Giraph原理：

Giraph原理由Google PageRank的图形处理平台Pregel继承而来。

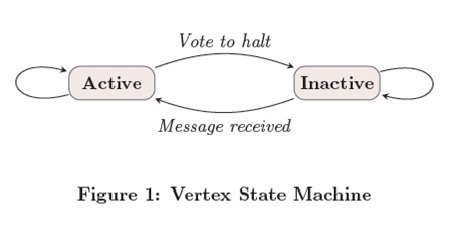
Pregel的计算过程由一系列被称为超级步(superstep)的迭代(iterations)组成。在每一个超级步中，计算框架都会针对每个顶点调用用户自定义的函数，这个过程是并行的，即不是一个一个顶点的串行调用，同一时刻可能有多个顶点被调用。该函数描述的是一个顶点V在一个superstep S中需要执行的操作。该函数可以读取前一个超级步(S-1)中发送给V的消息，并发送消息给其他顶点，这些消息将会在下一个超级步(S+1)中被接收，并且在此过程中修改顶点V及其出边的状态。消息通常沿着顶点的出边发送，但一个消息可能会被发送到任意已知ID的顶点上去。

在Pregel计算模型中，输入是一个有向图，该有向图的每一个顶点都有一个相应的由String描述的顶点标识符。每一个顶点都有一个与之对应的可修改的用户自定义值。每一条有向边都和其源顶点关联，并且也拥有一个可修改的用户自定义值，并同时还记录了其目标顶点的标识符。

一个典型的Pregel计算过程如下：读取输入初始化该图，当图被初始化好后，运行一系列的超级步直到整个计算结束，这些超级步之间通过一些全局的同步点分隔，输出结果结束计算。

在每个超级步中，顶点的计算都是并行的，每个顶点执行相同的用于表达给定算法逻辑的用户自定义函数。每个顶点可以修改其自身及其出边的状态，接收前一个超级步(S-1)中发送给它的消息，并发送消息给其他顶点(这些消息将会在下一个超级步中被接收)，甚至是修改整个图的拓扑结构。边，在这种计算模式中并不是核心对象，没有相应的计算运行在其上。

算法是否能够结束取决于是否所有的顶点都已经“vote”标识其自身已经达到“halt”状态了。在第0个超级步，所有顶点都处于active状态，所有的active顶点都会参与所有对应superstep中的计算。顶点通过将其自身的status设置成“halt”来表示它已经不再active。这就表示该顶点没有进一步的计算需要执行，除非被再次被外部触发，而Pregel框架将不会在接下来的superstep中执行该顶点，除非该顶点收到其它顶点传送的消息。如果顶点接收到消息被唤醒进入active状态，那么在随后的计算中该顶点必须显式的deactive。整个计算在所有顶点都达到“inactive”状态，并且没有message在传送的时候宣告结束。这种简单的状态机如下图所示：



整个Pregel程序的输出是所有顶点输出的集合。通常来都是一个跟输入同构的有向图，但是这并不是系统的一个必要属性，因为顶点和边可以在计算的过程中进行添加和删除。比如一个聚类算法，就有可能是从一个大图中生成的非连通顶点组成的小集合；一个对图的挖掘算法就可能仅仅是输出了从图中挖掘出来的聚合数据等。

