目录

[1. 绪论 3](#_Toc476752629)

[1.1. 研究背景 3](#_Toc476752630)

[1.2. 论文工作 4](#_Toc476752631)

[1.3. 论文组织 4](#_Toc476752632)

[2. 图计算相关工作 4](#_Toc476752633)

[2.1. 问题定义 4](#_Toc476752634)

[2.2. 关键技术 6](#_Toc476752635)

[2.2.1. 图的划分 7](#_Toc476752636)

[2.2.2. 编程模型 7](#_Toc476752637)

[2.2.3. 计算泛型 7](#_Toc476752638)

[2.3. 常见系统 7](#_Toc476752639)

[2.4. 存在问题 7](#_Toc476752640)

[2.5. 本章小结 7](#_Toc476752641)

[3. 基于状态更新的流式图计算模型 7](#_Toc476752642)

[3.1. 图算法特征分析 7](#_Toc476752643)

[3.2. 模型定义 10](#_Toc476752644)

[3.3. 状态存储和更新 12](#_Toc476752645)

[3.4. 模型应用举例 15](#_Toc476752646)

[3.5. 本章小结 19](#_Toc476752647)

[4. GraphFlow系统的设计与实现 19](#_Toc476752648)

[4.1. 系统架构 19](#_Toc476752649)

[4.2. 模型实现 19](#_Toc476752650)

[4.3. 算法实现 19](#_Toc476752651)

[4.4. 本章小结 19](#_Toc476752652)

[5. 实验与分析 19](#_Toc476752653)

[5.1. 实验环境 19](#_Toc476752654)

[5.2. 实验结果 19](#_Toc476752655)

[5.3. 本章小结 19](#_Toc476752656)

[6. 结束语 19](#_Toc476752657)

[6.1. 工作总结 19](#_Toc476752658)

[6.2. 下一步工作 19](#_Toc476752659)

[参考文献 19](#_Toc476752660)

[发表文章目录 19](#_Toc476752661)

[作者简历 20](#_Toc476752662)

[7. 致谢 20](#_Toc476752663)

# 绪论

## 研究背景

图是计算机科学中常用的一类数据结构，它很好的表达了数据之间的关联性。现实世界中有很多数据都可以抽象成图数据，例如Web网页之间的链接、社交人物之间的互动以及买卖双方的交易都可以抽象成彼此关联而形成的图。而随着互联网的快速发展，图数据的总量也在急剧增加。如截至2016年第四季度Facebook包含了18.6亿个活跃用户，每个用户平均好友155个；web链接图顶点数达到Ｔ级，边的个数达到Ｐ级[1]。

因为图数据能够很好的表达数据之间的关联性和聚集情况，因此针对图数据表达的关联关系可以挖掘出很多有用信息。比如，通过为购物者之间的关系建模，就能很快找到口味相似的用户，并为之推荐商品；在社交网络中，通过传播关系发现意见领袖。图算法及相关的处理框架已经广泛运用在社交分析、商品推荐、舆论监测、欺诈检测等各个领域。

处理这些海量动态的图数据也对现有的图计算模型提出了挑战。一方面，这种超大规模的图数据很难一次性的全部导入内存中进行处理，即使能够借助外存一批一批的处理图数据，也使得计算延迟显著增加；另一方面，这些数据又是动态变化，实时更新的，现有的图计算模型要能够在这种动态的数据集上进行增量计算。

现有的成熟的图计算系统如Google Pregel[2], Spark GraphX[3], 这些图计算框架都采用了分布式的集群和BSP（Bulk Synchronized Parallel）[4]消息同步机制来处理图数据。然而这些系统都是在静态的图数据结构上进行的离线批量处理,即每次针对整体的图进行计算，当图动态变化时，需要在变化后的整个图上重新计算一遍。这使得用户等待周期长，无法满足实时计算的要求，也浪费了系统资源。[5]

因此针对图数据不断变化的情况，提出了很多在动态图上直接进行计算的方法。针对这类动态图计算的问题，大致可以分为两类：估计计算和准确计算。对于估计计算，大部分的算法是希望通过采样的方式来降低时间和空间开销，并通过特定的采样方法来减少真实值和实测值之间的差距。 如文献[6][7][8]研究了在流图上如何通过设计采样规则来估计图中三角形数目；文献[9][10][11] 通过将原始的图信息转化为简单的数据结构来保存图中元素，降低了内存消耗。虽然估计算法能够在一定程度上节约了内存和计算开销，但其估计的误差在实际的生产环境中往往变得不可控制，文献[12]指出，针对大体量的无法全部载入内存的图数据，近似算法的错误率在95%-133%之间。对于准确计算，现有的KineoGraph[13]和IncGraph[5]提出采用增量计算的模型进行实时计算，然而这种增量式的更新是串行执行的，实时性有限。SpecGraph[14]虽然在上述两个增量模型的基础上有所改进，提出了基于推测机制的并发更新模型，然而该模型中假设顶点的状态只依赖于顶点当前接收的信息，而与顶点之前的旧状态无关。这种假设使得系统的适用性差，很多算法不仅跟顶点接收消息有关，还跟顶点的旧状态有关，因此模型的表达能力有限。

