]->()-
[:REVIEWS]->()-[:IN_CATEGORY]->AS hotelReviews ORDER BY u.between DESC LIMIT 10
```

输出如下：

```
+-----------+-------------+--------------------+--------------+
| user      | betweenness | pageRank           | hotelReviews |
+-----------+-------------+--------------------+--------------+
| Misti     | 841707563   | 12.239516000000004 | 19           |
| Christine | 236269693   | 10.740770499999998 | 16           |
| Erica     | 235806844   | 10.993773          | 6            |
| Mike      | 215534452   | NULL               | 2            |
| J         | 192155233   | 11.431505999999997 | 103          |
| Michael   | 161335816   | 5.105143           | 31           |
| Jeremy    | 160312436   | 9.576763499999998  | 6            |
```

```
| Michael   | 139960910   | 11.460049          | 13           |
| Chris     | 136697785   | 5.838922499999999  | 5            |
| Connie    | 133372418   | 9.118103499999998  | 7            |
+-----------+-------------+--------------------+--------------+
```

我们在这里看到的一些人和之前在pagerank查询中看到的人一样，Mike是一个有趣的例外。他被排除在外，因为他没有审查足够的酒店（三家是底线），但似乎他在Las Vegas的Yelp世界中有相当好的联系。

为了接触更多的客户，我们将查看这些“连接者”显示的其他偏好，以了解我们应该推广什么。这些用户中的许多人也对餐馆进行了审查，因此我们编写以下查询，以找出他们最喜欢的餐馆：

```
// Find the top 50 users who have reviewed the Bellagio
MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"}) WHERE u.between > 4436409
WITH u ORDER BY u.between DESC LIMIT 50

// Find the restaurants those users have reviewed in Las Vegas
MATCH (u)-[:WROTE]->(review)-[:REVIEWS]-(business)
WHERE (business)-[:IN_CATEGORY]->(:Category {name: "Restaurants"}) AND (business)-[:IN_CITY]->(:City {name: "Las
Vegas"})

// Only include restaurants that have more than 3 reviews by these users
WITH business, avg(review.stars) AS averageReview, count(*) AS numberOfReviews WHERE numberOfReviews >= 3

RETURN business.name AS business, averageReview, numberOfReviews ORDER BY averageReview DESC, numberOfReviews DESC
LIMIT 10
```

这个查询找到了我们最具影响力的50个连接者，并找到了Las Vegas最具影响力的10个餐厅，其中至少有3个人评价过该餐厅。如果我们运行它，我们将看到这里显示的输出：

```
+----------------------------------+---------------+-----------------+
| business                         | averageReview | numberOfReviews |
+----------------------------------+---------------+-----------------+
| Jean Georges Steakhouse          | 5.0           | 6               |
| Sushi House Goyemon              | 5.0           | 6               |
| Art of_Flavors                   | 5.0           | 4               |
| é by José Andrés                 | 5.0           | 4               |
| Parma By Chef Marc               | 5.0           | 4               |
| Yonaka Modern Japanese           | 5.0           | 4               |
| Kabuto                           | 5.0           | 4               |
| Harvest by Roy Ellamar           | 5.0           | 3               |
| Portofino by Chef Michael LaPlaca | 5.0           | 3               |
| Montesano’s Eateria              | 5.0           | 3               |
+----------------------------------+---------------+-----------------+
```

我们现在可以建议Bellagio与这些餐厅联合举办一次促销活动，以吸引来自他们通常无法接触的群体的新客人。评价Bellagio的这些超级连接者，成为了我们评估哪些餐馆可能吸引新类型目标访客的代理。

既然我们已经帮助Bellagio接触到了新的群体，我们将了解如何使用社区检测来进一步改进我们的应用程序。

找到类似类别

当我们的最终用户使用该应用程序查找酒店时，我们希望展示他们可能感兴趣的其他业务。Yelp数据集包含1000多个类别，其中一些类别似乎彼此类似。我们将使用这种相似性为我们的用户可能会感兴趣的新业务提供应用内建议。

我们的图模型在类别之间没有任何关系，但是我们可以使用第二章的“单分图、二分图和K-分图”中描述的思想，根据企业如何对自己进行分类来构建一个类别相似性的图。

例如，假设只有一个企业将自己分类为酒店和历史旅游，如图7-8所示。

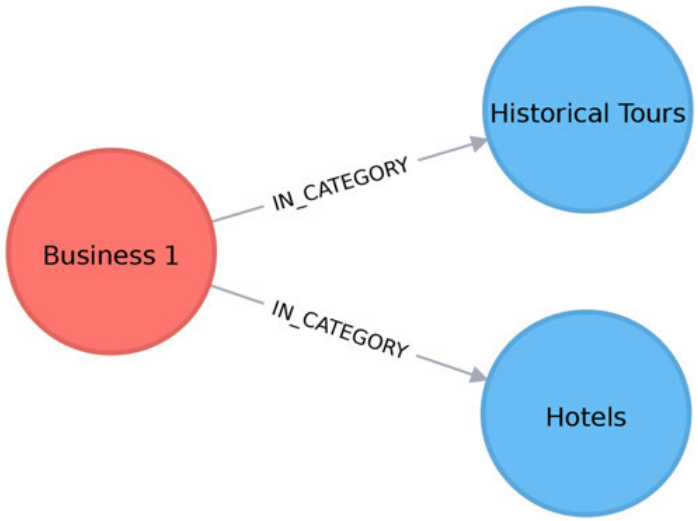


图7-8.有两个类别的企业

如图7-9所示，这将产生一个投影图，该图在酒店和历史旅游之间有一个权重为1的链接。

图7-9.投影类别图

在这种情况下，我们实际上不需要创建相似度图，而是可以运行社区检测算法，例如在投影的相似性图上进行标签传播。使用标签传播可以有效地将企业聚集在它们最为共同的超类别周围：

```
CALL algo.labelPropagation.stream(
      'MATCH (c:Category) RETURN id(c) AS id',
      'MATCH (c1:Category)<-[:IN_CATEGORY]-()-[:IN_CATEGORY]->(c2:Category)
       WHERE id(c1) < id(c2)
       RETURN id(c1) AS source, id(c2) AS target, count(*) AS weight',
      {graph: "cypher"}
)
YIELD nodeId, label
MATCH (c:Category) WHERE id(c) = nodeId
MERGE (sc:SuperCategory {name: "SuperCategory-" + label})
MERGE (c)-[:IN_SUPER_CATEGORY]->(sc)
```

让我们给这些超类别一个更友好的名称，用他们最大类别的名称：

```
MATCH (sc:SuperCategory)<-[:IN_SUPER_CATEGORY]-(category)
WITH sc, category, size((category)<-[:IN_CATEGORY]-()) as size ORDER BY size DESC
WITH sc, collect(category.name)[0] as biggestCategory
SET sc.friendlyName = "SuperCat " + biggestCategory
```

我们可以在图7-10中看到类别和超类别的示例。

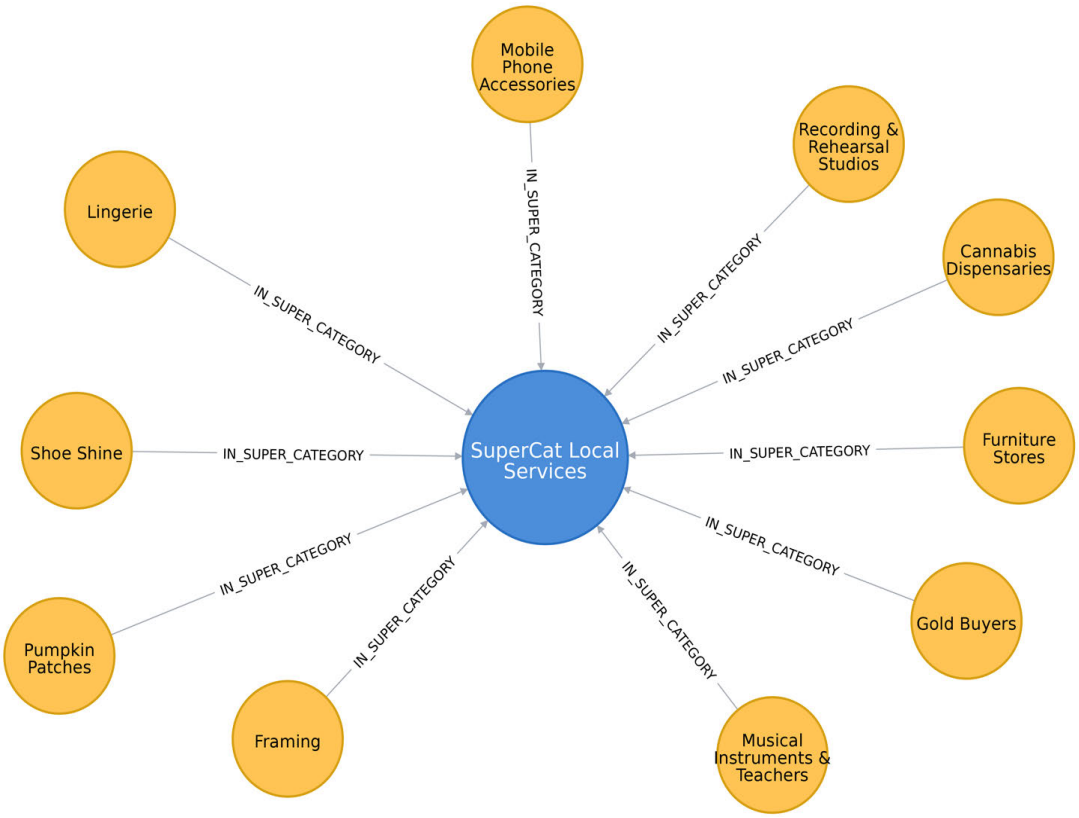


图7-10。类别和超类别

下面的查询查找与拉斯维加斯酒店最常见的类似类别：

```
MATCH (hotels:Category {name: "Hotels"}),
(lasVegas:City {name: "Las Vegas"}), (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-(otherCategory)

RETURN otherCategory.name AS otherCategory,size((otherCategory)<-[:IN_CATEGORY]-(:Business)- [:IN_CITY]->(lasVegas)) AS
businesses ORDER BY count DESC LIMIT 10
```

如果运行该查询，我们将看到以下输出：

```
+-------------------+------------+
| otherCategory     | businesses |
+-------------------+------------+
| Tours             | 189        |
| Car Rental        | 160        |
| Limos             | 84         |
| Resorts           | 73         |
| Airport Shuttles  | 52         |
| Taxis             | 35         |
| Vacation Rentals  | 29         |
| Airports          | 25         |
| Airlines          | 23         |
| Motorcycle Rental | 19         |
+-------------------+------------+
```

这些结果看起来很奇怪吗？显然，出租车和旅游不是酒店，但请记住，这是基于自我报告的分类。标签传播算法真正向我们展示的是这个相似性组中相邻的业务和服务。

现在，让我们在这些类别中找到一些评级高于平均水平的企业：

```
// Find businesses in Las Vegas that have the same SuperCategory as Hotels
MATCH (hotels:Category {name: "Hotels"}), (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-(otherCategory),
                                                                (otherCategory)<-[:IN_CATEGORY]-(business)
WHERE (business)-[:IN_CITY]->(:City {name: "Las Vegas"})

// Select 10 random categories and calculate the 90th percentile star rating
WITH otherCategory, count(*) AS count,
collect(business) AS businesses, percentileDisc(business.averageStars, 0.9) AS p90Stars
ORDER BY rand() DESC LIMIT 10

// Select businesses from each of those categories that have an average rating
// higher than the 90th percentile using a pattern comprehension
WITH otherCategory, [b in businesses where b.averageStars >= p90Stars] AS businesses

// Select one business per category
WITH otherCategory, businesses[toInteger(rand() * size(businesses))] AS business

RETURN otherCategory.name AS otherCategory, business.name AS business,
business.averageStars AS averageStars
```

在这个查询中，我们第一次使用模式理解（pattern comprehension）。模式理解是一种基于模式匹配创建列表的语法构造。它使用匹配子句和谓词的WHERE子句查找特定的模式，然后生成自定义投影。Cypher的这个功能是受GraphQL启发而来，GraphQL是一种用于API的查询语言。

如果运行该查询，将看到以下结果：

```
+----------------------+-----------------------------------+------------------+
| otherCategory        | business                          | averageStars     |
+----------------------+-----------------------------------+------------------+
| Motorcycle_Rental    | Adrenaline_Rush_Slingshot_Rentals | 5.0              |
| Snorkeling           | Sin_City_Scuba                    | 5.0              |
| Guest_Houses         | Hotel_Del_Kacvinsky               | 5.0              |
| Car_Rental           | The_Lead_Team                     | 5.0              |
| Food_Tours           | Taste_BUZZ_Food_Tours             | 5.0              |
| Airports             | Signature_Flight_Support          | 5.0              |
| Public_Transportation | JetSuiteX                        | 4.6875           |
| Ski_Resorts          | Trikke_Las_Vegas                  | 4.833333333333332 |
| Town_Car_Service     | MW_Travel_Vegas                   | 4.866666666666665 |
| Campgrounds          | McWilliams_Campground             | 3.875            |
+----------------------+-----------------------------------+------------------+
```

然后，我们可以根据用户的即时应用程序行为实时提出建议。例如，当用户在浏览Las Vegas的酒店时，我们现在可以标注出邻近的高分企业。我们可以将这些方法推广到任何地方的任何业务类别，如餐馆或剧院。

读者练习

- 你能否绘制出一个城市酒店的评论随时间的变化情况？
- 对于特定的酒店或其他业务怎么办？
- 是否有流行趋势（季节性或其他）？
- 最有影响力的评审员是否只与其他有影响力的评审员联系起来？

用Apache Spark分析航班数据

在本节中，我们将使用不同的场景来说明使用Spark对美国机场数据的分析。假设你是一个数据科学家，有一个相当大的旅行时间表，想深入了解航空公司航班和航班延误的信息。我们将首先研究机场和航班信息，然后深入研究两个特定机场的延误情况。社区检测将用于分析路线，并找到我们的常飞点的最佳利用。

美国交通统计局提供了大量的交通信息。为了进行分析，我们将使用他们2018年5月的航空旅行准时性能数据，其中包括当月在美国始发和结束的航班。为了添加更多关于机场的细节，例如位置信息，我们还将从一个单独的源openflights加载数据。

让我们把数据载入spark。与前几节中的情况一样，我们的数据以csv文件的形式存在，这些文件在图书的Github存储库中可用。

```
nodes = spark.read.csv("data/airports.csv", header=False)
cleaned_nodes = (nodes.select("_c1", "_c3", "_c4", "_c6", "_c7")
                      .filter("_c3 = 'United States'")
                      .withColumnRenamed("_c1", "name")
                      .withColumnRenamed("_c4", "id")s
                      .withColumnRenamed("_c6", "latitude")
                      .withColumnRenamed("_c7", "longitude")
                      .drop("_c3"))
cleaned_nodes = cleaned_nodes[cleaned_nodes["id"] != "\\N”]

relationships = spark.read.csv("data/188591317_T_ONTIME.csv", header=True)

cleaned_relationships = (relationships
                               .select("ORIGIN", "DEST", "FL_DATE", "DEP_DELAY",
                                       "ARR_DELAY", "DISTANCE", "TAIL_NUM", "FL_NUM",
                                        "CRS_DEP_TIME", "CRS_ARR_TIME",
                                        "UNIQUE_CARRIER")
```

```
                                .withColumnRenamed("ORIGIN", "src")
                                .withColumnRenamed("DEST", "dst")
                                .withColumnRenamed("DEP_DELAY", "deptDelay")
                                .withColumnRenamed("ARR_DELAY", "arrDelay")
                                .withColumnRenamed("TAIL_NUM", "tailNumber")
                                .withColumnRenamed("FL_NUM", "flightNumber")
                                .withColumnRenamed("FL_DATE", "date")
                                .withColumnRenamed("CRS_DEP_TIME", "time")
                                .withColumnRenamed("CRS_ARR_TIME", "arrivalTime")
                                .withColumnRenamed("DISTANCE", "distance")
                                .withColumnRenamed("UNIQUE_CARRIER", "airline")
                                .withColumn("deptDelay",
                                     F.col("deptDelay").cast(FloatType()))
                                .withColumn("arrDelay",
                                     F.col("arrDelay").cast(FloatType()))
                                .withColumn("time", F.col("time").cast(IntegerType()))
                                .withColumn("arrivalTime",
                                     F.col("arrivalTime").cast(IntegerType()))
                                )
g = GraphFrame(cleaned_nodes, cleaned_relationships)
```

我们必须对节点进行一些清理，因为有些机场没有有效的机场代码。我们将为这些列提供更具描述性的名称，并将一些项转换为适当的数字类型。我们还需要确保有名为id、dst和src的列，正如Spark的graphframes库所预期的那样。

我们还将创建一个单独的DataFrame，将航空公司代码映射到航空公司名称。我们将在本章的后面部分使用：

```
airlines_reference = (spark.read.csv("data/airlines.csv")
          .select("_c1", "_c3")
          .withColumnRenamed("_c1", "name")
          .withColumnRenamed("_c3", "code"))
    airlines_reference = airlines_reference[airlines_reference["code"] != "null"]
```

探索性分析

让我们从一些探索性的分析开始，看看数据是什么样子的。

首先，让我们看看我们有多少机场：

