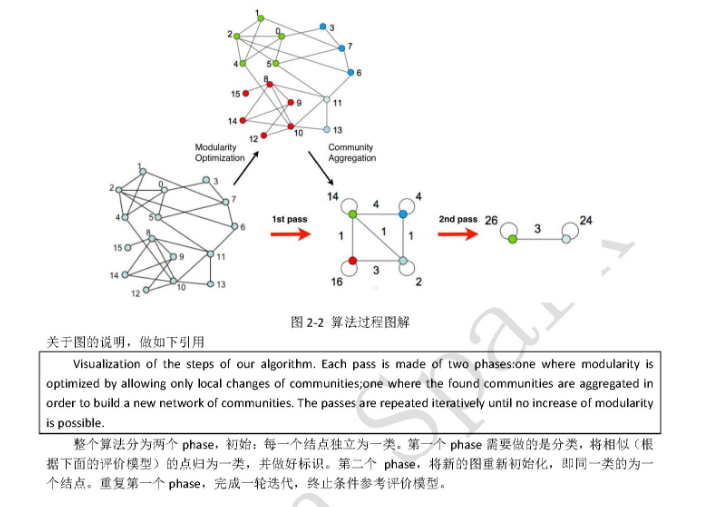
社团划分——Fast Unfolding(也叫Louvain)算法



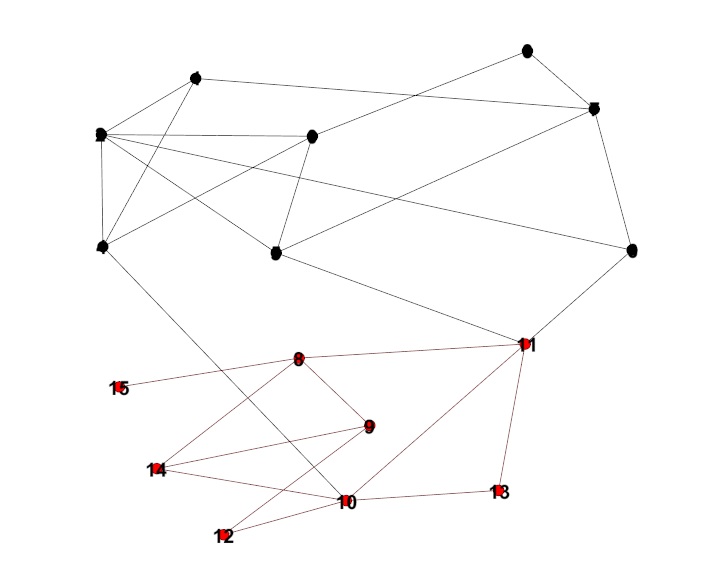
一、社区划分问题

1、社区以及社区划分

在社交网络中，用户相当于每一个点，用户之间通过互相的关注关系构成了整个网络的结构，在这样的网络中，有的用户之间的连接较为紧密，有的用户之间的连接关系较为稀疏，在这样的的网络中，连接较为紧密的部分可以被看成一个社区，其内部的节点之间有较为紧密的连接，而在两个社区间则相对连接较为稀疏，这便称为社团结构。

(Newman and Gievan 2004) A community is a subgraph containing nodes which are more densely linked to each other than to the rest of the graph or equivalently, a graph has a community structure if the number of links into any subgraph is higher than the number of links between those subgraphs.

如下图：



用红色的点和黑色的点对其进行标注，整个网络被划分成了两个部分，其中，这两个部分的内部连接较为紧密，而这两个社区之间的连接则较为稀疏。如何去划分上述的社区便称为社区划分的问题。

2、社区划分的算法

在社区划分问题中，存在着很多的算法，如由Newman和Gievan提出的GN算法，标签传播算法(Label Propagation Algorithm, LPA)，这些算法都能一定程度的解决社区划分的问题，但是性能则是各不相同。总的来说，在社区划分中，主要分为两大类算法

1. 凝聚方法(agglomerative method)：添加边
2. 分裂方法(divisive method)：移除边

在后续的文章中，我们会继续关注不同的社区划分的算法，在这篇文章中，主要关注Fast Unfolding算法。

3、社区划分的评价标准

为了评价社区划分的优劣，Newman等人提出了模块度的概念，用模块度来衡量社区划分的好坏。简单来讲，就是将连接比较稠密的点划分在一个社区中，这样模块度的值会变大，最终，模块度最大的划分是最优的社区划分。

