Flink-Gelly: 通用 Graph API 指南

Gelly简介

Gelly是Flink的图API库，它包含了一组旨在简化Flink中图形分析应用程序开发的方法和实用程序。在Gelly中，可以使用类似于批处理API提供的高级函数来转换和修改图。Gelly提供了创建、转换和修改图的方法，以及图算法库。

使用Gelly

在项目中为了能方便地使用Gelly，可以在`pom.xml`中引入以下依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-gelly_2.11</artifactId>
    <version>1.7.0</version>
</dependency>
```

在运行Gelly程序之前，Gelly库jar在opt目录下的Flink发行版中提供(对于超过Flink 1.2的版本，可以从Maven Central手动下载)。要运行Gelly示例，必须将Flink - Gelly(用于Java)或Flink - gely - Scala(用于Scala) jar复制到Flink的lib目录

```
cp opt/flink-gelly_*.jar lib/
cp opt/flink-gelly-scala_*.jar lib/
```

图API

Graph Representation

在Gelly中，一个图（`Graph`）由顶点的数据集（`DataSet`）和边的数据集（`DataSet`）组成。图中的顶点由`Vertex`类型来表示，一个`Vertex`由唯一的ID和一个值来表示。其中`Vertex`的ID必须是全局唯一的值，且实现了`Comparable`接口。如果节点不需要由任何值，则该值类型可以声明成`NullValue`类型。

```
// create a new vertex with a Long ID and a String value
Vertex<Long, String> v = new Vertex<Long, String>(1L, "foo");

// create a new vertex with a Long ID and no value
Vertex<Long, NullValue> v = new Vertex<Long, NullValue>(1L, NullValue.getInstance()
);
```

图中的边由`Edge`类型来表示，一个`Edge`通常由源顶点的ID，目标顶点的ID以及一个可选的值来表示。其中源顶点和目标顶点的类型必须与`Vertex`的ID类型相同。同样的，如果边不需要由任何值，则该值类型可以声明

成NullValue类型。

```
Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);

// reverse the source and target of this edge
Edge<Long, Double> reversed = e.reverse();

Double weight = e.getValue(); // weight = 0.5
```

在Gelly中，一个Edge总是从源顶点指向目标顶点。如果图中每条边都能匹配一个从目标顶点到源顶点的Edge，那么这个图可能是个无向图。同样地，无向图可以用这个方式来表示。

Graph Creation

我们可以通过以下几种方式创建一个Graph:

- 从一个**Edge**数据集合和一个**Vertex**数据集合中创建图。

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Vertex<String, Long>> vertices = ...
DataSet<Edge<String, Double>> edges = ...

Graph<String, Long, Double> graph = Graph.fromDataSet(vertices, edges, env);
```

- 从一个表示边的**Tuple2**数据集合中创建图。Gelly会将每个Tuple2转换成一个'Edge'，其中第一个元素表示源顶点的ID，第二个元素表示目标顶点的ID，图中的顶点和边的value值均被设置为NullValue。

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Tuple2<String, String>> edges = ...

Graph<String, NullValue, NullValue> graph = Graph.fromTuple2DataSet(edges, env);
```

- 从一个**Tuple3**数据集和一个可选的**Tuple2**数据集中生成图。在这种情况下，Gelly会将每个Tuple3转换成Edge，其中第一个元素域是源顶点ID，第二个域是目标顶点ID，第三个域是边的值。同样的，每个Tuple2会转换成一个顶点Vertex，其中第一个域是顶点的ID，第二个域是顶点的value。

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Tuple2<String, Long>> vertexTuples = env.readCsvFile("path/to/vertex/input"
).types(String.class, Long.class);

DataSet<Tuple3<String, String, Double>> edgeTuples = env.readCsvFile("path/to/edge/
input").types(String.class, String.class, Double.class);
```

```
Graph<String, Long, Double> graph = Graph.fromTupleDataSet(vertexTuples, edgeTuples
, env);
```