```
g.vertices.count()
1435
```

我们在这些机场之间有几条连接线？

```
g.edges.count()
616529
```

热门机场

哪些机场起飞的航班最多？我们可以使用度中心性算法计算出从机场起飞的航班数：

```
airports_degree = g.outDegrees.withColumnRenamed("id", "oId”)

    full_airports_degree = (airports_degree
                            .join(g.vertices, airports_degree.oId == g.vertices.id)
                            .sort("outDegree", ascending=False)
                            .select("id", "name", "outDegree”))

full_airports_degree.show(n=10, truncate=False)
```

如果运行该代码，我们将看到以下输出：

```
+-----+-----------------------------------------------------+-----------+
| id  | name                                                | outDegree |
+-----+-----------------------------------------------------+-----------+
| ATL | Hartsfield_Jackson_Atlanta_International_Airport | 33837     |
| ORD | Chicago_O’Hare_International_Airport             | 28338     |
| DFW | Dallas_Fort_Worth_International_Airport          | 23765     |
| CLT | Charlotte_Douglas_International_Airport          | 20251     |
| DEN | Denver_International_Airport                     | 19836     |
| LAX | Los_Angeles_International_Airport                | 19059     |
| PHX | Phoenix_Sky_Harbor_International_Airport         | 15103     |
| SFO | San_Francisco_International_Airport              | 14934     |
| LGA | La_Guardia_Airport                               | 14709     |
| IAH | George_Bush_Intercontinental_Houston_Airport     | 14407     |
+-----+-----------------------------------------------------+-----------+
```

大多数美国大城市都有受欢迎的机场，如Chicago、Atlanta、Los Angeles和New York。我们还可以使用以下代码创建外出航班的可视化表示：

```
plt.style.use('fivethirtyeight')
ax = (full_airports_degree.toPandas().head(10).plot(kind='bar', x='id', y='outDegree', legend=None))
ax.xaxis.set_label_text("")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

结果图表如图7-11所示。

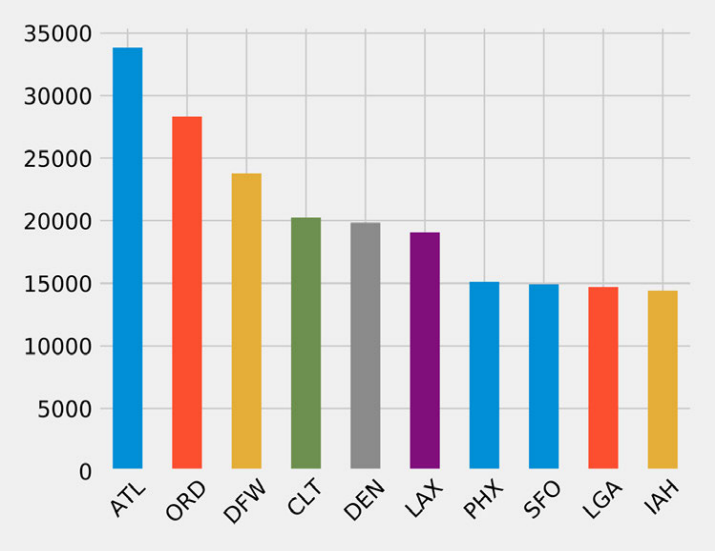


图7-11.机场离港航班

令人惊讶的是，航班数量突然减少的趋势。DEN（Denver International Airport）是第五大最受欢迎的机场，其出港航班数仅为ATL（Hartsfield Jackson Atlanta International Airport，ATL）的一半以上。

ORD机场的延迟

在我们的场景中，我们经常在西海岸和东海岸之间旅行，希望通过ORD（Chicago O’ Hare International Airport）这样的中间枢纽看到航班延误情况。这个数据集包含航班延误数据，所以我们可以直接进入。

以下代码按目的地机场分组得出从ORD起飞的航班的平均延误：

```
delayed_flights = (g.edges
                        .filter("src = 'ORD' and deptDelay > 0")
                        .groupBy("dst")
                        .agg(F.avg("deptDelay"), F.count("deptDelay"))
                        .withColumn("averageDelay",
                                    F.round(F.col("avg(deptDelay)"), 2))
                        .withColumn("numberOfDelays",
                                    F.col("count(deptDelay)")))
    (delayed_flights
     .join(g.vertices, delayed_flights.dst == g.vertices.id)
     .sort(F.desc("averageDelay"))
     .select("dst", "name", "averageDelay", "numberOfDelays")
     .show(n=10, truncate=False))
```

一旦我们计算了按目的地分组的平均延迟，我们就用一个包含所有顶点的数据框连接生成的Spark DataFrame，这样我们就可以打印目的地机场的全名。

运行此代码将返回延迟最严重的10个目的地：

```
+-----+-----------------------------------------------+--------------+----------------+
| dst | name                                          | averageDelay | numberOfDelays |
+-----+-----------------------------------------------+--------------+----------------+
| CKB | North Central West Virginia Airport           | 145.08       | 12             |
| OGG | Kahului Airport                               | 119.67       | 9              |
| MQT | Sawyer International Airport                  | 114.75       | 12             |
| MOB | Mobile Regional Airport                       | 102.2        | 10             |
| TTN | Trenton Mercer Airport                        | 101.18       | 17             |
| AVL | Asheville Regional Airport                    | 98.5         | 28             |
| ISP | Long Island Mac Arthur Airport                | 94.08        | 13             |
| ANC | Ted Stevens Anchorage International Airport   | 83.74        | 23             |
| BTV | Burlington International Airport              | 83.2         | 25             |
| CMX | Houghton County Memorial Airport              | 79.18        | 17             |
+-----+-----------------------------------------------+--------------+----------------+
```

这很有趣，但有一个数据点非常突出：从ORD到CKB的12次航班平均延误超过2小时！让我们找出这些机场之间的航班，看看发生了什么：

```
from_expr = 'id = "ORD"'
to_expr = 'id = "CKB"'
ord_to_ckb = g.bfs(from_expr, to_expr)

ord_to_ckb = ord_to_ckb.select(
    F.col("e0.date"),
    F.col("e0.time"),
    F.col("e0.flightNumber"),
    F.col("e0.deptDelay"))
```

然后我们可以用以下代码绘制航班：

```
ax = (ord_to_ckb.sort("date").toPandas().plot(kind='bar', x='date', y='deptDelay', legend=None))
ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

如果我们运行这个代码，我们将得到图7-12中的图表。

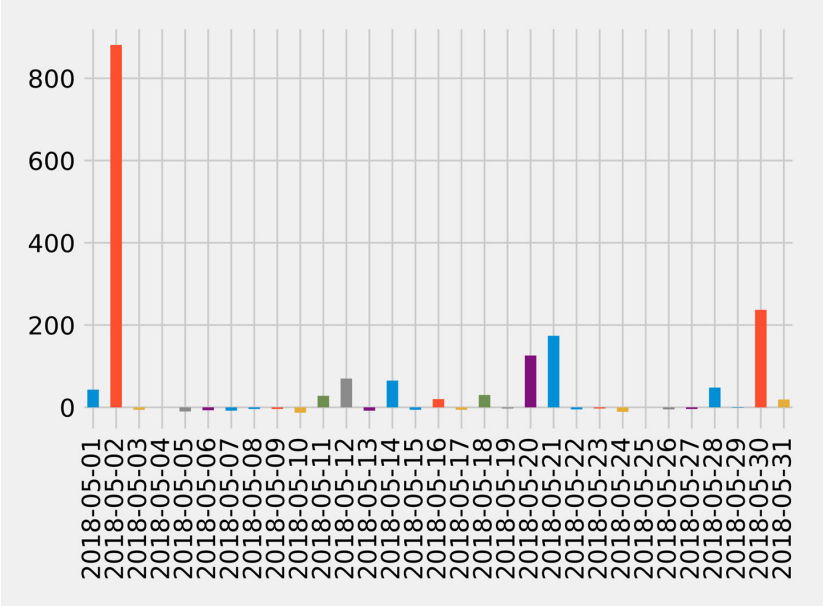


图7-12。从ORD到CKB的航班

大约一半的航班被延误，但2018年5月2日超过14小时的延误已经严重扭曲了平均水平。

如果我们想发现进出沿海机场的延误怎么办？这些机场经常受到恶劣天气条件的影响，因此我们可能会发现一些有趣的延误。

在SFO糟糕的一天

让我们考虑一个机场的延误与雾有关的“低天花板”问题：SFO（San Francisco International Airport）。分析的一种方法是看motif，它们是反复出现的子图或模式。

在Neo4j中和motif相当的概念是图模式（graph pattern）。图模式是用MATCH之类子句表的Cypher查询模式。

GraphFrame允许我们搜索motif，因此我们可以使用航班结构作为查询的一部分。让我们用主题来找出2018年5月11日进出旧金山的航班延误最多。以下代码将发现这些延迟：

```
motifs = (g.find("(a)-[ab]->(b); (b)-[bc]->(c)")
             .filter("""(b.id = 'SFO') and
                     (ab.date = '2018-05-11' and bc.date = '2018-05-11') and
                     (ab.arrDelay > 30 or bc.deptDelay > 30) and
                     (ab.flightNumber = bc.flightNumber) and
                     (ab.airline = bc.airline) and
(ab.time < bc.time)"""))
```

这个motif (a)-[ab]->(b); (b)-[bc]->(c) 查找同一个机场进出航班。然后，我们过滤生成的模式以查找具有以下特征的航班：

- 第一班航班到达SFO，第二班航班离开SFO，两个航班形成一个sequence
- 抵达或离开旧金山时的延误超过30分钟
- 航班号和航空公司相同

然后我们可以获取结果并选择我们感兴趣的列：

```
result = (motifs.withColumn("delta", motifs.bc.deptDelay - motifs.ab.arrDelay)
             .select("ab", "bc", "delta")
             .sort("delta", ascending=False))
    result.select(
        F.col("ab.src").alias("a1"),
        F.col("ab.time").alias("a1DeptTime"),
        F.col("ab.arrDelay"),
        F.col("ab.dst").alias("a2"),
        F.col("bc.time").alias("a2DeptTime"),
        F.col("bc.deptDelay"),
        F.col("bc.dst").alias("a3"),
        F.col("ab.airline"),
        F.col("ab.flightNumber"),
        F.col("delta")
).show()
```

我们还计算到港航班和离港航班之间的增量，这样可以看看哪些因素是归因于SFO。

如果执行此代码，将得到以下结果：

```
+---------+--------------+-----+------------+----------+-----+------------+-----------+-----+-------+
| airline | flightNumber | a1  | a1DeptTime | arrDelay | a2  | a2DeptTime | deptDelay | a3  | delta |
+---------+--------------+-----+------------+----------+-----+------------+-----------+-----+-------+
| WN      | 1454         | PDX | 1130       | -18.0    | SFO | 1350       | 178.0     | BUR | 196.0 |
| OO      | 5700         | ACV | 1755       | -9.0     | SFO | 2235       | 64.0      | RDM | 73.0  |
| UA      | 753          | BWI | 700        | -3.0     | SFO | 1125       | 49.0      | IAD | 52.0  |
| UA      | 1900         | ATL | 740        | 40.0     | SFO | 1110       | 77.0      | SAN | 37.0  |
| WN      | 157          | BUR | 1405       | 25.0     | SFO | 1600       | 39.0      | PDX | 14.0  |
| DL      | 745          | DTW | 835        | 34.0     | SFO | 1135       | 44.0      | DTW | 10.0  |
```

```
| WN      | 1783         | DEN | 1830       | 25.0     | SFO | 2045       | 33.0      | BUR | 8.0   |
| WN      | 5789         | PDX | 1855       | 119.0    | SFO | 2120       | 117.0     | DEN | -2.0  |
| WN      | 1585         | BUR | 2025       | 31.0     | SFO | 2230       | 11.0      | PHX | -20.0 |
+---------+--------------+-----+------------+----------+-----+------------+-----------+-----+-------+
```

最严重的情况是WN1454，它出现在最上面一排；它来得早，但出发晚了近三个小时。我们还可以看到arrDelay列中有一些负值；这意味着到SFO的航班偏早。

还请注意，一些航班，如WN5789和WN1585，在SFO的地面上弥补了时间，如图所示的负增量。

航空公司的互联机场

现在，假设我们已经旅行了很多次，我们决定使用那些频繁的飞行点，以尽可能有效地查看尽可能多的目的地，很快就会过期。如果我们从一个特定的美国机场出发，我们可以访问多少个不同的机场，然后使用同一家航空公司返回起始机场？

让我们首先确定所有的航空公司，并计算出每个航空公司有多少航班：

```
airlines = (g.edges
     .groupBy("airline")
     .agg(F.count("airline").alias("flights"))
     .sort("flights", ascending=False))
    full_name_airlines = (airlines_reference
                          .join(airlines, airlines.airline
                                == airlines_reference.code)
                          .select("code", "name", "flights"))
```

现在我们创建一个柱状图来呈现航班情况。

```
    ax = (full_name_airlines.toPandas()
          .plot(kind='bar', x='name', y='flights', legend=None))
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

如果我们运行这个查询，我们将在图7-13中得到输出。

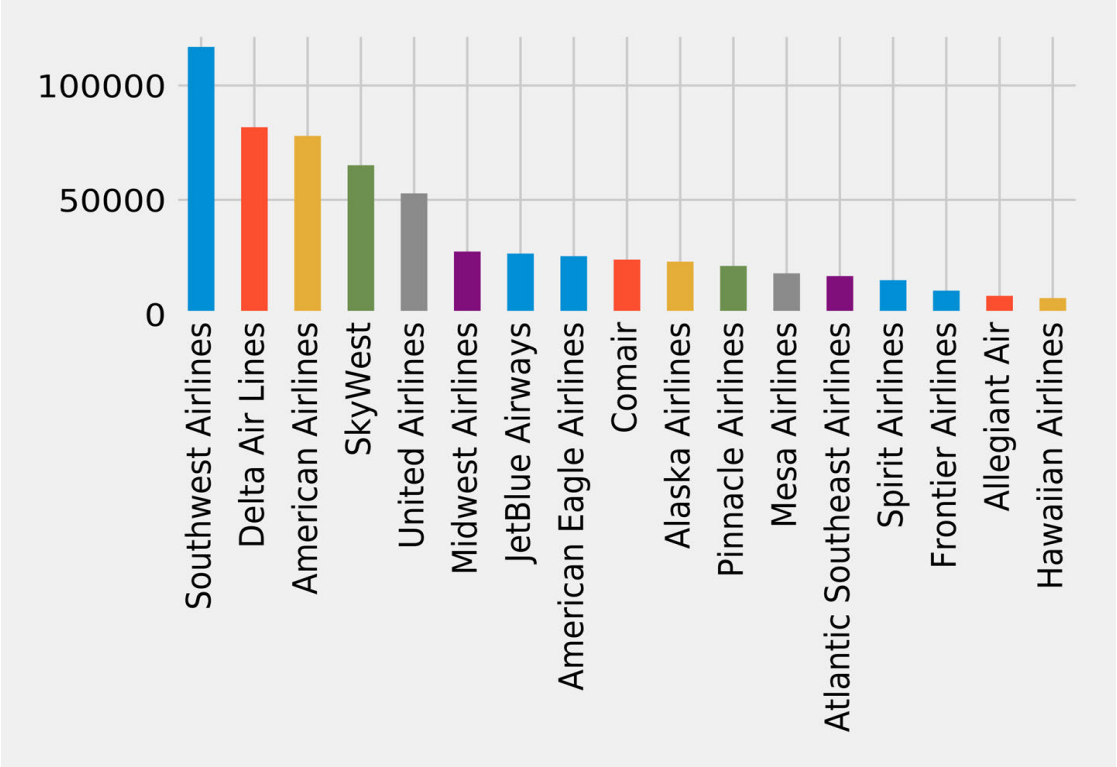


图7-13。航空公司的航班数

现在，让我们编写一个函数，该函数使用强连接组件算法查找每个航空公司的机场分组，其中所有机场都有往返该分组中所有其他机场的航班：

```
def find_scc_components(g, airline):
# Create a subgraph containing only flights on the provided airline airline_relationships = g.edges[g.edges.airline ==
airline] airline_graph = GraphFrame(g.vertices, airline_relationships)
        # Calculate the Strongly Connected Components
        scc = airline_graph.stronglyConnectedComponents(maxIter=10)
        # Find the size of the biggest component and return that
return (scc
.groupBy("component") .agg(F.count("id").alias("size")) .sort("size", ascending=False) .take(1)[0]["size"])
```

我们可以编写以下代码来创建一个数据框架，其中包含每个航空公司及其最大强连接组件的机场数量：

```
# Calculate the largest strongly connected component for each airline
airline_scc = [(airline, find_scc_components(g, airline))
for airline in airlines.toPandas()["airline"].tolist()]
    airline_scc_df = spark.createDataFrame(airline_scc, ['id', 'sccCount'])

# Join the SCC DataFrame with the airlines DataFrame so that we can show
# the number of flights an airline has alongside the number of
# airports reachable in its biggest component
airline_reach = (airline_scc_df
     .join(full_name_airlines, full_name_airlines.code == airline_scc_df.id)
```

```
     .select("code", "name", "flights", "sccCount")
     .sort("sccCount", ascending=False))
And now let's create a bar chart showing our airlines:
    ax = (airline_reach.toPandas()
          .plot(kind='bar', x='name', y='sccCount', legend=None))
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

如果我们运行这个查询，我们将在图7-14中得到输出。

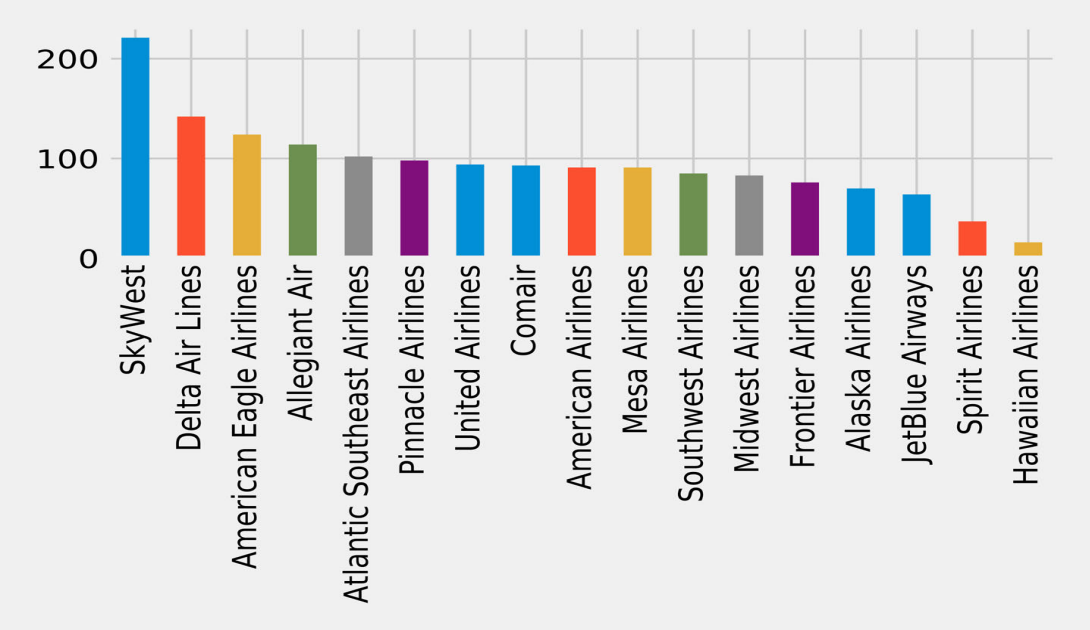


图7-14.航空公司可到达的机场数量

SkyWest拥有最大的社区，拥有200多个连接紧密的机场。这可能部分反映了其作为联属航空公司的商业模式，该公司运营合作航空公司航班中使用的飞机。另一方面，Southwest拥有最多的航班，但仅连接大约80个机场。

现在，让我们假设我们拥有的大多数常飞点都是Delta Airlines（DL）。我们能找到在特定航空公司网络内形成社区的机场吗？