二、模块度的概念

1、模块度的公式

社区划分的目标是使得划分后的社区内部的连接较为紧密，而在社区之间的连接较为稀疏，通过模块度的可以刻画这样的划分的优劣，模块度越大，则社区划分的效果越好 ，模块度的公式如下所示：

Q=12m∑i,j[Ai,j−kikj2m]δ(ci,cj)

其中，m=12∑i,jAi,j表示的是网络中的所有的权重，Ai,j表示的是节点i和节点j之间的权重，ki=∑jAi,j表示的是与顶点i连接的边的权重，ci表示的是顶点被分配到的社区，δ(ci,cj)用于判断顶点i与顶点j是否被划分在同一个社区中，若是，则返回1，否则，返回0。

2、模块度公式的简化形式

上述的模块度的计算可以得到以下的简化形式：

Q=∑c⎡⎣∑in2m−(∑tot2m)2⎤⎦

其中，∑in表示的是社区c内部的权重，∑tot表示的是与社区c内部的点连接的边的权重，包括社区内部的边以及社区外部的边。

3、模块度公式的解释

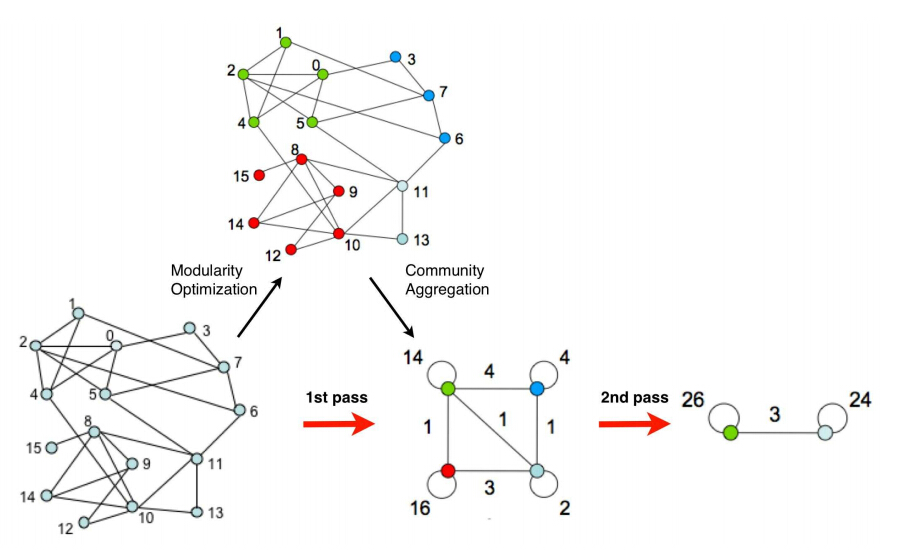
模块度（modularity）指的是网络中连接社区结构内部顶点的边所占的比例，减去在同样的社团结构下任意连接这两个节点的比例的期望值。

三、Fast Unfolding算法

1、Fast Unfolding算法的思路

模块度成为度量社区划分优劣的重要标准，划分后的网络模块度值越大，说明社区划分的效果越好，Fast Unfolding算法便是基于模块度对社区划分的算法，Fast Unfolding算法是一种迭代的算法，主要目标是不断划分社区使得划分后的整个网络的模块度不断增大。

2、Fast Unfolding算法的过程

Fast Unfolding算法主要包括两个阶段，如下图所示：   


第一阶段称为Modularity Optimization，主要是将每个节点划分到与其邻接的节点所在的社区中，以使得模块度的值不断变大；第二阶段称为Community Aggregation，主要是将第一步划分出来的社区聚合成为一个点，即根据上一步生成的社区结构重新构造网络。重复以上的过程，直到网络中的结构不再改变为止。

具体的算法过程如下所示：

1. 初始化，将每个点划分在不同的社区中；
2. 对每个节点，将每个点尝试划分到与其邻接的点所在的社区中，计算此时的模块度，判断划分前后的模块度的差值ΔQ是否为正数，若为正数，则接受本次的划分，若不为正数，则放弃本次的划分；
3. 重复以上的过程，直到不能再增大模块度为止；
4. 构造新图，新图中的每个点代表的是步骤3中划出来的每个社区，继续执行步骤2和步骤3，直到社区的结构不再改变为止。