## 论文工作

针对动态图计算的实时性和准确性要求，本文在上述已有的研究基础之上，提出了**基于状态更新的动态图计算模型**，能够在原有图状态上并发的计算增量信息对状态的影响，而无需在整个图上重新计算，同时通过控制更新影响范围来提高并发性，实现状态的并发更新。本文的工作主要有以下3点：

（1）分析现有的图计算的特点，抽象出在流式场景下图算法的典型特征。从影响范围和计算次数两个维度分析了4个典型的图算法：节点度分布（DD, Degree Distribution），三角形数目（TC, Triangle Count），单源点最短路径（SSSP, Single Source Shortest Path），和PageRank（PR）。

（2）根据这些流式图算法的特点，建立基于状态更新的动态图计算模型，该模型能够允许用户自定义状态，并且采用并发更新的方式来快速计算结果。

（3）在上述建立的基于状态更新的动态图计算模型基础上实现了典型的动态图算法：TC和连通子图（CC, Connected Components），并且从正确性和实时性对算法进行评估，结果表明：基于状态更新的动态图计算模型构建的算法能够得到较为正确的计算结果，计算偏差在1%以内；而且算法能够在5ms内返回增量计算的结果，符合实时性要求。

## 论文组织

论文的后续章节，按照如下的方式进行组织：

# 图计算相关工作

## 问题定义

图是计算机中一种常用的数据结构，常常用来表达个体之间的关联性。现实世界中很多情景都可以用图来进行表达。如在社交领域，微博、微信、Facebook等社交平台中每个个体可以抽象成图中的一个点，个体之间的关联可以抽象成边，这样形成了超大规模的关系网络图，针对这样的关系网络，可以进行社交分析，为其推荐好友或者侦测社区；在电商领域，个体对商品的浏览记录和购买记录也可以抽象成图数据，可以根据用户的这些记录为她们推荐商品；在万维网中，网页以及网页之间的相互链接可以形成巨型网页图，根据链接信息来对网页的重要性进行排名；由此可见，图数据的运用是非常广泛的。

图计算相关的问题，可以从数据特性和算法特点两个角度来进行定义和分析。

从数据特性来看，随着信息的传播和互联网的蓬勃发展，这些图数据呈现“海量”和“动态”两个特性。“海量”是指图数据的体量非常庞大，仅以Facebook为例，其在2016年第四季度的月用户活跃量已达18.6亿，平均拥有155个好友；“动态”是指图数据是在不断动态变化的，如Facebook在2010年第四季度的月用户活跃量在6亿左右，6年时间增加到18.6亿，平均增速高达2亿/年。详细数据见图2-1所示。



图 ‑1 Facebook月活跃量统计图

数据来源：<https://www.statista.com>

从算法特点来看，图计算具有“局部性差”和“迭代计算”的特点。图结构很好的表述了实体之间的关联，然而现实生活中这种关联往往呈现Power Law规则，满足这种规则的图数据分布极不均匀，这给分布式存储和计算带来巨大困难，导致节点之间的通信代价提高。此外，图相关的算法往往不是一次计算就可以获得最终结果，而是需要通过多次迭代计算，达到收敛条件或者最大迭代次数时算法才运行结束。

针对图数据的海量和动态特性，以及图算法的局部性差和迭代计算的特点，下面我们列举并分析了现有的图计算的关键技术点。

## 关键技术

图2-2从底层细节、模型、算法、系统和应用五个角度分别阐述图计算的关键技术。在本节中我们重点选取了底层细节和模型两个角度来阐述，而在算法和系统层面，我们会在其他章节讨论。



图 ‑2 图计算技术概览

从底层细节角度来看，分布式环境下的图计算可以分为两步：划分和计算。即通过某种划分规则，将原来超大规模的图数据合理的分配到各个计算节点上，然后各个计算节点在该子图上进行计算。一方面这种划分要保证负载均衡，即希望能够将整个图数据均匀的划分到各个计算节点上；另一方面又希望这种划分能够较好的维持图数据的关联性，即相互关联的节点希望它们能够分配到同一个计算节点上，以减少节点之间的通信。

从模型角度来看，图计算模型可以分解为编程模型和计算泛型：编程模型是面向图计算系统的应用开发者，他们应该按照何种编程模式来编写图的处理逻辑，如常用Vertex-Centric、SG、GSA都是以顶点为中心的编程模型等；而计算模型是指图计算引擎采用同步模式还是异步模式来执行计算，我们知道图计算往往需要经过多轮的迭代计算，如果在本轮的迭代过程中，节点的变化能够立即被其他节点看到并使用，则称这种计算模式为异步执行模型，相反如果所有节点的变化只有在下一轮的迭代过程中才能被其他节点看见和使用，则称这种计算模式为同步执行模型。