```
airline_relationships = g.edges.filter("airline = 'DL'")
    airline_graph = GraphFrame(g.vertices, airline_relationships)
    clusters = airline_graph.labelPropagation(maxIter=10)
    (clusters
     .sort("label")
     .groupby("label")
     .agg(F.collect_list("id").alias("airports"),
          F.count("id").alias("count"))
     .sort("count", ascending=False)
     .show(truncate=70, n=10))
```

如果运行该查询，我们将看到以下输出：

```
+---------------+--------------------------------------------------------------------------+-------+
| label         | airports                                                                 | count |
+---------------+--------------------------------------------------------------------------+-------+
| 1606317768706 | [IND, ORF, ATW, RIC, TRI, XNA, ECP, AVL, JAX, SYR, BHM, GSO, MEM, C…   | 89    |
| 1219770712067 | [GEG, SLC, DTW, LAS, SEA, BOS, MSN, SNA, JFK, TVC, LIH, JAC, FLL, M…   | 53    |
| 17179869187   | [RHV]                                                                    | 1     |
| 25769803777   | [CWT]                                                                    | 1     |
| 25769803776   | [CDW]                                                                    | 1     |
| 25769803782   | [KNW]                                                                    | 1     |
| 25769803778   | [DRT]                                                                    | 1     |
| 25769803779   | [FOK]                                                                    | 1     |
| 25769803781   | [HVR]                                                                    | 1     |
| 42949672962   | [GTF]                                                                    | 1     |
+---------------+--------------------------------------------------------------------------+-------+
```

DL使用的大多数机场都分为两组，让我们深入研究一下。这里有太多的机场要显示，所以我们只显示最大度（degree）的机场（进出航班）。我们可以编写以下代码来计算每个机场的度：

```
all_flights = g.degrees.withColumnRenamed("id", "aId")
```

然后，我们将把它与属于最大集群的机场结合起来：

```
(clusters
     .filter("label=1606317768706")
     .join(all_flights, all_flights.aId == result.id)
.sort("degree", ascending=False)
     .select("id", "name", "degree")
     .show(truncate=False))
```

如果我们运行这个查询，我们将得到这个输出：

```
+-----+-------------------------------------------------------------+--------+
| id  | name                                                        | degree |
+-----+-------------------------------------------------------------+--------+
| DFW | Dallas_Fort_Worth_International_Airport                     | 47514  |
| CLT | Charlotte_Douglas_International_Airport                     | 40495  |
| IAH | George_Bush_Intercontinental_Houston_Airport                | 28814  |
| EWR | Newark_Liberty_International_Airport                        | 25131  |
| PHL | Philadelphia_International_Airport                          | 20804  |
| BWI | Baltimore/Washington_International_Thurgood_Marshall_Airport | 18989  |
| MDW | Chicago_Midway_International_Airport                        | 15178  |
| BNA | Nashville_International_Airport                             | 12455  |
| DAL | Dallas_Love_Field                                           | 12084  |
| IAD | Washington_Dulles_International_Airport                     | 11566  |
| STL | Lambert_St_Louis_International_Airport                      | 11439  |
| HOU | William_P_Hobby_Airport                                     | 9742   |
| IND | Indianapolis_International_Airport                          | 8543   |
```

```
| PIT | Pittsburgh_International_Airport                          | 8410   |
| CLE | Cleveland_Hopkins_International_Airport                   | 8238   |
| CMH | Port_Columbus_International_Airport                       | 7640   |
| SAT | San_Antonio_International_Airport                         | 6532   |
| JAX | Jacksonville_International_Airport                        | 5495   |
| BDL | Bradley_International_Airport                             | 4866   |
| RSW | Southwest_Florida_International_Airport                   | 4569   |
+-----+-----------------------------------------------------------+--------+
```

在图7-15中，我们可以看到这个集群实际上集中在美国中西部的东海岸。

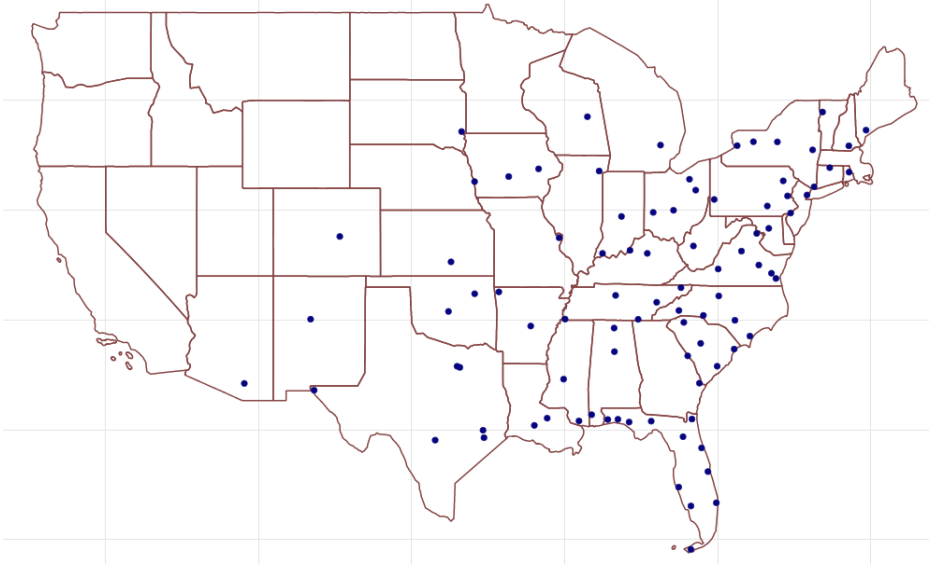


图7-15。集群1606317768706机场

现在让我们对第二大集群做同样的事情：

```
(clusters
     .filter("label=1219770712067")
     .join(all_flights, all_flights.aId == result.id)
     .sort("degree", ascending=False)
     .select("id", "name", "degree")
     .show(truncate=False))
```

如果运行该查询，我们将得到以下输出：

```
+-----+-----------------------------------------------------------+--------+
| id  | name                                                      | degree |
+-----+-----------------------------------------------------------+--------+
| ATL | Hartsfield Jackson Atlanta International Airport          | 67672  |
| ORD | Chicago O'Hare International Airport                      | 56681  |
| DEN | Denver International Airport                              | 39671  |
| LAX | Los Angeles International Airport                         | 38116  |
| PHX | Phoenix Sky Harbor International Airport                  | 30206  |
| SFO | San Francisco International Airport                       | 29865  |
| LGA | LA Guardia Airport                                        | 29416  |
```