- 从一个表示边数据的**CSV**文件和一个可选的表示节点的**CSV**文件中生成图。在这种情况下，Gelly会将表示边的CSV文件中的每一行转换成一个`Edge`，其中第一个域表示源顶点ID，第二个域表示目标顶点ID，第三个域表示边的值。同样的，表示节点的CSV中的每一行都被转换成一个`Vertex`，其中第一个域表示顶点的ID，第二个域表示顶点的值。为了通过`GraphCsvReader`生成图，需要指定每个域的类型，可以使用下列之一的方法：
- types(Class<K> vertexKey, Class<VV> vertexValue,Class<EV> edgeValue): both vertex and edge values are present.
- edgeTypes(Class<K> vertexKey, Class<EV> edgeValue): the Graph has edge values, but no vertex values.
- vertexTypes(Class<K> vertexKey, Class<VV> vertexValue): the Graph has vertex values, but no edge values.
- keyType(Class<K> vertexKey): the Graph has no vertex values and no edge values.

```
// create a Graph with String Vertex IDs, Long Vertex values and Double Edge values
Graph<String, Long, Double> graph = Graph.fromCsvReader("path/to/vertex/input", "pa
th/to/edge/input", env)
                    .types(String.class, Long.class, Double.class);


// create a Graph with neither Vertex nor Edge values
Graph<Long, NullValue, NullValue> simpleGraph = Graph.fromCsvReader("path/to/edge/i
nput", env).keyType(Long.class);
```

- 从一个边的集合和一个可选的顶点的集合中生成图。如果在图创建的时候顶点的集合没有传入，Gelly会依据数据的边数据集合自动地生成一个`Vertex`集合。这种情况下，创建的节点是没有值的。或者，我们也可以像下面一样，在创建图的时候提供一个`MapFunction`方法来初始化节点的值。

```
List<Vertex<Long, Long>> vertexList = new ArrayList...

List<Edge<Long, String>> edgeList = new ArrayList...

Graph<Long, Long, String> graph = Graph.fromCollection(vertexList, edgeList, env);

// initialize the vertex value to be equal to the vertex ID
Graph<Long, Long, String> graph = Graph.fromCollection(edgeList,
                new MapFunction<Long, Long>() {
                    public Long map(Long value) {
                        return value;
                    }
                }, env);
```

Graph Properties

Gelly提供了下列方法来查询图的属性和指标：

```
// get the Vertex DataSet
DataSet<Vertex<K, VV>> getVertices()

// get the Edge DataSet
DataSet<Edge<K, EV>> getEdges()

// get the IDs of the vertices as a DataSet
DataSet<K> getVertexIds()

// get the source-target pairs of the edge IDs as a DataSet
DataSet<Tuple2<K, K>> getEdgeIds()

// get a DataSet of <vertex ID, in-degree> pairs for all vertices
DataSet<Tuple2<K, LongValue>> inDegrees()

// get a DataSet of <vertex ID, out-degree> pairs for all vertices
DataSet<Tuple2<K, LongValue>> outDegrees()

// get a DataSet of <vertex ID, degree> pairs for all vertices, where degree is the
 sum of in- and out- degrees
DataSet<Tuple2<K, LongValue>> getDegrees()

// get the number of vertices
long numberOfVertices()

// get the number of edges
long numberOfEdges()

// get a DataSet of Triplets<srcVertex, trgVertex, edge>
DataSet<Triplet<K, VV, EV>> getTriplets()
```

Graph Transformations

- **Map**：Gelly提供了专门的用于转换顶点值和边值的方法。`mapVertices`和`mapEdges`会返回一个新图，图中的每个顶点和边的ID不会改变，但是顶点和边的值会根据用户自定义的映射方法进行修改。这些映射方法同时也可以修改顶点和边的值的类型。示例如下：

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Graph<Long, Long, Long> graph = Graph.fromDataSet(vertices, edges, env);

// increment each vertex value by one
Graph<Long, Long, Long> updatedGraph = graph.mapVertices(
                new MapFunction<Vertex<Long, Long>, Long>() {
                    public Long map(Vertex<Long, Long> value) {
                        return value.getValue() + 1;
```

```
                }
            });
```

- **Translate**：Gelly还提供了专门用于根据用户定义的函数转换顶点和边的ID和值的值及类型的方法（`translateGraphIDs`/`translateVertexValues`/`translateEdgeValues`），是Map功能的升级版，因为Map操作不支持修订顶点和边的ID。示例如下：

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Graph<Long, Long, Long> graph = Graph.fromDataSet(vertices, edges, env);

// translate each vertex and edge ID to a String
Graph<String, Long, Long> updatedGraph = graph.translateGraphIds(
            new MapFunction<Long, String>() {
                public String map(Long id) {
                    return id.toString();
                }
            });

// translate vertex IDs, edge IDs, vertex values, and edge values to LongValue
Graph<LongValue, LongValue, LongValue> updatedGraph = graph
            .translateGraphIds(new LongToLongValue())
            .translateVertexValues(new LongToLongValue())
            .translateEdgeValues(new LongToLongValue())
```