**注意：**在步骤2中计算节点的顺序对模块度的计算是没有影响的，而是对**计算时间有影响**。

社区发现算法FastUnfolding的GraphX实现

时间 2014-12-05 14:24:35  [淘宝技术部](http://www.tuicool.com/sites/ZNFNFv)

原文  [http://rdc.taobao.org/?p=2406](http://rdc.taobao.org/?p=2406&utm_source=tuicool&utm_medium=referral)

主题 [算法](http://www.tuicool.com/topics/11000083)

现实生活中存在各种各样的网络，诸如人际关系网、交易网、运输网等等。对这些网络进行社区发现具有极大的意义，如在人际关系网中，可以发现出具有不同兴趣、背景的社会团体，方便进行不同的宣传策略；在交易网中，不同的社区代表不同购买力的客户群体，方便运营为他们推荐合适的商品；在资金网络中，社区有可能是潜在的洗钱团伙、刷钻联盟，方便安全部门进行相应处理；在相似店铺网络中，社区发现可以检测出商帮、价格联盟等，对商家进行指导等等。总的来看，社区发现在各种具体的网络中都能有重点的应用场景，图1展示了基于图的拓扑结构进行社区发现的例子。

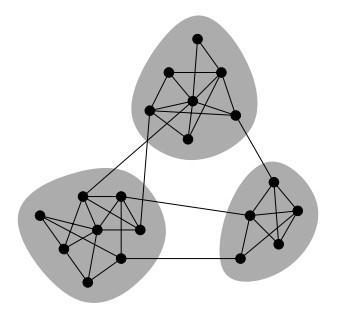


图1. 基于图的拓扑结构进行社区发现

**社区发现综述**

社区发现作为网络科学的经典问题之一，长期受到研究者的广泛关注。

* Girvan等人使用 [***GN算法***](http://arxiv.org/pdf/cond-mat/0112110.pdf) 进行求解，首先求解每条边的介数（betweenness），然后将介数最大的边删去，再重新求解每条边新的介数，依此循环。对应图1，连接不同社区的边的介数最大，把它们删去后即可得若干个独立的社区。但是求解介数时间复杂度高，在大图上并不实用，这时候需要考虑对 [***图的抽样***](https://cs.brown.edu/people/matteo/papers/RiondatoKornarop-BetweennessSampling.pdf) 等问题。
* [***Label Propagation Algorithm***](http://arxiv.org/pdf/0709.2938.pdf) 使用邻居节点的信息来决定当前节点的社区，并且也可应用到 [***多社区（Overlapping）的发现***](http://arxiv.org/ftp/arxiv/papers/0910/0910.5516.pdf) 中，但会存在有结果震荡等问题，性能不稳定。
* 此外，也可以结合 [***Game Theory***](http://www.cs.bgu.ac.il/~snean142/wiki.files/21-A%20game-theoretic-framework-to-identify-overlapping-communities.pdf) 、 [***Particle Competition***](http://www.fabriciobreve.com/artigos/aici2009.pdf) 、 ***[KCore](http://arxiv.org/pdf/1403.2226.pdf)*** 等方面的知识，求解社区发现问题。如 ***[Peng](http://arxiv.org/pdf/1403.2226.pdf)*** 等人认为，图的部分重要节点可以决定社区的整体框架，所以可以首先求解图的KCore节点，缩小图的规模确定社区基本框架，再逐步将其他点添加进已有的社区中。

**Modularity计算**

除了上述方法外，也有不少社区发现的算法是基于优化Modularity值的方向进行的。

[***Modularity值***](http://arxiv.org/pdf/physics/0602124.pdf) 用于评估社区发现的效果，对比社区发现结果与随机图（Null Model）的差异。对于同一个输入图进行不同的社区发现策略，取得Modularity值较高的策略性能较好。具体计算公式如下：

_1

其中，Aij 表示节点i与节点j之间的边的权重；ki 表示所有连接到节点i的边的权重之和；ci 表示当前节点i归属的社区；而当u等于v时，函数δ(u,v)的值为1，否则为0。