```
| LAS | McCarran International Airport                                | 27801  |
| DTW | Detroit Metropolitan Wayne County Airport                     | 27477  |
| MSP | Minneapolis-St Paul International/Wold-Chamberlain Airport    | 27163  |
| BOS | General Edward Lawrence Logan International Airport           | 26214  |
| SEA | Seattle Tacoma International Airport                          | 24098  |
| MCO | Orlando International Airport                                 | 23442  |
| JFK | John F Kennedy International Airport                          | 22294  |
| DCA | Ronald Reagan Washington National Airport                     | 22244  |
| SLC | Salt Lake City International Airport                          | 18661  |
| FLL | Fort Lauderdale Hollywood International Airport               | 16364  |
| SAN | San Diego International Airport                               | 15401  |
| MIA | Miami International Airport                                   | 14869  |
| TPA | Tampa International Airport                                   | 12509  |
+-----+---------------------------------------------------------------+--------+
```

在图7-16中，我们可以看到这个集群显然更加集中于枢纽，沿途还有一些西北方向的站点。

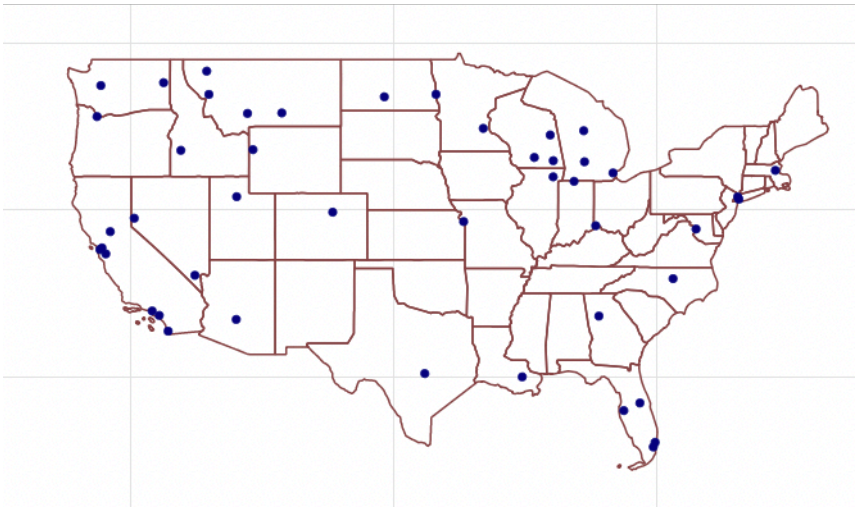


图7-16。集群1219770712067机场

我们用来生成这些映射的代码可以在书的Github存储库中找到。

在查看DL网站上的飞行常客计划时，我们注意到一个use-two-get-one-free（买二赠一）的促销计划。如果我们将积分用于两个航班，我们将免费获得另一个，但前提是我们在两个集群中的一个内飞行！用好我们的时间和得出来的成果，让我们在一个集群中飞行。

读者练习

- 使用最短路径算法评估从你家到Bozeman Yellowstone International Airport（BZN）的航班数量。

- 如果使用关系权重，有什么区别吗？

总结

在前面几章中，我们详细介绍了Apache Spark和Neo4j中用于路径查找、中心性和社区检测的关键图算法是如何工作的。

在本章中，我们介绍了一些工作流程，其中包括在其他任务和分析中使用几种算法。我们使用旅行业务场景分析Neo4j中的Yelp数据，并使用个人航空旅行场景评估Spark中的美国航空公司数据。

下一步，我们将研究图算法越来越重要的用途：用图来增强机器学习。

第八章 用图算法增强机器学习

我们已经陆续介绍了几种算法，例如LPA；但是，直到现在，我们还是强调图算法用于一般性图分析。由于图在机器学习（ML）中的应用越来越多，我们现在将研究如何使用图算法来增强ML的工作流程。

在本章中，我们重点介绍用图算法改进机器学习的最实用办法：提取连接相关的特征，并用于在关系预测。首先，我们将介绍一些基本的ML概念，并了解对提升预测的上下文数据的重要性。然后，我们将快速了解图功能的应用方式，包括用于垃圾邮件发送者欺诈、检测和链接预测。

我们将演示如何创建机器学习管道，并训练和评估连接预测的模型，并将Neo4j和Spark集成到我们的工作流程中。我们的示例将基于Citation Network Dataset，它包含作者、论文、作者关系和引用关系。我们将使用几个模型来预测研究作者将来是否有可能合作，并展示图形算法如何改进结果。

机器学习与上下文的重要性

机器学习不是人工智能，而是实现人工智能的一种方法。ML使用算法通过特定的示例和基于预期结果的渐进式改进来训练软件，而无需为如何实现这些更好的结果进行显式编程。训练包括向模型提供大量数据，并使其能够学习如何处理和合并这些信息。

从这个意义上讲，学习意味着算法迭代，不断地进行更改以接近目标，例如与训练数据相比减少分类错误。ML也是动态的，当有更多的数据时，它能够自我修改和优化。这可以发生在许多批次的使用前训练中，也可以作为使用过程中的在线学习。

最近在ML预测、大数据集的可访问性和并行计算能力方面取得的成功使ML更适合开发用于人工智能应用的概率模型。随着机器学习越来越广泛，记住它的基本目标很重要：做出与人类相似的选择。如果我们忘记了这一目标，我们最终可能只会得到另一个高目标、基于规则的软件版本。

为了提高机器学习的准确性，同时使解决方案更广泛地适用，我们需要整合大量的上下文信息，就像人们应该使用上下文来做出更好的决策一样。人类使用他们周围的环境，而不仅仅是直接的数据点，来找出在一种情况下什么是必要的，估

计缺失的信息，并决定如何将经验教训应用到新的情况。上下文帮助我们改进预测。

图、上下文和准确性

如果没有周边和相关信息，试图预测行为或针对不同情况提出建议的解决方案需要更大量的训练和规定性规则。这就是为什么人工智能擅长于特定的、定义明确的任务，但却难以处理模糊性的部分原因。图增强的ML可以帮助填充丢失的上下文信息，这些信息对于更好的决策非常重要。

从图论和现实生活中我们都知道，关系往往是行为的最强预测因子。例如，如果一个人投票，他们的朋友、家人甚至同事投票的可能性就会增加。图8-1说明了R. Bond等人在2012年的论文“A 61-Million-Person Experiment in Social Influence and Political Mobilization”中研究的被报道的投票（reported voting）和Facebook朋友所产生的连锁反应。

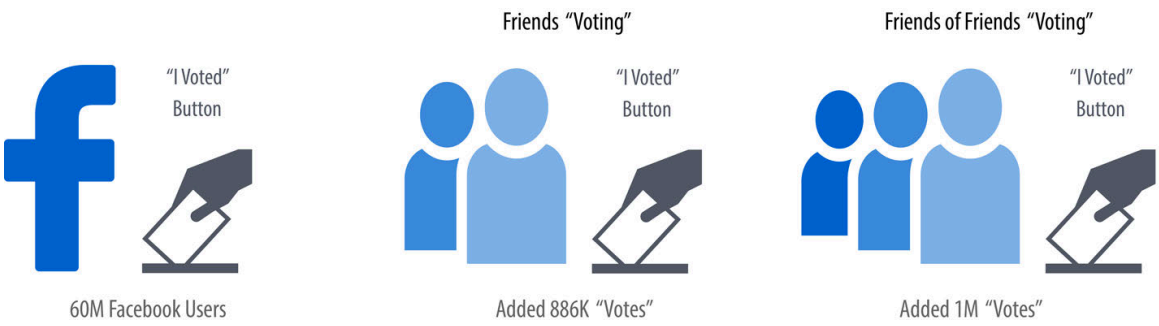


图8-1.人们受其社交网络的影响而投票。在这个例子中，两个跳点距离处的朋友就总体而言比直接的关系影响更大。

作者发现，报告投票的朋友影响了另外1.4%的用户声称他们投票了，有趣的是，朋友的朋友又增加了1.7%。小的百分比可能会产生显著的影响，我们可以在图8-1中看到，两个跳点处的人总比直接的朋友有更多的影响。Nicholas Christakis和James Fowler在《Connected》（Little、Brown和Company）一书中介绍了投票和其他社会网络如何影响我们的例子。

添加图特征和上下文可以改进预测，特别是在连接很重要的情况下。例如，零售公司不仅使用历史数据，还使用有关客户相似性和在线行为的上下文数据来使产品建议更加个性化。Amazon的Alexa使用了多层上下文模型（several layers of

contextual models），以提高准确性。2018年，亚马逊在回答新问题时，还引入了“上下文转移（context carryover）”，将以前的参考资料纳入对话中。

不幸的是，今天许多机器学习方法都缺少大量丰富的上下文信息。这是由于ML依赖于从元组构建的输入数据，而忽略了许多预测关系和网络数据。此外，上下文信息并不总是容易获得，或者太难访问和处理。对于传统的方法来说，即使找到四个或更多个跃点以外的连接也是一个规模上的挑战。使用图，我们可以更容易地访问和合并连接的数据。

连接特征（Connected feature）提取与选择

特征提取和选择有助于我们通过获取原始数据，创建一个合适的子集和格式来训练我们的机器学习模型。这是一个基本的步骤，当执行良好时，会使得ML产生更一致的准确预测。

特征提取与选择

特征提取是将大量数据和属性提取为一组具有代表性的描述性属性的方法。该过程为输入数据中的独特特征或模式导出数值（特征），以便我们可以在其他数据中区分类别。特征提取，在数据难以被模型直接使用时会用到，难以使用的原因可能是数据量、格式或者需要偶然比较（incidental comparison)。

特征选择是确定对目标最重要或影响最大的提取特征子集的过程。它被用来表明预测的重要性以及效率。例如，如果我们有20个特征，其中13个共同解释了我们需要预测的92%，那么我们可以在模型中消除7个特征。

将正确的特征组合在一起可以提高准确性，因为它从根本上影响我们的模型的学习方式。因为即使是适度的改进也会产生显著的差异，所以我们在本章中的重点是关联特征。连接特征(connected feature)是从图数据结构中提取的特征。这些特征可以从基于节点周围图形部分的图局部查询，用于连接特征提取的图全局查询。这些图全局查询用图算法在关系数据中识别可预测元素。

不仅要获得正确的功能组合，而且要消除不必要的功能，以降低我们的模型被错误计算（hypertargeted)。这使我们创建避免只在训练数据（称为过度拟合）上工作良好的模型，并大大扩展了适用性。我们还可以使用图算法来评估这些特征，并确定哪些特征对我们的关联特征选择模型最有影响。例如，我们可以将特征映射到图中的节点，根据相似的特征创建关系，然后计算特征的中心性。特征关系可以

通过保存数据点的聚类密度来定义。该方法是由K.Henniab、N.Mezghani和C.Gouin Vallerand在“Unsupervised Graph-Based Feature Selection Via Subspace and PageRank Centrality”中使用高维和低样本量的数据集来描述的。

图嵌入（Graph Embedding）

图嵌入是将图中的节点和关系表示为特征向量。这些仅仅是具有维度映射的特征的集合，如图8-2所示的（x，y，z）坐标。

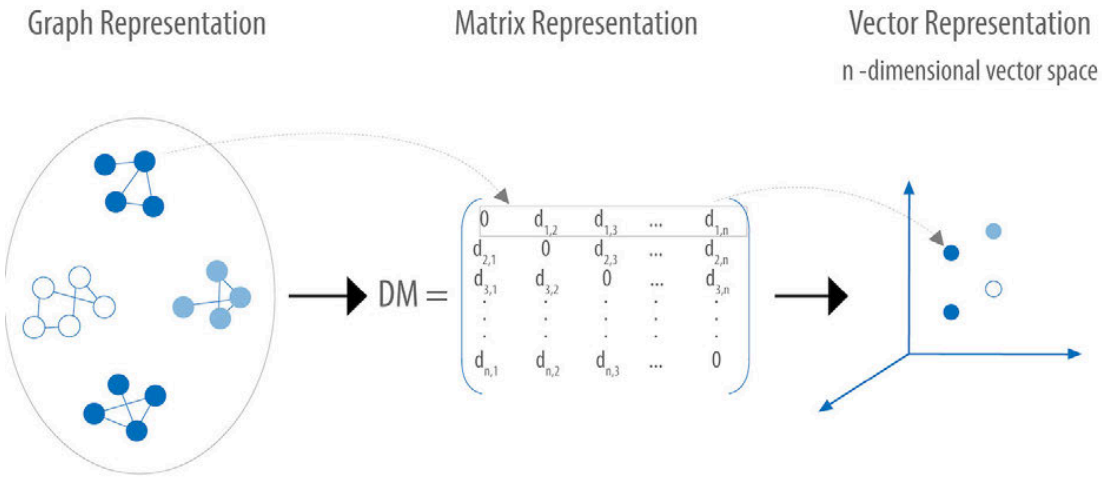


图8-2.图嵌入将图数据映射到可以可视化为多维坐标的特征向量中。

图形嵌入使用的图数据与连接特征提取略有不同。它使我们能够以数字格式表示整个图形或图形数据的子集，为机器学习任务做好准备。这对于无监督的学习尤其有用，因为数据通过关系获取更多的上下文信息，因此数据不进行分类。图嵌入还可用于数据挖掘、计算实体之间的相似性以及减少维数以辅助统计分析。（区分图嵌入和图特征提取，前者主要为无监督学习服务，后者主要为监督学习服务。本章主要致力于后者。）

这是一个快速发展的空间，有多种选择，包括node2vec、 struc2vec、GraphSAGE,、DeepWalk和 DeepGL。

现在，让我们看看一些连接特征的类型以及它们是如何使用的。

图特征(graphy features)

图形特征包括关于我们的图形的任何数量的连接相关度量，例如进入或退出节点的关系数、潜在三角形数和共同的邻居。在我们的示例中，我们将从这些度量开始，因为它们很容易收集，并且是对早期假设的良好测试。此外，当我们精确地知道我们在寻找什么时，我们可以使用特征工程。

例如，如果我们想知道有多少人拥有一个最多四个跳点处的欺诈账户。这种方法使用图遍历来非常有效地查找关系的深层路径，查看诸如标签、属性、计数和推断关系等内容。

我们还可以轻松地自动化这些流程，并将这些预测性图形功能交付到我们现有的管道中。例如，我们可以提取欺诈者关系的计数，并将该数字作为节点属性添加，以用于其他机器学习任务。

图算法特征(graph algorithm features)

我们也可以使用图算法来寻找我们所寻找的一般结构的特征，而不是精确的模式。举例来说，让我们假设我们知道某些类型的社区群表示欺诈；也许有一个典型的密度或层次关系。在这种情况下，我们不需要一个固定的组织特征，而需要一个灵活的全局相关结构。在我们的示例中，我们将使用社区检测算法来提取连接的特征，但是中心性算法（如PageRank）也经常被应用。

此外，结合多种连接特征的方法似乎比坚持使用一种方法要好。例如，我们可以将连接特征与基于Louvain算法发现的社区、使用PageRank的影响节点以及三个跃点处已知欺诈者的度量指标相结合来预测欺诈。

图8-3展示了一种组合方法，作者将Pagerank和Coloring等图形算法与入链和出链等图度量相结合。此图摘自S.Fakhraei等人的论文“Collective Spammer Detection in Evolving Multi-Relational Social Networks”。

在该论文中，图结构（Graph Structure ）章节说明了使用几种图算法进行的连接特征提取。有趣的是，作者发现从多种关系类型中提取关联特征比简单地添加更多特征更具预测性。报告子图部分（Report Subgraph）显示了如何将图形功能转换为ML模型可以使用的功能。通过在一个图增强的ML工作流程中结合多种方法，作者能够改进先前的检测方法，并分类70%以前需要手动标记的垃圾邮件发送者，准确率为90%。

即使我们提取了连接特征，我们也可以通过使用像PageRank这样的图算法来优化影响最大的特征。这使我们能够充分地表示数据，同时消除可能降低结果或处理速度的噪声变量。利用这类信息，我们还可以识别出高共现性的特征，通过特征约简进一步调整模型。这一方法在D. IENCO、R. MEO和M. Botta的研究论文“Using PageRank in Feature Selection”中进行了概述。

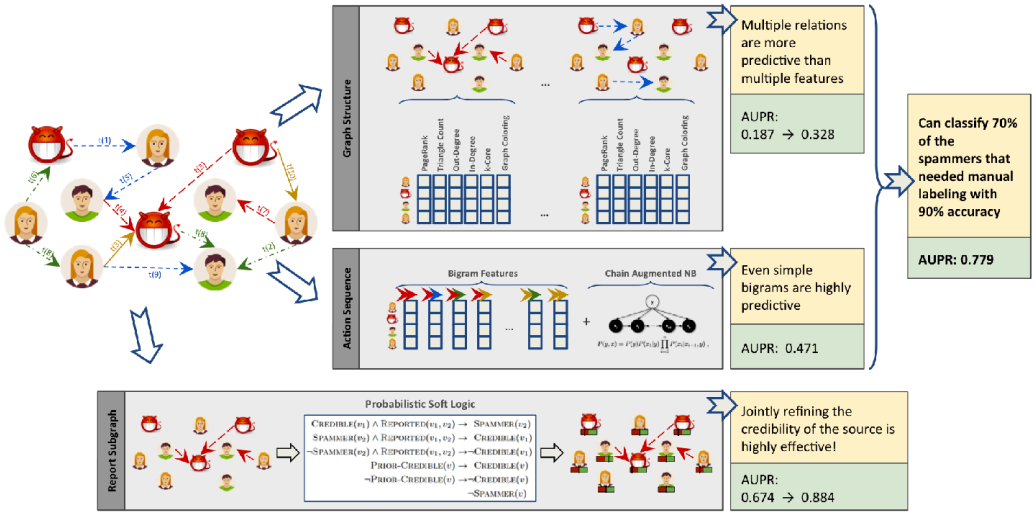


图8-3.关联特征提取可以与其他预测方法相结合，以提高结果。AUPR是指精确召回曲线下的区域，数值越大越好。

我们已经讨论了连接特征如何应用于涉及欺诈和垃圾邮件发送者检测的场景。在这些情况下，活动通常隐藏在多个模糊层和网络关系中。传统的特征提取和选择方法在没有图形所带来的上下文信息的情况下可能无法检测到这种行为。

连接特征能够增强机器学习的另一个领域（以及本章其余部分的重点）是链接预测。链接预测是一种估计关系未来形成的可能性的方法，或者该关系可能已经出现在我们的图中，但由于数据不完整而丢失。由于网络是动态的，可以快速增长，因此能够预测即将生成的链接具有广泛的适用性，从产品建议到药物重定目标，甚至推断犯罪关系。

图中的连接特征通常用来改进链接预测，在此过程中使用到了基本的图特征，以及从中心性和社区等算法中提取的特征。

基于节点邻近性或相似性的链接预测也是标准的，在论文“The Link Prediction Problem for Social Networks”中D. Liben Nowell和J. Kleinberg认为，仅网络结构就可能包含足够的潜在信息，用来检测节点邻近性，并优于直接的测量。

既然我们已经研究了连接特征如何增强机器学习，那么让我们深入到我们的链接预测示例中，看看如何应用图算法来改进预测。

实用图和机器学习：链接预测

本章的其余部分将演示一个基于引文网络数据集（Citation Network Dataset）的实践示例，该数据集是从DBLP、ACM和MAG中提取的一个研究数据集。该数据集在J. Tang等人的论文“ArnetMiner : Extraction and Mining of Academic Social Networks”中进行了描述。最新版本包含3,079,007篇论文、1,766,547位作者、9,437,718位作者关系和25,166,994位引文关系。

我们将在一个子集上工作，重点关注以下出版物中出现的文章：

- Lecture Notes in Computer Science
- Communications of the ACM
- International Conference on Software Engineering
- Advances in Computing and Communications

我们生成的数据集包含51,956篇论文、80,299位作者、140,575位作者关系和28,706个引文关系。我们将根据合作论文的作者创建一个合作者图，然后预测未来两个作者之间的合作。我们只对没有合作过的作者之间的合作感兴趣，在合作者之间有多个合作的，不是我们的关注重点。

在本章的其余部分，我们将建立所需的工具，并将数据导入Neo4j。然后我们将介绍如何正确平衡数据（均匀化），并将样本拆分为Spark DataFrame，以便进行训练和测试。在此之后，我们会在Spark中创建机器学习管道之前，解释链路预测假设和方法。

最后，我们将通过训练和评估各种预测模型，从基本的图特征开始，添加更多使用Neo4j提取的图算法特征。

工具和数据

让我们从设置工具和数据开始。然后我们将探索我们的数据集并创建一个机器学习管道。

在我们做其他事情之前，让我们先设置本章中使用的库：

库	意义
Py2neo	一个与Python数据科学生态系统完美集成的Neo4j Python库
pandas	一个高性能的库，用于数据库外部的数据打包，包含易于使用数据结构和数据分析工具
Spark MLlib	Spark的机器学习库

我们使用MLlib作为机器学习库的一个示范。本章所示的方法可以与其他ML库，如Scikit learn等等，结合使用。

所有显示的代码都将在PySpark REPL中运行。我们可以通过运行以下命令来启动REPL：

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
--driver-memory 2g \
--executor-memory 6g \
--packages julioasotodv:spark-tree-plotting:0.2
```

这类似于我们在第3章中启动REPL时使用的命令，但是我们加载的不是GraphFrames，而是spark-tree-plotting这个包。在编写时，Spark的最新发布版本是spark-2.4.0-bin-hadoop2.7，但由于这一版本可能在你阅读本文时有所更改，请确保适当更改Spark版本环境变量。

启动后，我们将导入以下要使用的库：

```
from py2neo import Graph
import pandas as pd
from numpy.random import randint
from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql.types import *
from pyspark.sql import functions as F
from sklearn.metrics import roc_curve, auc
from collections import Counter
from cycler import cycler
```

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

现在让我们创建一个到Neo4j数据库的连接：

```
graph = Graph("bolt://localhost:7687", auth=("neo4j", "neo"))
```

将数据导入Neo4j

现在我们准备将数据加载到Neo4j中，并为我们的训练和测试创建一个平衡的分割。我们需要下载数据集版本10的zip文件，解压它，并将内容放在导入文件夹中。我们应该有以下文件：

- dblp-ref-0.json
- dblp-ref-1.json
- dblp-ref-2.json
- dblp-ref-3.json

在导入文件夹中包含这些文件后，需要将以下属性添加到Neo4j设置文件中，以便使用APOC库处理这些文件：

```
apoc.import.file.enabled=true
apoc.import.file.use_neo4j_config=true
```

首先，我们将创建约束以确保不创建重复的文章或作者：

```
CREATE CONSTRAINT ON (article:Article)
ASSERT article.index IS UNIQUE;

CREATE CONSTRAINT ON (author:Author)
ASSERT author.name IS UNIQUE;
```

现在，我们可以运行以下查询从JSON文件导入数据：

```
CALL apoc.periodic.iterate(
    'UNWIND ["dblp-ref-0.json","dblp-ref-1.json",
    "dblp-ref-2.json","dblp-ref-3.json"] AS file
    CALL apoc.load.json("file:///" + file)
    YIELD value
    WHERE value.venue IN ["Lecture Notes in Computer Science",
                          "Communications of The ACM",
                          "international conference on software engineering",
                          "advances in computing and communications"]
```

```
    return value',
    'MERGE (a:Article {index:value.id})
    ON CREATE SET a += apoc.map.clean(value,["id","authors","references"],[0])
    WITH a,value.authors as authors
    UNWIND authors as author
    MERGE (b:Author{name:author})
    MERGE (b)<-[:AUTHOR]-(a)'
, {batchSize: 10000, iterateList: true});
```

这将生成图8-4中的图模式。

图8-4.引文关系图

这是一个连接文章和作者的简单图表，因此我们将添加更多可以从关系中推断的信息，以帮助预测。

合著关系图

我们希望预测未来作者之间的合作，所以我们将从创建一个合著关系图开始。以下Neo4j Cypher查询将在每对合作论文的作者之间创建合作作者关系：

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
MERGE (a1)-[coauthor:CO_AUTHOR {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

我们在查询中对合作作者关系设置的Year属性是这两个作者协作的最早年份。我们只对第一次有两位作者合作感兴趣，随后的合作与此无关。

图8-5是创建图的一部分示例（注意，这是一个很小的局部）。我们已经可以看到一些有趣的社区结构。

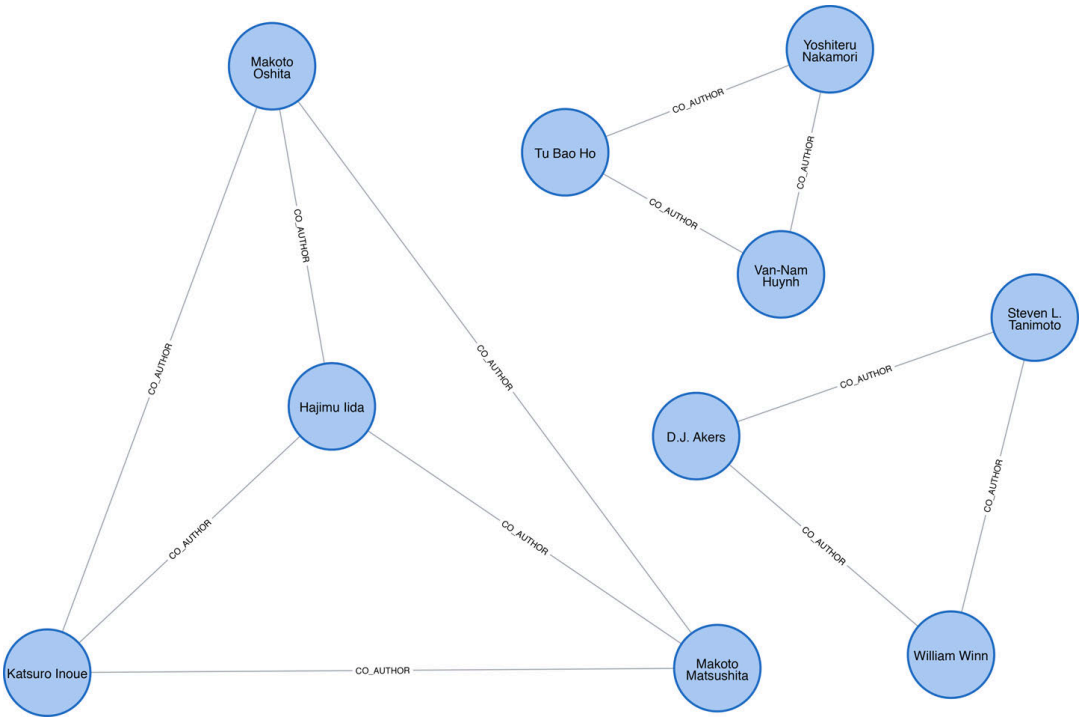


图8-5.合著者图

图中的每个圆代表一个作者，它们之间的线是CO_AUTHOR关系，因此我们有四个作者在左侧彼此协作，然后在右侧有三个合作作者的两个示例。现在我们已经加载了数据和一个基本的图表，让我们创建训练和测试需要的两个数据集。

创建均衡（balanced）的训练和测试数据集

对于链接预测问题，我们希望尝试并预测未来创建的链接。这个数据集可以很好地实现这一点，因为我们在文章中有可以用来分割数据的日期。我们需要计算出用哪一年来定义我们的训练/测试划分。我们将在那一年之前对我们的模型进行各方面的训练，然后用那一年之后的样本创建链接上测试模型。

让我们先看看这些文章是什么时候发表的。我们可以编写以下查询来获取按年份分组的文章数：

```
query = """
MATCH (article:Article)
RETURN article.year AS year, count(*) AS count
ORDER BY year
"""
by_year = graph.run(query).to_data_frame()
```

