GitHub: <https://github.com/DuanSky22/gelly-streaming>

相关资料：

https://github.com/vasia/gelly-streaming

http://www.slideshare.net/vkalavri/graphs-as-streams-rethinking-graph-processing-in-the-streaming-era

http://www.slideshare.net/vkalavri/gellystream-singlepass-graph-streaming-analytics-with-apache-flink

# 1. 简介

我们实现了一个轻量级的分布式的流图处理模型，该模型能够实时在线处理图数据，图聚合，近似估计，图窗口，并将这些运行一遍在无界的流图数据上。这里有两个重要的抽象：**GraphStream**和**GraphWindowStream**。

# 2. Basic API

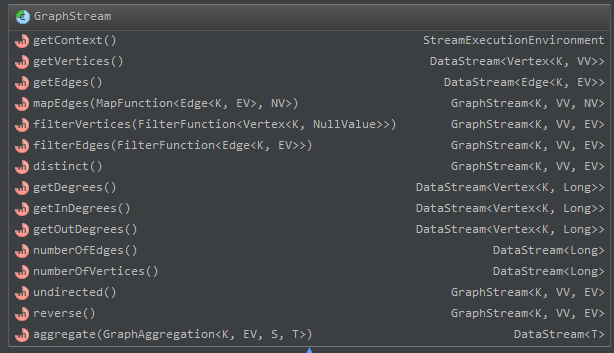
## 2.1 GraphStream

GraphStream是一个抽象类，用来定义一个运行在流上的图的基本运算，它总共定义了15种操作，这15种操作大致可以分为一下几类：

（1）获取Graph中诸如入度、出度、顶点数目，边数目等属性信息；

（2）运行在Graph顶点或边上的transformation等；

（3）运行在Graph上的**aggregate**.



## 2.2 SimpleEdgeStream

### 2.2.1 类图

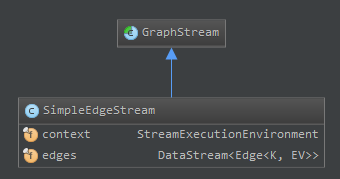
**SimpleEdgeStream**继承自**GraphStream**,用来定义一个以边为流的图。总共定义了23种操作。这23种操作大致可以分为以下几类：

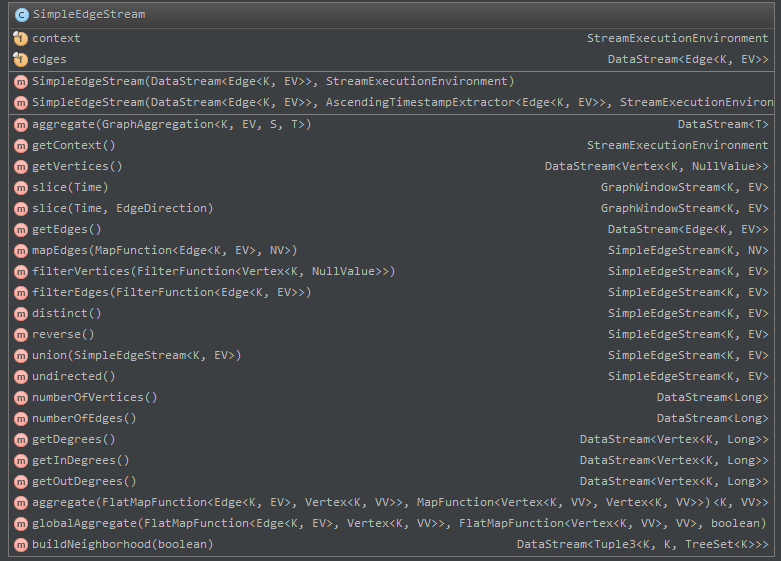
（1）从DataStream类型的边流数据源上创建SimpleEdgeStream;

（2）获取Graph中诸如入度、出度、顶点数目，边数目等属性信息；

（3）运行在Graph顶点或边上的诸如filter, union, distinct的transformation等；

（4）运行在Graph上的aggregate，slice等.





### 2.2.2 源码

（1）如何将edge stream转换成vertex stream?

