Flink GElly

# Graph API

## Graph Representation

1）图是由顶点的数据集（DataSet of vertices）和边的数据集（DataSet of edges）组成的。

2）图的节点是Vertex类型的，由唯一的ID和值组成。其ID需要实现Comparable接口，其值如果为空，则被设为NullValue类型。

3）图的边是Edge类型的，由Source ID、Target ID和值组成。其值为边的权重weight。

*// create a new vertex with a Long ID and a String value*

**val** v **=** **new** **Vertex(**1L**,** "foo"**)**

*// create a new vertex with a Long ID and no value*

**val** v **=** **new** **Vertex(**1L**,** **NullValue.**getInstance**())**

*// create a new Edge with Source ID , Target ID and value*

**val** e **=** **new** **Edge(**1L**,** 2L**,** 0.5**)**

*// reverse the source and target of this edge*

**val** reversed **=** e**.**reverse

**val** weight **=** e**.**getValue *// weight = 0.5*

## Graph Creation

可以根据以下几种方式来创建一个图。

1. From DataSet :

* edges and optional vertices[通过边和顶点的数据集创建图]

val env = ExecutionEnvironment.getExecutionEnvironment

val vertices: DataSet[Vertex[String, Long]] = ...

val edges: DataSet[Edge[String, Double]] = ...

val graph = Graph.**fromDataSet**(vertices, edges, env)

* Tuple2 representing the edges[通过二维数组创建图]

val env = ExecutionEnvironment.getExecutionEnvironment

val edges: DataSet[(String, String)] = ...

val graph = Graph.**fromTuple2DataSet**(edges, env)

val vertexTuples = env.readCsvFile[String, Long]("path/to/vertex/input")

val edgeTuples = env.readCsvFile[String, String, Double]("path/to/edge/input")

val graph = Graph.**fromTupleDataSet**(vertexTuples, edgeTuples, env)

1. From CSV :

*// create a Graph with String Vertex IDs, Long Vertex values and Double Edge values*

val graph = Graph.**fromCsvReader**[String, Long, Double](

pathVertices = "path/to/vertex/input",

pathEdges = "path/to/edge/input",

env = env)

1. From Collection :

val graph = Graph.**fromCollection**(vertexList, edgeList, env)

如果顶点没有提供，Gelly会自动的根据边的源和目标ID信息创建顶点ID。

## Graph properties

***1)***获得顶点集

getVertices**:** **DataSet[Vertex[K**, **VV]]**

***2)***获得边集

getEdges**:** **DataSet[Edge[K**, **EV]]**

***3)***获得顶点ID

getVertexIds**:** **DataSet[K]**

***4)***获得边集ID

getEdgeIds**:** **DataSet[(K**, **K)]**

***5)***获得入度

inDegrees**:** **DataSet[(K**, **LongValue)]**

***6)***获得出度

outDegrees**:** **DataSet[(K**, **LongValue)]**

***7)***获得度数

getDegrees**:** **DataSet[(K**, **LongValue)]**

***8)***顶点的个数

numberOfVertices**:** **Long**

***9)***边的个数

numberOfEdges**:** **Long**

***10)***获取边的数组形式

getTriplets**:** **DataSet[Triplet[K**, **VV**, **EV]]**

## Graph Transformations

1. Map:

mapVertices和mapEdges，返回一个新图，原顶点不变。

1. Translate
2. Filter
3. Join
4. Reverse
5. Undirected
6. Union
7. Difference
8. Intersect

## Graph Mutations

## Neighborhood Methods

## Graph Validation

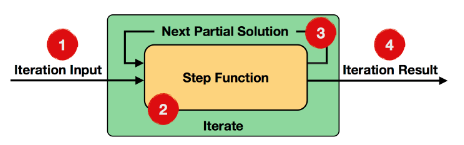
# Iterative Graph Processing

问题：Flink中的迭代是什么？

迭代在数值分析中是通过从一个初始估计出发，寻找一系列近似解来解决问题的方法。机器学习以及图分析中都有很多迭代算法，可以帮助找到数据中的有价值的信息。Flink是通过定义一个**step function**并将其嵌入到特殊的迭代操作中来实现迭代算法的，包括两种形式：Iterate和Delta Iterate。这两种操作方式在当前的迭代状态上，重复调用step function，直到达到特定的终止状态。