- **Filter**：Gelly支持在图中的顶点上或边上执行一个用户指定的filter转换。`filterOnEdges`会根据提供的在边上的断言在原图的基础上生成一个新的子图，注意，顶点的数据不会被修改。同样的`filterOnVertices`在原图的顶点上进行filter转换，不满足断言条件的源节点或目标节点会在新的子图中移除。该子图方法支持同时对顶点和边应用filter函数。示例如下：

```
Graph<Long, Long, Long> graph = ...

graph.subgraph(
        new FilterFunction<Vertex<Long, Long>>() {
                public boolean filter(Vertex<Long, Long> vertex) {
                    // keep only vertices with positive values
                    return (vertex.getValue() > 0);
                }
           },
        new FilterFunction<Edge<Long, Long>>() {
                public boolean filter(Edge<Long, Long> edge) {
                    // keep only edges with negative values
                    return (edge.getValue() < 0);
                }
        })
```

Filter

- **Join**：Gelly提供了专门的方法用于将节点和边的数据集合与其他的输入数据集进行连接。`joinWithVertices`用于连接节点和一个输入的`Tuple2`数据集，连接操作通过使用节点的ID和输入的`Tuple2`数据集的第一个域作为连接的Key值。该方法会根据用户定义的转换方法返回一个新图。类似的，一个数据集合也可以通过边进行连接，通过边进行连接有三种方式：`joinWithEdges`的输入是一个`Tuple3`数据集，并将边的源顶点ID和目标顶点ID作为一个联合的Key用于连接。`joinWithEdgesOnSource`和`joinWithEdgesOnTarget`均用于连接一个`Tuple2`数据集，其中`joinWithEdgesOnSource`针对`Tuple2`的第一个域进行连接，而`joinWithEdgesOnTarget`针对`Tuple2`的第二个域进行连接。值得注意的是，如果数据集中同一个Key出现多次，Gelly中所有的Join方法仅针对第一个相同Key值得数据进行连接操作。示例如下：

```
Graph<Long, Double, Double> network = ...

DataSet<Tuple2<Long, LongValue>> vertexOutDegrees = network.outDegrees();

// assign the transition probabilities as the edge weights
Graph<Long, Double, Double> networkWithWeights = network.joinWithEdgesOnSource(vert
exOutDegrees,
                new VertexJoinFunction<Double, LongValue>() {
                    public Double vertexJoin(Double vertexValue, LongValue inputVal
ue) {
                        return vertexValue / inputValue.getValue();
                    }
                });
```

- **Reverse**：Gelly中得`reverse()`方法用于在原图的基础上，生成一个所有边方向与原图相反的新图。
- **Undirected**：在前面的内容中，我们提到过，Gelly中的图通常都是有向的，而无向图可以通过对所有边添

加反向的边来实现，出于这个目的，Gelly提供了`getUndirected()`方法，用于获取原图的无向图。

- **Union**：Gelly的`union()`操作用于联合当前图和指定的输入图，并生成一个新图，在输出的新图中，相同的节点只保留一份，但是重复的边会保留。如下图所示：

Union

- **Difference**：Gelly提供了`difference()`方法用于发现当前图与指定的输入图之间的差异。
- **Intersect**：Gelly提供了`intersect()`方法用于发现两个图中共同存在的边，并将相同的边以新图的方式返回。相同的边指的是具有相同的源顶点，相同的目标顶点和相同的边值。返回的新图中，所有的节点没有任何值，如果需要节点值，可以使用`joinWithVertices()`方法去任何一个输入图中检索。示例如下：

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create first graph from edges {(1, 3, 12) (1, 3, 13), (1, 3, 13)}
List<Edge<Long, Long>> edges1 = ...
Graph<Long, NullValue, Long> graph1 = Graph.fromCollection(edges1, env);

// create second graph from edges {(1, 3, 13)}
```

```
List<Edge<Long, Long>> edges2 = ...
Graph<Long, NullValue, Long> graph2 = Graph.fromCollection(edges2, env);

// Using distinct = true results in {(1,3,13)}
Graph<Long, NullValue, Long> intersect1 = graph1.intersect(graph2, true);

// Using distinct = false results in {(1,3,13),(1,3,13)} as there is one edge pair
Graph<Long, NullValue, Long> intersect2 = graph1.intersect(graph2, false);
```

Graph Mutations

Gelly内置下列方法以支持对一个图进行节点和边的增加/移除操作：

