# Gelly:Graph API



# 图的表示

在Gelly中， 图(Graph)由顶点(vertex)的DataSet 和边(edge)的DataSet表示。

图的顶点由Vertex类表示。Vertex由一个唯一ID 和一个value 定义。VertexID 应该实现Comparable接口。要表示没有value的顶点，可以将value的类型设为NullType。

**Java** | **Scala**

```
// 用Long 类型的ID 和String 类型的 value 新建一个顶点
Vertex<Long, String> v = new Vertex<Long, String>(1L, "foo");

// 用一个Long 类型的ID 和空value 新建一个顶点
Vertex<Long, NullValue> v = new Vertex<Long, NullValue>(1L,
NullValue.getInstance());
```

图的边用Edge类表示。Edge由一个源ID (即源Vertex的ID)，一个目的ID (即目的Vertex的ID)，一个可选的value定义。源ID 和目的ID 应该与Vertex的ID 属于相同的类。没有值的边，它的value 类型为NullValue。

**Java** | **Scala**

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

// 反转一条边的两个点
Edge<Long, Double> reversed = e.reverse();

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

在Gelly中，Edge永远从源端点指向目的端点。对一个Graph而言，如果每条Edge 都对应着另一条从目的端点指向源端点的Edge，那么它可能是无向的。

Back to top

# 创建图

你可以通过如下方法创建一个Graph：

- 根据一个由边组成的DataSet，可选参数是一个由顶点组成的DataSet：

Java Scala

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

DataSet<Vertex<String, Long>> vertices = ...

DataSet<Edge<String, Double>> edges = ...

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

- 根据一个由表示边的Tuple2类组成的DataSet。Gelly 将把每个Tuple2转换成Edge，其中第一个field 将作为源ID，第二个field 将作为目的ID。顶点和边的值都会被置为NullValue。

Java Scala

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

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

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

- 根据一个由Tuple3组成的DataSet，可选参数是一个由Tuple2组成的DataSet。这种情况下，Gelly 将把每个Tuple3转换成Edge，其中第一个field 将成为源ID，第二个field 将成为目的ID，第三个field 将成为边的value。同样地，每个Tuple2将被转换为一个Vertex，其中第一个field 将成为端点的ID，第二个field 将成为端点的value。

Java Scala

```
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);
```

- 根据一个包含边数据的CSV文件，可选参数是一个包含端点数据的CSV文件。这种情况下，Gelly 将把边CSV文件的每一行转换成一个Edge，其中第一个field 将成为源ID， 第二个field 将成为目的ID， 第三个field (如果存在的话)将成为边的value。同样地，可选端点CSV文件的每一行将被转换成一个Vertex，其中第一个field 将成为端点的ID，第二个field（如果存在的话）将成为端点的value。想从GraphCsvReader得到Graph，必须用下面的某种方法指定类型：

- `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.

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

// 生成一个Vertex ID为String 类型、Vertex value为Long 类型，Edge value为Double 类型的图
Graph<String, Long, Double> graph = Graph.fromCsvReader("path/to/vertex/input",
"path/to/edge/input", env)
                                        .types(String.class, Long.class, Double.class);


// 生成一个Vertex 和Edge 都没有value 的图
Graph<Long, NullValue, NullValue> simpleGraph = Graph.fromCsvReader("path/to/edge/input",
env).keyType(Long.class);
```

- 根据一个由边组成的Collection，可选参数是一个由端点组成的Collection:

Java Scala

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

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

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

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

如果创建图时没有提供端点数据，Gelly 会根据边的输入自动生成一个Vertex的DataSet。这种情况下，生成的端点是没有值的。另外，将MapFunction 作为构建函数的一个参数传进去，也可以初始化Vertex的:

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

// 初始化时，将端点的值设为端点的ID
Graph<Long, Long, String> graph = Graph.fromCollection(edgeList,
                                new MapFunction<Long, Long>() {
                                        public Long map(Long value) {
                                                return value;
                                        }
                                }, env);
```

