# 1. 实验设计（Benchmark）

## 1.1 如何衡量一个算法的好坏？

（1）准确性：计算的结果是否准确？

（2）扩展性：在不同规模的数量级上算法的性能趋势？

（3）可重复性：在相同数据集上多次重复试验的结果是否相同？

应用参数的影响。

## 1.2 如何设计图的benchmark？

（1）纵向比较

a. 图的性质：有向/无向图？随机图？全连通图？空图？超立方体图？

b. 图的规模：顶点和边的数量？

（2）横向比较

a. Flink和Spark做对比？

b. 自己实现的做对比？

# 2. Flink图算法

表1. Flink图算法表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 名称 | 描述 | 应用 | 难度 |
|  | Community Detection | **社区发现算法**  社区，从直观上来看，是指网络中的一些密集群体，每个社区内部的结点间的联系相对紧密，但是各个社区之间的连接相对来说却比较稀疏。社区发现有一些列成熟的算法。 | 发现社区 |  |
|  | Label Propagation | **标签传播算法（LPA）**  它是一种**基于图的半监督学习**方法，其基本思路是用已标记节点的标签信息去预测未标记节点的标签信息。 | 广泛地应用到多媒体信息分类、虚拟社区挖掘等领域中。 | LPA算法的优点是简单、高效、快速；缺点是每次迭代结果不稳定，准确率不高。 |
|  | Connected Components |  |  |  |
|  | GSA Connected Components |  |  |  |
|  | PageRank |  |  |  |
|  | GSA PageRank |  |  |  |
|  | Single Source Shortest Paths |  |  |  |
|  | GSA Single Source Shortest Paths |  |  |  |
|  | Triangle Count |  |  |  |
|  | Triangle Listing |  |  |  |
|  | Triangle Enumerator |  |  |  |
|  | Hyperlink-Induced Topic Search | **HITS连接分析算法**  HITS算法的目的即是通过一定的技术手段，在海量网页中找到与用户查询主题相关的高质量“Authority”页面和“Hub”页面，尤其是“Authority”页面，因为这些页面代表了能够满足用户查询的高质量内容，搜索引擎以此作为搜索结果返回给用户。 | 网页质量分析 |  |
|  | Summarization | **图摘要算法**  用一个vertex代替多个vertex，生成原图的简略图形. |  | 资料较少 |
|  | Adamic-Adar | **节点相似性算法** |  | 资料较少 |
|  | Jaccard Index | **节点相似性算法**  用来计算两个有限集合的相似度. |  |  |
|  | Local Clustering Coefficient | **局部聚类系数算法**  在图论中，集聚系数是图中的点倾向于集聚在一起的程度的一种度量。 |  |  |
|  | Global Clustering Cofficient | **全局聚类系数算法** |  |  |

## 2.1 Flink 图模型



图1. Flink图模型

## 2.2 Gelly Library

### 2.2.1 算法框架

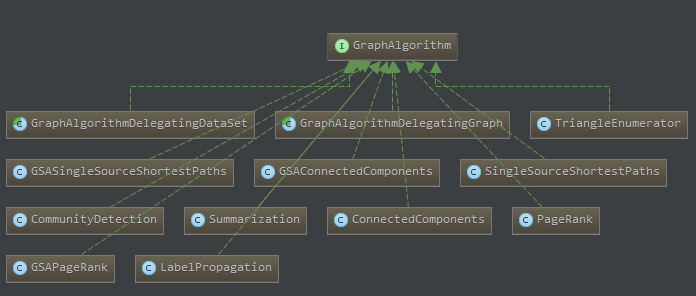
**（1）GraphAlgorithm**

**GraphAlgorithm**是算法的高度抽象接口。其接口只定义了一个run()方法,如下所示：

*/\*\*  
 \** ***@param*** <*K*> *key type  
 \** ***@param*** <*VV*> *vertex value type  
 \** ***@param*** <*EV*> *edge value type  
 \** ***@param*** <*T*> *the return type  
 \*/*public interface GraphAlgorithm<K, VV, EV, T> {  
  
 public T run(Graph<K, VV, EV> input) throws Exception;  
}

*/\*\*  
 \** ***@param*** <*K*> *key type  
 \** ***@param*** <*VV*> *vertex value type  
 \** ***@param*** <*EV*> *edge value type  
 \** ***@param*** <*T*> *the return type  
 \*/*

很多图算法的实现，都直接或间接实现了此接口，如下图所示：



如上图所示，在GraphAlgorithm接口的实现类中，有两个代理类，一个是**GraphAlgorithmDelegatingDataSet**，另外一个是**GraphAlgorithmDelegatingGraph**。

**（2）GraphAlgorithmDelegatingDataSet**

GraphAlgorithm将输入的图经过特定运算后输出某个结果。而GraphAlgorithmDelegatingDataSet则通过使用代理对象包装算法输出结果为DataSet的算法。当相同的算法运行在相同的可合并配置的输入上时，代理对象能够被替换掉。这允许算法由隐式可重用算法组成，而不公开共享中间数据集。原文如下：

*/\*\*  
 \* A {****@link*** *GraphAlgorithm} transforms an input {****@link*** *Graph} into an output of type {****@code*** *T}. A {****@code*** *GraphAlgorithmDelegatingDataSet} wraps the resultant {****@link*** *DataSet} with a delegating proxy object. The delegated object can be replaced when the same algorithm is run on the same input with a mergeable configuration. This allows algorithms to be composed of implicitly reusable algorithms without publicly sharing intermediate {****@link*** *DataSet}s.*

GraphAlgorithmDelegatingDataSet的数据结构如下：

public abstract class GraphAlgorithmDelegatingDataSet<K, VV, EV, T>  
implements GraphAlgorithm<K, VV, EV, DataSet<T>> {  
  
 // each algorithm and input pair may map to multiple configurations  
 private static Map<GraphAlgorithmDelegatingDataSet, List<GraphAlgorithmDelegatingDataSet>> *cache* =  
 Collections.*synchronizedMap*(new HashMap<GraphAlgorithmDelegatingDataSet, List<GraphAlgorithmDelegatingDataSet>>());  
  
 private Graph<K,VV,EV> input;  
  
 private Delegate<DataSet<T>> delegate;

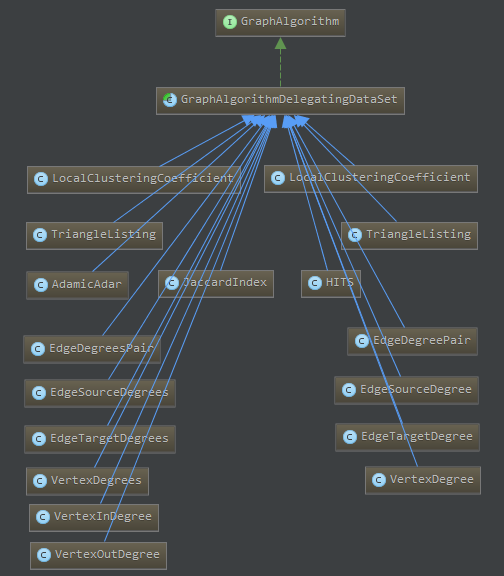
主要有3类：

* cache: 全局缓存
* input: 算法所需要的输入的图
* delegate: 算法输出的数据集的代理对象

run()方法如下：

public final DataSet<T> run(Graph<K, VV, EV> input)  
 throws Exception {  
 this.input = input;  
  
 if (*cache*.containsKey(this)) {  
 for (GraphAlgorithmDelegatingDataSet<K, VV, EV, T> other : *cache*.get(this)) {  
 if (mergeConfiguration(other)) {  
 // configuration has been merged so generate new output  
 DataSet<T> output = runInternal(input);  
  