下面，本文将从图的划分、编程模型和计算泛型这三个点来详细阐述图计算中的关键技术。

### 图的划分

对于分布式环境下的图计算，首先要解决的问题是如何将海量的图数据均衡的分配到各个计算节点上。而图数据之间的关联性使得这种划分往往比一般的非图数据变得复杂。其一考虑到节点负载的均衡性，希望能够将图数据平均分配到每个计算节点；其二考虑到图计算中邻接点的相互通信，又希望相互邻接点能够划分到一起来减少不同计算节点之间的通信代价。如果划分的不好，不仅会使某个计算节点的计算任务加重，还会使节点之间的通信代价显著提升，而图计算往往又是多轮迭代反复计算的，这使得这种不合理的划分所造成的影响会显著放大，因此，图的划分是图计算中至关重要的工作。

现在较为常用的图划分方式有切边法（Edge-Cut）和切点法（Vertex-Cut）。切边法是对边进行切割，然后将这两个顶点分配到不同的计算节点上，如图2-3左图所示，通过两条切线将原图分割成3个子图，这3个子图位于3个不同的计算节点上，图中顶点上的编号即为该顶点所在的计算节点编号。切点法是对顶点进行切割，然后将被切割的顶点的多个备份分配到不同的计算节点上，如图2-3右图所示，通过将中心节点切割，将该节点的3个副本分别分配到3台不同的计算节点上。由此可见，切边法使得图的边被切割，图的顶点只可能存在集群中的一个计算节点上；而切点法使得图的顶点被切割，但图的边只可能存在集群中的一个计算节点上。

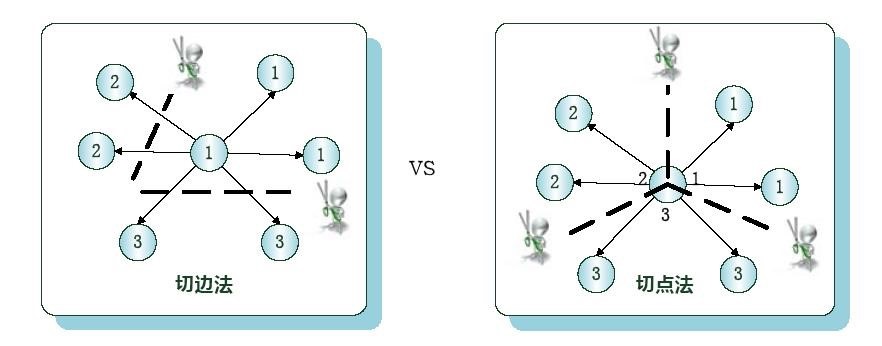


图 ‑3 切边法和切点法

图划分的问题

### 编程模型

### 计算泛型

## 常见系统

现有很多成熟的图计算框架和系统，本文按照图数据的特性，将这些系统分为批处理图系统和流处理图系统，即批处理图系统处理的图数据是静态的，而且只有等到全部数据处理完毕之后才能反馈计算结果，适合离线图计算情景； 而流处理图系统处理的图数据是动态的，针对动态变化的图数据，能够及时反馈中间计算结果，适合在线图计算情景。

### 批处理图系统

图计算一般需要多次迭代，计算过程依赖于顶点之间的通信，而传统的MapReduce模型更倾向于处理彼此独立的任务，因此其开源实现的Hadoop为代表的传统面向数据并行（Data-Parallel）的计算模型难以对图计算提供高效的支持。[17]

为了解决海量图计算问题，Google公司提出了基于BSP(Bulk Synchronous Parallel)思想的大规模分布式图计算平台Pregel。[18]专门解决网页链接分析、社交数据挖掘等图计算问题。Pregel使用了以顶点为中心的计算模型,将整个计算过程分解成由若干个顺序运行的超步（superstep），在每个超步中，活跃的节点(active vertex)将将接收上个超步中其他节点发送过来的消息，并执行用户自定义的计算函数，改变自己的状态，同时将更新的状态再发送给其他节点，这些消息会在下个超步中被其他节点接收并处理，然后该点进入不活跃状态（inactive vertex）。不活跃的点在下个超步中接收到其他节点的消息会变得活跃，反之如果没有接收其他节点的消息，将继续保持不活跃的状态，也不会向其他节点发送消息。超步内，各个节点可以并行处理，而超步之间会对消息进行同步，通过这样以超步为单位的方式迭代运行，直至所有节点都变得不活跃或没有新的消息产生。用户只需要自己定义超步内节点的计算逻辑，即可实现计算功能。计算模型如下图所示：



图 2‑4 Pregel计算模型

继Pregel之后，一大批以BSP为底层计算模型的系统涌现，如Spark GraphX，Flink Gelly，Giraph，Hama等底层都是依托于同步计算的BSP模型。

### 流处理图系统

流计算最初出现在20世纪末，是为了解决数据量大，数据不能完全保存在内存中的计算问题。通过将数据集转化为数据流，就可以依次处理流过的数据，而无需在内存中存储数据集，这样就解决了数据量大的问题。一开始，能够解决的问题领域也比较有限，主要都是针对数据流的计算。数据流上的算法研究成果有很多，如computation of frequency moments [1]，histograms[2]，Wavelet transforms[3]，distinct elements count[4]等

由于图数据和图算法的复杂性，图流计算晚几年才出现。到目前为止，图流上进行的图算法有Connectivity，Triangle Count[5]，PageRank[6]，Matching等