Back to top

# 图的属性

Gelly 提供了一些方法获取图的各种属性:

Java Scala

```
// 获取由端点构成的DataSet
DataSet<Vertex<K, VV>> getVertices()

// 获取边的DataSet
DataSet<Edge<K, EV>> getEdges()

// 获取由端点的ID构成的DataSet
DataSet<K> getVertexIds()

// 获取由边ID构成的source-target pair组成的DataSet
DataSet<Tuple2<K, K>> getEdgeIds()

// 获取端点的<端点ID， 入度> pair 组成的DataSet
DataSet<Tuple2<K, LongValue>> inDegrees()
```

```
// 获取端点的<端点ID，出度> pair 组成的DataSet
DataSet<Tuple2<K, LongValue>> outDegrees()

// 获取端点的<端点ID，度> pair 组成的DataSet，这里的度 = 入度 + 出度
DataSet<Tuple2<K, LongValue>> getDegrees()

// 获取端点的数量
long numberOfVertices()

// 获取边的数量
long numberOfEdges()

// 获取由三元组<srcVertex, trgVertex, edge> 构成的DataSet
DataSet<Triplet<K, VV, EV>> getTriplets()
```

Back to top

# 图的变换

- **Map**: Gelly 专门提供了一些方法，用来对端点的值和边的值进行map 变换。`mapVertices`和`mapEdges`返回一个新的`Graph`，它的端点(或者边)的ID保持不变，但是值变成了用户自定义的map 函数所提供的对应值。map 函数也允许改变端点或者边的值的类型。

Java Scala

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

// 把每个端点的值加1
