1. Flink 图算法

表 1. Flink 图算法表

编号	名称	描述	应用	难度
AF1	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	
AF2	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。
AF3	Connected Components			
AF4	GSA Connected Components			
AF5	PageRank			
AF6	GSA PageRank			
AF7	Single Source Shortest Paths			
AF8	GSA Single Source Shortest Paths			
AF9	Triangle Count			
AF10	Triangle Listing			
AF11	Triangle Enumerator			
AF12	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	网页质量分析	

		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。		
AF13	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
AF14	Adamic-Adar	**节点相似性算法**		资料较少
AF15	Jaccard Index	**节点相似性算法** 用来计算两个有限集合的相似度.		
AF16	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。		
AF17	Global Clustering Cofficient	**全局聚类系数算法**		

1.1 Flink 图模型

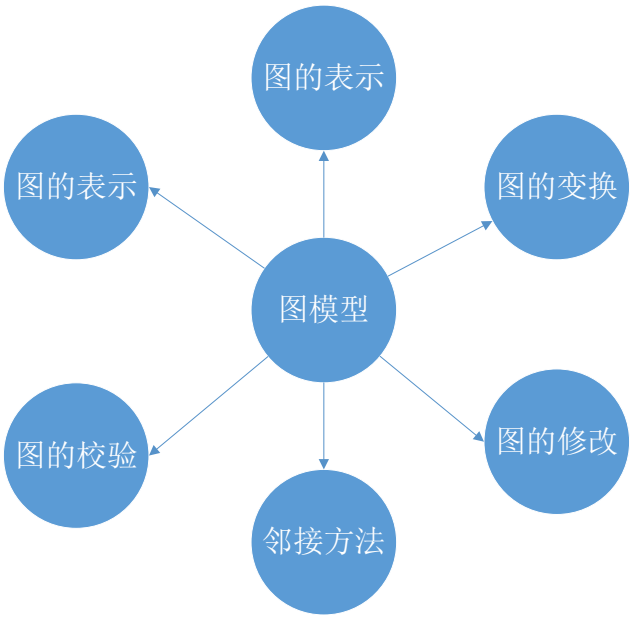


图 1. Flink 图模型

1.2 Gelly Library

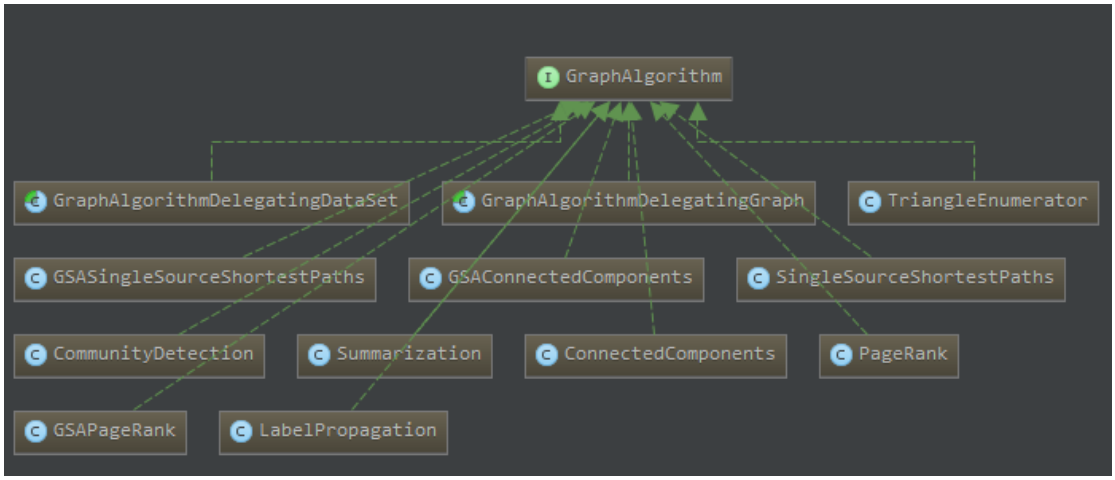
1.2.1 算法框架

（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：

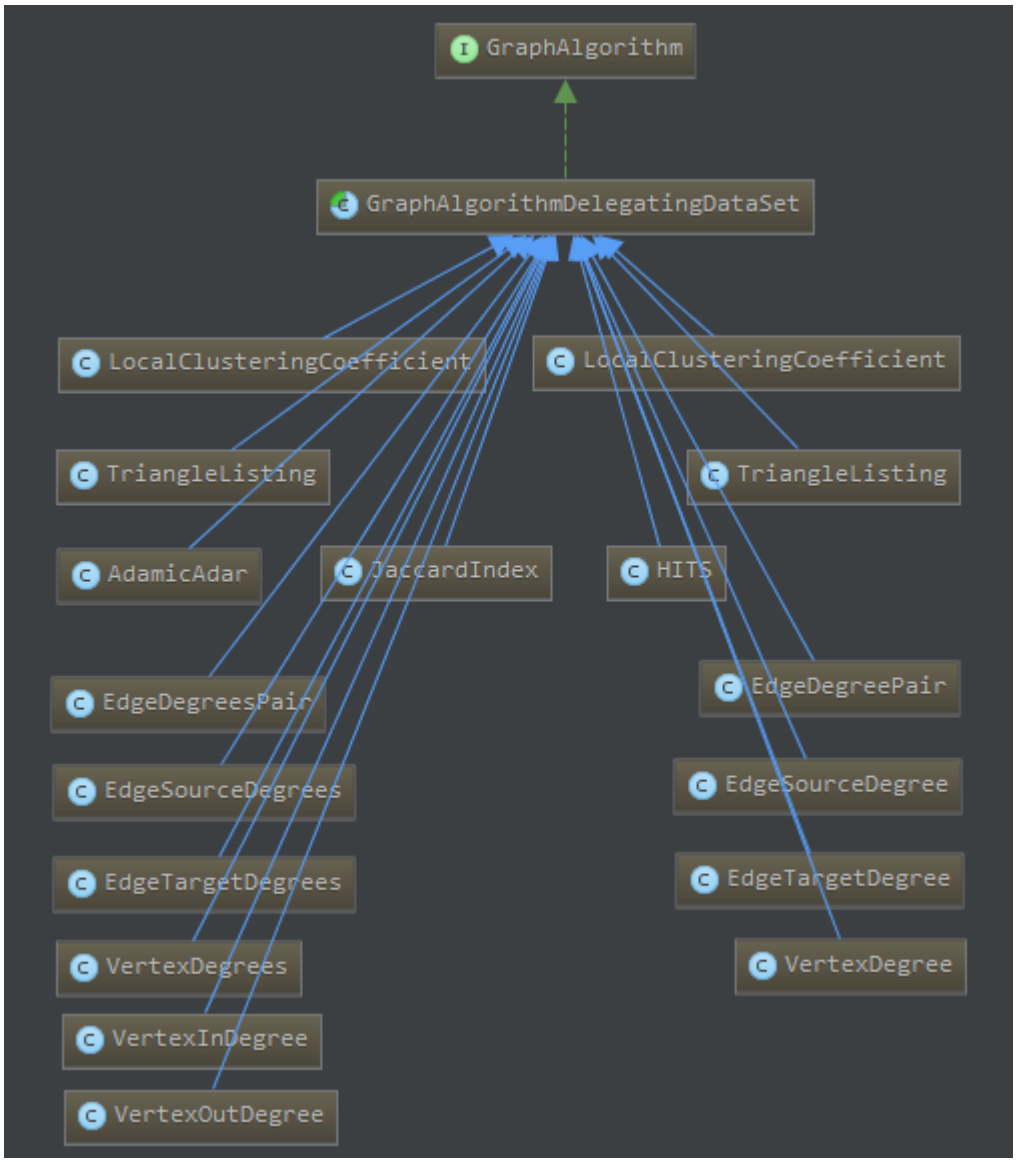
- cache: 全局缓存
- input: 算法所需要的输入的图
- delegate: 算法输出的数据集的代理对象