1. Iterate Operator

每次迭代中，step function会消费全部的输入（前一次迭代的结果或是初始数据集），并将计算的结果作为部分解的下一个版本，即下一次迭代的输入。当迭代次数达到最大值或是其他的终止条件时，产生最终的迭代结果。



1. Iteration Input：第一次迭代使用从数据源或是上一个算子中得到的初始化的输入
2. Step Function： 每一次迭代中都要执行的计算步骤。**是一个任意的数据流？**，包括map,reduce,join 等操作。
3. Next Partial Solution：每次迭代中，通过step function计算出的结果将被反馈到下一次的迭代中。
4. Iteration result：最后一次迭代的数据结果。

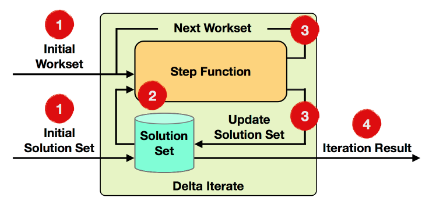
一次迭代的终止条件：达到最大迭代次数；自定义聚合器和收敛条件。

迭代操作可以抽象为以下步骤：

|  |
| --- |
| IterationState state **=** getInitialState**();**  **while** **(!**terminationCriterion**())** **{**  state **=** step**(**state**);**  **}**  setFinalState**(**state**);** |

1. Delta Iterate Operator

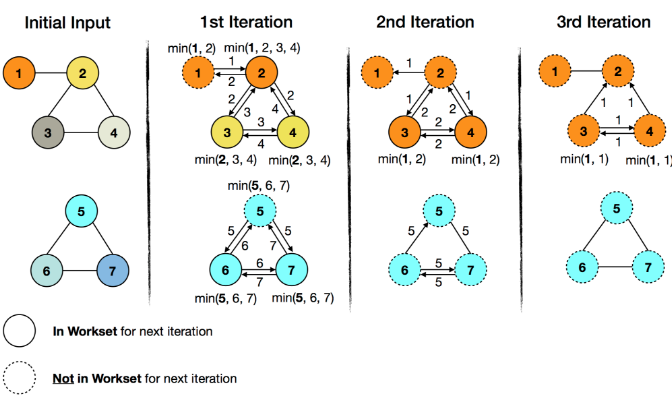
Delta迭代操作涵盖了增量迭代。增量迭代是选择性的修改解中的元素，递进求出解而不是全部重新计算。由于每次迭代并不是解集中的所有元素都修改，因此这种迭代方案能够实现更高效的算法。



1. Iteration Input：第一次迭代使用从数据源或是上一个算子中得到的初始化的WorkSet（工作集） 和SolutionSet（解集）。
2. Step Function： 每一次迭代中都要执行的计算步骤。**是一个任意的数据流？**，包括map,reduce,join 等操作。
3. Next Workset/Update Solution Set：下一个WorkSet工作集驱动迭代计算，并将反馈到下一个迭代。SolutionSet（解集）将被更新。两个集合都可以通过step function进行更新。
4. Iteration result：最后一次迭代的数据结果。

迭代终止的条件为Workset为空或达到最大迭代次数。也可以指定一个自定义的聚合器和收敛标准。

|  |
| --- |
| IterationState workset **=** getInitialState**();**  IterationState solution **=** getInitialSolution**();**  **while** **(!**terminationCriterion**())** **{**  **(**delta**,** workset**)** **=** step**(**workset**,** solution**);**  solution**.**update**(**delta**)**  **}**  setFinalState**(**solution**);** |



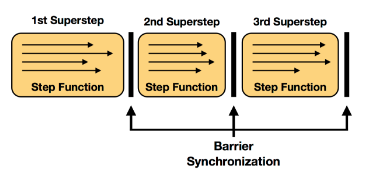
**Iterate VS Delta Iterate**

|  |  |  |
| --- | --- | --- |
|  | **Iterate** | **Delta Iterate** |
| **Iteration Input** | 部分解 | 工作集和解集 |
| **Step Function** | 任意数据流 | 任意数据流 |
| **State Update** | 下一个部分解 | 下一个工作集；更新解集 |
| **Iteration Result** | 上一个部分解 | 上一次迭代后的解集状态 |
| **Termination** | 最大迭代次数（默认）自定义聚合器收敛 | 最大迭代次数或工作集为空（默认）；自定义聚合器收敛 |