```
// adds a Vertex to the Graph.
//If the Vertex already exists, it will not be added again.
Graph<K, VV, EV> addVertex(final Vertex<K, VV> vertex)

// adds a list of vertices to the Graph.
//If the vertices already exist in the graph, they will not be added once more.
Graph<K, VV, EV> addVertices(List<Vertex<K, VV>> verticesToAdd)

// adds an Edge to the Graph.
//If the source and target vertices do not exist in the graph, they will also be ad
ded.
Graph<K, VV, EV> addEdge(Vertex<K, VV> source, Vertex<K, VV> target, EV edgeValue)

// adds a list of edges to the Graph.
// When adding an edge for a non-existing set of vertices,
//the edge is considered invalid and ignored.
Graph<K, VV, EV> addEdges(List<Edge<K, EV>> newEdges)

// removes the given Vertex and its edges from the Graph.
Graph<K, VV, EV> removeVertex(Vertex<K, VV> vertex)

// removes the given list of vertices and their edges from the Graph
Graph<K, VV, EV> removeVertices(List<Vertex<K, VV>> verticesToBeRemoved)

// removes *all* edges that match the given Edge from the Graph.
Graph<K, VV, EV> removeEdge(Edge<K, EV> edge)

// removes *all* edges that match the edges in the given list
Graph<K, VV, EV> removeEdges(List<Edge<K, EV>> edgesToBeRemoved)
```

Neighborhood Methods

邻接方法允许每个顶点针对其所有的邻接顶点或边执行某个集合操作。`reduceOnEdges()`可以用于计算顶点所有邻接边的值的集合。`reduceOnNeighbors()`可以用于计算邻接顶点的值的集合。这些方法采用联合和交换集合，并在内部利用组合器，显著提高了性能。邻接的范围由`EdgeDirection`来确定，它有三个枚举值，分别是：`IN / OUT / ALL`，其中IN只考虑所有入的邻接边和顶点，OUT只考虑所有出的邻接边和顶点，而ALL考虑所有的邻接

边和顶点。举个例子，如下图所示，假设我们想要知道图中出度最小的边权重。

Neighborhood Methods

下列代码会为每个节点找到出的边集合，然后在集合的基础上执行一个用户定义的方法`SelectMinWeight()`。

```
Graph<Long, Long, Double> graph = ...

DataSet<Tuple2<Long, Double>> minWeights = graph.reduceOnEdges(new SelectMinWeight(
),
EdgeDirection.OUT);

// user-defined function to select the minimum weight
static final class SelectMinWeight implements ReduceEdgesFunction<Double> {

        @Override
        public Double reduceEdges(Double firstEdgeValue, Double secondEdgeValue) {
            return Math.min(firstEdgeValue, secondEdgeValue);
        }
}
```

结果入下图所示：

result

同样的，假设我们需要知道每个顶点的所有邻接边上的权重的值之和，不考虑方向。可以用下面的代码来实现：

```
Graph<Long, Long, Double> graph = ...

DataSet<Tuple2<Long, Long>> verticesWithSum = graph.reduceOnNeighbors(new SumValues
(),
EdgeDirection.IN);

// user-defined function to sum the neighbor values
static final class SumValues implements ReduceNeighborsFunction<Long> {

            @Override
            public Long reduceNeighbors(Long firstNeighbor, Long secondNeighbor) {
                return firstNeighbor + secondNeighbor;
        }
}
```

结果如下图所示

result

Graph Validation

Gelly提供了一个简单的工具用于对输入的图进行校验操作。由于应用程序上下文的不同，根据某些标准，有些图可能有效，也可能无效。例如用户需要校验图中是否包含重复的边。为了校验一个图，可以定义一个定制的GraphValidator并实现它的validate()方法。`InvalidVertexIdsValidator`是Gelly预定义的一个校验器，用来校验边上所有的顶点ID是否有效，即边上的顶点ID在顶点集合中存在。示例如下：

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create a list of vertices with IDs = {1, 2, 3, 4, 5}
List<Vertex<Long, Long>> vertices = ...

// create a list of edges with IDs = {(1, 2) (1, 3), (2, 4), (5, 6)}
List<Edge<Long, Long>> edges = ...

Graph<Long, Long, Long> graph = Graph.fromCollection(vertices, edges, env);

// will return false: 6 is an invalid ID
graph.validate(new InvalidVertexIdsValidator<Long, Long, Long>());
```