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：



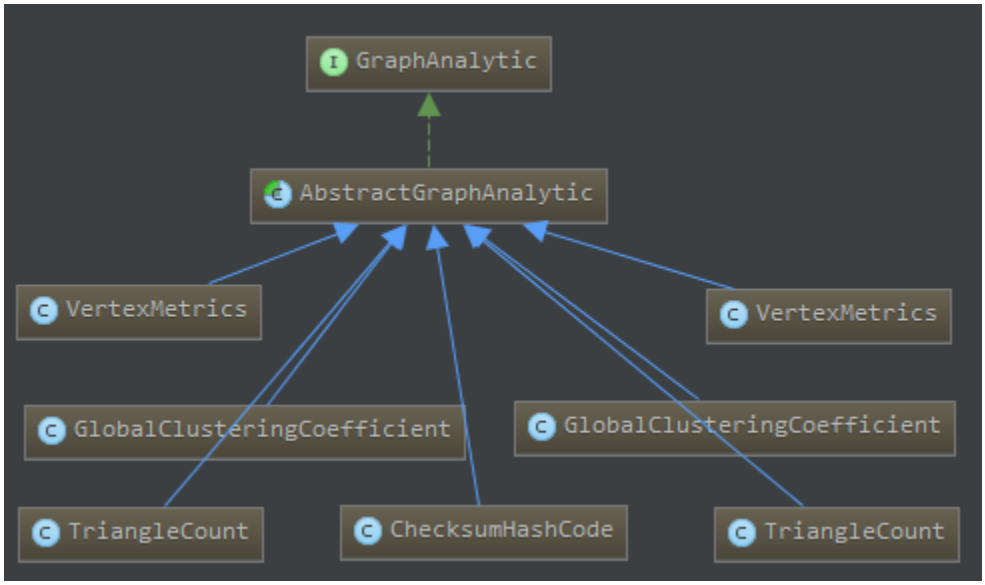
GraphAnalytic 和 GraphAlgorithm 接口很像，不同之处在于 GraphAnalytic 是末端的（即算法的最后一个算子），而且需要通过 accumulator 来获取计算结果。一个 Flink 程序是单点执行的，GraphAnalytic 推迟执行，允许多个 analytics 和 algorithms 在一个单独的程序里。