即源码中的getVertices()函数。方法是先将edge flatMap成源点和目标点，然后再对这些点进行过滤，去掉重复的点。即可得到vertex stream.

public DataStream<Vertex<K, NullValue>> getVertices() {  
 return this.edges  
 .flatMap(new EmitSrcAndTarget<K, EV>())  
 .keyBy(0)  
 .filter(new FilterDistinctVertices<K>());  
}

（2）如何在edge流上获取vertex/degree/edge number?

|  |
| --- |
| edge number  统计时使用一个计数器，每次map时将计数器加1即可。这样就将原来的边流一一映射成一个计数器流。每条边对应一个计数器。 |
| */\*\*  \* @return a data stream representing the number of all edges in the streamed graph, including possible duplicates  \*/* public DataStream<Long> numberOfEdges() {  return this.edges.map(new TotalEdgeCountMapper<K, EV>()).setParallelism(1); }  private static final class TotalEdgeCountMapper<K, EV> implements MapFunction<Edge<K, EV>, Long> {  private long edgeCount;   public TotalEdgeCountMapper() {  edgeCount = 0;  }   @Override  public Long map(Edge<K, EV> edge) throws Exception {  edgeCount++;  return edgeCount;  } } |

|  |
| --- |
| vertex number  统计时调用globalAggregate()函数，该函数接受三个参数，这三个参数也是统计vertex number的三个步骤：首先将原来的edge stream映射成(vertexId,1L)的顶点流，因为这些顶点可能有重复，所以第二步是将这些顶点流去重（去重采用的方式是使用HashSet）并且映射成vertex number流，第三步是这个vertex number stream可能会出现 2,3,3,3,3这样的情况（因为到达的边可能不会增加顶点数目），所以需要去重这些重复出现的顶点数目情况。 |
| */\*\*  \* @return a continuously improving data stream representing the number of vertices in the streamed graph  \*/* public DataStream<Long> numberOfVertices() {  return this.globalAggregate(new DegreeTypeSeparator<K, EV>(true, true),  new VertexCountMapper<K>(), true); }  //step 1.  private static final class DegreeTypeSeparator <K, EV>  implements FlatMapFunction<Edge<K, EV>, Vertex<K, Long>> {  private final boolean collectIn;  private final boolean collectOut;   public DegreeTypeSeparator(boolean collectIn, boolean collectOut) {  this.collectIn = collectIn;  this.collectOut = collectOut;  }   @Override  public void flatMap(Edge<K, EV> edge, Collector<Vertex<K, Long>> out) throws Exception {  if (collectOut) {  out.collect(new Vertex<>(edge.getSource(), 1L));  }  if (collectIn) {  out.collect(new Vertex<>(edge.getTarget(), 1L));  }  } }  //step 2.  private static final class VertexCountMapper<K> implements FlatMapFunction<Vertex<K, Long>, Long> {  private Set<K> vertices;   public VertexCountMapper() {  this.vertices = new HashSet<>();  }   @Override  public void flatMap(Vertex<K, Long> vertex, Collector<Long> out) throws Exception {  vertices.add(vertex.getId());  out.collect((long) vertices.size());  } }  //step 3.  private static final class GlobalAggregateMapper<VV> implements FlatMapFunction<VV, VV> {  VV previousValue;   public GlobalAggregateMapper() {  previousValue = null;  }   @Override  public void flatMap(VV vv, Collector<VV> out) throws Exception {  if (!vv.equals(previousValue)) {  previousValue = vv;  out.collect(vv);  }  } } |

|  |
| --- |
| degree  degree的统计分为入度、出度、和度三个层次的统计；统计方式都较为相似，即分两步进行：第一步将原来的edge stream转换成vertex stream，第二步将vertex stream映射成degree stream。在统计degree时使用了HashMap，来记录当前时刻各个节点的度。 |
| */\*\*  \* Get the degree stream  \*  \* @return a stream of vertices, with the degree as the vertex value  \* @throws Exception  \*/* @Override public DataStream<Vertex<K, Long>> getDegrees() throws Exception {  return this.aggregate(new DegreeTypeSeparator<K, EV>(true, true),  new DegreeMapFunction<K>()); }  */\*\*  \* Get the in-degree stream  \*  \* @return a stream of vertices, with the in-degree as the vertex value  \* @throws Exception  \*/* public DataStream<Vertex<K, Long>> getInDegrees() throws Exception {  return this.aggregate(new DegreeTypeSeparator<K, EV>(true, false),  new DegreeMapFunction<K>()); }  private static final class DegreeMapFunction <K>  implements MapFunction<Vertex<K, Long>, Vertex<K, Long>> {  private final Map<K, Long> localDegrees;   public DegreeMapFunction() {  localDegrees = new HashMap<>();  }   @Override  public Vertex<K, Long> map(Vertex<K, Long> degree) throws Exception {  K key = degree.getId();  if (!localDegrees.containsKey(key)) {  localDegrees.put(key, 0L);  }  localDegrees.put(key, localDegrees.get(key) + degree.getValue());  return new Vertex<>(key, localDegrees.get(key));  } }  */\*\*  \* Get the out-degree stream  \*  \* @return a stream of vertices, with the out-degree as the vertex value  \* @throws Exception  \*/* public DataStream<Vertex<K, Long>> getOutDegrees() throws Exception {  return this.aggregate(new DegreeTypeSeparator<K, EV>(false, true),  new DegreeMapFunction<K>()); } |