对上述公式进行化简，如下：

_2

其中，∑in 表示一个社区内部的连线数，∑tot 表示一个社区所有节点的度数之和。对比公式（1），它少了判断两个节点是否属于同一个社区的δ(u,v)函数，在后面的章节中我们可以看到，这种化简带来的计算量上的好处。

**FastUnfolding算法**

综合数据规模、运行时间等多方面的考虑，本文选择Blondel等人提出的 ***[FastUnfolding算法](http://arxiv.org/pdf/0803.0476.pdf)*** 进行实现。算法的基本步骤如下：

1.初始化，将每个节点划分在不同的社区中。

2.逐一选择各个节点，根据公式（3）计算将它划分到它的邻居社区中得到的Modularity增益。如果最大增益大于0，则将它划分到对应的邻居社区；否则，保持归属于原社区。

_3

3.重复步骤2，直到节点的社区不再发生变化。

4.构建新图。新图中的点代表上一阶段产生的不同社区，边的权重为两个社区中所有节点对的边权重之和。重复步骤2，直到获得最大的Modularity值。

可以将上述步骤分为两阶段（Pass）

* 1st Pass: 包含步骤1至3，用于设定各节点的归属社区，直到不再发生变化
* 2nd Pass: 由步骤4组成，用于构建新图，并重新执行1st Pass的操作，直到Modularity值不再增加

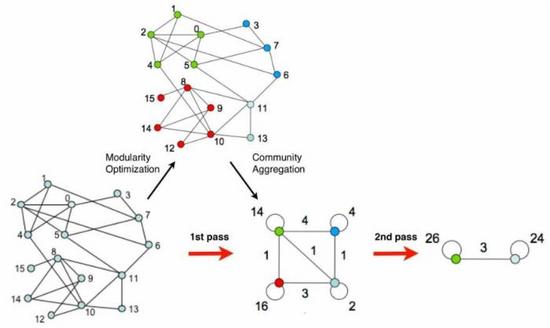


图2.FastUnfolding算法示意图

**串行化实现**

原始的FastUnfolding算法采用的是串行化实现方式：逐个选择节点，重新计算它的社区，不断进行迭代。这种串行化的计算方式，对分布式计算框架非常不友好，因为在选择一个节点计算它的增益的时候，其它的节点是不能进行变化的。这样不能进行并行化计算，也不能充分利用分布式框架的高并发、集群计算优势。另外Spark对于这种细粒度的操作，也非常的不合适，它为了改变单个结点的值，也需重新生成一个包含所有数据的RDD，开销非常的大。

**并行化实现**

为了将算法搬到分布式框架和集群上运行，我们需要对算法进行并行化改造。并行化的算法实现，会在每轮迭代中同步更新多个节点的信息，即根据t-1轮中邻居节点的信息来更新t轮中节点的信息，从而充分发挥高并发性的优势。

对照FastUnfolding的算法思路，定义一个新的数据结构VertexData，结构如下：

<code>**class VertexData() extends Serializable{**

var degree: Int = 0 *// 该节点度值*

var community: Long = 0 *// 该节点所属社区*

var communityDegreeSum: Long = -1 *// 该社区的度数之和*

var neighDegree: Int = 0 *// 目标节点的度值*

var neighCommunity: Long = -1 *// 目标节点所属社区*

var neighCommunityDegreeSum: Long = -1 *// 目标节点的社区总权重*

var edgeCount: Long = -1 *// 该节点与目标节点的连线条数*

……

}

</code>

|  |
| --- |
| < code > class VertexData ( ) extends Serializable {  var degree : Int = 0    // 该节点度值  var community : Long = 0    // 该节点所属社区  var communityDegreeSum : Long = - 1 // 该社区的度数之和  var neighDegree : Int = 0    // 目标节点的度值  var neighCommunity : Long = - 1 // 目标节点所属社区  var neighCommunityDegreeSum : Long = - 1 // 目标节点的社区总权重  var edgeCount : Long = - 1    // 该节点与目标节点的连线条数  ……  }  < / code > |

有了上述信息，就可以记录当前节点及它的邻居节点的信息。同时，为了提升性能，后续还可以使用kryo的序列化方法来替代Java的Serializable方法，获得时间和空间性能上的提升。