 // update delegatee object and reuse delegate  
 other.delegate.setObject(output);  
 delegate = other.delegate;  
  
 return delegate.getProxy();  
 }  
 }  
 }  
  
 // no mergeable configuration found so generate new output  
 DataSet<T> output = runInternal(input);  
  
 // create a new delegate to wrap the algorithm output  
 delegate = new Delegate<>(output);  
  
 // cache this result  
 if (*cache*.containsKey(this)) {  
 *cache*.get(this).add(this);  
 } else {  
 *cache*.put(this, new ArrayList(Collections.*singletonList*(this)));  
 }  
  
 return delegate.getProxy();  
}

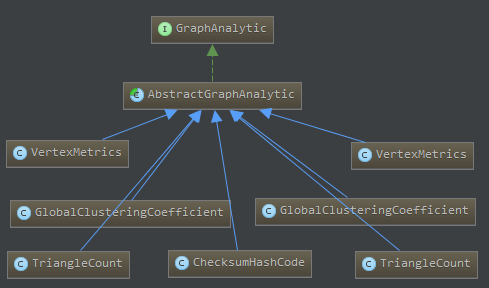
**GraphAlgorithmDelegatingDataSet**的类图如下：



**GraphAnalytic**和GraphAlgorithm接口很像，不同之处在于GraphAnalytic是末端的（即算法的最后一个算子），而且需要通过accumulator来获取计算结果。一个Flink程序是单点执行的，GraphAnalytic推迟执行，允许多个analytics和algorithms在一个单独的程序里。

GraphAnalytic的接口如下所示：

*/\*\*  
 \* A {****@code*** *GraphAnalytic} is similar to a {****@link*** *GraphAlgorithm} but is terminal  
 \* and results are retrieved via accumulators. A Flink program has a single  
 \* point of execution. A {****@code*** *GraphAnalytic} defers execution to the user to  
 \* allow composing multiple analytics and algorithms into a single program.  
 \*  
 \** ***@param*** <*K*> *key type  
 \** ***@param*** <*VV*> *vertex value type  
 \** ***@param*** <*EV*> *edge value type  
 \** ***@param*** <*T*> *the return type  
 \*/*public interface GraphAnalytic<K, VV, EV, T> {  
  
 */\*\*  
 \* This method must be called after the program has executed:  
 \* 1) "run" analytics and algorithms  
 \* 2) call ExecutionEnvironment.execute()  
 \* 3) get analytic results  
 \*  
 \** ***@return*** *the result  
 \*/* T getResult();  
  
 */\*\*  
 \* Execute the program and return the result.  
 \*  
 \** ***@return*** *the result  
 \** ***@throws*** *Exception  
 \*/* T execute() throws Exception;  
  
 */\*\*  
 \* Execute the program and return the result.  
 \*  
 \** ***@param*** *jobName the name to assign to the job  
 \** ***@return*** *the result  
 \** ***@throws*** *Exception  
 \*/* T execute(String jobName) throws Exception;  
  
 */\*\*  
 \* All {****@code*** *GraphAnalytic} processing must be terminated by an  
 \* {****@link*** *OutputFormat}. Rather than obtained via accumulators rather than  
 \* returned by a {****@link*** *DataSet}.  
 \*  
 \** ***@param*** *input input graph  
 \** ***@return*** *this  
 \** ***@throws*** *Exception  
 \*/* GraphAnalytic<K, VV, EV, T> run(Graph<K, VV, EV> input) throws Exception;  
}



### 2.2.2 算法实现

#### AF07 Single Source Shortest Paths

|  |  |
| --- | --- |
| 名称 |  |
| 描述 |  |
| 应用 |  |
| 逻辑 |  |
| 程序 |  |
| 实验 |  |
| 结果 |  |
| 参考 |  |

#### AF09 Triangle Count

**名称：**

Triangle Count

**描述：**

统计（有向/无向）图中不同三角形的数目

**应用：**

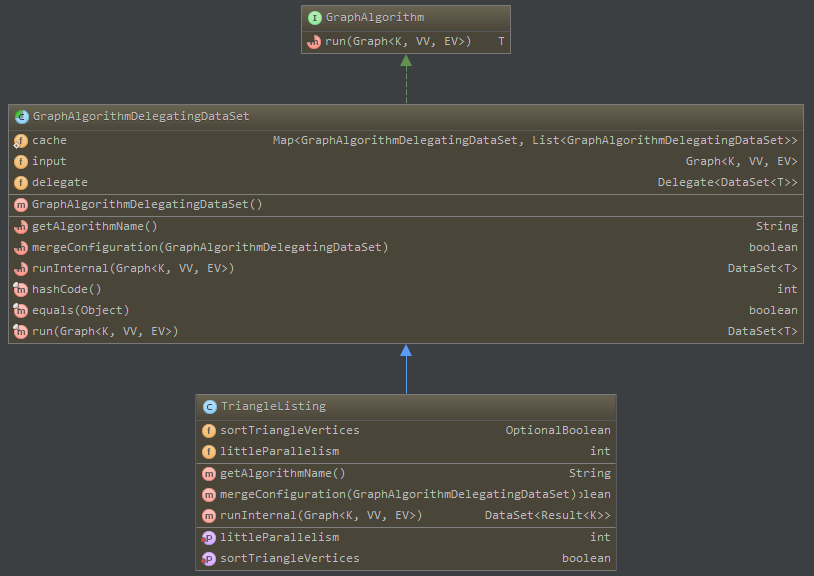
一般运用在社交网络分析中。社交网络中的三角形越多，说明关系网越强。（聚集系数）

**逻辑：**

1. **def** triangleCount(graph):
2. count = 0
3. tringle = []
4. **for** srcId **in** graph:  //1. 遍历图中的每个节点
5. srcSet = graph.get(srcId)  //2. 针对节点srcId，找出它的所有邻接点
6. **for** destId **in** srcSet:
7. **if** (destId > srcId):
8. destSet = graph.get(destId)  //3. 针对邻接点desctId,找出他的邻接点
9. //4. 如果srcId和descId的邻接点相同，则构成三角形
10. **for** vertexId **in** destSet:
11. **if**(vertexId **in** srcSet) **and** (vertexId > destId):
12. count += 1
13. tringle.append((srcId, destId, vertexId))