（3）如何在edge流上创建neighborhood?

|  |
| --- |
| buildNeighborhood  目的是将edge转换成带有edge源点的neighborhood标注的流。采用TreeSet来保留源点所有的neighborhood. |
| */\*\*  \* Builds the neighborhood state by creating adjacency lists.  \* Neighborhoods are currently built using a TreeSet.  \*  \* @param directed if true, only the out-neighbors will be stored  \* otherwise both directions are considered  \* @return a stream of Tuple3, where the first 2 fields identify the edge processed  \* and the third field is the adjacency list that was updated by processing this edge.  \*/* public DataStream<Tuple3<K, K, TreeSet<K>>> buildNeighborhood(boolean directed) {   DataStream<Edge<K, EV>> edges = this.getEdges();  if (!directed) {  edges = this.undirected().getEdges();  }  return edges.keyBy(0).flatMap(new BuildNeighborhoods<K, EV>()); }  private static final class BuildNeighborhoods<K, EV> implements FlatMapFunction<Edge<K, EV>, Tuple3<K, K, TreeSet<K>>> {   Map<K, TreeSet<K>> neighborhoods = new HashMap<>();  Tuple3<K, K, TreeSet<K>> outTuple = new Tuple3<>();   public void flatMap(Edge<K, EV> e, Collector<Tuple3<K, K, TreeSet<K>>> out) {  TreeSet<K> t;  if (neighborhoods.containsKey(e.getSource())) {  t = neighborhoods.get(e.getSource());  } else {  t = new TreeSet<>();  }  t.add(e.getTarget());  neighborhoods.put(e.getSource(), t);   outTuple.setField(e.getSource(), 0);  outTuple.setField(e.getTarget(), 1);  outTuple.setField(t, 2);  out.collect(outTuple);  } } |

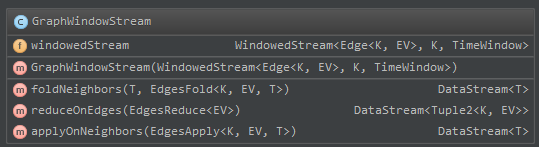
（4）GraphStream如何转换为GraphWindowStream?

|  |
| --- |
| Slice  Slice的目的是将原来连续的edge stream流切成一个一个的window窗口。怎样进行切割呢？这里是按照In/Out/All三种方式进行切割。 |
| */\*\*  \* Discretizes the edge stream into tumbling windows of the specified size.  \* <p>  \* The edge stream is partitioned so that all neighbors of a vertex belong to the same partition.  \* The KeyedStream is then windowed into tumbling time windows.  \*   \* @param size the size of the window  \* @param direction the EdgeDirection to key by  \* @return a GraphWindowStream of the specified size, keyed by  \*/* public GraphWindowStream<K, EV> slice(Time size, EdgeDirection direction)  throws IllegalArgumentException {   switch (direction) {  case *IN*:  return new GraphWindowStream<K, EV>(  this.reverse().getEdges().keyBy(new NeighborKeySelector<K, EV>(0)).timeWindow(size));  case *OUT*:  return new GraphWindowStream<K, EV>(  getEdges().keyBy(new NeighborKeySelector<K, EV>(0)).timeWindow(size));  case *ALL*:  getEdges().keyBy(0).timeWindow(size);  return new GraphWindowStream<K, EV>(  this.undirected().getEdges().keyBy(  new NeighborKeySelector<K, EV>(0)).timeWindow(size));  default:  throw new IllegalArgumentException("Illegal edge direction");  } } |

## 2.3 GraphWindowStream

### 2.3.1 类图

**GraphWindowStream**是建立在edge window stream上的离散的流图。每个window窗口代表一个Graph。该类的对象可由**SimpleEdgeStream**对象的slice(Time)方法创建。创建的方法为根据边的源顶点或目标顶点对原来的edge stream进行切割，使得源顶点或目标顶点的边的流都分到一个窗口内。GraphWindowStream主要定义了运行在其上（本质是运行在window edge stream上）的三个transformation：Fold, Reduce, And Apply.