整体上，我们使用mrTriplets函数来实现算法，在map阶段，每个节点生成它所有邻居节点的VertexData消息，在reduce阶段将其合并，组成一个数组，包含这个节点的所有邻居信息。有了一个节点的所有邻居信息后，我们就可以使用公式（3）来计算它新归属的社区。上述操作对应于1st Pass的过程。

<code>*// 初始化图，每个节点置于不同的社区*

**var** newGraph = generateInitGraph(graph, degreeSum).cache()

**do** {

*// 每个节点获得邻居节点的信息*

val vertexRdd = newGraph.mapReduceTriplets(edgeMapFunc, \_ ++ \_).cache()

*// 根据上一轮中邻居的信息，更新节点的社区*

val idCommunity = vertexRdd.map {

**case** (vid, vdArray) => (vid, getBestCommunity(vdArray, curDegree))

}.cache()

*// 根据新的节点社区，获得更新信息*

val updateMessage = getUpdateMessage(idCommunity)

*// 更新图*

newGraph = newGraph.joinVertices(updateMessage) {...}

...

} **while** (changeRate > minThreshold && i < maxIterations)

</code>

|  |
| --- |
| < code > // 初始化图，每个节点置于不同的社区  var newGraph = generateInitGraph ( graph , degreeSum ) . cache ( )  do {  // 每个节点获得邻居节点的信息  val vertexRdd = newGraph . mapReduceTriplets ( edgeMapFunc , \_ ++ \_ ) . cache( )  // 根据上一轮中邻居的信息，更新节点的社区  val idCommunity = vertexRdd . map {  case ( vid , vdArray ) = > ( vid , getBestCommunity ( vdArray , curDegree ) )  } . cache ( )  // 根据新的节点社区，获得更新信息  val updateMessage = getUpdateMessage ( idCommunity )  // 更新图  newGraph = newGraph . joinVertices ( updateMessage ) { . . . }  . . .  } while ( changeRate > minThreshold && i < maxIterations )  < / code > |

对于2nd Pass的操作，直接使用RDD处理起来更为直观。当前图的边信息保存在edgeRdd中，每行为节点对< srcId, dstId >。同时，我们将节点Id与它归属的社区信息保存在communityRdd中，每行为节点对< nodeId, communityId >。所以，执行两次的leftOuterJoin即可得到新图的边信息，具体代码如下：

<code>edgeRdd.leftOuterJoin(communityRdd)

.map{ case (srcId, (dstId, srcComm)) => (dstId, srcComm.getOrElse(0L)) }

.leftOuterJoin(communityRdd)

.map{ case (dstId, (srcComm, dstComm)) =>(srcComm, dstComm.getOrElse(0L)) }

</code>

|  |
| --- |
| < code > edgeRdd . leftOuterJoin ( communityRdd )  . map { case ( srcId , ( dstId , srcComm ) ) = > ( dstId , srcComm . getOrElse ( 0L )) }  . leftOuterJoin ( communityRdd )  . map { case ( dstId , ( srcComm , dstComm ) ) = > ( srcComm , dstComm . getOrElse ( 0L ) ) }  < / code > |

有了新图的边信息后，使用Graph.fromEdgeTuples即可构建新图，完成2nd Pass的过程。

**并行化问题及解决策略**

进行并行化处理时，我们主要遇到两个问题：一是中间计算量过大，二是消息滞后。

* **中间计算量过大**

如果直接使用公式（1）进行Modularity计算，会导致中间计算量过大，因为它需要考虑两两节点对的情况（pairwise），即n平方的量级（n为节点个数），在大数据量情况下并不可行。