**实现：**

**（1）类图**



**（2）核心代码**

主要以如下图为例子，结合有向图的Triangle Listing代码进行讲解



该图的顶点集合为：

v={1,2,3,4,5}

边集合为：

e={(1,2),(1,3),(1,4),(3,2),(3,4),(4,1),(4,3)}

**第1步：**排序边中顶点的顺序，使得对于边（u,v）有u<v。该操作将得到如下数据集

DataSet<Tuple3<K, K, ByteValue>> **filteredByID**

(源顶点，目标顶点，方向)，其中方向值的含义 3：双向；1：→；2：←

(1,4,3)

(1,2,1)

(1,3,1)

(2,3,2)

(3,4,3)

**第2步**：计算每条边的源顶点和目标顶点的度的值，将得到如下数据集DataSet<Edge<K, Tuple3<EV, Degrees, Degrees>>> **pairDegrees**

（源顶点，目标顶点，（（边值），（源顶点Degree, in-degree, out-degree），（目标顶点Degree, in-degree, out-degree）））

(1,2,((null),(3,3,1),(2,0,2)))

(1,3,((null),(3,3,1),(3,2,2)))

(1,4,((null),(3,3,1),(2,2,2)))

(3,2,((null),(3,2,2),(2,0,2)))

(3,4,((null),(3,2,2),(2,2,2)))

(4,1,((null),(2,2,2),(3,3,1)))

(4,3,((null),(2,2,2),(3,2,2)))

**第3步：**在第2步的基础上，根据顶点的度来进行排序，使得对于输出的边（u, v）有deg(u) < deg(v) or (deg(u) == deg(v) and u < v)将得到如下数据集

DataSet<Tuple3<IntValue, IntValue, ByteValue>> **filteredByDegree**

（源顶点，目标顶点，方向（12：双向；4：→；8：←））

(2,1,8)

(4,1,12)

(4,3,12)

(1,3,4)

(2,3,8)

**第4步**：在第3步的基础上，构建半开三角形（u, v, w）满足 (u, v) and (u, w) are edges in graph的数据集

DataSet<Tuple4<IntValue, IntValue, IntValue, ByteValue>> **triplets**

（u, v, w, bitmask）

(2,1,3,40)

(4,1,3,60)

**第5步**：在第1步和第4步的基础上，构建封闭的三角形（u, v, w）满足(u, v), (u, w), and (v, w) are edges in graph and v < w的数据集

DataSet<TriangleListing.Result<IntValue>> **triangles**

（u, v, w, bitmask）

(2,1,3,41)

(4,1,3,61)

算法的核心函数如下：

public DataSet<Result<K>> runInternal(Graph<K, VV, EV> input)  
 throws Exception {  
 // first step: u, v, bitmask where u < v  
 DataSet<Tuple3<K, K, ByteValue>> filteredByID = input  
 .getEdges()  
 .map(new OrderByID<K, EV>())  
 .setParallelism(littleParallelism)  
 .name("Order by ID")  
 .groupBy(0, 1)  
 .reduceGroup(new ReduceBitmask<K>())  
 .setParallelism(littleParallelism)  
 .name("Flatten by ID");  
  
 // second step: u, v, (deg(u), deg(v))  
 DataSet<Edge<K, Tuple3<EV, Degrees, Degrees>>> pairDegrees = input  
 .run(new EdgeDegreesPair<K, VV, EV>()  
 .setParallelism(littleParallelism));  
  
 //third step: u, v, bitmask where deg(u) < deg(v) or (deg(u) == deg(v) and u < v)  
 DataSet<Tuple3<K, K, ByteValue>> filteredByDegree = pairDegrees  
 .map(new OrderByDegree<K, EV>())  
 .setParallelism(littleParallelism)  
 .name("Order by degree")  
 .groupBy(0, 1)  
 .reduceGroup(new ReduceBitmask<K>())  
 .setParallelism(littleParallelism)  
 .name("Flatten by degree");  
  
 // forth step: u, v, w, bitmask where (u, v) and (u, w) are edges in graph  
 DataSet<Tuple4<K, K, K, ByteValue>> triplets = filteredByDegree  
 .groupBy(0)  
 .sortGroup(1, Order.*ASCENDING*)  
 .reduceGroup(new GenerateTriplets<K>())  
 .setParallelism(littleParallelism)  
 .name("Generate triplets");  
  
 // u, v, w, bitmask where (u, v), (u, w), and (v, w) are edges in graph  
 DataSet<Result<K>> triangles = triplets  
 .join(filteredByID, JoinOperatorBase.JoinHint.*REPARTITION\_HASH\_SECOND*)  
 .where(1, 2)  
 .equalTo(0, 1)  
 .with(new ProjectTriangles<K>())  
 .setParallelism(littleParallelism)  
 .name("Triangle listing");  
  
 if (sortTriangleVertices.get()) {  
 triangles = triangles  
 .map(new SortTriangleVertices<K>())  
 .name("Sort triangle vertices");  
 }  
  
 return triangles;  
}

**参考：**

<http://blog.csdn.net/u010376788/article/details/50223157>

<http://book.51cto.com/art/201409/451628.htm>

**附（文献阅读）：**

P1

Tangwongsan K, Pavan A, Tirthapura S. Parallel triangle counting in massive streaming graphs[C]//Proceedings of the 22nd ACM international conference on Information & Knowledge Management. ACM, 2013: 781-786.

**Parallel Triangle Counting in Massive Streaming Graphs**

**Abstract**

图的triangle count在复杂网络分析、链接标签和推荐等多个领域都是非常基础重要的度量。在这些应用中，图变得越来越大，而且动态变化。这篇文章陈述了一个快速的并行的在大体量的无向图中估计triangle count的方法，而这个大体量的无向图的边像流一样动态流过的。我们的算法被设计成多核共享内存的形式，充分利用并行和多级内存的优势。我们提供了理论上的边界和精确度，我们的实验时运行在真实数据集上的，结果表明我们的实验结果是精确的，并且和一个优化了的序列算法相比，我们的算法在速度上有显著提高。

**1. Introduction**

经过调研，我们发现如下论文[1, 14, 6, 20, 15, 23, 13]致力于解决graph体量足够大以至于无法一次性放进内存的流式处理算法；但是他们却无法高效的利用并行计算的优势，性能上有待提高。另外一方面，有[26, 9]论文致力于快速的处理大体量的静态数据，但是他们无法高效的处理动态数据。考虑到现在的数据不经体量巨大，而且经常动态变化，现存的算法无法充分利用并行优势，兼顾高效的处理大体量的图。

在这片文章中，我们设置一个快速的共享内存的并行算法，能够充分结合流式算法和并行算法的优势。这个算法提供一个可调的错误率参数：given 0 < ε, δ ≤ 1, a random variable Xˆ is an (ε, δ)-approximation of the true quantity X if |Xˆ − X| ≤ εX with probability at least 1 − δ.

**1.1 Our Contributions**

介绍了我们的工作：设计并实现了一个协调的小批量的并行算法。说出这个算法的优势和特点。

**1.2 Related Work**

在这里详细列举了目前关于Triangle Count的各种研究，介绍的非常细致。需要的相关资料都可以在这里找。

**2. Preliminaries**

**P2**

Chu S, Cheng J. Triangle listing in massive networks and its applications[C]//Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011: 672-680.

