痛点：

在GraphX和GraphFrames中图也是不可变的对象，不可以在图中增量地添加边或节点。

spark-shell --packages graphframes:graphframes:0.5.0-spark2.1-s\_2.11

//=------

**图分析算法**  
  
图算法能帮助在图数据集上执行分析，而不用自己实现这些算法。下面给出了一些算法，来帮助找出图中的模式。

* PageRank
* Connected components
* Label propagation
* SVD++
* Strongly connected components
* Triangle count
* Single-Source-Shortest-Paths
* Community Detection

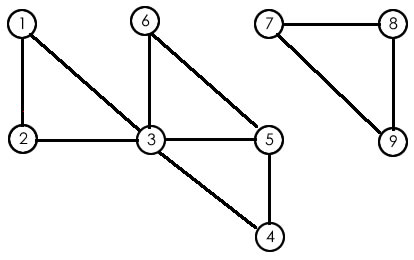
Spark GraphX中已经包含了一些预构造的图算法来进行图数据处理和分析工作。这些算法在org.apache.spark.graphx.lib包中。调用这些算法很简单，就像从Graph类中调用一个方法一样简单。

//=------

**Spark机器学习程序库的用例**：在线欺诈检测就是这样的一个领域。考虑到欺诈攻击的快速演变特性，预测分析需要使用最近五分钟或者更短时间内所生成的特性。此外将图模型加入到特征混合中的方法，具有实现更加有效的预测算法的潜力。

[**graphx上的一些简单应用**](http://blog.csdn.net/u013045749/article/details/50501929)

## 看实验用的图：



该图可以使用如下代码来进行标示

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | val sc=new SparkContext();  val edge=List(//边的信息  (1,2),(1,3),(2,3),(3,4),(3,5),(3,6),  (4,5),(5,6),(7,8),(7,9),(8,9))    //构建边的rdd  val edgeRdd=sc.parallelize(edge).map(x=>{  Edge(x.\_1.toLong,x.\_2.toLong,None)  })  //构建图 顶点Int类型  val g=Graph.fromEdges(edgeRdd, 0) |

## 度分布

可以了解图中“超级节点”的个数和规模，以及所有节点度的分布曲线。

在graphx中求解度分布分享简单，一个API即可

|  |  |
| --- | --- |
| 1 | g.degrees.collect.foreach(println(\_)) |

同时将其度的rdd数据收集到Driver中打印出来,第一列表示顶点id，第二列表示各个顶点的度

(4,2)

(6,2)

(8,2)

(2,2)

(1,2)

(3,5)

(7,2)

(9,2)

(5,3)

## 二跳邻居

App中好友的好友的秘密，传播范围更加广，信息更加丰富

使用两次遍历，首先进行初始化的时候将自己的生命值设为2，第一次遍历向邻居节点传播自身带的ID以及生命值为1(2-1)的消息，第二次遍历的时候收到消息的邻居再转发一次，生命值为0，最终汇总统计的时候 只需要对带有消息为0ID的进行统计即可得到二跳邻居

上面的需求使用pregel很方便就能解决，按照最短路径的方法来做，首先是消息的更新、发送和合并方法的定义：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | type VMap=Map[VertexId,Int]    /\*\*  \* 节点数据的更新 就是集合的union  \*/  def vprog(vid:VertexId,vdata:VMap,message:VMap)  :Map[VertexId,Int]=addMaps(vdata,message)  /\*\*  \* 发送消息  \*/  def sendMsg(e:EdgeTriplet[VMap, \_])={    //取两个集合的差集 然后将生命值减1  val srcMap=(e.dstAttr.keySet -- e.srcAttr.keySet).map { k => k->(e.dstAttr(k)-1) }.toMap  val dstMap=(e.srcAttr.keySet -- e.dstAttr.keySet).map { k => k->(e.srcAttr(k)-1) }.toMap    if(srcMap.size==0 && dstMap.size==0)  Iterator.empty  else  Iterator((e.dstId,dstMap),(e.srcId,srcMap))  }  /\*\*  \* 消息的合并  \*/  def addMaps(spmap1: VMap, spmap2: VMap): VMap =  (spmap1.keySet ++ spmap2.keySet).map {  k => k -> math.min(spmap1.getOrElse(k, Int.MaxValue), spmap2.getOrElse(k, Int.MaxValue))  }.toMap |