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;
}
```



1.2.2 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。

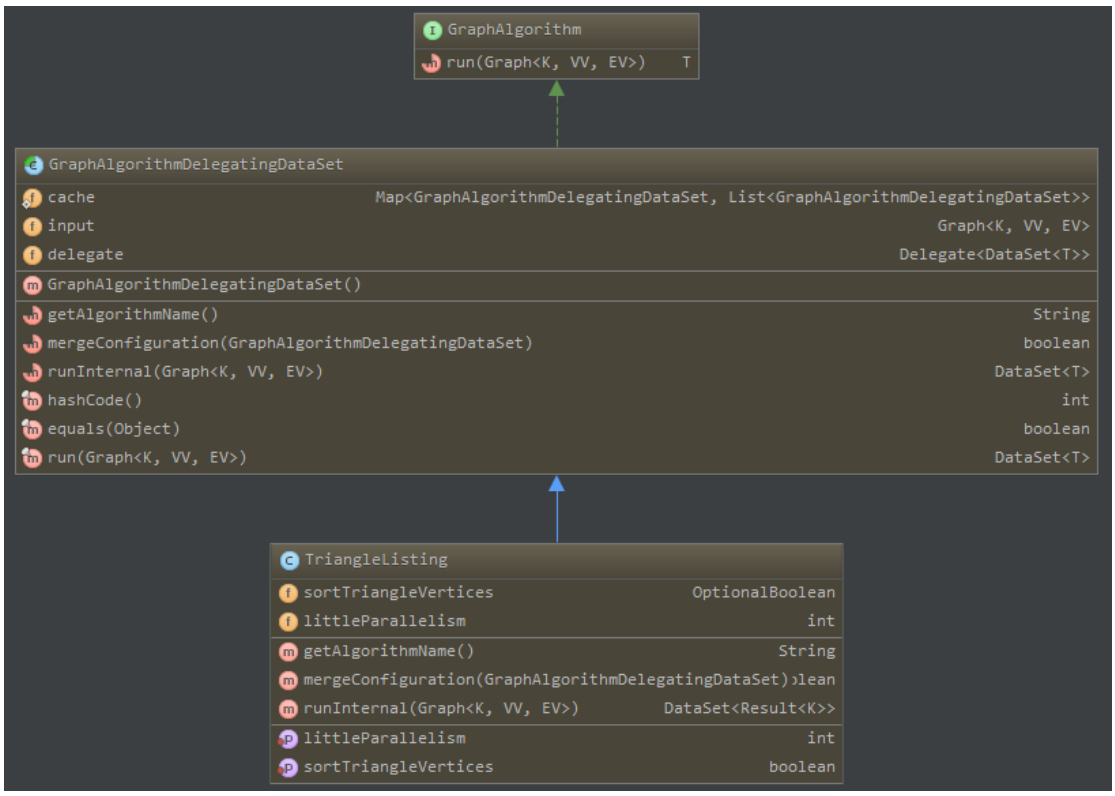
（聚集系数）

逻辑：

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

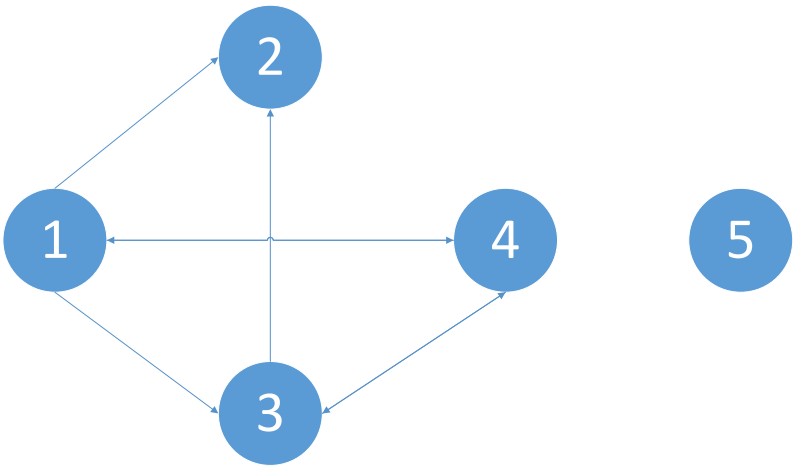
实现：

（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 $0 < \varepsilon, \delta \leq 1$, a random variable $\hat{X}$ is an (ε, δ)-approximation of the true quantity X if $|\hat{X} - X| \leq \varepsilon X$ with probability at least $1 - \delta$.

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：

Table 1: Notations

Symbol	Description
$G = (V_G, E_G)$	A simple undirected graph
$adj_G(v)$	The set of adjacent vertices of v in G
$deg_G(v)$	Degree of v in G
$\triangle_{uvw}$	A triangle formed by u, v and w
$\triangle(v)$	The set of triangles that contains v (Eq. 1)
$N_\triangle(v)$	The triangle number of v, $N_\triangle(v) = \lvert\triangle(v)\rvert$
$\triangle(G)$	The set of all triangles in G (Eq. 2)
$N_\triangle(G)$	The number of triangles in G (Eq. 3)
$N_\vee(v)$	The number of open triangles centered at v (Eq. 4)
M	Available main memory size
B	Disk block size
$scan(N)$	$\Theta(N/B)$ I/Os
$sort(N)$	$\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

一个以 v 为中心的开放三角形的数目为：

$$N_\vee(v) = \frac{1}{2} deg_G(v)(deg_G(v) - 1).$$

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

Algorithm 1 *In-Memory Triangle Listing*

Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$