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 提供专门的方法用来translate 端点和边的ID的类型和值(`translateGraphIDs`)，端点的值(`translateVertexValues`)，或者边的值(`translateEdgeValues`)。Translation 的过程是由用户定义的map 函数完成的，`org.apache.flink.graph.asm.translate` 这个包也提供了一些map 函数。同一个`MapFunction`，在上述三种方法里是通用的。

Java Scala

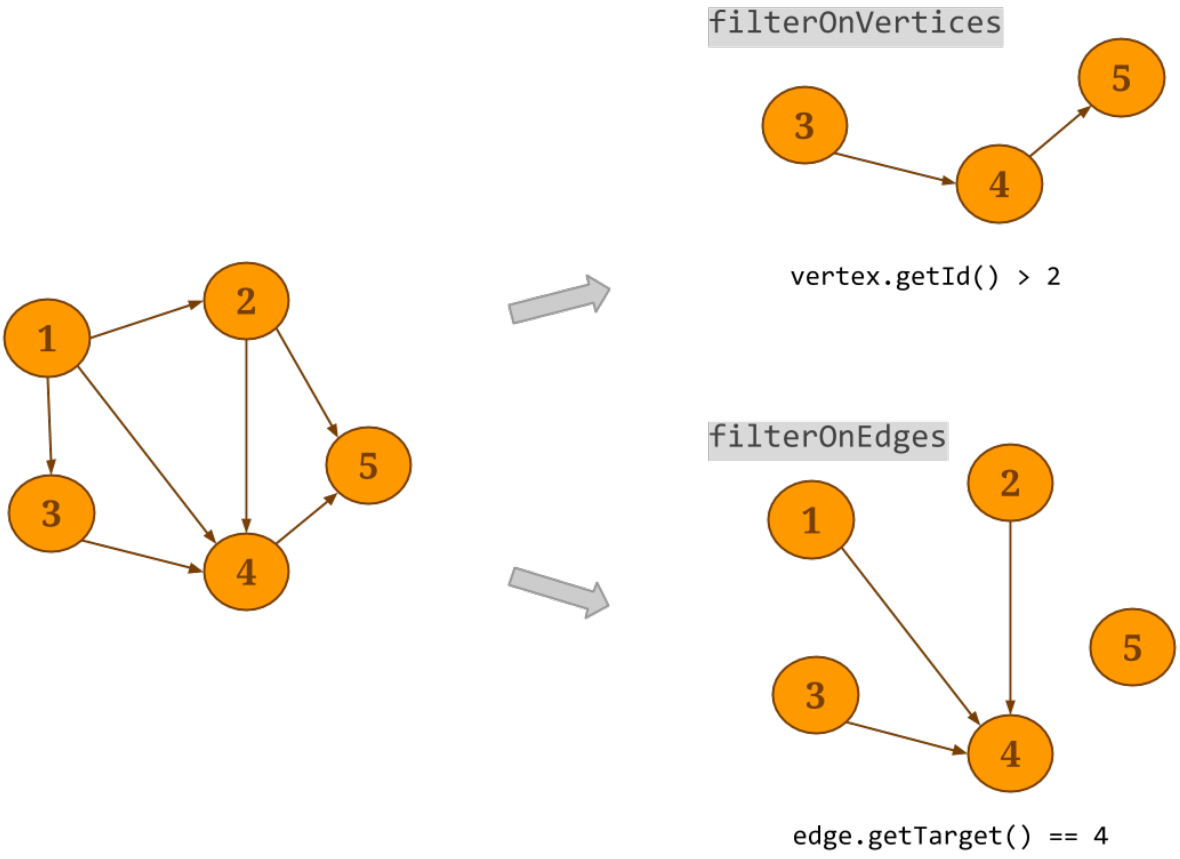
```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Graph<Long, Long, Long> graph = Graph.fromDataSet(vertices, edges, env);

// 将每个端点和边的ID translate 成String 类型
Graph<String, Long, Long> updatedGraph = graph.translateGraphIds(
                                new MapFunction<Long, String>() {
                                        public String map(Long id) {
                                                return id.toString();
                                        }
                                });

// 将端点ID，边ID，端点值，边的值 translage 成LongValue 类型
Graph<LongValue, LongValue, LongValue> updatedGraph = graph
                .translateGraphIds(new LongToLongValue())
```

```
                .translateVertexValues(new LongToLongValue())
                .translateEdgeValues(new LongToLongValue())
```