让我们将其可视化为条形图，使用以下代码就可以做到：

```
plt.style.use('fivethirtyeight')
ax = by_year.plot(kind='bar', x='year', y='count', legend=None, figsize=(15,8))
ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

我们可以在图8-6中看到执行此代码生成的图表。

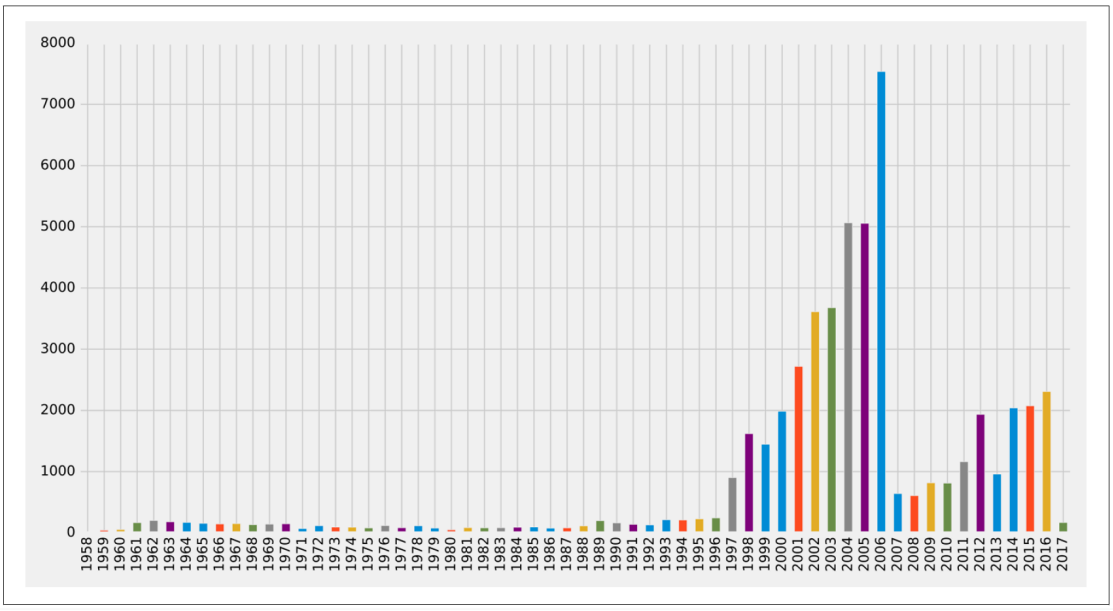


图8-6.按年份排列的文章

1997年之前发表的文章很少，2001年至2006年期间发表了大量文章，之后就减少了，然后是2011年以来的逐步攀升（2013年除外）。2006年似乎是一个很好的一年，可以将我们的数据分割开来，以训练我们的模型并做出预测。让我们看看那一年之前发表了多少论文，以及在那一年和那之后发表了多少论文。我们可以编写以下查询来计算：

```
MATCH (article:Article)
RETURN article.year < 2006 AS training, count(*) AS count
```

其结果如下，其中true是指2006年之前发表的论文：

```
+----------+-------+
| training | count |
+----------+-------+
| false    | 21059 |
| true     | 30897 |
+----------+-------+
```

结果还不错！60%的论文在2006年之前发表，40%在2006年期间或之后发表。对于我们的训练和测试来说，这是一个相当均衡的数据分割。

现在我们有了一个很好的文件分割，让我们使用2006年的相同分割来进行合著。我们将在第一次合作是在2006年之前的两位作者之间建立一种CO_AUTHOR_EARLY关系。

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year < 2006
MERGE (a1)-[coauthor:CO_AUTHOR_EARLY {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

然后，我们将在2006年或之后首次合作的两位作者之间建立CO_AUTHOR_LATE关系：

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year >= 2006
MERGE (a1)-[coauthor:CO_AUTHOR_LATE {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

在构建我们的训练和测试集之前，让我们检查有多少对节点之间有链接。以下查询将找到CO_AUTHOR_EARLY的数量：

```
MATCH ()-[:CO_AUTHOR_EARLY]->()
RETURN count(*) AS count
```

运行该查询将返回此处显示的结果：

```
+--------+
| count |
+--------+
| 81096 |
+--------+
```

此查询将找到CO_AUTHOR_LATE的数目：

```
MATCH ()-[:CO_AUTHOR_LATE]->()
RETURN count(*) AS count
```

运行该查询将返回此结果：

```
+-------+
| count |
+-------+
| 74128 |
+-------+
```

现在我们已经准备好构建我们的训练和测试数据集了。

前面一节解决的是训练集与测试集之间的比例，下面讲的是正例和反例之间的比例。

均衡和拆分数据

两个节点之间的CO_AUTHOR_EARLY和CO_AUTHOR_LATE关系将作为我们的积极（positive）例子，但我们也需要创建一些消极（negative）的例子。大多数现实世界中的网络都是稀疏的，只有局部密集，这个图也是这样。两个节点没有关系的样本数比有关系的样本数大得多。

如果我们查询我们的CO_AUTHOR_EARLY数据，我们会发现有45,018个作者有这种关系，但只有81,096个协作关系。这听起来可能不平衡，但事实更加严峻：我们的图可能具有的最大关系数是（45,018*45,017）/2=1,013,287,653，这意味着有很多负面的例子（没有链接）。如果我们用所有的负面例子来训练我们的模型，我们将有一个严重的类不平衡问题(class imbalance problem)。一个模型可以通过预测每对节点都没有关系来达到极高的精度。

在R. Lichtenwalter、J. Lussier和N. Chawla的论文“New Perspectives and Methods in Link Prediction”中，描述了解决这一挑战的几种方法。其中一种方法是通过在我们的邻近节点中找到我们当前没有连接的节点来构建负面的例子。（在正例附近寻找反例）

我们将通过找到两个到三个跃点之间的混合节点对来构建我们的负面示例，不包括那些已经有关系的节点对。然后我们将对这些节点对进行降采样，这样我们就可以得到等量的正负示例。

我们有314,248对节点，它们在两个跃点之间没有关系。如果我们增加距离到三个跃点，我们有967,677对节点。

下面的函数将用于对负面样本进行降采样：

```
def down_sample(df):
    copy = df.copy()
    zero = Counter(copy.label.values)[0]
    un = Counter(copy.label.values)[1]
    n = zero - un
    copy = copy.drop(copy[copy.label == 0].sample(n=n, random_state=1).index)
    return copy.sample(frac=1)
```

这个函数计算出正负两个例子的数量之差，然后对负的例子进行采样，从而得到相等的数字。然后，我们可以运行以下代码来构建一个具有平衡的正负示例的训练集：

```
train_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_EARLY]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

train_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_EARLY]-()
MATCH (author)-[:CO_AUTHOR_EARLY*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR_EARLY]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

train_missing_links = train_missing_links.drop_duplicates()
training_df = train_missing_links.append(train_existing_links, ignore_index=True)
training_df['label'] = training_df['label'].astype('category')
training_df = down_sample(training_df)
training_data = spark.createDataFrame(training_df)
```

我们现在将label列强制（coerce）为一个类别，其中1表示一对节点之间存在链接，0表示没有链接。我们可以通过运行以下代码来查看DataFrame中的数据：

```
training_data.show(n=5)

+-------+-------+-------+
| node1 | node2 | label |
+-------+-------+-------+
```

```
| 10019 | 28091 | 1     |
| 10170 | 51476 | 1     |
| 10259 | 17140 | 0     |
| 10259 | 26047 | 1     |
| 10293 | 71349 | 1     |
+-------+-------+-------+
```

结果向我们展示了节点对的列表，以及它们是否具有合著者关系；例如，节点10019和28091具有1个标签，表示有一个协作。

现在，让我们执行以下代码来检查DataFrame的内容摘要：

```
training_data.groupby("label").count().show()
```

结果如下：

```
+-------+-------+
| label | count |
+-------+-------+
| 0     | 81096 |
| 1     | 81096 |
+-------+-------+
```

我们用相同数量的positive和negative样本创建了我们的训练集。现在我们需要为测试集做同样的事情。以下代码将使用平衡的正负示例构建测试集：

```
test_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_LATE]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

test_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_LATE]-()
MATCH (author)-[:CO_AUTHOR*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

test_missing_links = test_missing_links.drop_duplicates()
test_df = test_missing_links.append(test_existing_links, ignore_index=True)
test_df['label'] = test_df['label'].astype('category')
test_df = down_sample(test_df)
test_data = spark.createDataFrame(test_df)
```

我们可以执行如下代码来查看DataFrame的内容：

```
test_data.groupby("label").count().show()
```

结果如下：

```
+-------+-------+
| label | count |
+-------+-------+
| 0     | 74128 |
| 1     | 74128 |
+-------+-------+
```

既然我们已经平衡了训练和测试数据集，那么让我们看看预测链接的方法。

如何预测缺少的链接

我们需要从一些基本假设开始，即数据中的哪些元素可以预测两位作者是否会在以后成为合著者。我们的假设因领域和问题而异，但在这种情况下，我们相信最具预测性的特征将与社区有关。我们将从以下假设开始，即以下要素增加了作者成为合著者的可能性：（猜想一些特征与最终的结果相关）

- 更多共同作者
- 作者之间潜在的三元关系
- 有更多任意类型关系的作者
- 同一社区的作者（这里的社区指的是community detection算法中的社区，并非物理上的实际的社区，下同）
- 同一个更紧密社区的作者

我们将基于我们的假设构建图特征，并使用这些特征训练二元分类器。二元分类是一种ML类型，其任务是根据规则预测元素所属的两个预定义组中的哪一个。我们使用分类器来根据分类规则预测一对作者是否有链接。对于我们的示例，值1表示有一个链接（有合著者关系），值0表示没有链接（没有合著者关系）。

我们将利用Spark中的随机森林实现我们的分类器。随机森林是一种用于分类、回归和其他任务的集成学习方法，如图8-7所示。

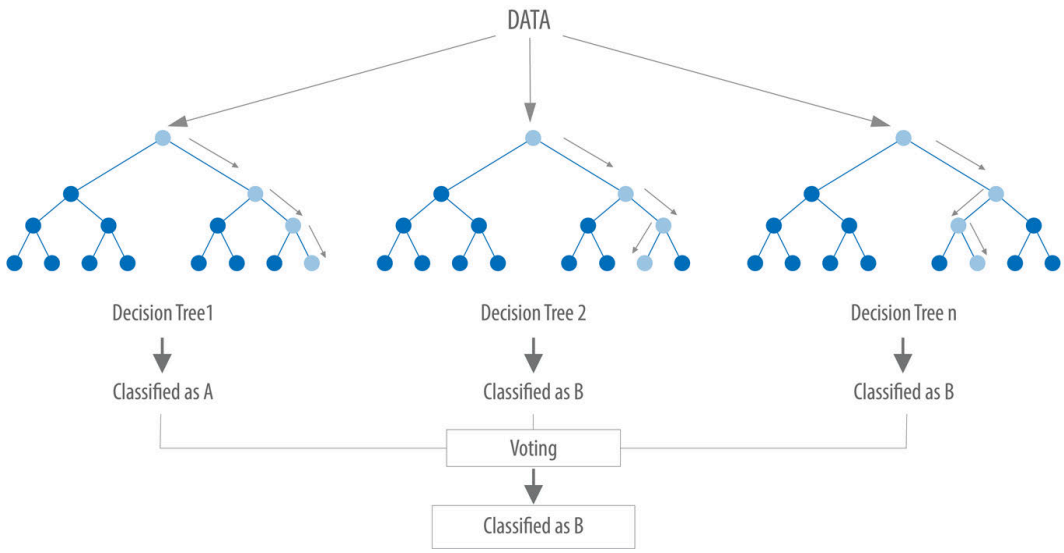


图8-7.一个随机森林建立一个决策树集合，然后将结果聚合为多数票（用于分类）或平均值（用于回归）。

我们的随机森林分类器将从我们训练的多个决策树中获取结果，并使用投票来预测分类。在这里的例子中，分类意味着是否存在链接（合著者关系）。现在让我们创建我们的工作流程。

创建机器学习的管道

我们将基于Spark中的随机森林分类器创建机器学习管道。这种方法非常适合，因为我们的数据集将由强特征(对结果影响较大的特征)和弱特征混合组成。虽然弱特征有时会有所帮助，但随机森林方法将确保我们不会创建只适合我们的训练数据的模型。

为了创建我们的ML管道（pipline），我们将传入一个特征列表作为字段变量，这些是我们的分类器将使用的特征。分类器期望将这些特征接收为一个单独的列features，因此我们使用VectorAssembler来将数据转换为所需的格式。

以下代码创建机器学习管道，并使用MLLib设置参数：

```
def create_pipeline(fields):
    assembler = VectorAssembler(inputCols=fields, outputCol="features")
    rf = RandomForestClassifier(labelCol="label", featuresCol="features",
                                numTrees=30, maxDepth=10)
    return Pipeline(stages=[assembler, rf])
```

RandomForestClassifier使用以下参数：

参数	意义
labelCol	包含要预测的变量的字段的名称；即一对节点是否具有链接
featuresCol	包含用于预测一对节点是否具有链接的变量的字段的名称
numTrees	形成随机森林的决策树的数目
maxDepth	决策树的最大深度

我们根据实验选择决策树的数量和深度。我们可以考虑超参数，比如可以调整以优化性能的算法设置。最佳超参数通常很难提前确定，而调整模型通常需要一些尝试和错误。

我们已经介绍了基础知识并建立了我们的管道，所以让我们开始创建模型并评估它的性能。

预测链接：基本的图特征

我们将首先创建一个简单的模型，该模型试图根据两位作者的Common authors、preferential attachment和the total union of neighbors中提取的特征来预测未来的协作：

- Common authors：查找两个作者之间的潜在三角形数。这抓住了这样一个观点，即两个有共同作者的作者将来可能会被介绍和合作。
- Preferential attachment（择优附着）通过将每对作者的合著者数量相乘，为每对作者生成一个分数。直觉是已经和其他人合作的作者更有可能开展合作。
- Total union of neighbors：查找每个作者的合著者总数，减去重复项。

在Neo4j中，我们可以使用Cypher查询计算这些值。以下函数将计算训练集的这些度量：

```
def apply_graphy_training_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
             size([(p1)-[:CO_AUTHOR_EARLY]-(a)-
```

```
                    [:CO_AUTHOR_EARLY]-(p2) | a]) AS commonAuthors,
            size((p1)-[:CO_AUTHOR_EARLY]-()) * size((p2)-
                    [:CO_AUTHOR_EARLY]-()) AS prefAttachment,
            size(apoc.coll.toSet(
                [(p1)-[:CO_AUTHOR_EARLY]-(a) | id(a)] +
                    [(p2)-[:CO_AUTHOR_EARLY]-(a) | id(a)]
            )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                                    for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                                    {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

下面的函数将为测试集计算它们：

```
def apply_graphy_test_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
            pair.node2 AS node2,
            size([(p1)-[:CO_AUTHOR]-(a)-[:CO_AUTHOR]-(p2) | a]) AS commonAuthors,
            size((p1)-[:CO_AUTHOR]-()) * size((p2)-[:CO_AUTHOR]-())
                    AS prefAttachment,
            size(apoc.coll.toSet(
                [(p1)-[:CO_AUTHOR]-(a) | id(a)] + [(p2)-[:CO_AUTHOR]-(a) | id(a)]
                )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                            for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                                    {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

这两个函数都采用一个DataFrame，其中包含列node1和node2中的节点对。然后我们构建一个包含这些对的映射数组，并计算每对节点的每个度量值。

在本章中，UNWIND子句对于获取大量节点对的集合，并在一个查询中返回它们的所有特征特别有用。

我们可以在Spark中使用以下代码将这些功能应用于我们的训练和测试数据帧：

```
training_data = apply_graphy_training_features(training_data)
test_data = apply_graphy_test_features(test_data)
```

让我们来探索一下我们训练集中的数据。以下代码将绘制commonAuthors频率的柱状图：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    common_authors = filtered.toPandas()["commonAuthors"]
    histogram = common_authors.value_counts().sort_index()
    histogram /= float(histogram.sum())
    histogram.plot(kind="bar", x='Common Authors', color="darkblue",
                   ax=axs[index], title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Common Authors")

plt.tight_layout()
plt.show()
```

我们可以看到图8-8中的图表。

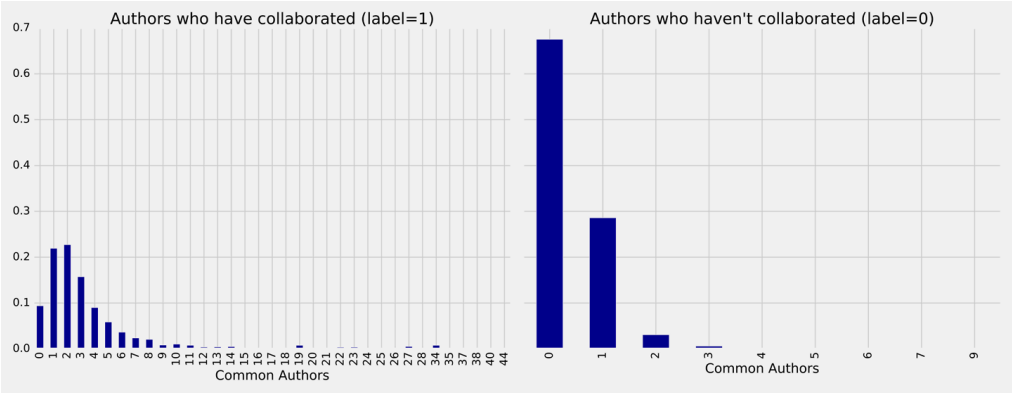


图8-8.commonAuthors的频率

左边是作者之间有合作时的commonAuthors频率，右边是没有合作时的commonAuthors频率。没有合作的commonAuthors的最大数量为9，但95%的值是1或0。这不奇怪，在那些没有合作过的人中，大多数人之间也没有其他的共同作者。对于那些合作过的人（左边），70%有少于5个共同作者，以有1~2个共同作者的情况居多。

现在我们要训练一个模型来预测丢失的链接。以下函数执行此操作：

```
def train_model(fields, training_data):
    pipeline = create_pipeline(fields)
    model = pipeline.fit(training_data)
    return model
```

我们将首先创建一个只使用commonAuthors的基本模型。我们可以通过运行以下代码来创建该模型：

```
basic_model = train_model(["commonAuthors"], training_data)
```

在我们的模型经过训练后，让我们检查它如何与一些dummy数据进行比较。以下代码根据commonAuthors的不同值评估代码：

```
eval_df = spark.createDataFrame(
    [(0,), (1,), (2,), (10,), (100,)],
    ['commonAuthors'])

(basic_model.transform(eval_df)
.select("commonAuthors", "probability", "prediction")
.show(truncate=False))
```