尝试的一个解决方法是，进行分步计算，如根据节点Id的hash值将数据划分成100个分区，每次只对分区内的节点进行计算。但是这种方法处理不直观，效率也不高。

经过反复尝试后，我们发现，更好的解决方法是使用化简后的公式（2）进行处理，避免了pairwise的过程。

* **消息滞后**

由于在并行化处理时，在t轮时每个节点根据t-1轮时的邻居社区信息进行更新，存在一定的消息滞后现象，会造成 **“互换社区”** 的问题，示意图如下：

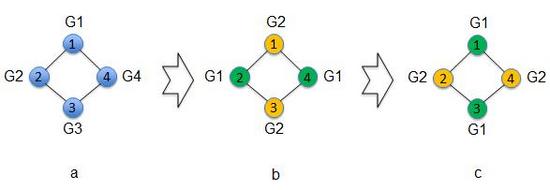


图3. “互换社区”问题示意图

变化情况如图3所示：

1. 每个节点被分配到不同的社区中（节点1属于G1，节点2属于G2，节点3属于G3，节点4属于G4）
2. 第二轮b图时，每个节点根据它邻居的信息进行更新（如节点1的新社区为邻居节点2在第一轮的社区G2）
3. 最终情况会导致不相连的节点反而归属同一社区（如节点1与3均受到节点2的影响，归属社区G2）
4. 第三轮c图类似，造成社区的互换。造成这种情况的原因在于，每个节点根据它的邻居前一轮的信息进行变化，而它的邻居也在同步改变。

类似的，还会存在有 **“社区归属延迟”** 问题。示意图如图4所示。节点1的归属社区受到节点2的影响，归属到社区2。但是节点2的社区也在同步变化，它可能归属于社区3，这样就造成只有节点1归属到社区2，成为一个孤立的点。

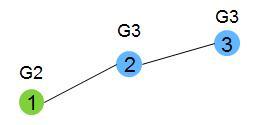


图4.”社区归属延迟”问题示意图

考虑有以下两种解决策略：

1. 添加随机值，即每轮迭代中会有部分节点的社区保持不变。如果阈值足够高，其实相当于逐个节点进行社区信息的更新，也即与串行的方法等价。使用随机值带来的问题是不能保证结果，得到的Modularity值有时高，有时低。并且，“互换社区”的问题不一定能解决。考虑到的一种解决思路是，多次运行，取最优。但是，这种方法也不太可靠，随机性较大。
2. 得到结果后构建逻辑图，求解连通区域，将同一个连通区域的点都归为一个社区。比如初始结果是互换社区的<1,2>,<2,1>（格式为<节点Id，归属社区>），求连通区域就可以将它们都归属同一社区。这种思路也可以解决 “社区归属延迟”的问题，如初始结果是<1,2>,<2,3>,<3,4>，节点1应该与归属社区2，但是节点2又归属于社区3，所以最终应该节点1,2,3都归属社区3。

对比上面两种方法，后一种策略充分考虑了图的特性，更为可取，能够保证结果的稳定性。大致代码如下：

<code>// 根据原始的<id, community>信息构建新图

val rawG = Graph.fromEdgeTuples(rawIdCommnity, 1)

// 获得连通区域

val connetedComponent = rawG.connectedComponents().vertices

// 得到最终结果

val idCommunity = rawIdCommunity.join(connetedComponent).map {

case(id, (rawCommunity, newCommunity)) => (id, newCommunity)

}

</code>

|  |
| --- |
| < code > // 根据原始的<id, community>信息构建新图  val rawG = Graph . fromEdgeTuples ( rawIdCommnity , 1 )  // 获得连通区域  val connetedComponent = rawG . connectedComponents ( ) . vertices  // 得到最终结果  val idCommunity = rawIdCommunity . join ( connetedComponent ) . map {  case ( id , ( rawCommunity , newCommunity ) ) = > ( id , newCommunity )  }  < / code > |

**总结**

FastUnfolding算法，基于结果Modularity值的优化进行，得到的社区发现效果比较理想，对比LPA算法会更稳定。并且，FastUnfolding算法会不断合并节点构造新图，大大减少了计算量，使得大规模图数据的计算成为可能。

原始的FastUnfolding算法采用串行化的实现思路，不适合面对海量数据。实现中需要进行算法并行化，充分利用并行化框架带来的计算优势。在将传统的串行化算法改造成并行化算法的过程中时，会遇到中间计算量过大、消息滞后造成的问题，如“互换社区”和“社区归属延迟”问题。解决的思路是考虑图的特性，对结果再次求解连通图区域，并通过重置社区得到最终结果。这样既保证了算法的准确性，又保证其性能，从而能够在大规模的网络上，进行实际的生产应用。

根据我们的初步测评，在三千万的用户数据上，可以在2个小时的级别，发现四万的社区，基本满足生产预期。