1. SuperStep 同步

Step function的多个并行实例在同一个SuperStep中完成，终止标准也将在superstep栅栏上评估。即Barrier分隔了相邻的superstep。

在同步迭代步骤中的计算叫做superstep（超步），每个superstep中的每个vertex节点执行一个用户定义的function。顶点之间通过messages（消息）进行通信。只要知道其他节点的唯一ID，便可以同其他顶点进行通信。在每个superstep中，所有顶点并行的执行相同的用户定义的计算逻辑。Superstep是同步执行的，保证一个superstep中的消息可以传递到下一个superstep中。

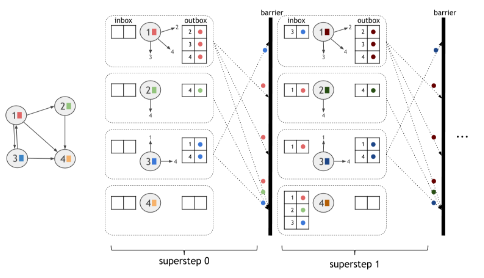


GElly 利用增量迭代（delta iteration）可以将各种图形处理模型应用到Flink数据流中，如 [vertex-centric](http://delivery.acm.org/10.1145/2490000/2484843/a22-salihoglu.pdf?ip=141.23.53.206&id=2484843&acc=ACTIVE%20SERVICE&key=2BA2C432AB83DA15.0F42380CB8DD3307.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=706313474&CFTOKEN=60107876&__acm__=1440408958_b131e035942130653e5782409b5c0cde), [partition-centric](http://researcher.ibm.com/researcher/files/us-ytian/giraph++.pdf), [gather-apply-scatter](http://www.eecs.harvard.edu/cs261/notes/gonzalez-2012.htm), [edge-centric](http://infoscience.epfl.ch/record/188535/files/paper.pdf), [neighborhood-centric](http://www.vldb.org/pvldb/vol7/p1673-quamar.pdf)等。在过去几年中，已经引入了用于分布式图形处理的许多不同的编程模型：以顶点为中心，以分区为中心，以聚集 - 应用 - 分散，以边缘为中心，以邻域为中心。 这些模型中的每一个都针对特定类别的图形应用程序，并且每个相应的系统实现分别优化运行时间。在Gelly，希望利用灵活的数据流模型和Flink的高效迭代，以支持在同一系统上的多个分布式图处理模型。目前提供以下几种模型：vertex-centric、scatter-gather、gather-sum-apply 。

## Vertex-centric [Pregel-like programming model]

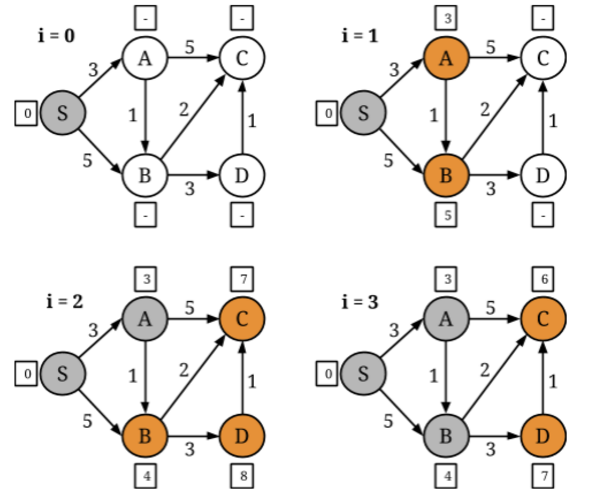
Gelly的以顶点为中心的编程模型利用了Flink的高效增量迭代操作。用户只需定义顶点的计算方法ComputerFunction。通过将这个function和迭代的次数作为参数传递给runVertexCentricIteration.可以用MessageCombiner（将需要传递给下一个superstep的消息先聚合起来）来降低通信开销。

Vertex-centric处理模型中每个超步中包括两个基本操作：1）**send Message**：向顶点发送信息；2）**Receive Message**：收集信息并更新状态。**每个超步中，所有活跃的顶点并行的执行相同的用户定义的操作，且每个超步是同步执行的，保证在一个超步期间发送的消息可以在下一个超步进行传递。**