无论是图流计算还是数据流计算，研究都集中在估计计算。估计计算主要有以下优点：a. 可以不存或者存储少量的数据，相对于整个图数据来说。这点对于Internet和社交网络等方面的分析有重大意义，因为它们的顶点和边数量庞大，甚至可能存在264条边，完全存储数据集是不现实的。b. 更新时间短，流算法对每个流数据要进行相应的计算和更新，而采用估计的流算法更新和计算时间短。在分析流计算估计模型之前，我们先看一下流模型。

依据流中数据的表达形式，主要有两种典型的模型[7]：

* Cash Register Model : 流中的每一项仅仅是数据集中一项，比如在distinct elements count 中，每一项就是一个数。数据集中的每一项以任意顺序形成数据流。
* Turnstile Model : 在该模型中，我们有一个初始化为空的集合D，流中的数据由两项组成，一项是数据集的某一项，另一项是一个标志位，可以对集合D进行动态改变。例如，流图中的每一项为(x, U)，如果U为＋，就将x加入D，如果U为－，就将x从D删除。这种模型更符合现实的一些场景，近期的参考文献也将这种模型成为动态流(dynamic stream)。

现在我们转向流的估计计算模型，下列模型主要是针对图流进行分析，当然一些模型可以应用于数据流，这里不细区分。依据模型是否存储流中的数据，将模型分为两类：

采样(sampling) : 该模型完全不存储流中的数据，在流经过时，对流数据进行采样和计算。该模型的内存消耗主要在采样线程上，要采n个样本，就要起n个线程对流进行采样，即每个线程只能采一个样。依赖该模型的算法的最终计算结果，取决于图结构和采样结果。例如，采用该模型的典型算法为triangle count，见[5]，该论文定义了两类图流arbitrary stream和incidence stream，并分别提出了1-pass和3-pass采样算法，以arbitrary stream的1-pass为例，该算法可以通过令采样次数，使得的概率为，其中，，

概要(summarization) : 该模型通过将图结构转化为简单的数据结构，保存图中元素，使得消耗内存量远远小于原图。同时，结构随数据流进行更新。依据做概要的方式，可以将该模型分为三类：

* 生成树(spanner) [8, 9]: 针对图流来说，该模型仅保留边的一个集合(set)，即将图结构转化为集合(set)，可以用于判断图的连通性(connectivity)和图中任意两点的距离。
* 稀疏图(sparsifier) [10, 11]: 针对图流来说，该模型仅保留边的一个权重矩阵，即将图结构转化为矩阵，可以用于估计图中每个连通分量(connected components)的权重。
* 草图(sketch) : 该模型又分为线性草图(linear sketch)[12, 13]和同构草图(homomorphic sketch)[14]，针对图流来说，线性草图仅保留点的一个向量和边的一个向量，即将图结构转化向量，因为丢失了图结构，线性草图支持的查询有限，如边权重和点的入度等；同构草图保留多个顶点矩阵，即将图结构转化为多个矩阵，保留了图结构，可以支持的查询有顶点查询，边查询，路径查询和子图查询等。

基于流的特征，我们可以将流图计算扩展到实时计算上，基本上上述的模型与算法均可以应用在实时计算。这样，就可以满足现代社会日益增长的实时计算的需求，研究有意义。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 系统类型 | 系统 | 数据划分 | 编程模型 | 计算模型 | 特点 |
| **离线计算**  图数据是静态的 | Spark GraphX | 节点随机哈希 | Vertex-centric | BSP |  |
| Flink Gelly | 节点随机哈希 | Vertex-centric | BSP |  |
| Pregel | 节点随机哈希 | Vertex-centric | BSP | 鼻祖 |
| GraphLab | 节点随机哈希 | GAS | 异步执行 | C++编写 |
| PowerGraph | 节点随机哈希，  Edge-Cut,  Vertex-Cut,  启发式v-cut | GAS | 混合 | 目前主流图计算系统中效率最高的 |
| Giraph | 节点随机哈希 | Vertex-centric | BSP | 建立在Hadoop平台上 |
| GraphChi(单机) | Edge-Cut,v排序 |  | 异步执行 |  |
| **增量计算**  图数据由原始数据和增量数据组成 | Percolator |  |  |  |  |
| Kineograph |  | Vertex-centric | 变化传播 |  |
| DryadInc |  |  | 结果缓存复用 |  |
| CBP |  |  |  | 建立在Hadoop平台上 |
| Incoop |  |  |  |
| IncMR |  |  |  |

## 存在问题

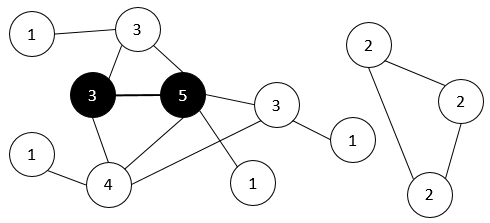
## 本章小结

# 基于状态更新的流式图计算模型

## 图算法特征分析

（1）Degree Distribution

DD算法是统计无向图中各个节点的度。如图2展示了在流式场景下如何统计节点的度。图2中的每个圆圈表示一个节点，圆圈之间的连线表示一条边，圆圈内部的数字表示当前时刻该节点的度。在某一时刻节点的度分布情况如图2(a)所示，当增加一条边（图2(b)中标黑的两个节点之间增加一条边）后，这条边对应的源顶点和目标顶点的度分别增加1。增加该条边所带来的影响如图2(b)所示。由此可见，在DD算法中，增加的一条边只影响了这条边的源点和目标点。