运行该代码将得到以下结果：

```
+---------------+----------------------------------------+------------+
| commonAuthors | probability                            | prediction |
+---------------+----------------------------------------+------------+
| 0             | [0.7540494940434322,0.24595050595656787] | 0.0        |
| 1             | [0.7540494940434322,0.24595050595656787] | 0.0        |
| 2             | [0.0536835525078107,0.9463164474921892]  | 1.0        |
| 10            | [0.0536835525078107,0.9463164474921892]  | 1.0        |
+---------------+----------------------------------------+------------+
```

如果我们的commonAuthors值小于2，有75%的概率作者之间不会有关系，所以我们的模型预测为0。如果我们的commonAuthors值为2或更大，则94%的概率表示作者之间存在关系，因此我们的模型预测1。

如何评估结果

现在让我们根据测试集评估我们的模型。尽管有几种方法可以评估模型的性能，但大多数方法都是根据一些基线预测指标得出的，如表8-1所示：

表8-1.预测指标

指标	公式	描述
准确率(accuracy)	TP+TN/(TP+TN+FP+FN)	被正确预测出来的样本占全部样本的比例。注意尤其当数据不均衡的时候，该结果会产生误导。比如一个图片数据集，有95只猫和5只狗，如果我们预测每个

		图片都是猫，就会取得95%的正确率，但其实没有预测对任何的一只狗。
精确率(precision)	TP/(TP+FP)	预测是正例中样本中，正确识别的比例。很低的精确率意味着很多false positive。没有false positive的模型，精确率是1.0。
召回率(recall，true positive rate, TPR)	TP/(TP+FN)	所有的正例的样本中被正确识别的比例。一个很低的召回率意味着更多的false negative。没有false negative的模型，召回率是1.0。
假正确率(false positive rate, FPR)	FP/(FP+TN)	所有的反例样本中，被错误识别的比例。很高的False positive值意味着更多的False positive。
ROC(Receiver operating characteristics，受试者操作特征)	X-Y图	ROC曲线是一个Recall(true positive rate)和假正确率(False positive rate)在不同的分类器阈值下的跑出的曲线。ROC曲线下从(0,0)到(1,1)曲线下的是AUC(Area under curve，AUC)。

其中，对于TP，FN，FP，TN的解释如下：

	预测值Positive	预测值Negative
实际值Positive	True Positive	False Negative
实际值Negative	False Positive	True Negative

对于二元分类，根据实际值和预测值的关系，共有四种分类：True Positive, True Negative, False Negative, False Positive。 它们的格式：(预测值==实际值）预测值。比如False Negative指的是预测分类是Negative，实际分类是Positive，所以预测正确与否是False。

ROC曲线：对于数据集，如果设置从0开始逐渐增大的概率阈值，意味着越来越多的样本被划分成为正例。在每一个阈值下，都得到一组（FPR，TPR）。最终可以形成曲线，叫做ROC，受试者操作特征。

我们将使用准确度（accuracy）、精确率（precision）、召回率（recall）和ROC曲线来评估我们的模型。准确度是一个粗略的度量，所以我们将重点提高我们的整体精度和召回度。我们将使用ROC曲线比较各个特征如何改变预测率。

根据我们的目标，我们可能希望采用不同的措施。例如，我们可能想消除所有疾病指标的假阴性（false negative），但我们不想把所有的预测都推到positive的结果中。对于不同的模型，我们可能会设置多个阈值，这些阈值将一些结果传递给对错误结果的再次似然检查。

降低分类阈值会导致更全面的positive结果，从而增加false positive和true positive。

让我们使用以下函数来计算这些预测指标：

```
def evaluate_model(model, test_data):

    # Execute the model against the test set
    predictions = model.transform(test_data)

    # Compute true positive, false positive, false negative counts
    tp = predictions[(predictions.label == 1) &
                     (predictions.prediction == 1)].count()
    fp = predictions[(predictions.label == 0) &
                     (predictions.prediction == 1)].count()
    fn = predictions[(predictions.label == 1) &
                     (predictions.prediction == 0)].count()

    # Compute recall and precision manually
    recall = float(tp) / (tp + fn)
    precision = float(tp) / (tp + fp)

    # Compute accuracy using Spark MLLib's binary classification evaluator
    accuracy = BinaryClassificationEvaluator().evaluate(predictions)

    # Compute false positive rate and true positive rate using sklearn functions
    labels = [row["label"] for row in predictions.select("label").collect()]
    preds = [row["probability"][1] for row in predictions.select
            ("probability").collect()]
    fpr, tpr, threshold = roc_curve(labels, preds)
    roc_auc = auc(fpr, tpr)

    return { "fpr": fpr, "tpr": tpr, "roc_auc": roc_auc, "accuracy": accuracy,
                "recall": recall, "precision": precision }
```

然后我们将编写一个函数，以更易于使用的格式显示结果：

```
def display_results(results):
    results = {k: v for k, v in results.items() if k not in
                     ["fpr", "tpr", "roc_auc"]}
return pd.DataFrame({"Measure": list(results.keys()),
                     "Score": list(results.values())})
```

我们可以使用此代码调用函数并显示结果：

```
basic_results = evaluate_model(basic_model, test_data)
display_results(basic_results)
```

常用作者模型的预测措施有：

```
+-----------+----------+
| measure   | score    |
+-----------+----------+
| accuracy  | 0.864457 |
| recall    | 0.753278 |
| precision | 0.968670 |
```

```
+-----------+----------+
```

这不是一个糟糕的开始，因为我们预测未来的合作只基于我们两个作者中的共同作者的数量。然而，如果我们把这些措施放在一起考虑的话，我们会得到一个更大的图景。例如，该模型的精度为0.968670，这意味着它非常擅长预测链接的存在。然而，我们的召回率是0.753278，这意味着它不善于预测何时链接不存在。

我们还可以使用以下函数绘制ROC曲线（true positive和false positive的相关性图）：

```
def create_roc_plot():
    plt.style.use('classic')
    fig = plt.figure(figsize=(13, 8))
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.rc('axes', prop_cycle=(cycler('color',
                   ['r', 'g', 'b', 'c', 'm', 'y', 'k'])))
    plt.plot([0, 1], [0, 1], linestyle='--', label='Random score
                   (AUC = 0.50)')
    return plt, fig

def add_curve(plt, title, fpr, tpr, roc):
    plt.plot(fpr, tpr, label=f"{title} (AUC = {roc:0.2})")
```

我们这样来调用它：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
               basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])
plt.legend(loc='lower right')
plt.show()
```

我们可以在图8-9中看到基本模型的ROC曲线。

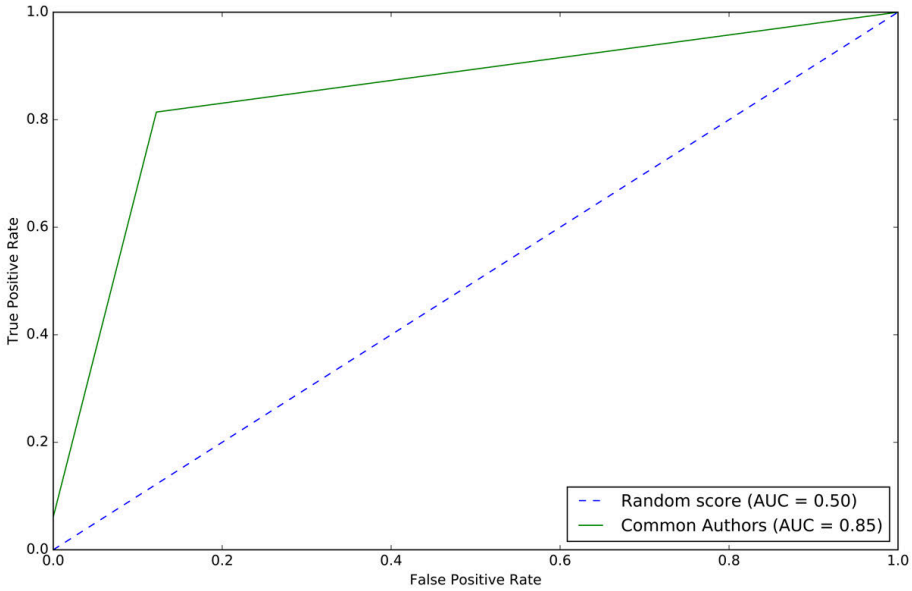


图8-9.基本模型的ROC曲线

常用作者模型给出曲线下0.86面积（AUC）评分。虽然这为我们提供了一个全面的预测指标，但我们需要图表（或其他指标）来评估这是否符合我们的目标。在图8-9中，我们看到当我们接近80%的true positive率时，我们的false positive率达到了大约20%。这在欺诈检测这样的情况下可能会有问题，在这些场景下，追查false positive的成本很高。

现在让我们使用其他的图表功能来看看我们是否可以改进我们的预测。在训练我们的模型之前，让我们看看数据是如何分布的。我们可以运行以下代码来显示每个图形功能的描述性统计信息：

```
(training_data.filter(training_data["label"]==1)
.describe()
.select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
.show())

(training_data.filter(training_data["label"]==0)
.describe()
.select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
.show())
```

我们可以在下表中看到运行这些代码位的结果：

```
+---------+-------------------+-------------------+-------------------+
| summary | commonAuthors     | prefAttachment    | totalNeighbors    |
+---------+-------------------+-------------------+-------------------+
```

```
| count   | 81096              | 81096              | 81096              |
| mean    | 3.5959233501035808 | 69.93537289138798  | 10.082408503502021 |
| stddev  | 4.715942231635516  | 171.47092255919472 | 8.44109970920685   |
| min     | 0                  | 1                  | 2                  |
| max     | 44                 | 3150               | 90                 |
+---------+--------------------+--------------------+————————————+

+---------+--------------------+------------------+--------------------+
| summary | commonAuthors      | prefAttachment   | totalNeighbors     |
+---------+--------------------+------------------+--------------------+
| count   | 81096              | 81096            | 81096              |
| mean    | 0.37666469369635985 | 48.18137762651672 | 12.97586810693499  |
| stddev  | 0.6194576095461857 | 94.92635344980489 | 10.082991078685803 |
| min     | 0                  | 1                | 1                  |
| max     | 9                  | 1849             | 89                 |
+---------+--------------------+------------------+--------------------+
```

链接（合著关系）和无链接（无合著关系）之间差异较大的功能应该更具预测性，因为两者之间的差距更大。prefAttachment的平均值对于合作过的作者比没有合作过的作者更高，这一差异在commonAuthors上更为显著。我们注意到totalNeighbors的值没有太大的差别，这可能意味着这个特征不能很好地预测。还有一个有趣的问题是prefAttachment的标准偏差大，以及它的最大值和最小值。这正是我们对具有集中集线器（超级连接者，superconnetors）的小世界网络的期望。

现在，让我们运行以下代码，来训练一个新的模型（graphy model），添加preferential attachment 和 total union of neighbors：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors"]
graphy_model = train_model(fields, training_data)
```

让我们评估一下这个模型，并显示结果：

```
graphy_results = evaluate_model(graphy_model, test_data)
display_results(graphy_results)
```

该模型（graphy model）的预测措施如下：

```
+-----------+----------+
| measure   | score    |
+-----------+----------+
| accuracy  | 0.978351 |
| recall    | 0.924226 |
| precision | 0.943795 |
+-----------+----------+
```

我们的准确度（accuracy）和召回率（recall）大大提高了，但是准确度下降了一点，我们仍然错误地分类了大约8%的链接。让我们绘制ROC曲线，并通过运行以下代码比较基本模型和图形模型：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
            basic_results["fpr"], basic_results["tpr"],
            basic_results["roc_auc"])

add_curve(plt, "Graphy",
            graphy_results["fpr"], graphy_results["tpr"],
            graphy_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

我们可以在图8-10中看到输出。

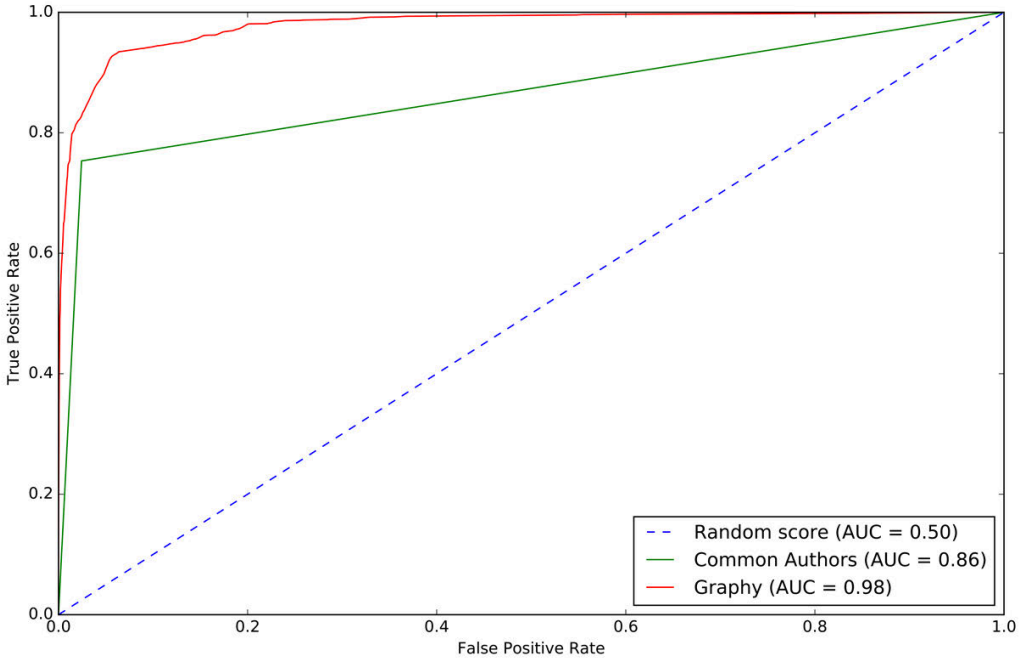


图8-10.graphy模型的ROC曲线

总的来说，我们似乎朝着正确的方向前进。可视化比较有助于这个，并了解不同的模型如何影响我们的结果。

现在我们有了多个特征，我们要评估哪些特征产生的差异最大。我们将使用特征重要性对不同特征对模型预测的影响进行排序。这使我们能够评估不同算法和统计数据对结果的影响。

为了计算特征重要性，Spark中的随机森林算法平均减少森林中所有树的杂质。杂质是随机分配标签错误的频率。

特征排名（feature rankings）是我们正在评估的一组特征进行比较，总是归一化为1。如果我们将一个特征排序，它的特征重要性是1.0，因为它对模型有100%的影响。

以下函数创建了一个图表，其中显示了最具影响力的功能：

```
def plot_feature_importance(fields, feature_importances):
    df = pd.DataFrame({"Feature": fields, "Importance": feature_importances})
    df = df.sort_values("Importance", ascending=False)
    ax = df.plot(kind='bar', x='Feature', y='Importance', legend=None)
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

我们这样调用它：

```
rf_model = graphy_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数的结果如图8-11所示。

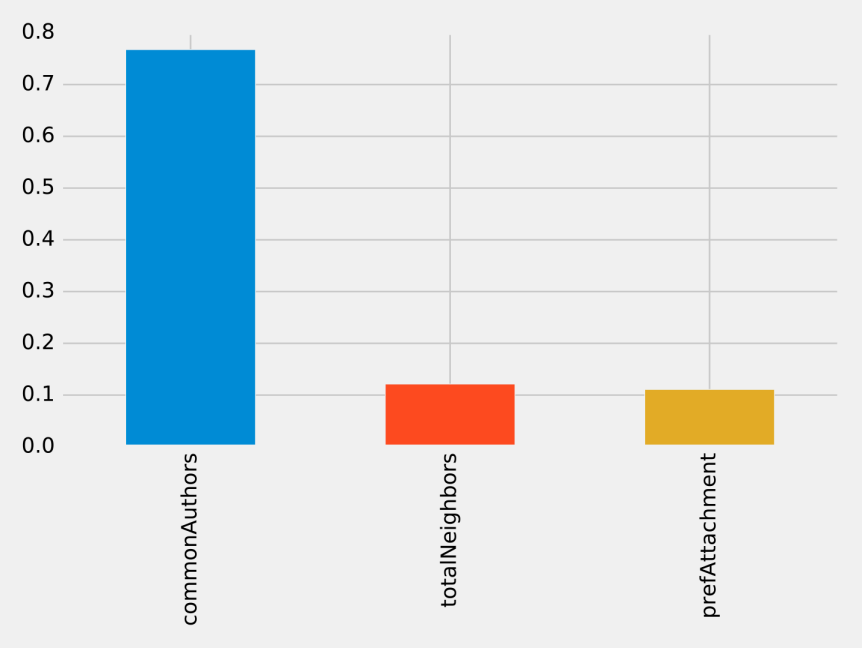


图8-11.特征重要性：图模型

在我们目前使用的三个特征中，commonAuthors在很大程度上是最重要的特征。

为了了解预测模型是如何创建的，我们可以使用spark-tree-plotting库来可视化随机林中的一个决策树。以下代码生成一个GraphViz文件：

```
from spark_tree_plotting import export_graphviz

dot_string = export_graphviz(rf_model.trees[0],
           featureNames=fields, categoryNames=[], classNames=["True", "False"],
           filled=True, roundedCorners=True, roundLeaves=True)

with open("/tmp/rf.dot", "w") as file:
    file.write(dot_string)
```