Vertex-centric利用了Flink的Delta Iteration操作：如果一个顶点的状态没有更改，它将不会发送信息到相邻接点。这样可以增量的去更新图中的热点部分（发生状态更新的顶点），那些没有状态更新的顶点将被冻结（不再被更新）。

**Single-Source-Shortest-Paths（SSSP）**：1）初始化，每个顶点有一个初始距离（源顶点的初始值为0）。2）第一个superstep中，每个顶点检查他所接收到的消息，并选择他们中间最小的距离。如果选择的这个距离小于顶点当前值，它会更新它的状态并向邻居节点传送消息。如果不改变自身状态，则不会向下一个superstep的顶点传递消息。



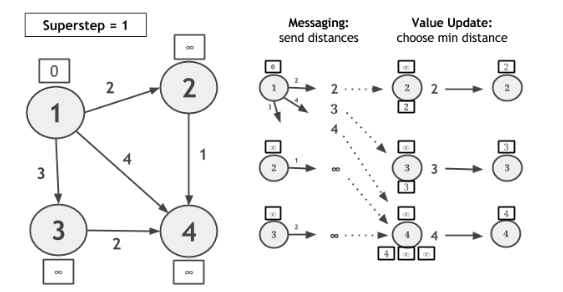
|  |
| --- |
| *// run the vertex-centric iteration, also passing the configuration parameters*  Graph**<**Long**,** Long**,** Double**>** result **=**  graph**.**runVertexCentricIteration**(**  **new** **Compute(),** **null,** maxIterations**,** parameters**);**  *// user-defined function*  **public** **static** **final** **class** **Compute** **extends** ComputeFunction **{**  LongSumAggregator aggregator **=** **new** **LongSumAggregator();**  **public** **void** **preSuperstep()** **{**  *// retrieve the Aggregator*  aggregator **=** getIterationAggregator**(**"sumAggregator"**);**  **}**  **public** **void** **compute(**Vertex**<**Long**,** Long**>** vertex**,** MessageIterator inMessages**)** **{**  *//do some computation*  Long partialValue **=** **...**  *// aggregate the partial value*  aggregator**.**aggregate**(**partialValue**);**  *// update the vertex value*  setNewVertexValue**(...);**  **}**  **}** |

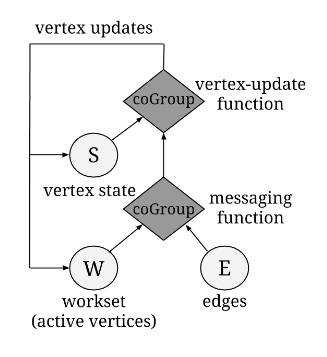
## Scatter-Gather Iterations

共分为两个步骤，scatter：产生发送给其他顶点的消息；gather：使用接收到的消息更新顶点值。用户只需实现两个function，ScatterFunction和GatherFunction。

ScatterFunction：向下一个超步中的顶点发送消息，核心方法是sendMessages。

GatherFunction：根据old状态和接收到的值更新顶点的状态，其核心方法是updateVertex。



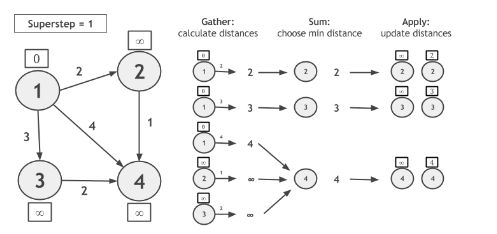


|  |
| --- |
| **CoGroup-Messaging Function:**  public void coGroup(Iterable<Edge<K, EV>> edges, Iterable<Vertex<K, Tuple3<VV, LongValue, LongValue>>> state, Collector<Tuple2<K, Message>> out) throws Exception {  final Iterator<Vertex<K, Tuple3<VV, LongValue, LongValue>>> stateIter = state.iterator();  if (stateIter.hasNext()) {  Vertex<K, Tuple3<VV, LongValue, LongValue>> vertexWithDegrees = stateIter.next();  nextVertex.setField(vertexWithDegrees.f0, 0);  nextVertex.setField(vertexWithDegrees.f1.f0, 1);  scatterFunction.setInDegree(vertexWithDegrees.f1.f1.getValue());  scatterFunction.setOutDegree(vertexWithDegrees.f1.f2.getValue());  scatterFunction.set(edges.iterator(), out, vertexWithDegrees.getId());  scatterFunction.sendMessages(nextVertex);  } } |