1. $\triangle(G) \leftarrow \emptyset$;
2. **for each** $u \in V_G$ **do**
3. **for each** $v \in adj_G(u)$, where $v > u$, **do**
4. **for each** $w \in (adj_G(u) \cap adj_G(v))$, where $w > v$, **do**
5. $\triangle(G) \leftarrow (\triangle(G) \cup \{\triangle_{uvw}\})$;
6. **return** $\triangle(G)$;

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。

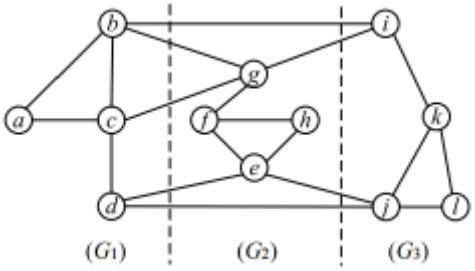
4.2 Correctness and Global-Completeness of Local Triangle Listing

我们的算法基于如下定理：

LEMMA 1. *Let* $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$ *be a partition of* G, *where* $\cup_{1 \le i \le p} V_{G_i} = V_G$ *and* $V_{G_i} \cap V_{G_j} = \emptyset$ *for* $1 \le i < j \le p$. *Then,* $\triangle(G) = \triangle 1 \cup \triangle 2 \cup \triangle 3$, *where* $\triangle 1$, $\triangle 2$, *and* $\triangle 3$ *are disjoint sets defined as follows.*

- $\triangle 1 = \cup_{1 \le i \le p} \{\triangle_{uvw} : u, v, w \in V_{G_i}\}$.
- $\triangle 2 = \cup_{1 \le i,j \le p \wedge i \ne j} \{\triangle_{uvw} : u, v \in V_{G_i}, w \in V_{G_j}\}$.
- $\triangle 3 = \cup_{1 \le i < j < k \le p} \{\triangle_{uvw} : u \in V_{G_i}, v \in V_{G_j}, w \in V_{G_k}\}$.

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：

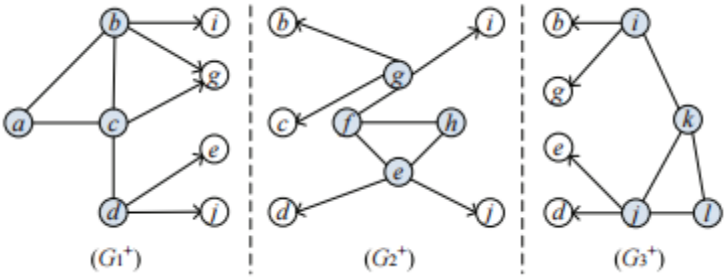


根据定理 1，一个三角形△uvw 可以通过搜索任意的子图 Gi，并且仅当 uvw 在子图 Gi 中时能够被找到（即 Type1 类型的 triangle）。然而这类的 triangle 数量有限，更关键的是我们不能移除掉任何边以及顶点即使我们已经列举出所有的 Type1 类型的三角形，因为一条边 (u, v) 极可能组成△uvw，也可能组成△uvx，而 x 在其他的子图中。

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**）。