然后，我们可以通过从终端运行以下命令来生成该文件的可视化表示：

```
dot -Tpdf /tmp/rf.dot -o /tmp/rf.pdf
```

该命令的输出如图8-12所示。

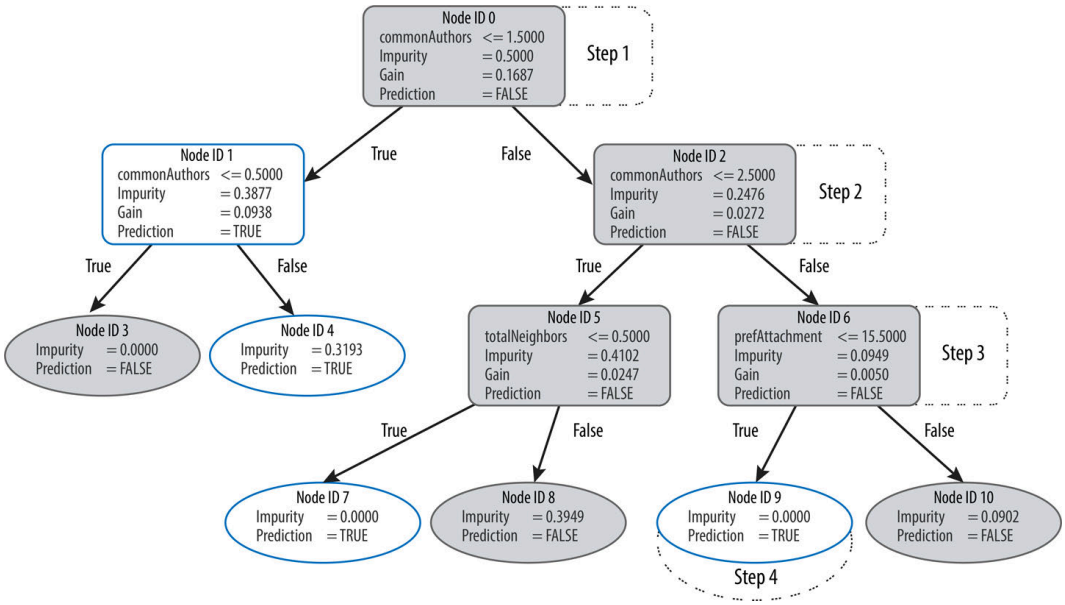


图8-12.可视化决策树

假设我们使用此决策树来预测一对具有以下特征的节点是否链接：

```
+---------------+----------------+----------------+
| commonAuthors | prefAttachment | totalNeighbors |
+---------------+----------------+----------------+
| 10            | 12             | 5              |
+---------------+----------------+----------------+
```

我们的随机森林通过几个步骤来建立一个预测：

1) 我们从Node ID 0开始，这里有超过1.5个commonAuthors，所以我们沿着False分支向下到Node ID 2。
2) 我们这里有超过2.5个commonAuthors，所以我们遵循False分支到Node ID 6。
3) 我们的prefAttachment得分低于15.5分，这将我们带到Node ID 9。
4) Node ID 9是这个决策树中的一个叶节点，这意味着我们不需要再检查任何条件。这个节点上的Pridiction(预测值，比如是True)就是决策树的预测。
5) 最后，随机森林根据这些决策树的集合对所预测的项目进行评估，并根据最可能的结果进行预测。

现在让我们看看添加更多的图特征。

预测链接：三角形和聚类系数

推荐方案通常是基于某种形式的三角形度量进行预测，因此让我们看看它们是否对我们的示例有进一步的帮助。我们可以通过执行以下查询来计算节点所属的三角形数及其聚类系数：

```
CALL algo.triangleCount('Author', 'CO_AUTHOR_EARLY', { write:true,
writeProperty:'trianglesTrain', clusteringCoefficientProperty:
'coefficientTrain'});

CALL algo.triangleCount('Author', 'CO_AUTHOR', { write:true,
writeProperty:'trianglesTest', clusteringCoefficientProperty:
'coefficientTest'});
```

以下函数将向我们的DataFrame添加这些功能：

```
def apply_triangles_features(data, triangles_prop, coefficient_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           apoc.coll.min([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS minTriangles,
           apoc.coll.max([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS maxTriangles,
           apoc.coll.min([p1[$coefficientProp], p2[$coefficientProp]])
                                              AS minCoefficient,
           apoc.coll.max([p1[$coefficientProp], p2[$coefficientProp]])
                                              AS maxCoefficient
    """
    params = {
           "pairs": [{"node1": row["node1"], "node2": row["node2"]}
                           for row in data.collect()],
           "trianglesProp": triangles_prop,
           "coefficientProp": coefficient_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2”])
```

注意，我们已经为三角形计数和聚类系数算法使用了最小和最大前缀。在无向图中，我们需要一种方法来防止我们的模型根据两个作者被传入的次序进行学习。为了做到这一点，我们已经用最小计数和最大计数将这些特征分割开来。

我们可以使用以下代码将此功能应用于我们的训练和测试DataFrame：

```
training_data = apply_triangles_features(training_data, "trianglesTrain", "coefficientTrain")

test_data = apply_triangles_features(test_data, "trianglesTest", "coefficientTest")
```

并运行此代码来显示每个三角形特征的描述性统计信息：

```
(training_data.filter(training_data["label"]==1)
.describe()
.select("summary", "minTriangles", "maxTriangles",
                  "minCoefficient", "maxCoefficient")
.show())

(training_data.filter(training_data["label"]==0)
.describe()
.select("summary", "minTriangles", "maxTriangles", "minCoefficient",
                                                  "maxCoefficient")
.show())
```

我们可以在下表中看到运行这些代码的结果。

```
+---------+--------------------+-------------------+--------------------+--------------------+
| summary | minTriangles       | maxTriangles      | minCoefficient     | maxCoefficient     |
+---------+--------------------+-------------------+--------------------+--------------------+
| count   | 81096              | 81096             | 81096              | 81096              |
| mean    | 19.478260333431983 | 27.73590559337082 | 0.5703773654487051 | 0.8453786164620439 |
| stddev  | 65.7615282768483   | 74.01896188921927 | 0.3614610553659958 | 0.2939681857356519 |
| min     | 0                  | 0                 | 0.0                | 0.0                |
| max     | 622                | 785               | 1.0                | 1.0                |
+---------+--------------------+-------------------+--------------------+--------------------+

+---------+-------------------+--------------------+---------------------+--------------------+
| summary | minTriangles      | maxTriangles       | minCoefficient      | maxCoefficient     |
+---------+-------------------+--------------------+---------------------+--------------------+
| count   | 81096             | 81096              | 81096               | 81096              |
| mean    | 5.754661142349808 | 35.651980368945445 | 0.49048921333297446 | 0.860283935358397  |
| stddev  | 20.639236521699   | 85.82843448272624  | 0.3684138346533951  | 0.2578219623967906 |
| min     | 0                 | 0                  | 0.0                 | 0.0                |
| max     | 617               | 785                | 1.0                 | 1.0                |
+---------+-------------------+--------------------+---------------------+--------------------+
```

注意，在这个比较中，coauthorship和no-coauthorship数据之间没有太大的区别。这可能意味着这些特征并不是具有预测性的。

我们可以通过运行以下代码来训练其他模型：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
       "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient"]

triangle_model = train_model(fields, training_data)
```

现在让我们评估模型并显示结果：

```
triangle_results = evaluate_model(triangle_model, test_data)
display_results(triangle_results)
```

triangles模型的预测指标如下表所示：

```
+-----------+----------+
| measure   | score    |
+-----------+----------+
| accuracy  | 0.992924 |
| recall    | 0.965384 |
| precision | 0.958582 |
+-----------+----------+
```

通过将每个新特征添加到以前的模型中，我们的预测措施得到了很好的提高。我们将三角形模型添加到ROC曲线图中，代码如下：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
           basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
           graphy_results["fpr"], graphy_results["tpr"],
                                  graphy_results["roc_auc"])

add_curve(plt, "Triangles",
           triangle_results["fpr"], triangle_results["tpr"],
                                  triangle_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

我们可以在图8-13中看到输出。

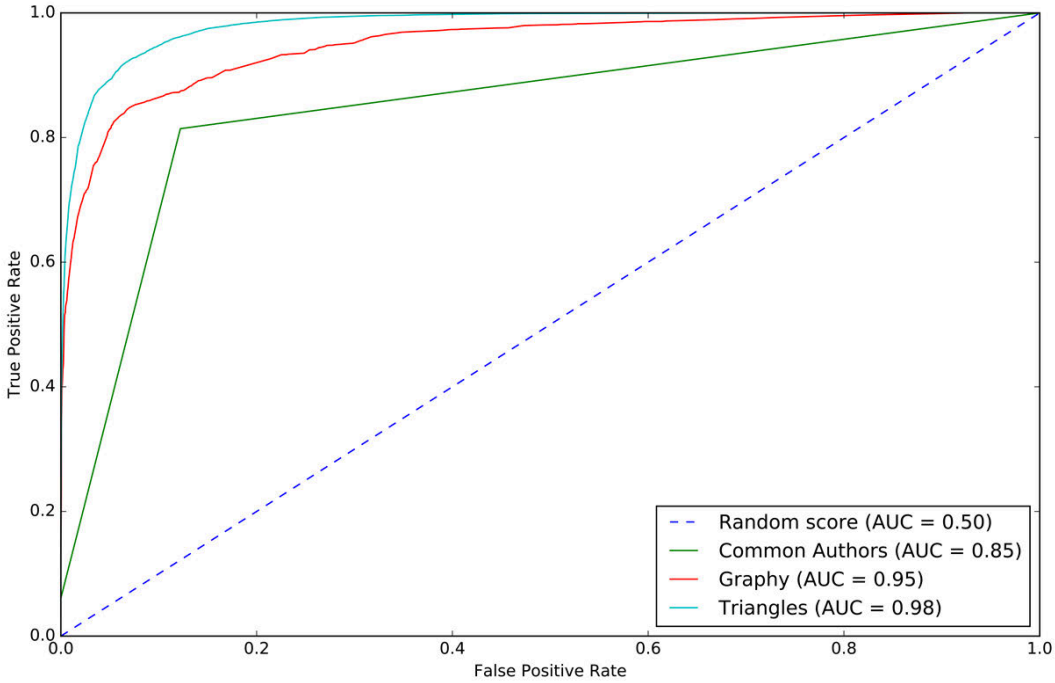


图8-13.triangles模型的ROC曲线

我们的模型已经得到了普遍的改进，并且我们在预测措施方面处于90%+的预测能力。提高通常是很困难的，因为最易获得的成果已经拿到了，但仍有改进的空间。让我们看看重要的特征是如何改变的：

```
rf_model = triangle_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数的结果如图8-14所示。

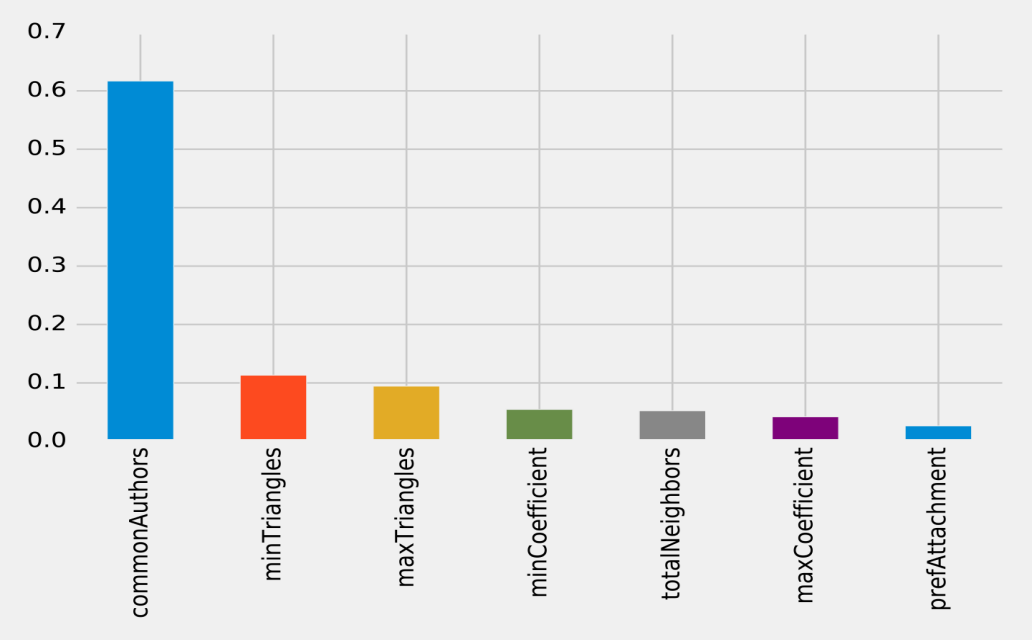


图8-14.特征的重要性：三角形（triangles）模型

common authors特征仍然对我们的模型有最大的单一影响。也许我们需要看看新的领域，看看当我们添加社区信息时会发生什么。

预测链接：社区检测

我们假设在同一个社区中的节点如果还没有连接的话，它们之间更有可能有联系。此外，我们相信社区越紧密，联系越可能。

首先，我们将使用Neo4j中的LPA来计算更粗粒度的社区。我们通过运行以下查询来实现这一点，该查询将社区存储在训练集的partitionTrain属性中，以及测试集的partitionTest属性中：

```
CALL algo.labelPropagation("Author", "CO_AUTHOR_EARLY", "BOTH",
{partitionProperty: "partitionTrain"});

CALL algo.labelPropagation("Author", "CO_AUTHOR", "BOTH",
{partitionProperty: "partitionTest"});
```

我们还将使用Louvain算法计算更细粒度的组。Louvain算法返回中间聚类，我们将这些聚类中最小的聚类存储在训练集的louvainTrain属性中，测试集的louvainTest属性中：

```
CALL algo.louvain.stream("Author", "CO_AUTHOR_EARLY",
                        {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTrain = smallestCommunity;

CALL algo.louvain.stream("Author", "CO_AUTHOR",
                        {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTest = smallestCommunity;
```

现在我们将创建以下函数来返回这些算法的值：

```
def apply_community_features(data, partition_prop, louvain_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           CASE WHEN p1[$partitionProp] = p2[$partitionProp] THEN
                      1 ELSE 0 END AS samePartition,
           CASE WHEN p1[$louvainProp] = p2[$louvainProp] THEN
                      1 ELSE 0 END AS sameLouvain
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]} for
                           row in data.collect()],
        "partitionProp": partition_prop,
        "louvainProp": louvain_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

我们可以使用以下代码将此功能应用于Spark中的训练和测试DataFrame：

```
training_data = apply_community_features(training_data,
                                         "partitionTrain", "louvainTrain")
test_data = apply_community_features(test_data, "partitionTest", "louvainTest")
```

我们可以运行此代码来查看节点对是否属于同一分区：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('samePartition',
                F.when(F.col("samePartition") == 0, "False")
           .otherwise("True"))
           .groupby("samePartition")
```

```
            .agg(F.count("label").alias("count"))
            .select("samePartition", "count")
            .toPandas())
    values.set_index("samePartition", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
            title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Partition")

plt.tight_layout()
plt.show()
```

我们可以在图8-15中看到运行该代码的结果。

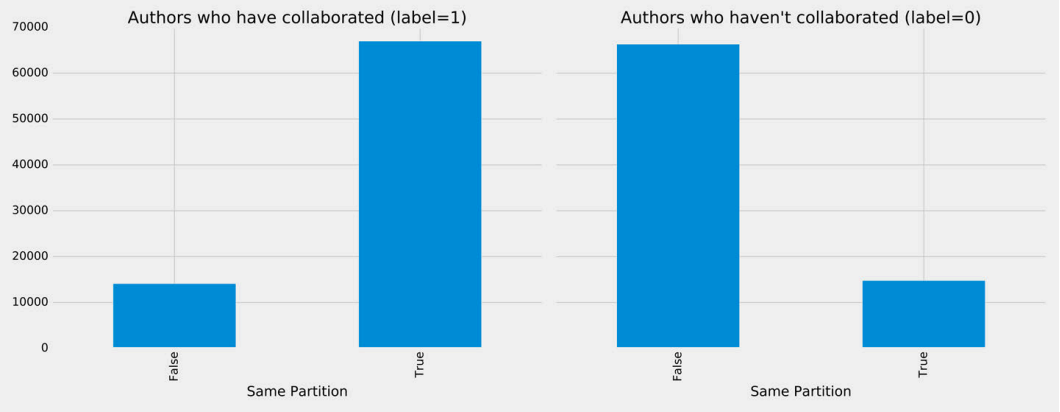


图8-15 同一分区

看起来这个特征可以很好地预测那些合作过的作者比那些没有合作过的作者更有可能在同一个分区中。我们可以通过运行以下代码为Louvain聚类做同样的事情：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('sameLouvain',
                F.when(F.col("sameLouvain") == 0, "False")
                                    .otherwise("True"))
                .groupby("sameLouvain")
                .agg(F.count("label").alias("count"))
                .select("sameLouvain", "count")
                .toPandas())
    values.set_index("sameLouvain", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
                title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Louvain")
plt.tight_layout()
plt.show()
```

我们可以在图8-16中看到运行该代码的结果。

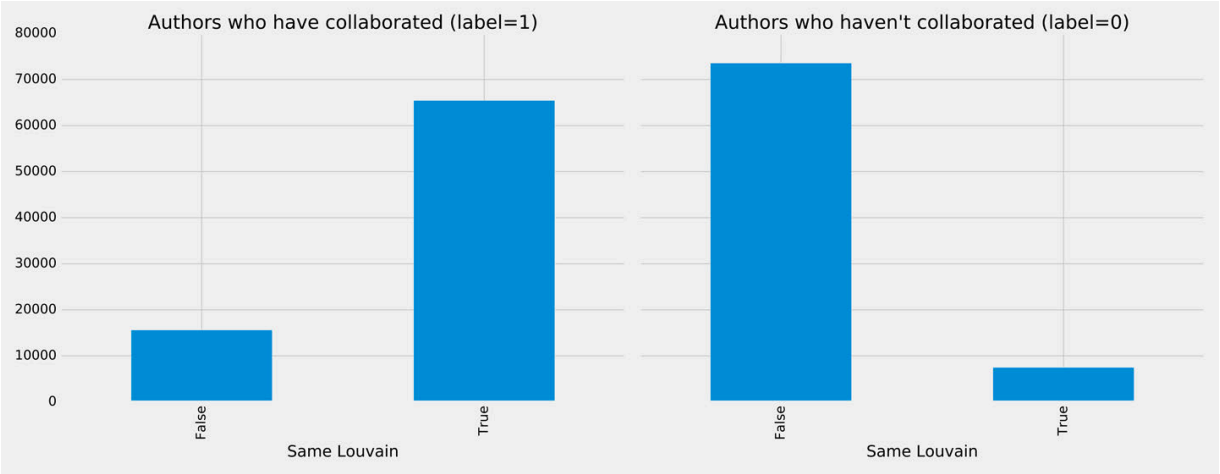


图8-16。同一个Louvain聚类

看起来这个特征是非常具有预测性的，而且合作过的作者很可能在同一个集群中，而那些没有合作过的作者则不太可能在同一个集群中。

我们可以通过运行以下代码来训练其他模型：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient",
          "samePartition", "sameLouvain"]
community_model = train_model(fields, training_data)
```