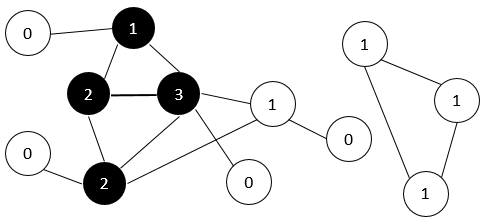
 

（a） （b）

图 2 节点度分布

（2）Triangle Count

TC算法是用来统计无向图中的不同三角形的数目。该算法在复杂网络分析、链接标签和推荐等多个领域中都是非常基础重要的度量，也是一些诸如复杂网络、聚集系数等图运算中的基本方法。图3展示了在流式场景下如何统计三角形的数目。图3中节点内的数字表示该节点所拥有的三角形的数目。图3(a)表示在某一时刻三角形的分布情况，当在三角形数目为0和1的节点之间增加一条边时，它会使得这两个节点的所有公共邻接点的三角形的数目都增加1，而这两个节点的三角形数目增加N,N为公共邻接点的数目。由此可见，对于TC算法，增加的一条边不仅影响了这条边的两个顶点，还影响了这两个顶点的公共邻接点。

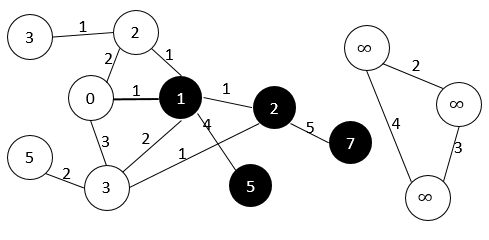


（a） （b）

图 3 三角形数目分布

（3）Single Source Shortest Path

SSSP算法是在有向图或无向图中，给定一个源点，求解这个源点到图中其它各个点的最短路径问题。最短路径问题是图论算法中的经典问题，也是诸如路径规划、物流规划、GPS导航、社交网络等现实世界中许多应用的基本问题。[16]图4展示了在流式场景下的SSSP的求解方法。图中边上的数字表示这两个节点之间的距离，节点内的数字表示当前时刻源点到该节点的最短距离，数字为0的节点为源点。图4(a)反应了某一时刻源点到各个节点的最短路径情况，图4(b)表示新增一条边（节点内数字为0的点和节点内数字为3的点新增了一条边，边上的权重为1）之后的情况。由图4(b)可知，新增的这条边会将影响沿着某条路径传播下去（即图中的0->1->2->7和图中的0->1->5），如图中新增的权重为1的这条边，会将原来值为3的节点更改为1，并且该节点的后续节点的值都会被更新。

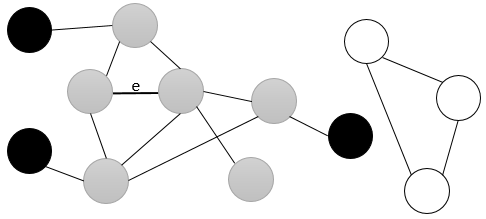
 

（a） （b）

图 4 单源点最短路径

（4）PageRank

PR算法[17]是基于网页链接分析来计算各个网页的重要程度。假设网页A引用了网页B，那么A就将一定的分数贡献给了B，该分数（即PageRank值，简称PR值）就是网页的重要程度的体现。而网页之间是相互引用的，因此经过若干次的迭代之后，网页的得分会趋于稳定，这个分数就反应了该网页的重要程度。图5(a)展示了某一时刻各个节点的PR值，当新增一条边e之后，图5(b)展示了这条边带来的影响：灰色的节点表示新增的这条边直接影响这些节点的值，随后灰色的节点又将这些影响继续往外传播给黑色节点，经过若干次的迭代之后各个节点的PR值保持稳定，算法运行结束。由此可见，对于PageRank算法，当图中新增一条边时，这条边会影响这条边所在的连通子图内的所有节点。

（a） （b）

图 5 PageRank

（5）总结

当新增一条边时，本文从影响范围、计算次数两个维度来分析这4个算法的特点。影响范围是指新增加的这条边可能会影响到哪些节点，而计算次数是指这种影响是否会被计算多次，例如在DD算法中，新增加一条边只需要将这条边的源点和目标点对应的度数加1，且这种运算只需要执行一次，而对于PR算法，新增加一条边时，这条边的源点和目标点的输出贡献将会发生变化，因此会首先影响它们的所有邻接点，这些邻接点在下一次的传播中会继续它们的邻接点，经过多次迭代计算之后各个节点的PR值会趋于稳定，在迭代计算的过程中，每个节点可能参与多次计算。