Definition 1 (EXTENDED SUBGRAPH). *Let* $H = (V_H, E_H)$ *be a subgraph of* G. *An* extended subgraph *of* H *in* G, *denoted by* H^+, *is a* directed *subgraph defined as* $H^+ = (V_{H^+}, E_{H^+})$, *where* $V_{H^+} = V_H \cup \{v : u \in V_H, v \in V_G \backslash V_H, (u,v) \in E_G\}$ *and* $E_{H^+} = \{(u,v) : (u,v) \in E_G, u \in V_H\}$.

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：

LEMMA 2. *Let* H^+ *be an extended subgraph of a subgraph* H *of* G. *Then:*

- *Let* $\triangle 1(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 1,\ u,v,w \in V_H\}$. *Then,* $\forall \triangle_{uvw} \in \triangle 1(H^+)$, $\triangle_{uvw}$ *can be listed by searching* H^+ *alone.*
- *Let* $\triangle 2(H^+) = \{\triangle_{uvw} : \triangle_{uvw} \in \triangle 2,\ u,v \in V_H\}$. *Then,* $\forall \triangle_{uvw} \in \triangle 2(H^+)$, $\triangle_{uvw}$ *can be listed by searching* H^+ *alone.*

In addition, for any edge $(u,v) \in E_H$, (u,v) *does not exist in any triangle in* $\triangle(G) \backslash (\triangle 1(H^+) \cup \triangle 2(H^+))$.

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：

Algorithm 2 *I/O-Efficient Triangle Listing*

Input: A graph $G = (V_G, E_G)$
Output: A listing of $\triangle(G)$

1. **while**(G is not empty)
2. Partition G into $\mathcal{P} = \{G_1, \ldots, G_i, \ldots, G_p\}$;
3. **for each** extended subgraph G_i^+ of $G_i \in \mathcal{P}$ **do**
4. List all triangles in $\triangle 1(G_i^+)$ and $\triangle 2(G_i^+)$ (by Algorithm 3);