现在让我们评估模型并显示结果：

```
community_results = evaluate_model(community_model, test_data)
display_results(community_results)
```

社区模型（community model）的预测指标如下有：

```
+-----------+----------+
| measure   | score    |
+-----------+----------+
| accuracy  | 0.995771 |
| recall    | 0.957088 |
| precision | 0.978674 |
+-----------+----------+
```

我们的一些度量指标已经改进，因此为了进行比较，让我们通过运行以下代码为所有模型绘制ROC曲线：

```
plt, fig = create_roc_plot()
```

```
add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
          graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
          triangle_results["roc_auc"])

add_curve(plt, "Community",
          community_results["fpr"], community_results["tpr"],
          community_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

我们可以在图8-17中看到输出。

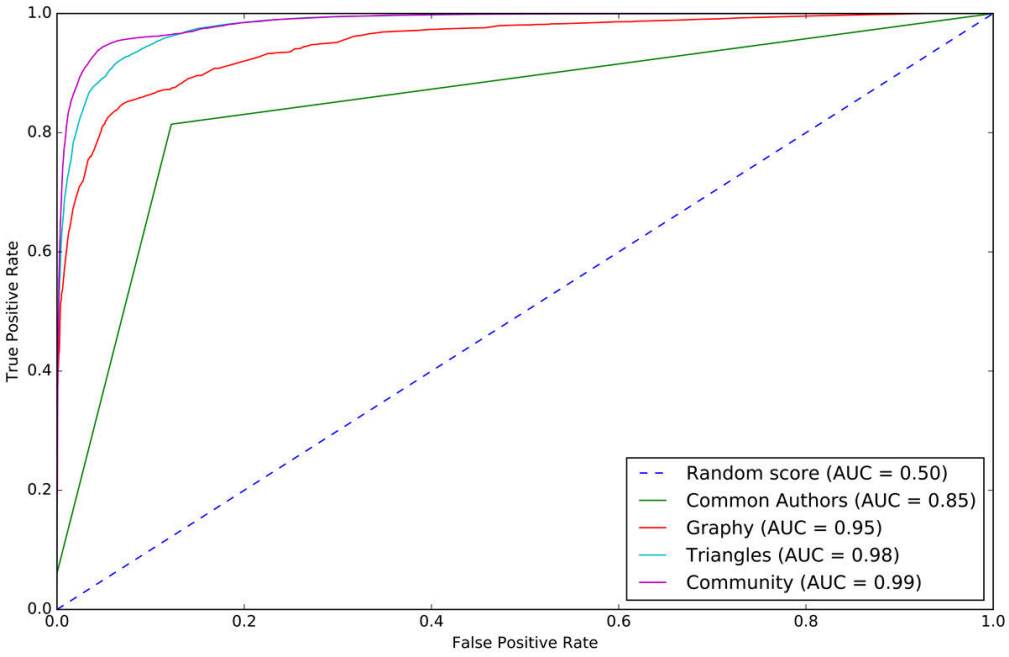


图8-17。社区模型的ROC曲线

我们可以通过添加社区模型看到改进，因此让我们看看哪些是最重要的特征：

```
rf_model = community_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数的结果如图8-18所示。

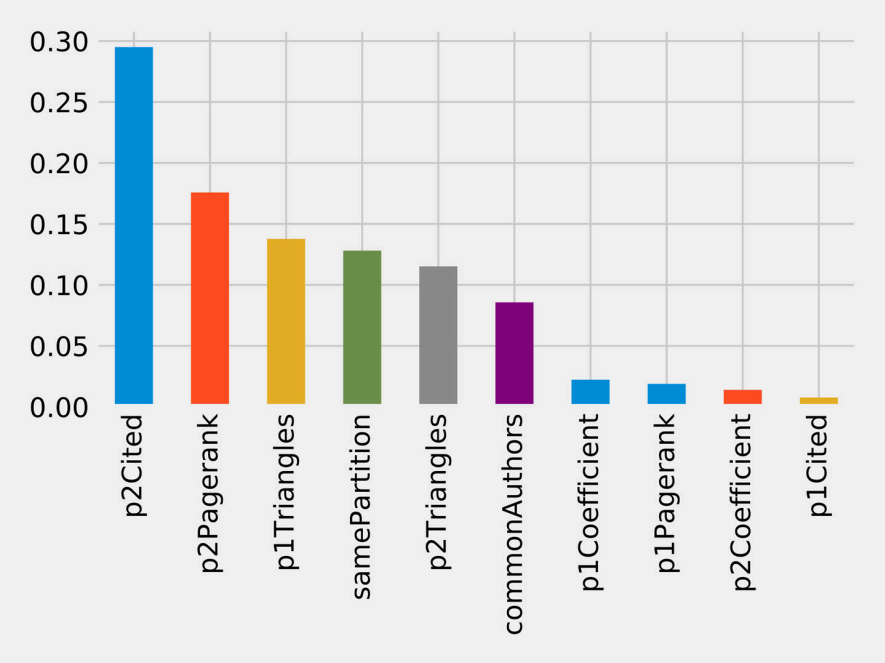


图8-18。特征重要性：社区模式

尽管common authors模型总体上非常重要，但最好避免有一个过度占主导地位的元素，它可能会扭曲对新数据的预测。社区检测算法中的所有特征在我们的最后一个模型中有很大的影响，这有助于完善我们的预测方法。

在我们的例子中，我们已经看到简单的基于图的特征是一个很好的开始，然后随着我们添加更多的基于graphy和图算法的特征，我们继续改进我们的预测措施。我们现在有了一个好的、平衡的模型来预测合著关系。

使用图进行关联特征提取可以显著提高我们的预测。理想的图特征和算法取决于数据的属性，包括网络域和图形形状。我们建议首先考虑数据中的预测元素，并在微调之前测试具有不同类型连接特征的假设。

读者练习

有几个领域需要调查，以及调查构建其他模型的方法。以下是一些进一步探索的想法：

- 我们的会议数据模型对我们没有包括的会议数据的预测性如何？
- 测试新数据时，删除某些功能会发生什么？
- 训练和测试的年数划分是否会影响我们的预测？
- 此数据集在论文之间也有引文；我们可以使用该数据生成不同的特征或预测未来引文吗？

总结

在本章中，我们研究了使用图形特征和算法来增强机器学习。我们介绍了一些初步概念，然后通过一个集成Neo4j和Apache Spark进行预测链接的详细示例。我们说明了如何通过随机森林分类器模型，并结合各种类型的关联特征来改进我们的结果。

本书总结

在本书中，我们介绍了图概念以及处理平台和分析。然后，我们介绍了在Apache Spark和Neo4j中如何使用图算法的许多实际示例，最后我们将介绍图形如何增强机器学习。

图算法是分析现实系统的强大后盾，从防止欺诈、优化呼叫路由到预测流感的传播。我们希望你加入我们，利用当今高度互联的数据，开发自己独特的解决方案。

附录A

补充信息和资源

在本节中，我们将快速介绍可能对某些读者有所帮助的其他信息。我们将研究其他类型的算法，将数据导入Neo4j的另一种方法，以及另一个procedure库。还有一些资源用于查找数据集、平台帮助和培训。

其他算法

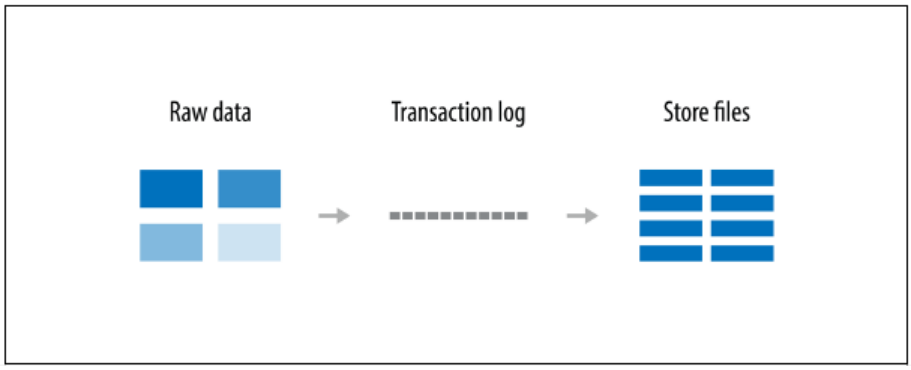
许多算法可用于图数据。在这本书中，我们重点介绍了那些最能代表经典图形算法的算法，以及那些对应用程序开发人员最有用的算法。有些算法，如着色(coloring)和启发式(heuristics)算法，被省略了，因为它们更关注学术领域，要么是很容易得到的。

其他算法，如基于边缘的社区检测（edge-based community detection），很有趣，但还没有在Neo4j或Apache Spark中实现。我们预计，随着图形分析的使用量的增加，两个平台中使用的图形算法将会增加。

还有一些算法与图一起使用，但本质上并不是严格的图。例如，我们在第8章中研究了机器学习环境中使用的一些算法。另一个值得注意的领域是相似性算法，它通常应用于推荐和链接预测。相似性算法通过使用不同的方法来比较节点属性等项，从而找出最相似的节点。

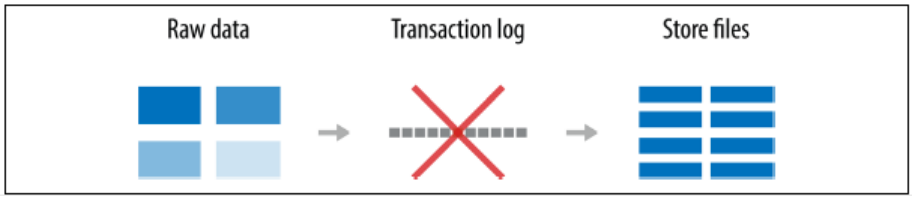
Neo4j批量数据导入和Yelp

Cypher查询语言使用事务性方法将数据导入Neo4j。图A-1说明了此过程的高层次的概述。



图A-1.基于Cypher的导入

虽然此方法在增量数据加载或千万级大容量加载时效果良好，但在导入初始大容量数据集时，Neo4j导入工具是更好的选择。该工具直接创建存储文件，跳过事务日志，如图A-2所示。



图A-2：使用Neo4j导入工具

Neo4j导入工具处理csv文件，并期望这些文件具有特定的头文件。图A-3显示了该工具可以处理的csv文件的示例。

Nodes

id:ID(User)	name
1234	Bob
1235	Alice
1236	Erika

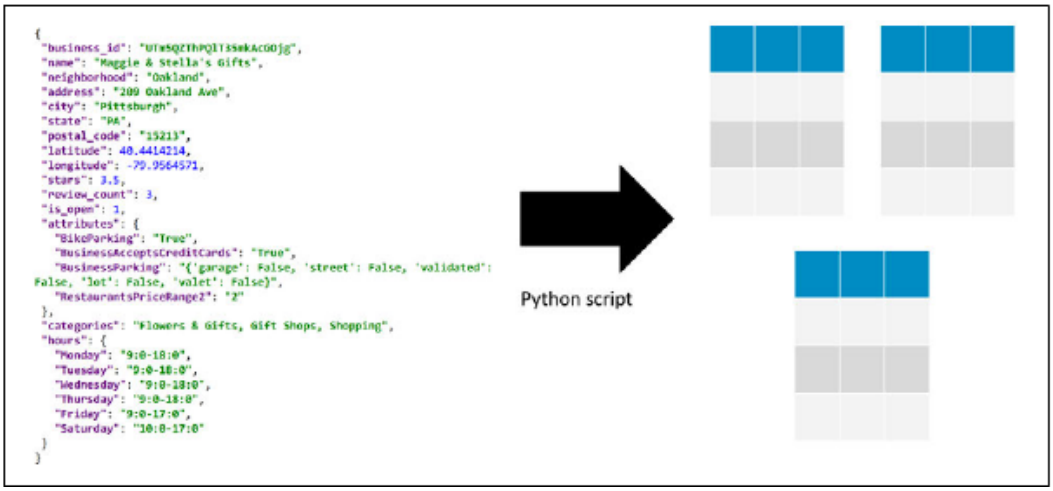
id:ID(Review)	text	stars
678	Awesome	3
679	Mediocre	2
680	Really bad	1

Relationships

:START_ID(User)	:END_ID(Review)
1234	678
1235	679
1236	680

图A-3.Neo4j导入处理的csv文件格式

Yelp数据集的大小意味着Neo4j导入工具是将数据转换为Neo4j的最佳选择。数据采用JSON格式，因此首先我们需要将其转换为Neo4j导入工具期望的格式。图A-4显示了我们需要转换的JSON的一个示例。



图A-4.将JSON转换为CSV

使用python，我们可以创建一个简单的脚本来将数据转换为csv文件。一旦我们将数据转换成这种格式，我们就可以将其导入Neo4j中。详细说明如何实现这一点的说明在书的资源库中。

APOC和其他Neo4j工具

Awesome Procedures on Cypher（APOC）是一个包含450多个过程（procedure）和功能（function）的库，用于帮助完成常见任务，如数据集成、数据清理和数据转换以及常规帮助功能。APOC是Neo4j的标准库。

Neo4j还有其他工具可以与他们的图算法库结合使用，例如用于无代码探索的算法“游乐场”应用程序。这些可以在他们的图算法开发网站上找到。

找到数据集

找到一个符合测试目标或假设的图数据集是一项挑战。除了回顾研究论文外，还考虑探索网络数据集的索引：

- SNAP（Stanford Network Analysis Project，SNAP）包括多个数据集以及相关论文和使用指南。
- ICON（Colorado Index of Complex Networks，ICON）是网络科学各个领域的研究质量网络数据集的可搜索索引。

- KONECT（Koblenz Network Collection ，KONECT）包括各种类型的大型网络数据集，以便进行网络科学研究。大多数数据集将需要一些信息来将其转换为更有用的格式。

Apache Spark和Neo4j平台在线资源

ApacheSpark和Neo4j平台有许多在线资源。如果您有具体问题，我们鼓励您联系他们各自的社区：

- 对于一般的Spark问题，请在Spark Community页面上订阅users@spark.apache.org。
- 对于GraphFrames问题，请使用Github issue tracker。
- 对于所有Neo4j问题（包括图算法），请访问在线Neo4j Community 。

培训信息

有很多优秀的资源可以用来开始图分析。搜索关于图算法、网络科学和网络分析的课程或书籍将发现许多选项。在线学习的几个很好的例子包括：

- Coursera上的Python课程： Applied Social Network Analysis
- YouTube上的Leonid Zhukov的Social Network Analysis系列
- 斯坦福大学的网络分析课程，包括视频讲座、阅读列表和其他资源
- Complexity Explorer提供的在线复杂科学的课程

关于原著作者

Mark Needham是Neo4j的图倡导者和开发人员关系工程师，他致力于帮助用户接受图和Neo4j，为具有挑战性的数据问题构建复杂的解决方案。Mark在图数据方面拥有深厚的专业知识，曾帮助构建Neo4j的因果集群系统。他在自己的热门博客https://markhneedham.com/blog/和tweets@markhneedham上写下了自己成为一名图设计师的经历。

Amy E. Hodler是Neo4j的网络科学爱好者和人工智能和图分析项目经理，她提倡使用图分析来揭示现实网络中的结构并预测动态行为。Amy帮助团队采用新颖的方法在EDS、Microsoft、Hewlett-Packard（HP）、Hitachi IoT和Cray Inc.等公司创造新的机会。Amy热爱科学和艺术，对复杂性研究和图论充满兴趣。她的推特是@amyhodler.

版权页标记

图算法（Graph Algorithms）封面上的动物是欧洲花园蜘蛛（Araneus diadematus），一种常见的欧洲和北美蜘蛛，在那里它无意中被欧洲殖民者引入。

欧洲花园中的蜘蛛不到一英寸长，斑驳的棕色，有苍白的斑纹，其中一些在它的背面是一个小的十字，这给了蜘蛛一个普通的名字“十字蜘蛛”。这些蜘蛛在它们的范围内很常见，而且大多数是在夏末被看到，这个时候它们长大到最大尺寸并开始织网。

欧洲的花园蜘蛛是圆形织布者，这意味着它们会旋转一个圆形的网，在其中捕捉小昆虫的猎物。网络经常在晚上被消耗和响应，以确保和保持其有效性。当蜘蛛看不到网的时候，它的一条腿靠在一条与网络相连的“信号线”上，在这个“信号线”上的移动提醒蜘蛛有挣扎的猎物。然后，蜘蛛快速移动，咬住猎物杀死它，并向它注入特殊的酶，使之能够被消耗（enable consumption）。当它们的网被掠食者扰乱或无意中受到干扰时，欧洲的花园蜘蛛会用它们的腿摇动它们的网，然后落在地上的丝线上。当危险过去时，蜘蛛会用这个线重新回到它的网中。

它们生活一年：在春天孵化后，蜘蛛在夏天成熟，在一年中晚些时候交配。雄性小心接近雌性，因为雌性有时会杀死并吃掉雄性。交配后，雌蜘蛛在秋天死去之前为自己的卵编织一个致密的茧。

由于这些蜘蛛很常见，并且很好地适应了人类受干扰的栖息地，所以人们对它们进行了很好的研究。1973年，两个名为Arabella和Anita的雌性花园蜘蛛在美国宇航局的太空实验室轨道飞行器上进行了实验，以测试零重力对蜘蛛网结构的影响。在适应失重环境的最初阶段之后，Arabella建立了一个部分的网，然后是一个完全成形的圆形网。

O’Reilly封面上的许多动物都濒临灭绝，它们对世界都很重要。

封面图片是Karen Montgomery的彩色插图，基于Meyers Kleines Lexicon的黑白版画。封面字体是Gilroy和Guardian Sans。文本字体为Adobe Minion Pro；标题字体为Adobe Myriad Condensed；代码字体为Dalton Maag的Ubuntu Mono。