**Triangle listing in massive networks and its applications**

**Abstract**

Triangle Listing 是一些诸如复杂网络、聚集系数等图运算中的基本方法。现有的Triangle Listing算法主要是基于内存的方法（即数据全部载入到内存之后再进行运算），这种方式显然不能够支撑今天大体量的不断增长的网络模型。当图的体量远远大于内存的容量时，Triangle Listing的计算就需要外存的支撑，而这又会显著正价IO的开销。一些流式计算和抽样的算法虽然能够解决内存不够用的问题，但是他们是近似的算法，也无法得到确定的解。我们提出了一种基于IO的高效的算法来计算Triangle Listing。我们的计算结果是准确的，而且有效避免了磁盘的使用。我们的实验结果表明我们的算法是可扩展的，而且比最先进的local triangle estimation算法要好。

**1. Introduction**

我们的目标是在一个无向图G中，列举出其所有的不重复的三角形。我们希望设计一种高效的算法来针对大体量而内存有限的图数据。

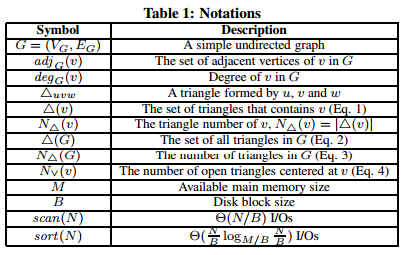
中间一段是分析现有的算法的不足。略。

我们的算法是采用迭代的方式，将原来输入进来的图G不断分割成一个个子图，使得该子图能够放入到内存中进行处理和计算Triangle List。为了确保根据本地子图而计算得到的结果的准确性和完整性，我们将Triangle分成三种类型。我们设计出一种机制，她能够列举出所有的Type1和Type2类型的triangle,然后在下一次的迭代过程中，通过一个新的分片，将Tpye3类型的triangle转换成Type1和Type2类型。为了限制总的迭代次数，我们将在每次迭代结束后，移除掉每个子图中的所有边，以此方式不断的缩小图G直至它变成空的。我们给出了两个高效的图分片算法。

我们在大量的真实数据集中测试我们的算法。该数据集有1亿（106million）多个点，18亿（1,877million）多条边，同时我们将和最好的内存算法以及近似算法进行比较。当数据能够全部载入内存时，我们的算法和内存算法拥有相似的计算性能，但是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，而我们的算法是精确的，在内存和时间上却相当接近。当我们希望降低算法的错误率例如降低到50%时，近似算法的运行速度已经和我们的精确算法有数量级的差异。

**2. Notations and Problem Definition**

相关术语和定义如下：

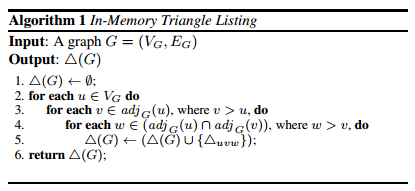


一个以v为中心的开放三角形的数目为：



直观来看，这个数目是顶点v能够形成的最多的封闭三角形的数目。

**3. In-Memory Triangle Listing**



上面这种算法和最优的内存算法相似，只不过**最优的内存算法先对顶点按照degree进行从小到大排序**，然后再进行计算。

**4. I/O-EFFICIENT TRIANGLE LISTING**

在本章节，我们将首先描述整个算法框架，然后在深入到细节。

**4.1 Algorithm Framework**

当无法将整个图导入到内存的时候，我们一次只能导入图的一部分（子图）进入内存。我们的算法就是迭代的在这样的子图上计算triangle listing.算法框架如下：

**在每一轮的迭代中：**

1. 将图G进行分割 P = {G1, . . . , Gi, . . . , Gp}, 使得得到的每个子图都能载入内存；

2. 将每个子图Gi载入到内存之中，并且计算它的triangle listing；

3. 将Gi从图G中移除掉，因为在以后的triangle listing计算中将不再起作用。

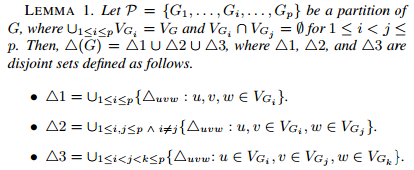
重复上面的迭代过程，直至整个图G为空。

我们的算法的主要思想就是采用迭代划分的方式不断的切割原图，然后在每个子图上独立的计算Triangle listing，以避免随机访问任意顶点及其邻接点。

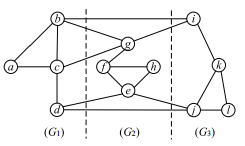
这个思想是非常简单的，但是中间有若干的挑战：（1）如何从每次的本地迭代中，确保最终结果的正确性和完整性；（2）一个高效的Triangle listing划分算法；（3）确定整体IO复杂度的边界（例如每一步的迭代中IO的复杂度以及迭代次数等）。下面我们将在每个小节中详细讨论上述问题。

**4.2 Correctness and Global-Completeness of Local Triangle Listing**

我们的算法基于如下定理：

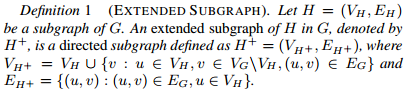


此时，我们将triangle △1, △2和△3分别称为Type 1,Type 2,Type 3这三类三角形。如下图所示：



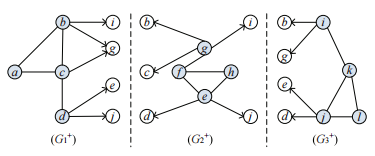
根据定理1，一个三角形△uvw可以通过搜索任意的子图Gi, 并且仅当uvw在子图Gi中时能够被找到（即Type1 类型的triangle）。然而这类的triangle数量有限，更关键的是我们不能移除掉任何边以及顶点即使我们已经列举出所有的Type1类型的三角形，因为一条边 (u, v) 极可能组成△uvw，也可能组成△uvx，而x在其他的子图中。

为了确保在列举完所有的三角形之后，能够安全的删除这些边而不会影响最终结果的完整性，我们引入扩展子图的概念（**extended subgraph**）。

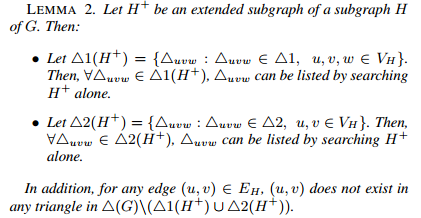


例子如下：上面分割得到的G1，G2，G3的子图的扩展子图表示如下：

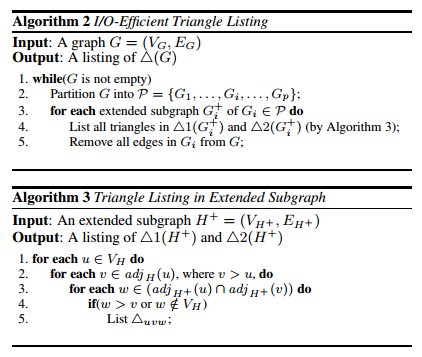
深颜色的顶点是原来子图的顶点，有向的边及其指向的顶点是Gi的扩展。



根据上面的定义1，我们又有如下的针对扩展子图的定理：



因此，我们可以采用如下的算法来统计这两类的三角形的数目：



在上述的算法中，只能够列举出第一类和第二类的triangle的数目，但是仍然缺少对第三类triangle的枚举。我们设计了一个高效的算法，它能够将第三类的triangle转换成第一类和第二类的triangle以使得所有的三角形都能够被列举出来，与此同时还能够显著减少IO开销。