5. Remove all edges in G_i from G;

Algorithm 3 *Triangle Listing in Extended Subgraph*

Input: An extended subgraph $H^+ = (V_{H^+}, E_{H^+})$
Output: A listing of $\triangle 1(H^+)$ and $\triangle 2(H^+)$

1. **for each** $u \in V_H$ **do**
2. **for each** $v \in adj_H(u)$, where $v > u$, **do**
3. **for each** $w \in (adj_{H^+}(u) \cap adj_{H^+}(v))$ **do**
4. **if**($w > v$ or $w \notin V_H$)
5. List $\triangle_{uvw}$;

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

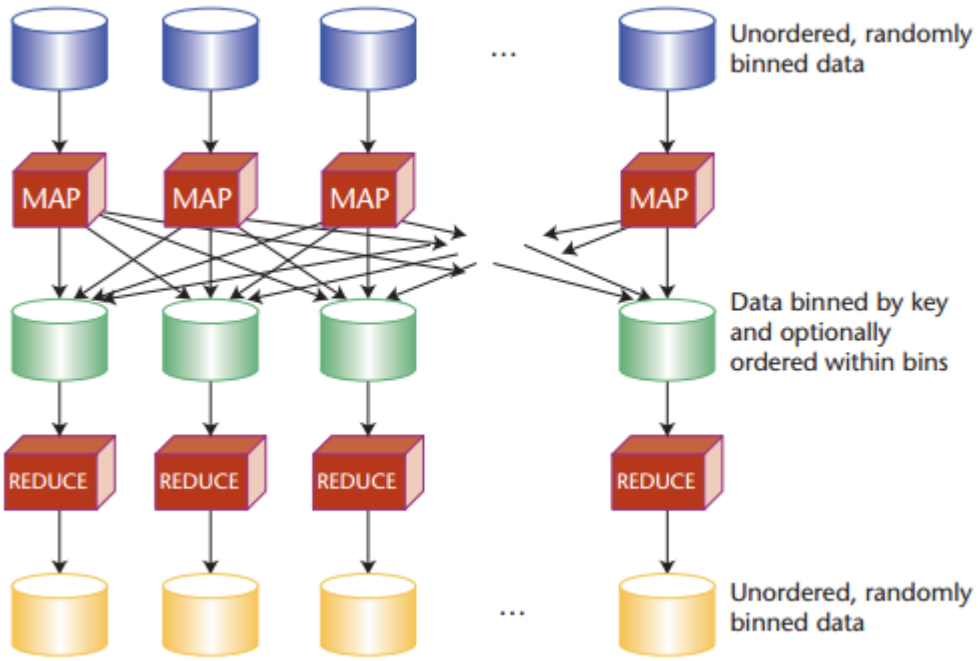
这篇文章首先讲述了如何将一个图的操作分解成一系列的 MapReduce 操作。这样的分解可以使得图算法能够运行在 Cloud, Streaming 或者单机环境下。

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：

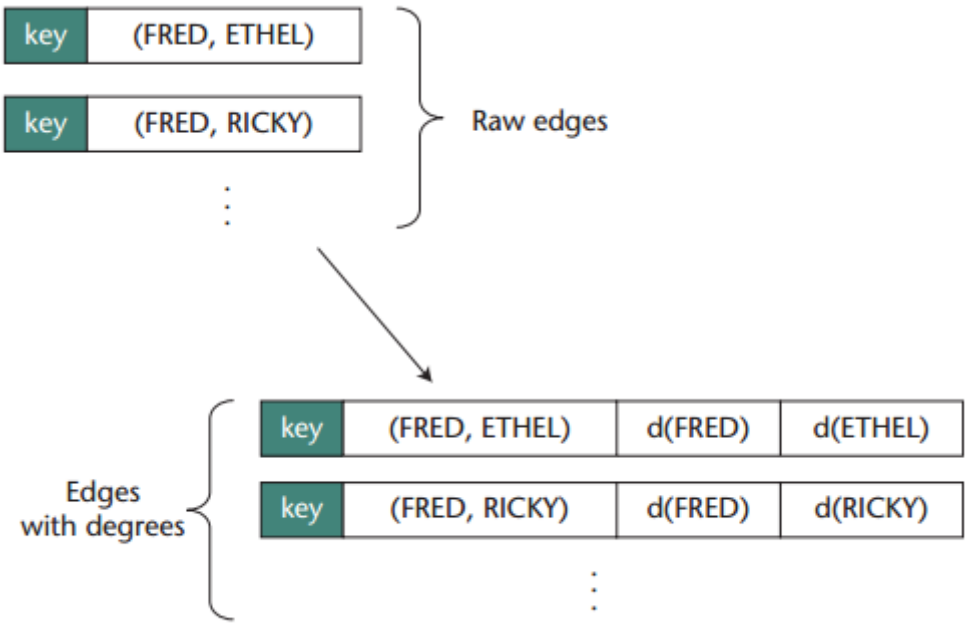


Environmental Assumptions（环境假设）

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。

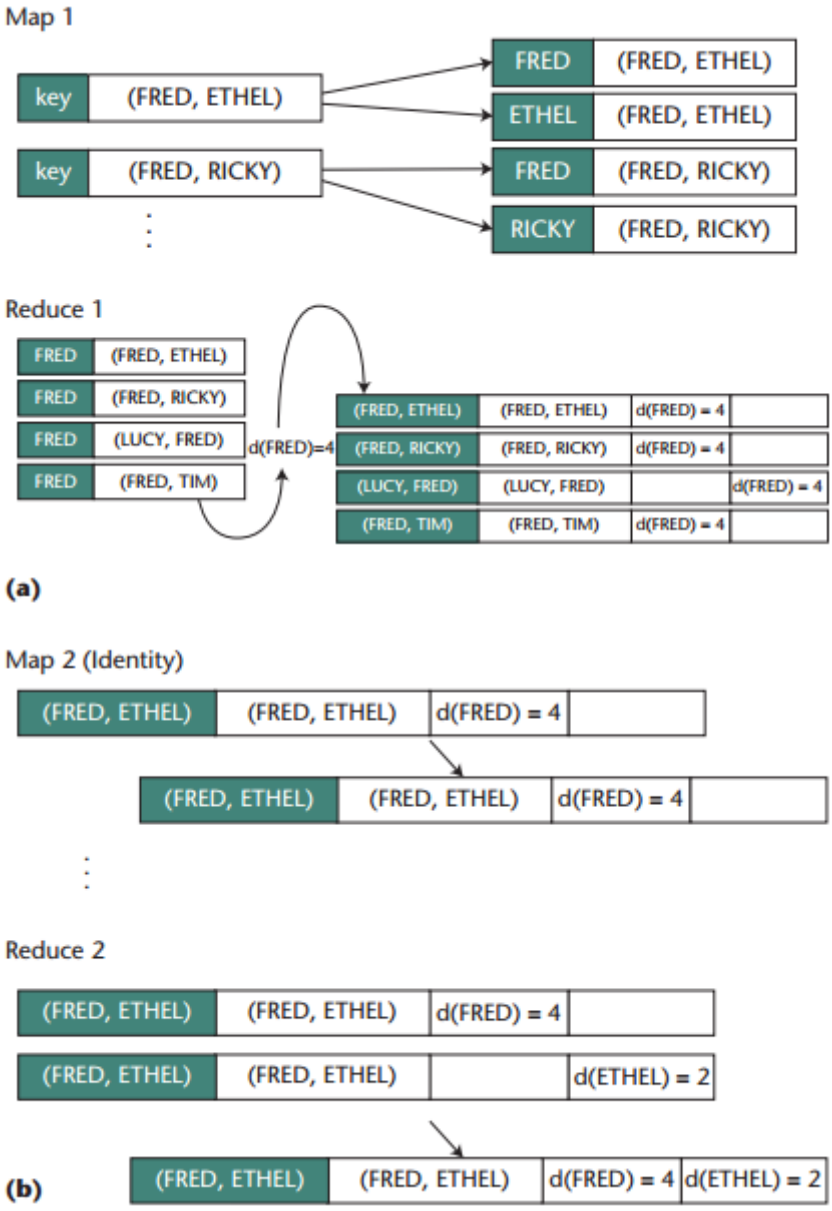
Graph Algorithms

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

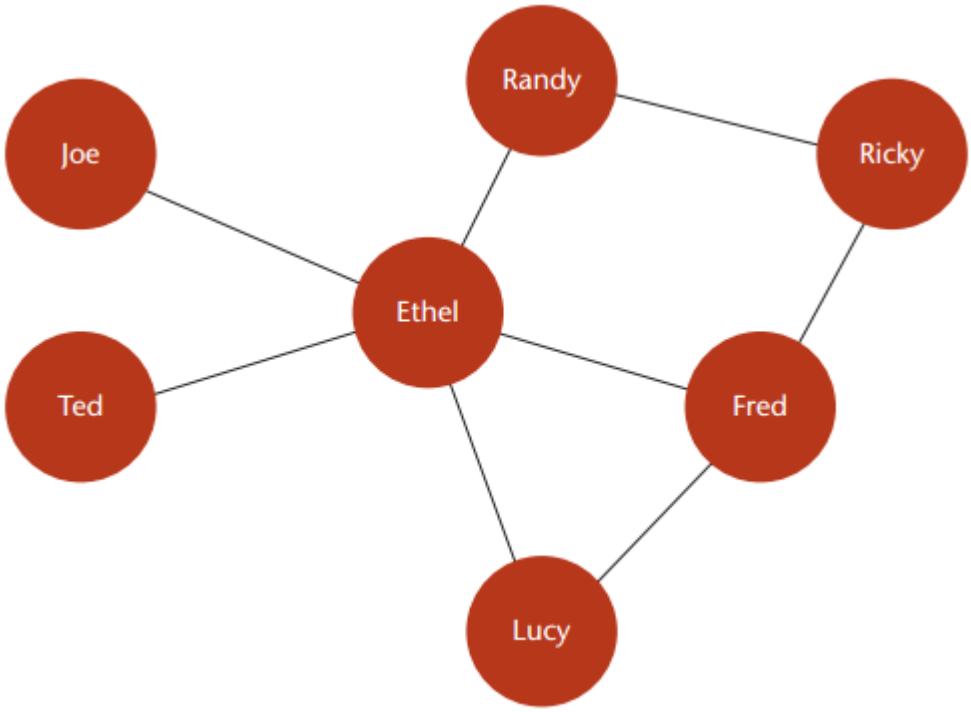
的度。具体的实施过程如下图所示：



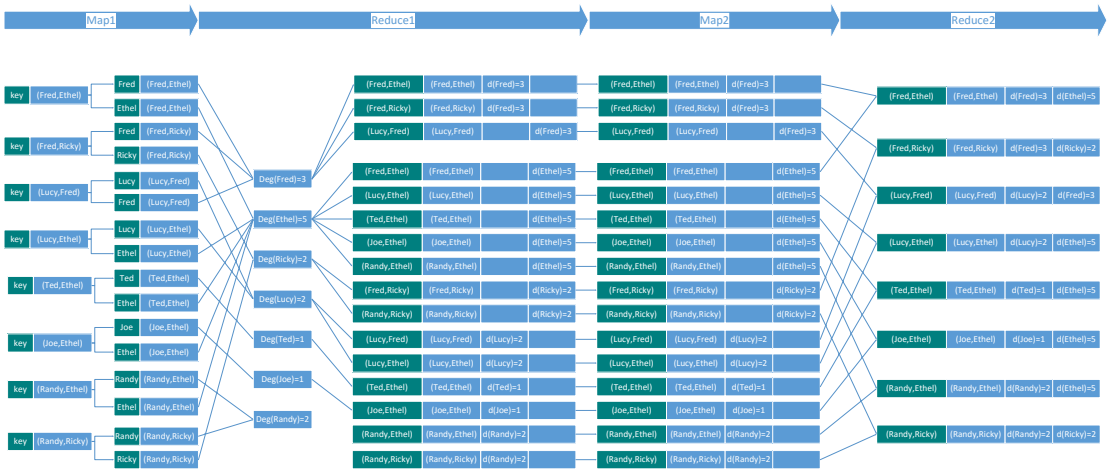
在第一次的 Map-Reduce 过程中，先将原来的每条边(key,(Fred, Ethel))map 成两个元素：（Fred,(Fred, Ethel)）和（Ethel,(Fred, Ethel)）。其他的边依次内推，这样在 Reduce 的时候，只需要 count 分到同一个 key 中的元素数目，就可以得到这个 key 对应的度。于是得到若干个类似（（Fred, Ethel）,d(Fred)=4）的元素。

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：

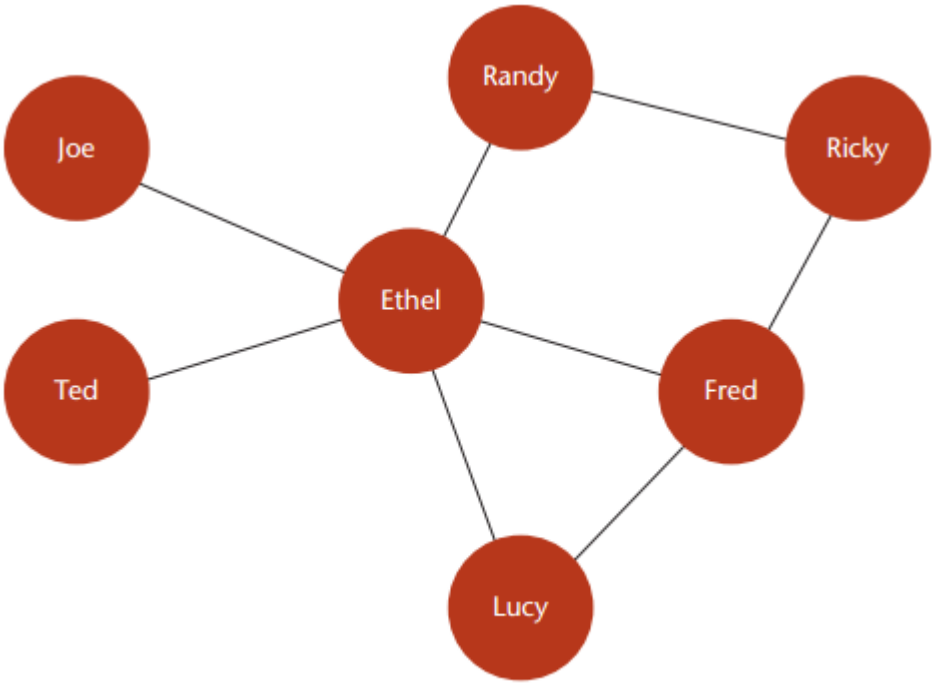

Randy
Joe
Ricky
Ethel
Ted
Fred
Lucy

图 1. 一个简单的无向图

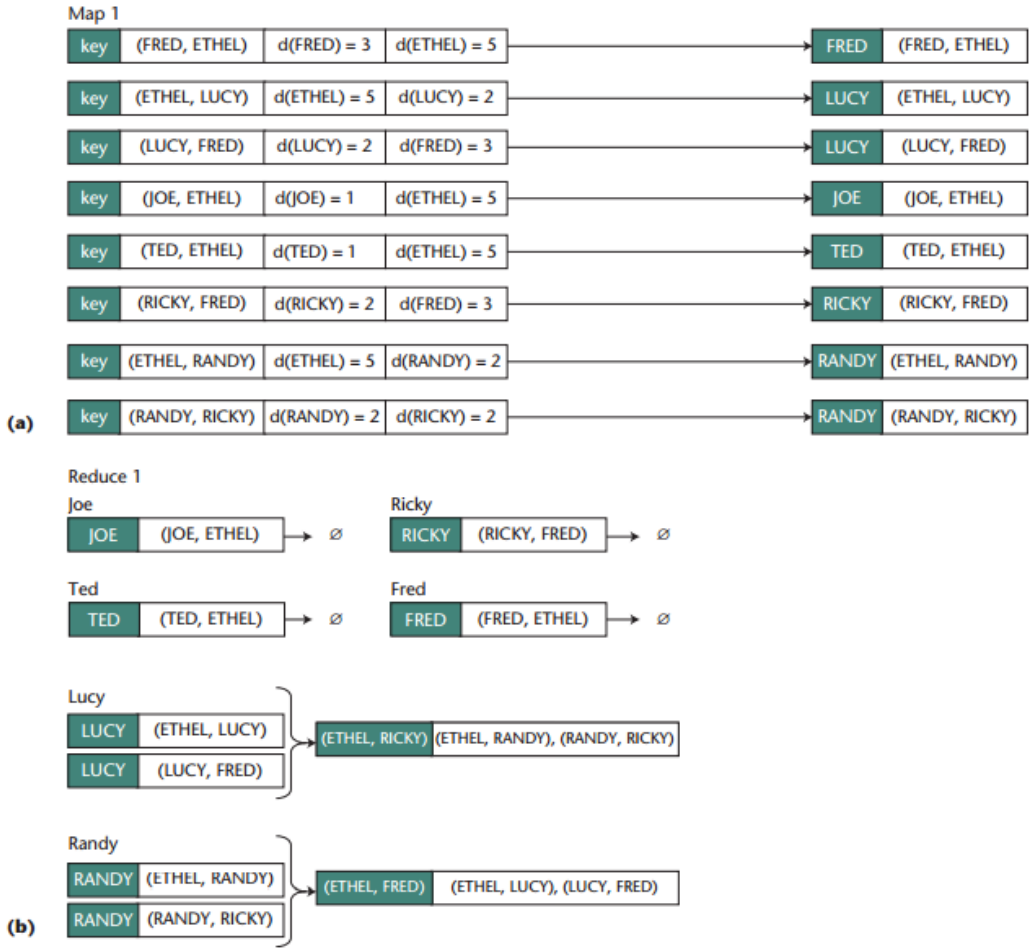

Map 1
key (FRED, ETHEL) d(FRED) = 3 d(ETHEL) = 5 FRED (FRED, ETHEL)