- **Filter**: Filter 变换将用户自定义的filter 函数作用于`Graph`中的顶点/边。`filterOnEdges` 生成原始图的一个sub-graph，只留下那些满足预设条件的边。注意，端点的dataset 将不会变动。对应地，`filterOnVertices` 在图的端点上应用filter。那些源/目的端点不满足vertex条件的边，将从最终的边组成的 dataset中删除。可以使用`subgraph` 方法，同时在端点和边上应用filter 函数。

Java Scala

```
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);
                                }
                })
```



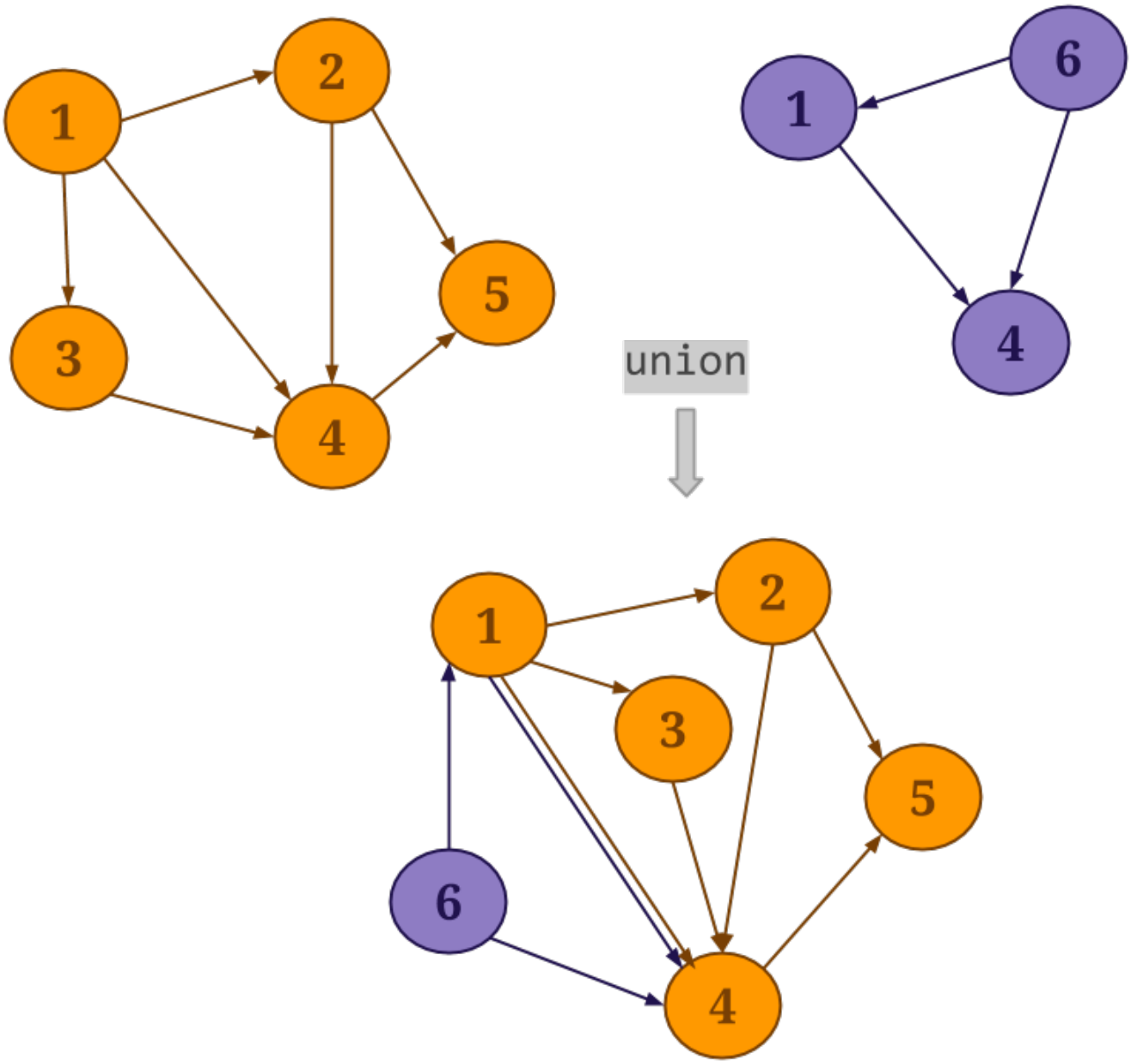

- **Join**: Gelly 提供一些专门的方法，对vertex 和edge 的dataset 与其它输入的dataset 做join 操作。`joinWithVertices` 将端点与输入的一个`Tuple2`组成的dataset 做join。Join 操作使用的key 是端点的ID和`Tuple2` 的第一个field。这个方法返回一个新的`Graph`，其中端点的值已经根据用户定义的转换函数更

新过了。
类似地，使用下面三种方法，输入的dataset 也可以和边做join。`joinWithEdges` 的期望输入是`Tuple3` 组成的 `DataSet`，join 操作发生在源端点和目的端点的ID 形成的组合key 上。`joinWithEdgesOnSource` 的期望输入是`Tuple2` 组成的`DataSet`，join 操作发生在边的源端点和输入的第一个field上。`joinWithEdgesOnTarget` 的期望输入是`Tuple2` 组成的`DataSet`，join 操作发生在边的目的端点和输入的第一个field上。以上的三种方法，都是在边和输入的dataset上应用变换函数。
注意，输入的dataset 如果包含重复的key，Gelly 中所有的join 方法都只会处理它遇到的第一个 value。

Java Scala

```
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(vertexOutDegrees,
                                    new VertexJoinFunction<Double, LongValue>()
{
                                            public Double vertexJoin(Double
vertexValue, LongValue inputValue) {
                                                    return vertexValue /
inputValue.getValue();
                                            }
                                    });
```