P3

**Graph Twiddling in a MapReduce World**

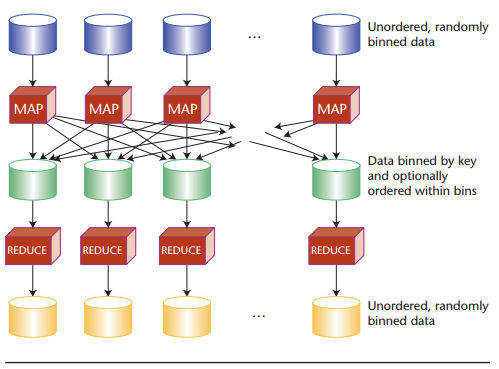
这篇文章首先讲述了如何将一个图的操作分解成一系列的MapReduce操作。这样的分解可以使得图算法能够运行在Cloud, Streaming或者单机环境下。

在多数情况下，我的MapReduce方法并不直接实现现有的graph算法，与之相反，他们抛弃的现有的算法，并且寻找能够产生相同输出新的规程。像我一样，很多人都会发现这样的规程其实是将一个问题分解成一系列的排序操作（而不是图的遍历操作），这一开始是一个障碍，但最后却发现非常有意义。

**The MapReduce Construct**

**The Process**

map和reduce操作都是非常普通的。他们接收一系列的记录（record），并且每条记录产生一个或多个输出，一条记录是由键值对组成的。一个简单的MapReduce Job如下所示：

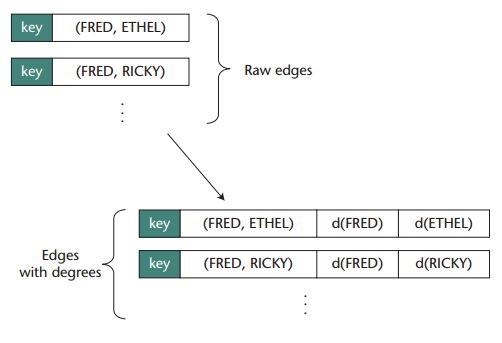


**Environmental Assumptions（环境假设）**

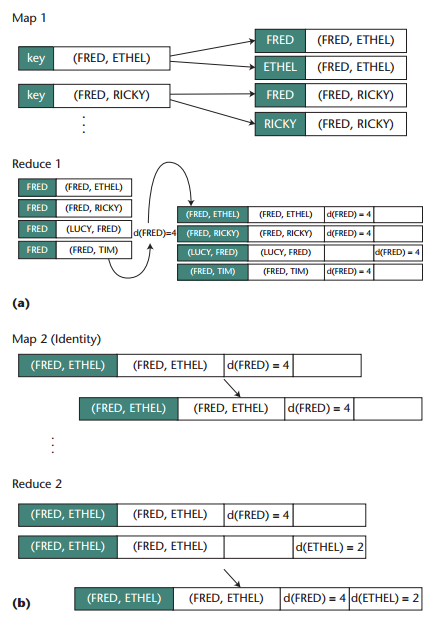
尽管用户可以在单机上使用MapReduce，但不得不说的是MapReduce计算框架是为分布式的云平台计算环境而设计的。MapReduce给并行计算带来的简化是只有当创建和访问文件的时候，才需要进行同步。Hadoop，一个用Java实现的MapReduce框架，能够在各自独立的机器中运行mapper和reducer实例。

Graph Algorithms

**An Example: Augmenting Edges with Degrees（给边增加节点的度）**



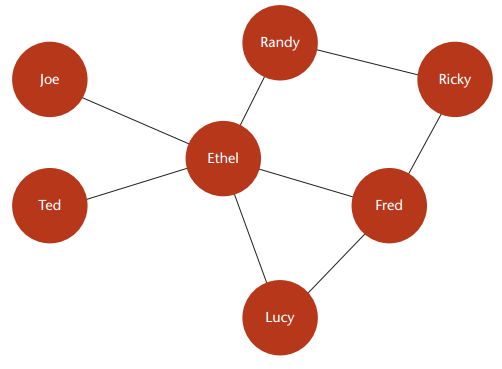
如上图所示，我们希望能够给每条边增加两个标注来分别表示源点和目标点的度。具体的实施过程如下图所示：



在第一次的Map-Reduce过程中，先将原来的每条边(key,(Fred, Ethel))map成两个元素：（Fred,(Fred, Ethel)）和（Ethel,(Fred, Ethel)）。其他的边依次内推，这样在Reduce的时候，只需要count分到同一个key中的元素数目，就可以得到这个key对应的度。于是得到若干个类似（（Fred, Ethel）,d(Fred)=4）的元素。

在第二次的Map-Reduce过程中，在map阶段，直接读取在第一阶段的Reduce结果，然后以边为key值，直接Reduce，合并源顶点和目标顶点的度，即可完成对每条边的标记。

下面是完整的示例：



上图中的每步迭代过程如下图所示：



**Simplifying the Graph**

许多图算法的首要阶段是对输入的图进行简化。即移除图中的循环以及重复出现的边。但有时候，会将边的重复的次数视为这条边连接的两个顶点之间的关联度，因此在合并相同边的时候需要考虑类似这样的问题。而且在移除相同边的时候，无向图和有向图的处理略微不同。例如针对（A,B）这条边，对于无向图来说，（B，A）就是重复的，需要将其移除；而对于有向图来说（B，A）不是重复的，需要被保留。这一小节后面的描述没有读懂，但是Flink的实现很简单。

**Enumerating Triangles**

MapReduce框架非常适合寻找Triangles。枚举所有的triangle一般可以分两步进行：第一步枚举出开放的元组（triplet）（A,B,C）满足（A，B），（A, C）是图中存在的边，那么第二步只需要验证（B, C）也是图中的边，即可构成封闭的三元组，即Triangle。注意到并不需要枚举出所有的triplet来定位triangle，其实一个triangle只需要一个triplet即可。

假设我有一个排序的顶点列表，进一步假设我记录了每条边的low-order member（这里应该是从小到大顺序，即对于边（a, b）有a<b），这样我就可以保证每个triangle都将只有一个顶点，这个顶点会收到它相关的两条边。这个顶点是这个triplet的两条边的交点，我可以选择一个顶点的顺序，我可以选择一个顶点的顺序，将所有的边根据他们最小顶点进行装箱，然后检测每个箱内的triplet是否能够被第三条边封住构成一个triangle.