接着直接使用pregel的接口即可：

|  |  |
| --- | --- |
| 1  2  3 | val two=2 //这里是二跳邻居 所以只需要定义为2即可  val newG=g.mapVertices((vid,\_)=>Map[VertexId,Int](vid->two))  .pregel(Map[VertexId,Int](), two, EdgeDirection.Out)(vprog, sendMsg, addMaps) |

现在可以看一下二次遍历之后各个顶点的数据：

|  |  |
| --- | --- |
| 1 | newG.vertices.collect().foreach(println(\_)) |

(4,Map(5 -> 1, 1 -> 0, 6 -> 0, 2 -> 0, 3 -> 1, 4 -> 2))

(6,Map(5 -> 1, 1 -> 0, 6 -> 2, 2 -> 0, 3 -> 1, 4 -> 0))

(8,Map(8 -> 2, 7 -> 1, 9 -> 1))

(2,Map(5 -> 0, 1 -> 1, 6 -> 0, 2 -> 2, 3 -> 1, 4 -> 0))

(1,Map(5 -> 0, 1 -> 2, 6 -> 0, 2 -> 1, 3 -> 1, 4 -> 0))

(3,Map(5 -> 1, 1 -> 1, 6 -> 1, 2 -> 1, 3 -> 2, 4 -> 1))

(7,Map(7 -> 2, 8 -> 1, 9 -> 1))

(9,Map(9 -> 2, 7 -> 1, 8 -> 1))

(5,Map(5 -> 2, 1 -> 0, 6 -> 1, 2 -> 0, 3 -> 1, 4 -> 1))

Map中的key表示周边的顶点id，其value就是对应顶点id的生命值，所以我们现在对该rdd再做一次mapValues处理即可得到最后的二跳邻居

|  |  |
| --- | --- |
| 1  2  3 | //过滤得到二跳邻居 就是value=0 的顶点  val twoJumpFirends=newG.vertices  .mapValues(\_.filter(\_.\_2==0).keys) |

之后将其打印出来

|  |  |
| --- | --- |
| 1 | twoJumpFirends.collect().foreach(println(\_)) |

(4,Set(1, 6, 2))

(6,Set(1, 2, 4))

(8,Set())

(2,Set(5, 6, 4))

(1,Set(5, 6, 4))

(3,Set())

(7,Set())

(9,Set())

(5,Set(1, 2))

## 连通图

连通图的检测可以弄清一个图有多少个连通部分以及每个连通部分有多少个节点，这样可以在小图上进行更加精细的操作

使用ConnectedComponents和StronglyConnectedComponents接口即可完成计算，其原理就是使用pregel模型，每次都是向邻居节点发送自己的ID，然后合并消息和更新消息方法都是保留最小的ID即可，在任意两条边需要传播的ID一样时迭代停止。

关于连通的Spark实现可以看这个[使用Spark求解大图的连通组件(第二版)](http://kubicode.me/2015/06/18/Spark/Connect-Component-From-Baidu-Interview-Using-Spark-V2/)，当然在graphx源码中org.apache.spark.graphx.lib.ConnectedComponents就是相关源码的实现。

## 多图合并工具

直接使用outerJoinVertices就可以进行很方便的操作

## 能量传播模型

加权网络是经典的能量传播图模型之一

最简单的使用随机游走模型，每次都是将自己的能量值x传播强度传播给邻居，不断迭代。

下面是淘宝的信誉度的检测：  
  
模型的思路是：物以类聚，人以群分。常和信誉度高的用户进行交易的，信誉度自然较高；常和信誉度差的用户有业务来往的，信誉度自然较低。

首先以用户为点，买卖关系为边生成图，对选出来的用户分别赋予相同的正负能量值（比如信誉高的种子的trustRank=1，信誉度低的种子badRank=1），然后进行两轮随机游走，一轮是高信誉用户传播trustRank，另一轮是低信誉用户传播badRank，两轮结束之后对每个用户进行finalRank=trustRank-badRank的计算，finalRank高的即为信誉较好的用户。

下面是淘宝的改进：

但是这种方法得到的AUC很低（之前初始的传播强度为0.85），需要给每条边加上一个组合权重（比如由交易次数，计算金额等多个特征计算出来的一个权重），通过使用partialDerivativeAUC方法，在训练集上计算AUC，然后对AUC求偏导，得到每个关系维度的独立权重和偏移量（这里不知道怎么得到的-\_-），生成新的权重调节器（WeightAdjustor），然后对图上所有的边进行权重更新，再进行一次大的迭代，直到AUC趋于稳定，终止计算。