# 3. Library

## 3.1 Window Triangles

名称：Window Triangles

描述：统计一个事件窗口内的Triangle Count.

输入：edge stream.

输出：triangle count stream.

算法：

|  |
| --- |
| Window Triangles |
| DataStream<Tuple2<Integer, Long>> triangleCount =   edges.slice(*windowTime*, EdgeDirection.*ALL*) //slice the edge to time window.  .applyOnNeighbors(new GenerateCandidateEdges())  .keyBy(0, 1).timeWindow(*windowTime*) .apply(new CountTriangles()) .timeWindowAll(*windowTime*).sum(0); |

分析：

下面以如下图例来分析整个运算过程。图中edge流到达的顺序按照a b c d … i 进行。详细完整的图见Model Design.vsdx – Window Triangles.



算法的流程图如下：



结合上面的案例，算法每一步的运算结果和详细的运算过程如下：



## 3.2 Exact Triangles

名称：Exact Triangles

描述：统计一个edge stream中准确的triangle count.

输入：edge stream.

输出：triangle count stream.

算法：

|  |
| --- |
| Exact Triangles |
| SimpleEdgeStream<Integer, NullValue> edges = *getGraphStream*(env);  DataStream<Tuple2<Integer, Integer>> result =  edges.buildNeighborhood(false)  .map(new ProjectCanonicalEdges())  .keyBy(0, 1).flatMap(new IntersectNeighborhoods())  .keyBy(0).flatMap(new SumAndEmitCounters()); |

分析：

下面以如下图例来分析整个运算过程。图中edge流到达的顺序按照a b c d … i 进行。详细完整的图见Model Design.vsdx – Exact Triangles。原理是对于新增的一条边（a,b），先找出a, b两个顶点的所有邻接点，假设集合为Ea, Eb,那么只需要找出这两个集合的交集即可找出因为这条新增的边所增加的三角形的数目。



## 3.3 Broadcast Triangles

3.4

# 4. 分析

## 4.1 Batch Graph VS Streaming Graph

Batch Graph共提供了78种操作。这些操作大致可以分为以下5类：

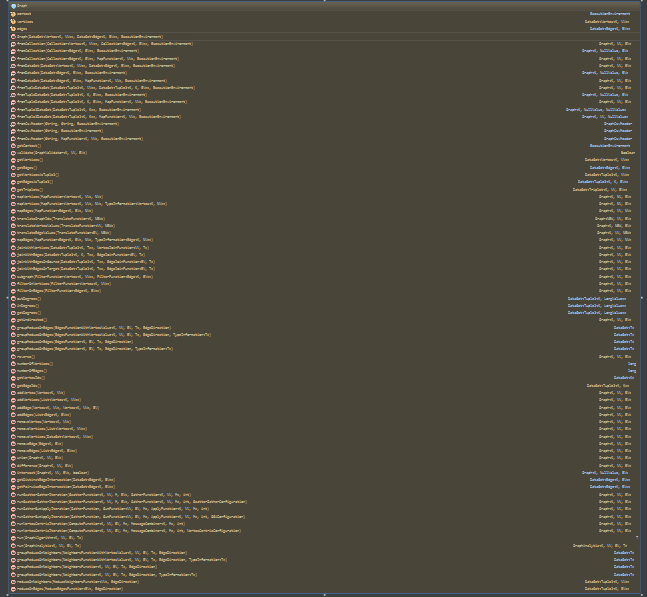
（1）从集合、文本文件、DataSet等不同数据源生成Graph;

（2）获取Graph中诸如入度、出度、顶点数目，边数目等属性信息；

（3）运行在Graph顶点或边上的transformation等；

（4）修改Graph的数据，诸如添加边，删除边等（注意并不能在原图上修改返回一个新图）；

（5）运行在Graph上的算法，aggregate和迭代模型；



# 5. 总结