这种方法的一个可能的问题是二次爆炸，其可以通过排出记录在箱中的边对而产生。为了避免这个问题，我们可以针对顶点排序设置一个巧妙的规则：根据度来排序。我根据低度优先的规则来记录每条边（low-degree member，即度数小的顶点放在前边，度数大的顶点放在后边），这样一来，度数大的顶点很少有边会分到一个箱中，因此所有的箱都不会变的很大。因此，二次方的搜索规模也不会成为问题（因为每个箱都很小）。当然，精心构造的图也会使得这种方式失效，但是大部分自然的图都不会出现这个情况。

首先根据前面两节讲述的Simplify和Degree，我们将图转换成带有顶点的度的标记的简单的无环图。在此之后，我们将进行两个MapReduce Job,如图2和图3所示。

在一个MapReduce阶段，Map的输入是上面的**Augmenting Edges with Degrees**中的输出，即带有节点的度标记的每条边作为输入，然后选择边的两个顶点中度较小的顶点作为key值，边作为value值，作为输出（如果边的两个顶点的度相同，可以选择标号较小的顶点作为key值）。在Reduce时，在每个箱中，都有一个顶点和该顶点相关联的边。（这是因为在map阶段我们是选用的顶点作为key值的）。reducer的工作就是按照某种规则，将箱中的边都组合成对边的形式：组合方式为对边所组合而成的triplet的两条边相交的顶点是该箱的key.而这条记录的key值必须是这个triplet所缺的第三条边的两个顶点组成，这样就可以在下次的mapper过程中将含有这两个顶点的边分配到一个箱中。注意到选取最小的度的顶点作为分箱是为了减少每个箱中的边的数目，从而有效的减少了边对的数目。

在第二个MapReduce阶段，这需要两个输入：一个是来自于Reduce1的结果，另一个是来自带有度标记的边集。它的工作就是量这两个记录结合起来。特别需要注意的是边集中每条边中顶点的排列规则必须和triplet中另外两个顶点的排列规则保持一致。比如都是从小到大排列。因为只有这样才能保证他们能够映射到相同的key上。当一条边和一个triplet分到同一个箱的时候，表明他们可以构成一个封闭的triangle。一般情况下，一个箱中会有至多一条边和任意数目的triplet,有多少个triplet就对应的有多少个triangle。

下面我们将以如下图来进行说明：

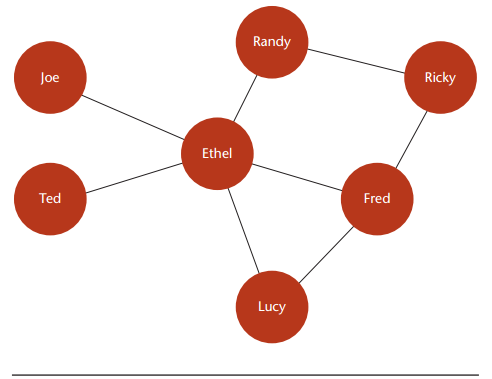


图1. 一个简单的无向图

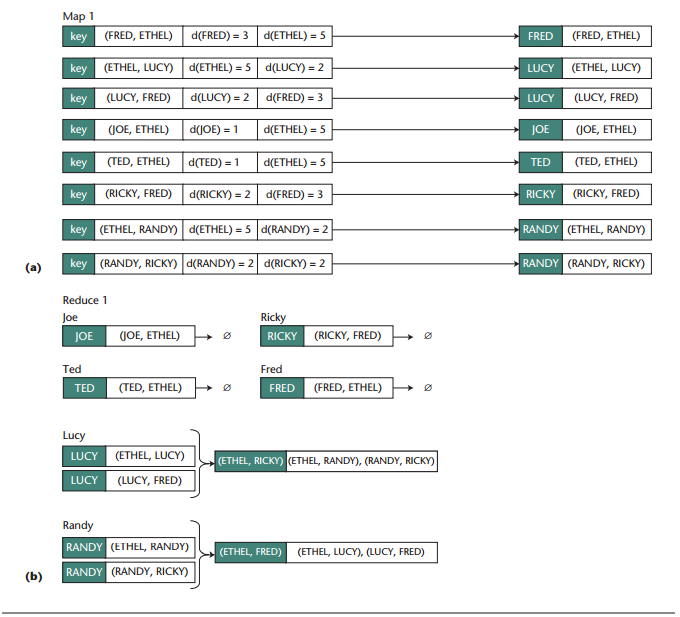


图2. Triangle Count MapReduce1

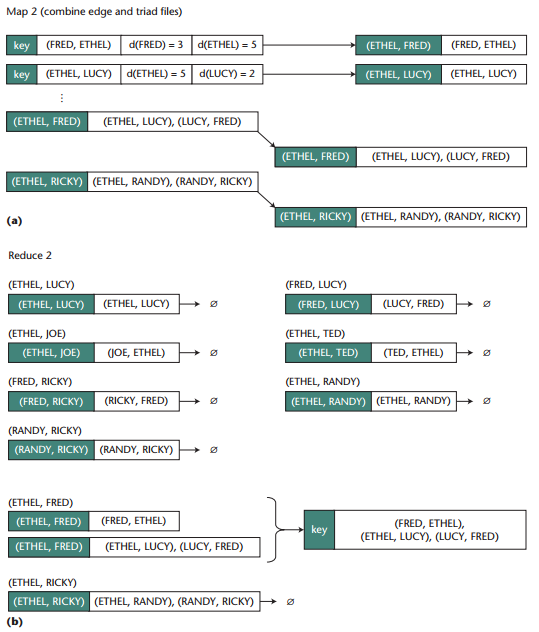


图3. Triangle Count MapReduce2

**思考1，为什么这种装箱不会漏掉解？**

等价于证明对于一个三角形的三条边，不会拆分到三个不同的箱中。

**思考2，为什么这种方式的解不会重复？**

在合并的过程中保证了一个三角形只会有一个triplet，所以不会重复。

#### AF16 Local Clustering Coefficient

**资料：**

聚类系数定义：

<https://en.wikipedia.org/wiki/Clustering_coefficient>

<http://blog.csdn.net/minenki/article/details/8606515>

<http://blog.sina.com.cn/s/blog_439371b501012lgw.html>

计算规则：

<http://blog.csdn.net/pennyliang/article/details/6838956>

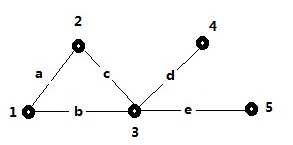
**定义：**

在图论中，集聚系数是图中的点倾向于集聚在一起的程度的一种度量。证据显示：在多数实际网络以及特殊的社会网络中，结点有形成团的强烈倾向。在实际网络中，这种可能性比随机生成的均匀网络的两个结点间连接的可能性大。

这一度量有两种版本的方法：全局的和局部的。全局的方法旨在度量整个网络的集聚（性），而局部的（方法）给出了单个结点的嵌入性的度量。

**三元组（triplet）**：有两条（开三元组）或者三条（闭三元组）无向边连接的三个节点。

计算示例如下：



可以用下面结构定义一个triplet

struct triplet

{

int key;

set<int> pair;

};