表1 图算法特征表

|  |  |  |
| --- | --- | --- |
|  | **影响范围** | **计算次数** |
| **DD** | 影响新增这条边的源点和目标点 | 被影响的节点只参与计算一次 |
| **TC** | 影响新增这条边的源点和目标点，以及这两个点的公共邻接点 | 被影响的节点只参与计算一次 |
| **SSSP** | 以这条边的某个节点为起点，沿着某条路径往其他节点传播影响 | 被影响的节点可能会参与计算多次 |
| **PR** | 影响这条边的源点和目标点所在的整个连通子图内的所有节点 | 被影响的节点一般会参与计算多次 |

## 模型定义

传统的图计算模型（例如BSP模型）中，图数据是静态的，即在计算的过程中图数据不会发生变化；本文提出的基于状态更新的动态图计算模型，能够很好的解决动态图计算问题，它将动态图在每个时刻抽象成一个状态（State），将流动的图数据抽象成一系列事件流（Event Stream），事件（Event）触发了图由一个状态转换（Transform）成另一个状态。



图 6 基于状态更新的动态图计算模型

基于状态更新的动态图计算模型有如下几个定义：

（1）状态（State）：反应了图当前的特征信息，这些特征信息可以以顶点为单位进行体现，也可以由用户自定义的特征信息来体现，状态是由因子（Factor）组成，因子是指组成状态的基本单位，如状态可以以顶点的方式组织，那么这里的因子就是顶点。需要注意的是，状态反应了用户的关注点，虽然是根据流动的图数据而动态计算生成的，但并不等价于图数据本身，即状态不直接存储原始的图数据，而只存储用户关心的图的某些特征信息。这使得系统无需存储庞大的原始图数据，只需要存储设计精巧的状态信息即可反应图的特征信息。例如当统计图的边数，此时State可以设计为一个计数器，该计数器反应了当前时刻流入系统中的图的边数，每次新增或者删除边时，增加或减少这个计数器的值，即可实时反应当前图的边数信息。

在这里，本文将状态抽象成一个接口，该接口的基本方法表见表2，用户可以扩展该接口来实现更加复杂的状态信息。

表2 State接口方法表

|  |  |
| --- | --- |
| **方法签名** | **方法作用** |
| State GET-STATE(Factor) | 获取指定因子的状态 |
| SET-STATE(Factor, State) | 设置指定因子的状态 |
| SET-STATE(State) | 向图中添加一个状态 |
| Map GET-STATE() | 获取整个图的状态 |

（2）事件（Event）：触发图由T1时刻的State1转换为T2时刻的State2的事件，例如在T2时刻新增加了一条边，将使得图由State1经过某种运算得到State2。事件是由事件值（Event Value）和事件类型(Event Type)组成。如“增加一条边e(v1,v2)”这个事件中，e(v1,v2)是事件的值，“增加”是事件的类型。一般来说，事件的值分为两种：（顶点编号，顶点的值）和（边起点，边终点，边值）；而事件的类型分为三种：新增（ADD），删除（DELETE），更新（UPDATE）。这样总共可以组成6种事件：新增边，删除边，更新边；新增顶点，删除顶点，更新顶点。这6种事件基本涵盖了所有的图变化的情形。事件的接口方法表见表3.

表3 Event接口方法表

|  |  |
| --- | --- |
| **方法签名** | **方法作用** |
| Value GET-VALUE(Event) | 获取指定事件的值 |
| Type GET-TYPE(Event) | 获取指定事件的类型 |

（3）转换（Transform）：由事件触发的图的更新过程，即图是如何根据相应的事件来由State1转换成State2。如图6所示，在T1时刻图的状态为State1，在T2时刻，图接收了一条边e，这个事件会触发图状态转换函数（Transform），将图的状态转换为State2。转换函数是动态图计算模型中的计算逻辑，详细定义了图如何根据到达的事件，从一个状态转变成另外一个状态，可以称之为状态更新的图计算模型的驱动程序，驱动图从一系列的事件流转换成一系列对应的状态流。

## 状态存储和更新

（1）状态类型

基于状态更新的动态图计算模型中，一个核心问题是状态如何存储和更新。状态是从用户的视角来进行设定的。即用户关心什么数据，就可以将该数据设置为图的一个状态，这些状态可以以顶点为单位进行保存：图的状态由各个顶点的状态组成，也可以以边或者其他的方式来组织。相比较传统的顶点编程模型或边编程模型来说，用一个高度可自定义的状态能够直接反应用户关心的结果，使得模型的表达能力更强。

根据上述3.2节中图算法分析，在图计算中大致分为两类状态：独立状态和关联状态。所谓独立状态，是指状态内的各个因子之间是独立的，一个因子的状态的变化不会引起其他因子的状态的变化，如DD算法就是属于独立状态范围，每增加一条边，这个事件只会影响增加这条边的两个节点，不会影响到其他的节点；所谓关联状态，是指状态内的各个因子之间相互关联，一个因子的状态的变化会影响到其它因子状态的变化，诸如TC、SSSP、PR算法中增加一条边，不仅会影响增加这条边的两个顶点的状态，还会影响到这两个顶点的公共邻接点，甚至整个连通子图内的所有顶点。可以用下图表示这两种情况。