key (ETHEL, LUCY) d(ETHEL) = 5 d(LUCY) = 2 LUCY (ETHEL, LUCY)
key (LUCY, FRED) d(LUCY) = 2 d(FRED) = 3 LUCY (LUCY, FRED)
key (JOE, ETHEL) d(JOE) = 1 d(ETHEL) = 5 JOE (JOE, ETHEL)
key (TED, ETHEL) d(TED) = 1 d(ETHEL) = 5 TED (TED, ETHEL)
key (RICKY, FRED) d(RICKY) = 2 d(FRED) = 3 RICKY (RICKY, FRED)
key (ETHEL, RANDY) d(ETHEL) = 5 d(RANDY) = 2 RANDY (ETHEL, RANDY)
key (RANDY, RICKY) d(RANDY) = 2 d(RICKY) = 2 RANDY (RANDY, RICKY)
(a)
Reduce 1
Joe
JOE (JOE, ETHEL) ∅
Ricky
RICKY (RICKY, FRED) ∅
Ted
TED (TED, ETHEL) ∅
Fred
FRED (FRED, ETHEL) ∅
Lucy
LUCY (ETHEL, LUCY)
LUCY (LUCY, FRED)
(ETHEL, RICKY) (ETHEL, RANDY), (RANDY, RICKY)
Randy
RANDY (ETHEL, RANDY)
RANDY (RANDY, RICKY)
(ETHEL, FRED) (ETHEL, LUCY), (LUCY, FRED)
(b)

图 2. Triangle Count MapReduce1

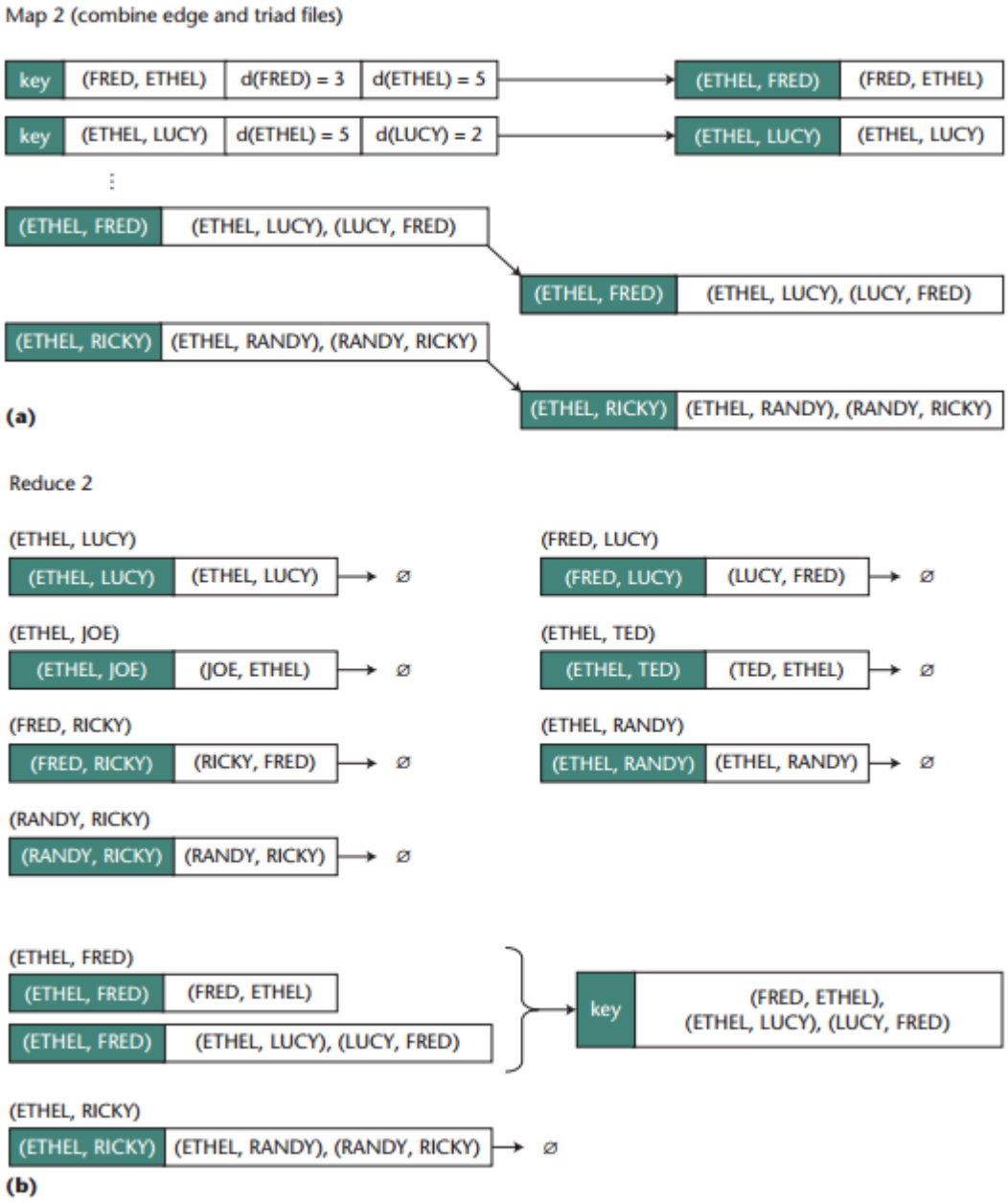


图 3. Triangle Count MapReduce2

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

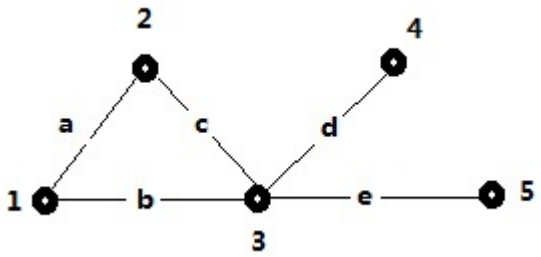
定义：

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

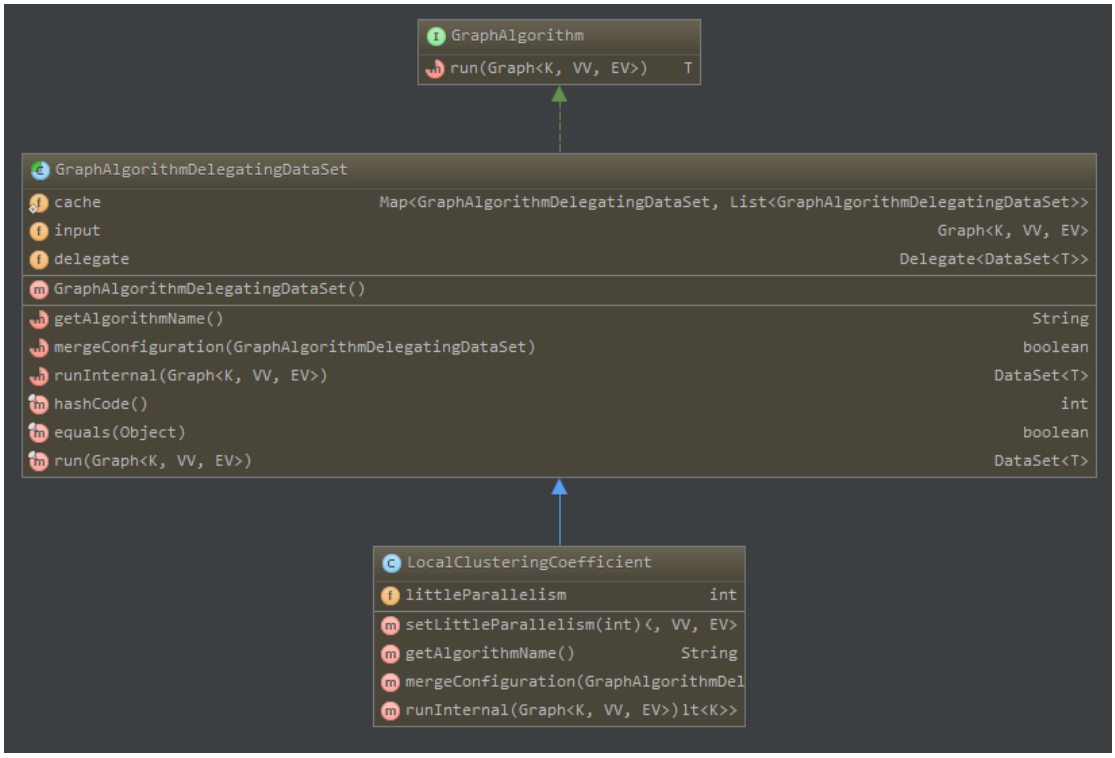
4 节点的邻居节点（3），他们之间构成的边有 0 条，可能构成的边 0 条，因此 0

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

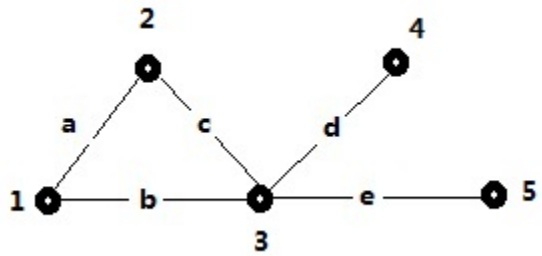
实现：

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

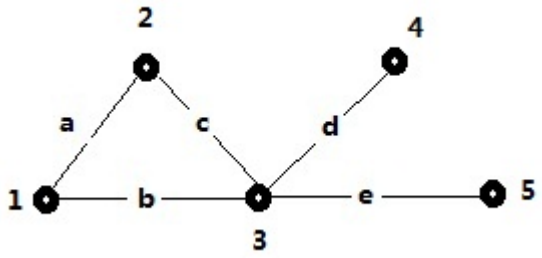
```
public DataSet<Result<K>> runInternal(Graph<K, VV, EV> input)
      throws Exception {
   // u, v, w
   DataSet<Tuple3<K,K,K>> triangles = input
      .run(new TriangleListing<K,VV,EV>()
         .setLittleParallelism(littleParallelism));