例如上图{1，(2,3)}构成的triplet是封闭的，{3,（4,5）}构成的triplet是开放的。

对于局部聚集系数：

**Local Clustering Cofficient = number of edges / number of supposed edges**

针对某个节点v,找出v的所有邻接点，这些邻接点之间的边的数目/这些邻接点之间总共应该有的边的数目（C(n,2)）。

例如：

1节点的邻居节点（2,3），他们之间构成的边有1条，可能构成的边1条，因此1/1=1

2节点的邻居节点（1,3），他们之间构成的边有1条，可能构成的边1条，因此1/1=1

3节点的邻居节点（1,2,4,5），他们之间构成的边有1条，可能构成的边(4\*3)/2条，因此1/6=1/6

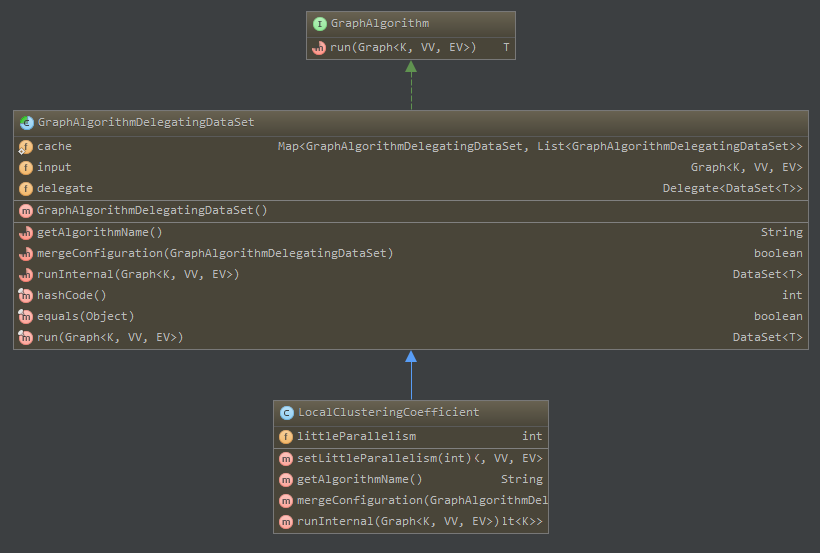
4节点的邻居节点（3），他们之间构成的边有0条，可能构成的边0条，因此0

5节点的邻居节点（3），他们之间构成的边有0条，可能构成的边0条，因此0

则，5个节点平均local Clustering coefficient = (1+1+1/6)/5=13/30

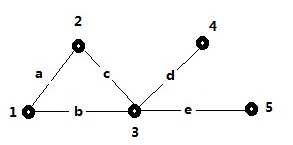
**实现：**

**a. 类图**



**b.核心代码**

以如下图的例子解释代码运行逻辑：



该图的边集为：（1,2,a）（1,3,b）（3,2,c）（4,3,d）（3,5,e）

**第一步：**利用TriangleListing算法，计算所有的三角形，得到如下数据集

**DataSet<Tuple3<K,K,K>> triangles** （u, v, w）

(1,2,3)

**第二步：**根据第一步的计算结果，将每个点u映射成（u,1）类型。

**DataSet<Tuple2<IntValue, LongValue>> triangleVertices**

(1,1)

(2,1)

(3,1)

**第三步**：根据第二步的计算结果，合并每个点u。（u, triangle count）

**DataSet<Tuple2<IntValue, LongValue>> vertexTriangleCount**

(3,1)

(1,1)

(2,1)

**第四步**：计算每个顶点的度。（u, deg(u)）。

**DataSet<Vertex<IntValue, LongValue>> vertexDegree**

(3,4)

(1,2)

(5,1)

(2,2)

(4,1)

**第五步**：统计结果（u, deg(u), triangle count）：

**DataSet<Vertex<T, Tuple2<LongValue, LongValue>>> result**

(3,(4,1))

(1,(2,1))

(5,(1,0))

(2,(2,1))

(4,(1,0))

**第六步**：计算LCC的值。

Vertex ID: 3, vertex degree: 4, triangle count: 1, local clustering coefficient: 0.16666666666666666

Vertex ID: 1, vertex degree: 2, triangle count: 1, local clustering coefficient: 1.0

Vertex ID: 5, vertex degree: 1, triangle count: 0, local clustering coefficient: NaN

Vertex ID: 2, vertex degree: 2, triangle count: 1, local clustering coefficient: 1.0

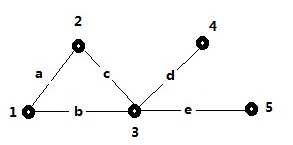
Vertex ID: 4, vertex degree: 1, triangle count: 0, local clustering coefficient: NaN

public DataSet<Result<K>> runInternal(Graph<K, VV, EV> input)  
 throws Exception {  
 // u, v, w  
 DataSet<Tuple3<K,K,K>> triangles = input  
 .run(new TriangleListing<K,VV,EV>()  
 .setLittleParallelism(littleParallelism));  
  
 // u, 1  
 DataSet<Tuple2<K, LongValue>> triangleVertices = triangles  
 .flatMap(new SplitTriangles<K>())  
 .name("Split triangle vertices");  
  
 // u, triangle count  
 DataSet<Tuple2<K, LongValue>> vertexTriangleCount = triangleVertices  
 .groupBy(0)  
 .reduce(new CountTriangles<K>())  
 .setCombineHint(CombineHint.*HASH*)  
 .name("Count triangles");  
  
 // u, deg(u)  
 DataSet<Vertex<K, LongValue>> vertexDegree = input  
 .run(new VertexDegree<K, VV, EV>()  
 .setParallelism(littleParallelism)  
 .setIncludeZeroDegreeVertices(true));  
  
 // u, deg(u), triangle count  
 return vertexDegree  
 .leftOuterJoin(vertexTriangleCount)  
 .where(0)  
 .equalTo(0)  
 .with(new JoinVertexDegreeWithTriangleCount<K>())  
 .setParallelism(littleParallelism)  
 .name("Clustering coefficient");  
}

#### AF17 Global Clustering Cofficient

对于全局聚集系数：

**Global Clustering Cofficient = number of closed triplet / number of triplet**



以上图为例:

closed triplet ={1，(2,3)}，{2，(1,3)}，{3，(1,2)}

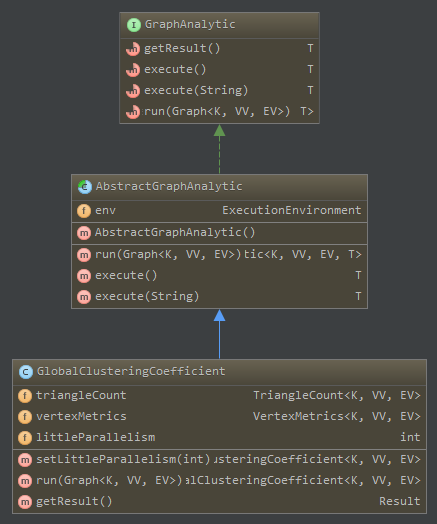
all triplet = {1，(2,3)}，{2，(1,3)}，{3，(1,2)}，{3，（2,4）}，{3，（4,5）}，{3，（1,5）}，{3，（2,5）}，{3，（1,4）}

number of closed triplet = 3

number of triplet = 8

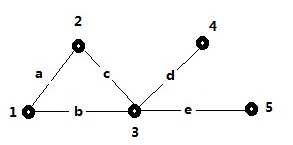
number of triplet / number of triplet = 3/8