图 7 状态分类图

（2）独立状态的存储和更新

在独立状态中，因为状态内的各个因子之间不会相互影响，因此独立状态可以分布式的存储和并发的更新。即可以按照状态的组织形式，分布式的存储在多个节点上，而且每个节点上的状态都可以同时进行更新，并向用户实时反馈更新结果。这样充分利用了分布式的特点，提高存储和计算效率。状态的并发更新过程如图8所示：



图 8 独立状态的存储和更新过程图

该图演示了独立状态的并行存储和更新的过程。系统接收到事件流（Event Stream）之后，将事件流按照某种分片规则（即特定的图的划分算法），分发到不同的计算节点上（如图所示的No.1, No.2, No.3这3个计算节点），然后分别在各个计算节点上独立进行状态更新（如图所示对应计算节点的状态从State1转换到了State2），这样使得图由T1时刻State 1，更新成了T2时刻的State 2，注意到这种更新过程充分利用了分布式的优势，多个节点同时独立进行更新，提高了计算效率。

（3）关联状态的存储和更新

在关联状态中，一个因子的状态的变化会影响到其它因子状态的变化，因此，多个事件触发的更新可能会影响同一个因子，引起更新冲突问题。解决更新冲突的方法有很多，最为简单的方式是将多个事件的更新串行化，即对两个事件A和B，事件A先触发更新，A更新完毕后事件B再触发更新，在事件A触发更新期间，其他任何事件都不得触发更新，以免引起更新冲突问题。诸如IncGraph[5]就是采用这样的更新模型，这使得即使不会发生更新冲突的两个事件也不能同时更新，无法充分利用多机并行的优势。因此本文提出了两种解决更新冲突的方法：基于分区的并行更新策略和基于细粒度锁的并行更行策略。

基于分区的并行更新策略是将原来的图划分成若干个子图，使得子图内部的节点之间联系比较紧密，子图之间的节点之间几乎没有边相连或者联系较少。这样可以假设子图内节点更新的影响范围只限于子图内部，不会传播到其它子图中节点。因此子图与子图之间的更新可以同时进行，而子图内部的更新则需要串行进行，这样在一定程度上能够提高更新的并行度。如图9所示。



图 9 基于分区的并行更新策略

图9展示了基于分区的并行更新策略，按照连通性将原来的图分成如图9所示的四个连通子图，在每个连通子图的更新是串行的，如在左上角子图中a节点和f节点的更新需串行进行，而连通子图之间的节点的更新是并行的，如a节点、e节点、k节点和f节点的更新可以同时进行。如果分区策略划分的好，可以充分利用分布式的优势，实现多个分区并行更新。

基于分区的并行更新策略需要谨慎的选择子图划分算法，该分区算法要能够很好的将原来的大图切分成若干个子图，保证子图之间节点的联系是松散的，子图内部的节点之间的联系是紧密耦合的。关于图划分的问题，现有的研究工作也很多，Ioanna Filippidou和Yannis Kotidis[18]提出了一种基于精简生成树结构的图分割算法，它不仅能对任意的演变图进行图分割，还允许不同的应用按需来调整分区; Stanton和Kliot[19]提出了一种只依赖于图结构的启发式算法，相对于基于散列的分割方法和METIS，分割效果有很大提升。另外，Charalampos E. Tsourakakis[20]等人提出了一个新颖的one-pass流图分割算法，该算法统一了两个看似正交的启发式算法：将新到达的顶点放置在具有最大数量邻居结点的分区中或者具有最小数量的非邻居结点的分区中。相对于METIS，平衡分割时间更短，效果更好。本文的工作重心不是比较这些图划分算法的优劣，而是希望借鉴这些现有的图划分算法来提高系统的并行更新能力。

基于分区的并行更新策略不可避免的会出现多个节点的更新会集中在一个子图上的情况，这种情况会严重影响系统整体的并行度。基于分区的并行更新策略本质上是一个范围锁，锁住一定范围内的所有节点数据。这样粗粒度的锁会大大影响系统的并发性，因此本文又提出了基于细粒度锁的并行更新策略，即每次只需要锁住组成状态的单个因子本身，而不需要锁住范围内的所有节点。

传统的BSP模型是将整个图的迭代计算过程分解为若干个超步，超步内部的节点之间并行计算，超步之间进行同步。这使得在每个超步内，计算最慢的节点拖慢整个超步的计算速度，因此会出现短板效应，而本文的基于细粒度锁的并行更新策略有效弥补了这个不足，因子（这里的因子等价于BSP模型中的节点）与因子之间的更新都是并行的，只有属于一个因子的多个更新请求才会被串行执行，这样真正实现了多个因子的并行更新策略，而且没有显示的同步过程，消除了短板效应。但这需要因子的更新满足无序性，即对于任何一个节点的若干个更新，这些更新的顺序不会影响整个节点的最终状态。大多数算法（如本文中的DD、TC、SSSP等算法）都满足这个条件，因此模型的表达能力不会受到太大影响。