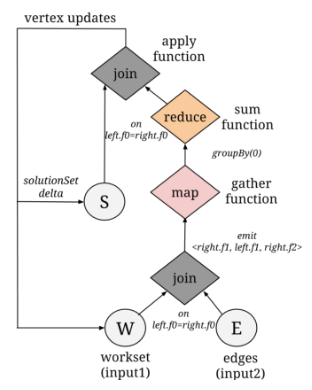
## Gather-Sum-Apply（GSA） Iteration

同Scatter-Gather一样使用同步迭代步骤。每个superstep中包括三个步骤gather、sum、apply。

1. Gather：根据用户定义的function从相邻顶点和边上收集值。 **【主动获取数据】**
2. Sum：通过使用用户定义的reducer，在Gather阶段收集到的值会被聚合成一个值
3. Apply：通过提供的function比较计算Sum阶段的值，并更新。



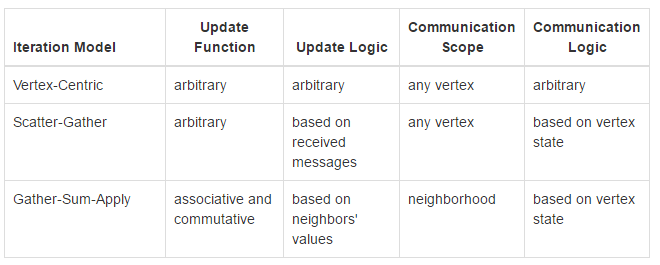
每个超步中，活跃的顶点和边做join操作，为每个顶点产生其neighborhood信息。Gather function 通过map function从每个neighborhood处收集值；然后结果按照顶点ID分组，并使用Sum function进行reduce操作；最后，sum阶段的结果与当前的顶点的值（solutionSet）相比较，符合更新条件的进行值更新，产生一个新的工作集，将作为下一次迭代的输入。



|  |
| --- |
| *// Gather*  **private** **static** **final** **class** **CalculateDistances** **extends** GatherFunction**<**Double**,** Double**,** Double**>** **{**  **public** Double **gather(**Neighbor**<**Double**,** Double**>** neighbor**)** **{**  **return** neighbor**.**getNeighborValue**()** **+** neighbor**.**getEdgeValue**();**  **}**  **}**  *// Sum*  **private** **static** **final** **class** **ChooseMinDistance** **extends** SumFunction**<**Double**,** Double**,** Double**>** **{**  **public** Double **sum(**Double newValue**,** Double currentValue**)** **{**  **return** Math**.**min**(**newValue**,** currentValue**);**  **}**  **}**  *// Apply*  **private** **static** **final** **class** **UpdateDistance** **extends** ApplyFunction**<**Long**,** Double**,** Double**>** **{**  **public** **void** **apply(**Double newDistance**,** Double oldDistance**)** **{**  **if** **(**newDistance **<** oldDistance**)** **{**  setResult**(**newDistance**);**  **}**  **}**  **}** |

## Iteration Abstractions Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Step-1** | **Step-2** | **特点** |
| Vertex-centric | Message | Update | 从顶点的角度来执行计算操作。一个超步中，每个顶点执行用户定义的function【遍历接收到的信息，更新顶点状态，将消息发送给下一个超步中的顶点】，且一个顶点可以向图中任意的顶点发送消息，只要提供顶点ID即可。**将所有的逻辑写到了一个函数里面** |
| Scatter-Gather | Scatter | Gather | 产生数据和更新数据逻辑分离，不仅容易执行且对性能产生积极的影响，低内存开销（不需要同时的访问定点的收件箱和发件箱）【像强连通图和近似最大权重匹配问题需要一个顶点同时访问它的收件箱和发件箱，这样使用该模式将会出现问题，**最直接的影响是顶点不能在同一个阶段产生消息并更新状态**】 |
| GSA | Gather+Sum | Apply | 和Scatter-Gather相近。区别是GSA的Gather阶段**作用于边**，而Scatter-Gather则作用于顶点；GSA和neighbor通信，而Scatter-Gather和任何节点通信。如果Gather阶段有很重的操作，使用GSA并扩展计算比将重任压到单个顶点上更有效。另外对于倾斜的图（即一些顶点比其他顶点有过多的相邻接点），在边上并行化是一种更有效的方法。 |