**a.类图**



**b. 核心代码**

还是以该图为例进行分析



**第一步：**计算三角形的数目。

**private TriangleCount<K, VV, EV> triangleCount;**

1

**第二步：**计算三元组的数目。

**VertexMetrics<IntValue,NullValue, NullValue> vertexMetrics**

在这里借助了**VertexMetrics**类，VertexMetrics用来记录图中的顶点数目，边的数目和三元组的数目。

VertexMetrics是如何计算三元组的数目呢？

首先，VertexMetrics计算每个顶点的度，接着，根据顶点的度来计算三元组的数目，即三元组的数目等于deg(u) \* (deg(u)-1) / 2。

**第三步**：计算GCC。

gcc = 3 \* **triangleCount / tripletCount**

GlobalClusteringCofficient核心代码如下：

public GlobalClusteringCoefficient<K, VV, EV> run(Graph<K, VV, EV> input)  
 throws Exception {  
 super.run(input);  
  
 triangleCount = new TriangleCount<K, VV, EV>()  
 .setLittleParallelism(littleParallelism);  
  
 input.run(triangleCount);  
  
 vertexMetrics = new VertexMetrics<K, VV, EV>()  
 .setParallelism(littleParallelism);  
  
 input.run(vertexMetrics);  
  
 return this;  
}

VertexMetrics核心代码如下：

public VertexMetrics<K, VV, EV> run(Graph<K, VV, EV> input)  
 throws Exception {  
 super.run(input);  
  
 DataSet<Vertex<K, LongValue>> vertexDegree = input  
 .run(new VertexDegree<K, VV, EV>()  
 .setIncludeZeroDegreeVertices(includeZeroDegreeVertices)  
 .setReduceOnTargetId(reduceOnTargetId)  
 .setParallelism(parallelism));  
  
 vertexDegree  
 .output(new VertexMetricsHelper<K>(id))  
 .name("Vertex metrics");  
  
 return this;  
}

# 3. Spark图算法

表2. Spark 图算法表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 名称 | 描述 | 应用 | 难度 |
|  | Label Propagation |  |  |  |
|  | Connected Components |  |  |  |
|  | PageRank |  |  |  |
|  | Shortest Paths |  |  |  |
|  | Strongly Connected Components |  |  |  |
|  | SVDPPlusPlus |  |  |  |
|  | Triangle Count |  |  |  |

# #. 问题

1. Flink和Spark的编程模型趋同，而且在调试的时候发现Flink也属于延迟计算类型，只记录数据源和算子，并没有真正的去执行计算（需要深入理解）=> 总结Flink和Spark的编程模型，并且给出系统全面的比较。（10）interesting.

(1) <https://www.zhihu.com/question/30151872>

(2)<http://blog.madhukaraphatak.com/introduction-to-flink-for-spark-developers-flink-vs-spark/>

2. Flink和Spark的图算法的布局都趋于相同。（6）

数据分析和运算是Python/R/Matlab的强项，现在在Flink/Spark上构建数据分析运算模块是否合理？ give up.

3. 可以扩展的地方：（8）

AF9算法中Flink的实现版本是比较直观的，网上还给出了一种基于矩阵运算的算法，是否能够加速？=>Flink是否支持矩阵运算？=>图论中有很多利用矩阵来进行运算的例子，而Flink的算法是建立在顶点和边的集合上的运算，如果能够将这种运算转换为矩阵运算，构建矩阵运算模型来代替图运算模型，是否能够加速呢？maybe difficult, give up.

4. Flink的Graph模型是建立在DataSet上的，即批处理上，如何在流处理上建立Graph模型，或者跑Graph相关算法？（10）key problem

5. 之前上过一门课程是社交媒体大数据挖掘，里面提到非常多的算法，是否可以在Flink/Spark上实现其中的某些有价值的算法？（10）maybe interesting.

6. 如果提出针对图算法的测试Benchmark，算不算成果？（10）

7. Benchmark

测试方法：

（1）顶点和边的数量

（2）边的稀疏度

（3）机器的扩展性

数据来源：

（1）真实的DataSet，twitter等；

（2）人为生成的。

性能评价：

扩展性：数据量增大时的曲线图。

可靠性：出错？

可重复性：多次跑同样的数据集，结果是否一致或相差不大？

8. 优化思路

算法角度：

系统角度：

结合。

9. 当提交的任务占用内存资源过多时，会报OOM错误，整个集群宕机。这种问题如何侦测，解决？（云计算环境下，如何检测用户提交的任务不是非法的，不会影响到整个集群）

10. 现在有很多更加专一和强大的计算框架，为什么不直接用例如Giraph / Pregel等这样的图计算框架而要研究Flink?

# #. 记录

## 2016.10.17

**郑莹莹**

1. Iterative Graph Processing

barrier是如何实现的？barrier加在哪里？

2. scatter-gather

发送时，发送的顺序？

接收时，接收有顺序？接收了所有的之后（缓存）再更新，还是接收了之后就立刻更新。

3. 单源点最短路径 => 任意两两之间的最短路径的实现

4. 三种算法的区别？

scatter：以顶点为视角，发送边；

gather：以边为视角，收集边；

两段式，三段式的设计区别和举例。

**深入理解框架模型的设计理念。**

5. Scatter-Gather 和任何节点通信？如何理解？

任意的，满足交换律，结合律。

**重点：深入理解三种模型。**

6. Graph能否画出来？（D3?）

7. Spark VS Flink算法实现的对比=>迭代模型的对比。

8. 强连通图的实现细节，边排序？

9. 算法使用什么模式来实现的？读取数据的过程和迭代计算的过程？

## 2016.11.07

1. 基本算子的使用：map, flatMap, join等等。

2. 算法在执行时考虑并行度问题。

3. 幂律分布。

4. 和师兄讨论毕设问题。

## 2016.11.11

1. 分片的策略如何。

2. 数据分布图。数据流图。数据之间的依赖。

3. 如何进行pipeline划分的。

## 2016.11.17

1. DataSet 和 DataStream有什么区别和联系？

2. 是否可以统一DataSet和DataStream模型？

3. Flink对算子执行的优化：map().first() 和 first().map()

4. Flink中的算子并行度如何起作用？算子是如何执行的？

5. 记录并调试Flink Stream的执行过程。

6. Stream Graph.

重心放在图上，Gelly,

算法的实现原理. -> 流式该如何转.

中期答辩，把算法的思路讲清楚.

算法的实现，如果边改变，算法该如何适应.

3个算法

7. 如何衡量论文和实践的差距？

## 2016.11.25

**赵伟**

1. Semi-streaming

2. triangle count 精确计算和近似计算？

3. random-walks问题，用在什么方面，解决什么问题。

4. 窗口定义。

5. 找出来三四个算法，在流处理上做是有意义的。

6. Dynamic Graph和Streaming Graph区别。

7. Triangle Count Spark实现细节。考虑Triangle Count的算法，哪一种可以实现在Streaming上。

batch如何实现 | Streaming 如何实现

## 2016.12.16

段世凯

1. 抽象批处理和流处理的异同，细致的构建模型。

2. Delta Iteration 迭代模型。