## 模型应用举例

基于状态更新的动态图计算模型将图的状态从用户的角度出发，只保存用户关心的数据，相比较传统的基于顶点的编程模型来说表达能力更强。在此本文选取了TC算法和CC算法来说明如何在该模型上进行算法设计，前者采用的是传统的面向顶点编程方式，后者采用的是用户自定义状态的编程方式。

（1）Triangle Count

Triangle Count算法是用来统计有向/无向图中的不同三角形的数目。该算法在复杂网络分析、链接标签和推荐等多个领域中都是非常基础重要的度量，也是一些诸如复杂网络、聚集系数等图运算中的基本方法。

由前文所知，基于状态更新的动态图计算模型，有State, Event, Transform三个重要的概念。针对Triangle Count算法，这三个组件的定义如下：

State:图的State由每个顶点对应的邻接点的信息组成，即，其中，表示节点的邻接点的集合为，节点构成的三角形的数目为；

Event:图的Event为图到达一条边相关的事件，那么Event构成的序列就形成事件流，即，其中=(,TYPE)，TYPE ∈{ADD,UPDATE,DELETE},这三种状态对应的事件分别为增加一条边，更新一条边和删除一条边；

Transform:图在动态变化过程中，State的更新过程见算法1。

|  |
| --- |
| **算法 1 Dynamic Triangle Count** |
| 1 ***for*** each ∈ EventStream  2  ***do***← ∅  3 ← ∅  4  **(,)** ← GET-VALUE()  5 ← GET-STATE()  6 ←GET-STATE()  7 ***if*** ≠ ∅  8 ***then*** ← GET-NEIGHBOR() ∪ {}  9 ***else*** ← {}  10 ***if*** ≠ ∅  11 ***then*** ← GET-NEIGHBOR() ∪ {}  12 ***else*** ← {}  13 cross ←∩  14 type ← GET-TYPE()  16 ***for*** each  ∈ cross  17 ***do***  ← GET-STATE()  18 t ← GET-TRIANGLE()  19 N ← GET-NEIGHBOR()  20 ***if*** type = ADD ***then*** ←(,N,t+1)  21 ***elseif*** type = DELETE ***then***←(,N,t-1)  18 SET-STATE(,)  19 ← GET-TRIANGLE()  20 ← GET-TRIANGLE()  21 ***if*** type = ADD  22 ***then*** ← (**,**,+|cross|)  22 ← (**,**,+|cross|)  23 ***elseif*** type = DELETE  24 ← (**,**,-|cross|)  25 ← (**,**,-|cross|)  26 SET-STATE(**,**)  27 SET-STATE(**,**) |

在算法2中，为了能够统计图中三角形数目，用户需要扩展State的方法：GET-NEIGHBOR()用来获取因子中的邻接点集合，GET-TRIANGLE()用来获取因子中的三角形数目。有了这些方法和State接口原来提供的方法，就可以实现动态的TC算法。

（2）Connected Components

如果一个图中，每对顶点都有路径相连，则称其为**连通图**。如果图的子图中任意两个顶点都是可达的，则这个子图称之为图的**连通分支**。连通分支反应了一个大图中子图的聚集情况，可以根据连通分支将原来的大图分解成若干个连通分支，算法独立并行的在连通分支上进行。连通分支在好友推荐、循环引用判断等诸多问题上被广泛使用。下面将介绍在不断增加边的情况下，如何针对无向图做连通分量的计算。由前文所知，基于状态更新的动态图计算模型有三个概念：State, Event, Transform，下面将详细介绍如何定义这三个基本组件：

State:当前图的所有的连通分支，，其中表示第k个连通分支，, 其中集合表示由这些顶点构成了一个连通分支，是这些顶点中标号最小的点。

Event:图的Event为图中新增了一条边，那么这些Event构成的序列就形成事件流，即，其中，表示新增边 ；

Transform:图在动态变化过程中，State的更新过程见算法2：

|  |
| --- |
| **算法 2 Dynamic Connected Components** |
| 1 ***for*** each ∈ EventStream  2 ***do* (,)** ← GET-VALUE()  3 ← GET-STATE()  4 ← GET-STATE()  5 ***if*** ≠ ∅ ***and***≠ ∅  6 ***then*** UNION-STATE(,)  7 ***elseif*** ≠ ∅ ***and***= ∅  8 ***then*** UNION-STATE(,)  9 ***elseif*** = ∅ ***and***≠ ∅  10 ***then*** UNION-STATE(,)  11 ***else***  12 s ← (min(**,**),{**,**})  13 ADD-STATE(s) |

在算法2中，表示节点****所在的连通分支，表示节点****所在的连通分支，新增的这条边使得这两个节点所在的连通分支合并，构成一个大的连通分支，算法中用户自定义的UNION-STATE()函数完成两个连通分支的合并。

## 本章小结

# GraphFlow系统的设计与实现

## 系统架构

## 模型实现

## 算法实现

## 本章小结

# 实验与分析

## 实验环境

## 实验结果

## 本章小结

# 结束语

## 工作总结

## 下一步工作

# 参考文献

# 发表文章目录

# 作者简历

# 致谢