- **Reverse**: `reverse()` 反转所有边，然后返回一个新的`Graph`。
- **Undirected**: Gelly中，所有的`Graph` 永远是有向的。给图中所有边都加上方向相反的边，这样就可以表示无向图。因此，Gelly提供了`getUndirected()`方法。
- **Union**: Gelly 的`union()` 方法在指定图和当前图的端点和边的集合上取并集。在得到的`Graph` 中，重复的端点会被删除；如果存在重复边，重复的端点会被保留。




- **Difference**：Gelly 的`difference()`方法在指定图和当前图的端点和边的集合上取差异。
- **Intersect**: Gelly 的`intersect()`方法在指定图和当前图的端点和边的集合上取交集。结果是生成一个新的`Graph`, 包含两个图中都存在的所有边。如果两条边的源 identifier, 目的 identifier，value 都相同，那么就认为它们是相等的。生成的图中，所有的端点都没有value。 如果需要端点的value， 可以通过`joinWithVertices()`方法从输入图中获取。

根据`distinct` 参数存在与否，相等边在生成的`Graph` 中出现的次数要么是一次，要么是输入的图中存在的相等边的pair 的数量。

**Java** | **Scala**

```
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);
```

-

Back to top

# 图的变化

Gelly 提供如下方法，增加、删除输入Graph的端点或者边:

Java Scala

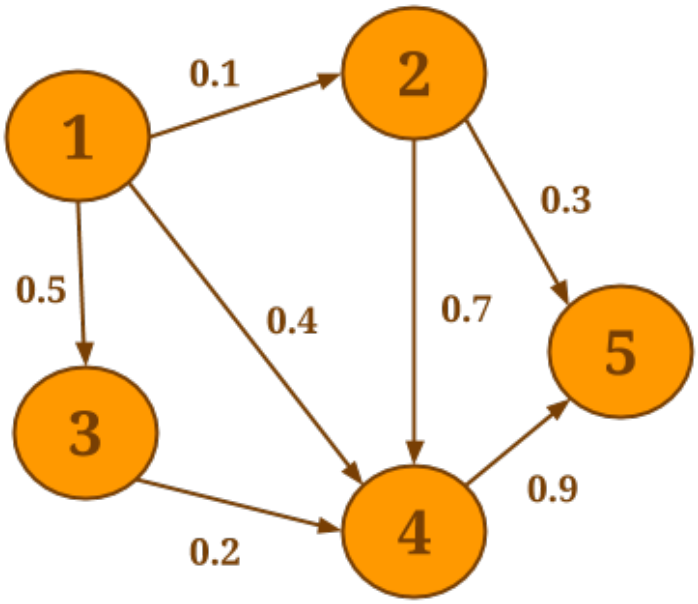
```
// 添加一个端点。如果端点已经存在，不会重复添加。
Graph<K, VV, EV> addVertex(final Vertex<K, VV> vertex)

// 添加一个端点的list。 如果图中已经存在端点，它们最多会被添加一次。
Graph<K, VV, EV> addVertices(List<Vertex<K, VV>> verticesToAdd)

// 添加一条边。如果源端点和目的端点在图中不存在，它们也会被添加。
Graph<K, VV, EV> addEdge(Vertex<K, VV> source, Vertex<K, VV> target, EV
edgeValue)

// 添加一个边的list。如果在一个不存在的端点集合上添加边，边将被视为不合法，而且会被忽略。
Graph<K, VV, EV> addEdges(List<Edge<K, EV>> newEdges)

// 从图中移除指定的端点，以及它的边。
Graph<K, VV, EV> removeVertex(Vertex<K, VV> vertex)

// 从图中移除指定的端点的集合，以及它们的边。
Graph<K, VV, EV> removeVertices(List<Vertex<K, VV>> verticesToBeRemoved)

// 移除图中*所有* 与某条给定边match 的边。
Graph<K, VV, EV> removeEdge(Edge<K, EV> edge)

// 给定一个边的list，移除图中*所有* 与list中的边match 的边。
Graph<K, VV, EV> removeEdges(List<Edge<K, EV>> edgesToBeRemoved)
```