   // 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)}

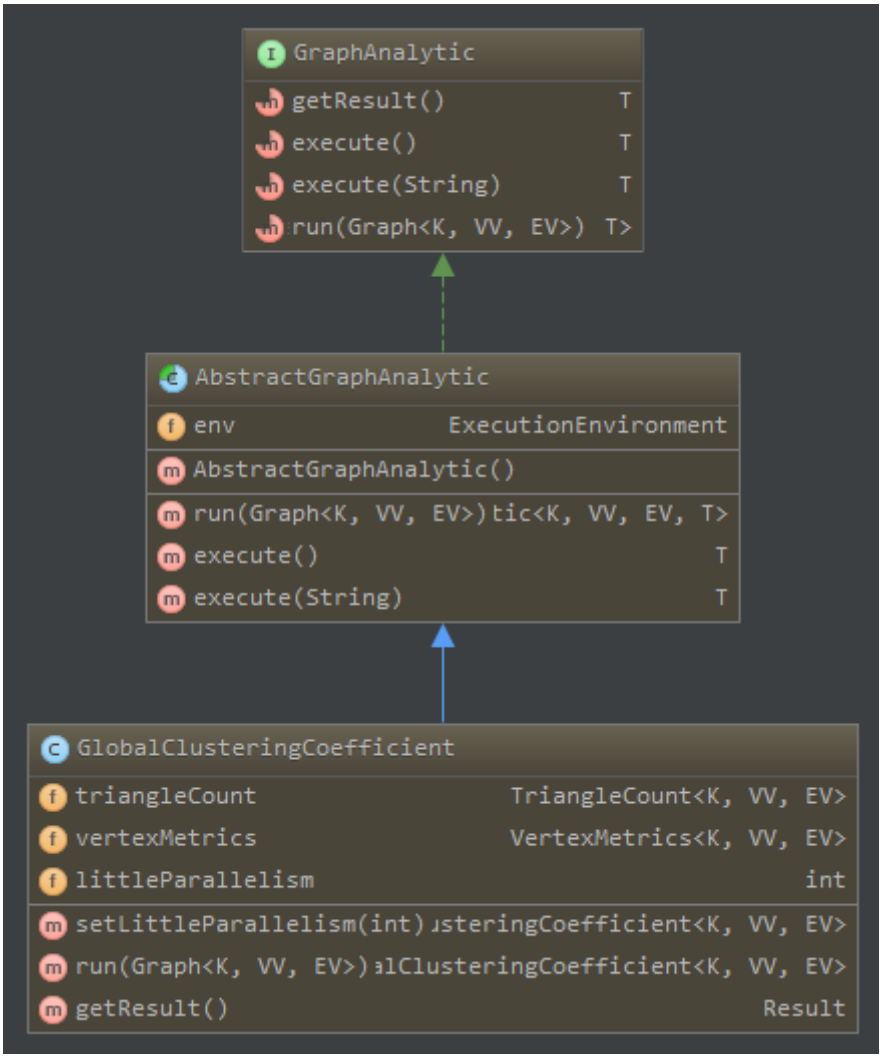
all triplet = {1，(2,3)}，{2，(1,3)}，{3，(1,2)}，{3，（2,4）}，{3，（4,5）}，{3，（1,5）}，{3，（2,5）}，{3，（1,4）}

number of closed triplet = 3

number of　triplet = 8

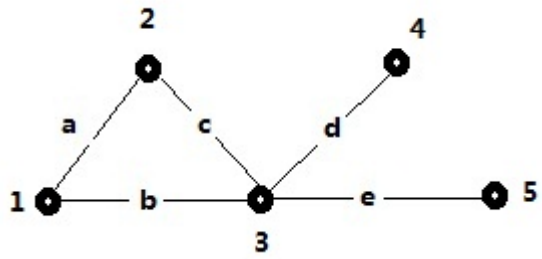
number of triplet / number of　triplet = 3/8

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;
}
```

2. Spark 图算法

表 2. Spark 图算法表

编号	名称	描述	应用	难度
AS1	Label Propagation			
AS2	Connected Components			
AS3	PageRank			
AS4	Shortest Paths			
AS5	Strongly Connected Components			
AS6	SVDPPlusPlus			
AS7	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?