# Graph Generators

Graph Generators的操作有以下几个步骤：

1）图特有的参数配置；2）公共参数配置；3）调用generate()。

|  |  |  |
| --- | --- | --- |
| Comlete Graph（完全图） | 每个顶点之间都恰连有一条边的简单图 |  |
| Cycle Graph （循环图） | 顶点连接成一个封闭的链 |  |
| Empty Graph（空图） | 没有边的图 |  |
| Grid Graph（网格图） | 形似右图的图 |  |
| Hypercube Graph（超立方体图） | 形似右图的图 |  |
| Path Graph | 一个无向图，其中所有边形成单个路径 |  |
| RMat Graph | 使用递归矩阵（R-Mat）模型生成的定向或无向功率定律图 |  |
| Singleton Edge Graph | 包含独立双路径的无向图 |  |
| Star Graph（星型图） | 一个无向图，其中一个中心定点有连接到其他所有叶节点的边 |  |

# Graph Algorithm

## Algorithm comparation

|  |  |  |
| --- | --- | --- |
| **常用算法** | **Spark** | **Flink** |
| Community Detection | **N** | Y |
| Label propagation | **N** | Y |
| **Connected Components** | **Y** | **Y** |
| Strongly connected components | Y | **N** |
| GSA Connected Components | **N** | Y |
| **PageRank** | **Y** | **Y** |
| GSA PageRank | **N** | Y |
| Single Source Shortest Paths | **N** | Y |
| GSA Single Source Shortest Paths | **N** | Y |
| **Triangle Count** | **Y** | **Y** |
| Triangle Listing | **N** | Y |
| Triangle Enumerator | **N** | Y |
| Hyperlink-Induced Topic Search | **N** | Y |
| Summarization | **N** | Y |
| Adamic-Adar | **N** | Y |
| Jaccard Index | **N** | Y |
| Local Clustering Coefficient | **N** | Y |
| Global Clustering Coefficient | **N** | Y |
| Label propagation | Y | **N** |
| SVD++ | Y | **N** |