# 邻域方法

邻域方法可以在端点的first-hop 的邻居上进行聚合。reduceOnEdges()方法可以对一个端点的相邻边的值进行聚合， reduceOnNeighbors() 方法可以对一个端点的相邻点的值进行聚合。这些方法的聚合具有结合性和交换性，利用了内部的组合，因此极大提升了性能。

邻域的范围由EdgeDirection 这个参数指定，可选值包括IN,OUT,ALL。IN 聚合一个端点所有的入边， OUT 聚合一个端点所有的出边， ALL 聚合一个端点所有的边。

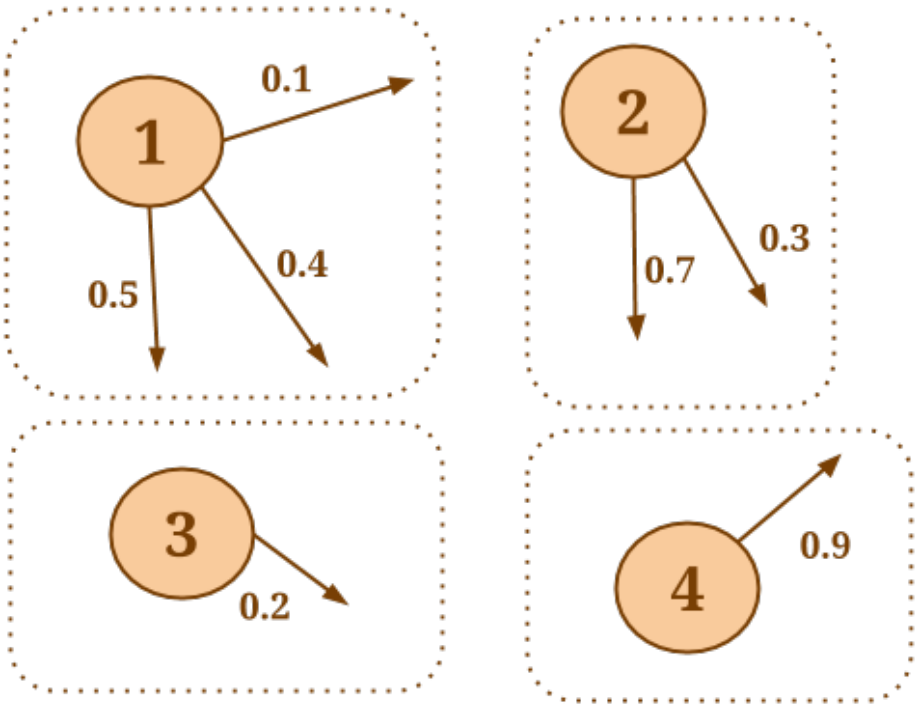
例如，假设你想从图中每个的端点的所有出边中选出最小weight：




下面的代码将计算每个端点的出边，并对得到的每个邻域应用自定义的SelectMinWeight()函数：

Java | Scala

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

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

// 用户自定义函数，用来选择最小weight
static final class SelectMinWeight implements ReduceEdgesFunction<Double> {

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

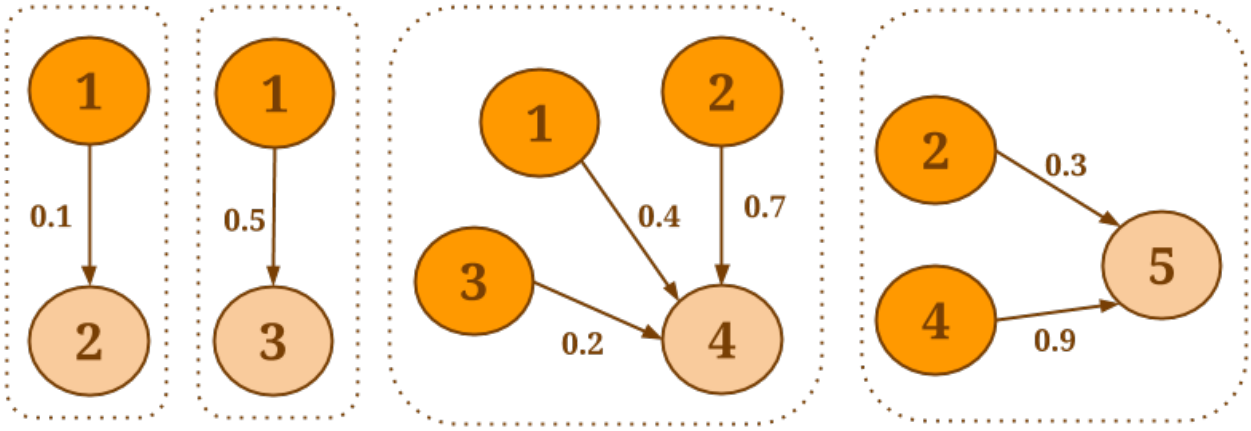



**result**
{ [1, 0.1], [2, 0.3], [3, 0.2], [4, 0.9] }

与之类似，假设你想计算每个端点的所有in-coming 邻居端点的value之和。下面的代码计算了每个端点的in-coming 邻居，并对每个邻居端点应用自定义的`SumValues()` 函数。

Java | Scala

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

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

// 自定义函数，用于计算邻居端点的value之和
static final class SumValues implements ReduceNeighborsFunction<Long> {

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




**result**
{ [2, 1], [3, 1], [4, 6], [5, 6] }

如果聚合函数不具有结合性和交换性，或者想从每个端点返回不止一个值，可以使用`groupReduceOnEdges()`和 `groupReduceOnNeighbors()` 这两个更一般性的方法。这些方法对每个端点返回0个，1个或者多个value，而且提供对所有邻居的访问。

例如，下面的代码将输出所有端点的pair，条件是连接它们的边的weight大于或者等于0.5：

Java | Scala

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

DataSet<Tuple2<Vertex<Long, Long>, Vertex<Long, Long>>> vertexPairs =
graph.groupReduceOnNeighbors(new SelectLargeWeightNeighbors(),
EdgeDirection.OUT);

// 用户自定函数，用来筛选用邻居端点，条件是连接它们的边的weight大于或者等于0.5
static final class SelectLargeWeightNeighbors implements
NeighborsFunctionWithVertexValue<Long, Long, Double,
                Tuple2<Vertex<Long, Long>, Vertex<Long, Long>>> {

                @Override
                public void iterateNeighbors(Vertex<Long, Long> vertex,
                                Iterable<Tuple2<Edge<Long, Double>,
Vertex<Long, Long>>> neighbors,
                                Collector<Tuple2<Vertex<Long, Long>,
Vertex<Long, Long>>> out) {

                        for (Tuple2<Edge<Long, Double>, Vertex<Long, Long>>
neighbor : neighbors) {
                                if (neighbor.f0.f2 > 0.5) {
                                        out.collect(new Tuple2<Vertex<Long,
Long>, Vertex<Long, Long>>(vertex, neighbor.f1));
                                }
                        }
                }
}
```

如果计算聚合值不需要访问端点的value （聚合计算应用在它身上），推荐使用两个效率更高的函数`EdgesFunction` 和 `NeighborsFunction`，或者是用户自定义的函数。如果需要访问端点的value，那么就应该使用`EdgesFunctionWithVertexValue` 和 `NeighborsFunctionWithVertexValue`。

Back to top

# 图的校验

Gelly 提供一种简单的工具来检测输入的图形的合法性。随着应用语境的变化，以某个标准衡量，一个图形既可能合法也可能不合法。例如，用户可能需要检查图形是否包含重复边，或者图的结构是否是二分的。要检查图的合法性，可以自己定义 `GraphValidator`并实现它的`validate()`方法。`InvalidVertexIdsValidator`是Gelly 中预定义的validator。它检测边的集合包含了合法的端点ID，换言之，所有边的ID 在端点的ID 集合中也存在。

**Java** | **Scala**

```
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>());
```