|  |
| --- |
| **Algorithm：**Connected Components（弱连通分量） |
| **Description：**  这是弱连通分量算法。如果有从一个顶点到另一个顶点的路径，在不考虑边的方向的情况下，这两个顶点属于同一个分量。求解连通分量是一个解决动态连通性的问题。   1. 深度优先搜索   使用id[]作为顶点索引的数组，如果v属于第i个连通分量，则id[v] = i;同时借助marked数组，标记访问过的顶点。构造函数会找出一个未被标记的顶点，并调用递归函数dfs进行深搜，标记并区分所有和他连通的顶点。如此重复直到所有顶点都被标记并区分。   1. Union-Find算法   开始将每个节点当做一个分量，每次对两个分量合并时，分量总数减一，最后得到连通分量数目。维护两个实例变量，一个是连通分量的总数，一个是数组id[]（用于存储所有节点所属的分量ID）。  **// QuickFind: 并查集实现**  **//** id[]数组中元素表示该节点所在的连通分量的ID,初始id[i]=i;  void find(int p){ return id[p];}  void Union(int p, int q){  int pID = find(p);  int qID = find(q);  // p和q已经在相同的连通分量不需要做处理  if (pID == qID){ return; }  // 遍历, 把所有id与p相同的元素都与q连接(id置为相同)  for (int i = 0; i < id.length(); i++){  if (id[i] == pID){ id[i] = qID;}  }  count--; // 连通分量数目减一  }  }  **// QuickUnion算法: 使用树结构实现并查集**  // id[]中的元素是同一个分量中另一个触点的名称，通过这种联系将同一个连通分量中的顶点链接起来。  int find(int p){  while(p!=id[p]) p = id[p];  return p;  }  void union(int p, int q){  int pRoot = find(p);  int qRoot = find(q);  if(pRoot == qRoot) return;  id[pRoot] = qRoot;  count--;  }  **// WeightedQuickUnion: 加权的QuickUnion, 用于减少构造树的深度**  // 在QuickUnion的基础上，增加一个数组size[]用来存放每个连通分量所在树的深度，并每次将小树连接到大树上。  int find(int p){  while(p!=id[p]) p = id[p];  return p;  }  void union(int p, int q){  int i = find(p);  int j = find(q);  if(i == j) return;  //将小树的根节点连接到大树的根节点上  If(size[i]<size[j]) {  Id[i] = j; size[j] += size[i];  }else{  Id[j] = I; size[i] += size[j];  }  Count--;  }  **Weighted QuickUnion with Path Compression**  int find(int i)  {  while (i != id[i])  {  // 将中间节点的父节点设为爷爷节点  // 使树扁平化  id[i] = id[id[i]];  i = id[i]; // DFS  }  }  比较：  算法 构造函数 Union find  -----------------------------------------------  QuickFind N N 1  -----------------------------------------------  QuickUnion N TreeHeight TreeHeight  -----------------------------------------------  Weighted N lgN lgN  QuickUnion  -----------------------------------------------  Weighted  QuickUnion N Very near to Very near  with Path 1 to 1  Compression  ----------------------------------------------- |
| **Application**：   1. 将节点抽象为计算机, 节点之间的连通性抽象为网络, 那么动态连接性问题可以让我们减少不必要的布线 2. 将节点抽象为用户, 节点之间的连通性抽象为共同的爱好, 那么一个高效的推荐系统就可以应用于产品之中 3. 将节点抽象为变量, 节点之间的连通性抽象为变量间的引用关系, 那么构造并查集就可以解决变量的循环引用问题, 避免引用环的存在, 方便引用技术和垃圾回收 |
| **Implementation**：   1. **Scatter-Gather ConnectedComponents**：   这个算法使用Scatter-Gather迭代模型。该实现使用可比较的顶点值作为初始分量的ID。顶点在每次迭代中传播其当前值。 在从其邻居接收分量ID时，如果其值低于其当前分量ID的值，则该顶点采用新的分量ID。当顶点不再更新其分量ID值或达到最大迭代次数时，算法收敛。  其中CCMessager=》ScatterFunction；CCUpdater=》GatherFunction  public static final class CCMessenger<K, VV extends Comparable<VV>>  extends ScatterFunction<K, VV, VV, NullValue>  implements ResultTypeQueryable<VV> {   private final TypeInformation<VV> typeInformation;   public CCMessenger(TypeInformation<VV> typeInformation) {  this.typeInformation = typeInformation;  }  @Override  public void sendMessages(Vertex<K, VV> vertex) throws Exception {  // send current minimum to neighbors  sendMessageToAllNeighbors(vertex.getValue());  }  @Override  public TypeInformation<VV> getProducedType() {  return typeInformation;  } }  public static final class CCUpdater<K, VV extends Comparable<VV>>  extends GatherFunction<K, VV, VV> {  @Override  public void updateVertex(Vertex<K, VV> vertex, MessageIterator<VV> messages) throws Exception {  VV current = vertex.getValue();  VV min = current;  for (VV msg : messages) {  if (msg.compareTo(min) < 0) {  min = msg;  }  }  if (!min.equals(current)) {  setNewVertexValue(min);  }  } }   1. **GSAConnectedComponents：**   使用GSA迭代模型。在gather阶段，每个顶点收集相邻顶点的值；在sum阶段，选择gather阶段收集到的值中的最小值；在apply阶段，将节点的当前值和sum阶段的之比较，选择最小的值。该算法在没有顶点更新值的时候或是迭代次数达到最大值的时候收敛。 |
| **Result**：  使用以下数据集运行验证。      运行结果：（k,v）->（顶点编号，所属连通分量编号）    迭代步骤如下：   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 1 | 1 | **1** | **2** | **2** | **4** | **5** | 7 | **7** | | 2 | 1 | 1 | **1** | **1** | **2** | **4** | 7 | 7 | | 3 | 1 | 1 | 1 | 1 | 1 | **2** | 7 | 7 | | 4 | 1 | 1 | 1 | 1 | 1 | **1** | 7 | 7 | |
| **Testing：**   1. **数据：**数据规模，图的类型 2. **配置**：参数包括edges列表（可从数据上分析）和maxIteration（最大迭代次数）。针对maxIteration，如果其设置的值小于图中各个连通分量中的树高的最大值，则会出错。（由于Scatter-Gather迭代模型以及GSA迭代模型在图中顶点不再有值的更新或是达到最大迭代次数时，便停止迭代。如果设置的最大迭代次数maxIteration不足以求出最终的结果，此时便会出错） 3. **扩展性：** |

|  |  |  |
| --- | --- | --- |
|  | Flink | Spark |
| Abstract | DataSet和DataStream是两个独立的抽象，不能合并在一起操作。 | 不管是批处理还是Streaming都是用RDD进行处理 |
| Memory | Flink从一开始就自己控制内存，把数据存储在自己管理的内存上，同时还直接操作二进制数据。 | 1.5之前使用java的内存管理做数据缓存，很容易造成OOM或GC。后来转向更加友好和精准的控制内存，即Tungsten项目。 |

CoGroup

@Override

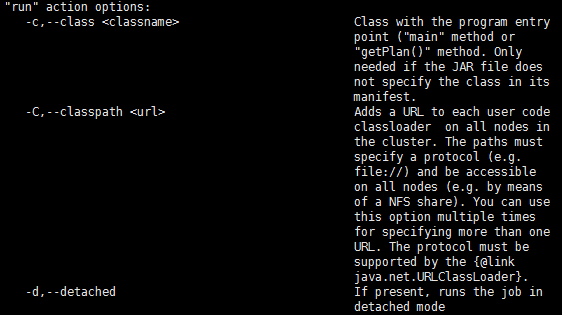
@Override  
public void coGroup(Iterable<Tuple2<K, Message>> messages,  
 Iterable<Vertex<K, VV>> vertex,  
 Collector<Vertex<K, VV>> out) throws Exception {  
 final Iterator<Vertex<K, VV>> vertexIter = vertex.iterator();  
 if (vertexIter.hasNext()) {  
 Vertex<K, VV> vertexState = vertexIter.next();  
 @SuppressWarnings("unchecked")  
 Iterator<Tuple2<?, Message>> downcastIter = (Iterator<Tuple2<?, Message>>) (Iterator<?>) messages.iterator();  
 messageIter.setSource(downcastIter);  
 gatherFunction.setOutput(vertexState, out);  
 gatherFunction.updateVertex(vertexState, messageIter);  
 }

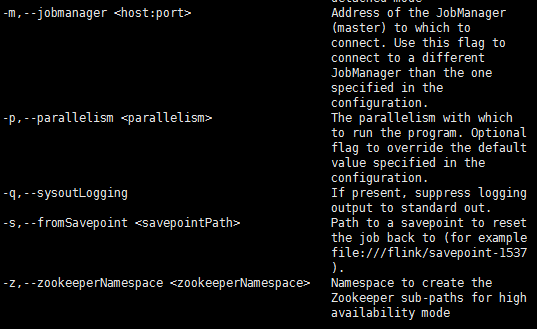
# Execution

bin/flink <ACTION> [OPTIONS] [ARGUMENTS]

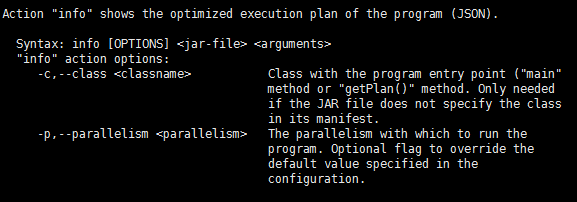
<ACTION>:

1. Action "run" compiles and runs a program. 【run 编译并运行一个程序】

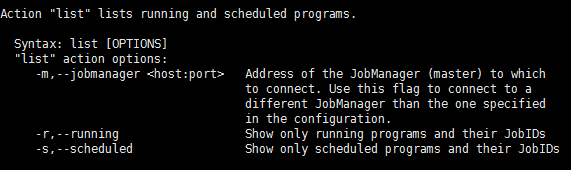




1. Action "info" shows the optimized execution plan of the program (JSON).



1. Action "list" lists running and scheduled programs.



1. Stop、cancel、savepoint