最短路径问题：

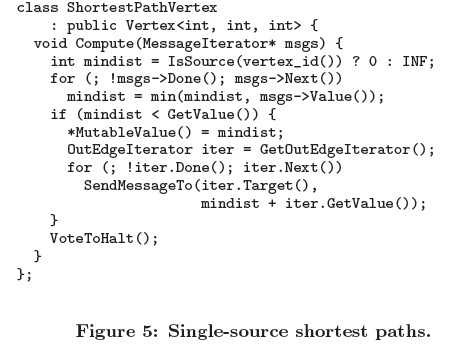
最短路径问题作为图问题中的一个重要问题有着广泛的应用，也有多种变种问题：

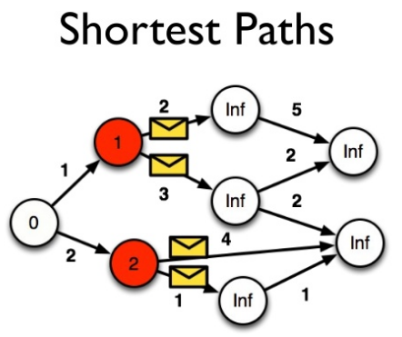
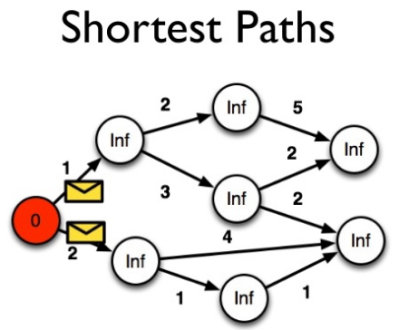
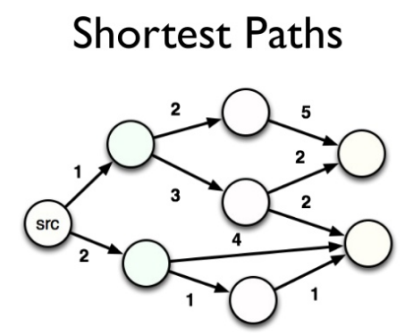
单源最短路径问题需要找到自单源顶点倒其他顶点的最短路径；s-t 做短路径问题需要找到顶点s到顶点t的最短路径，这个问题在导航方面有实际应用。All-pair最短路径问题由于其需求的存储空间代价过大，在大图中是不适用的。

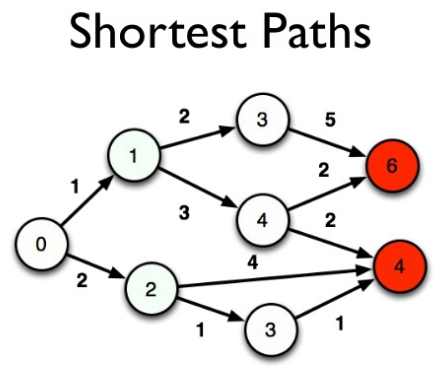
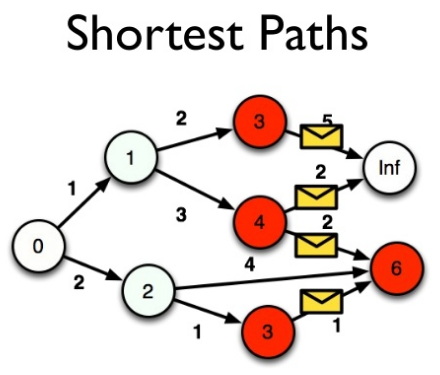
我们主要研究单源最短路径问题，在该算法中，我们假设与顶点关联的那个值被初始化为INF(比从源点到图中其他顶点的所有可能距离都大的一个常量)。在每个超级步中，每个顶点会首先接收到来自邻居传送过来的消息，该消息包含更新过的从源顶点到该顶点的潜在的最短距离，邻居节点发送过来的已经是源顶点到它本身的当前已知的最短距离+它到该顶点的距离了，所以该顶点接收到的已经是源顶点到它的距离了。如果这些更新里的最小值小于该顶点当前关联值，那么顶点就会更新这个值，并发送消息(该消息包含了该顶点的关联值+每个出边的关联值)给它的邻居。在第一个超级步中，只有源顶点会更新它的关联值(从INF改为0)，然后发送消息给它的直接邻居。然后这些邻居会更新它们的关联值，然后继续发送消息给它们的邻居，如此循环往复。当没有更新再发生的时候，算法就结束，之后所有顶点的关联值就是从源顶点到它的最短距离，若值为INF表示该顶点不可达。如果所有的边权重都是非负的，就可以保证该过程肯定会结束。

Singal source最短路径：

1. 初始化每个顶点对应值为INF（一个大于所有自源顶点的可能距离的常量值）
2. 在每一个superstep中
   1. 对每一个顶点，从邻居顶点收到信息
   2. 若最短距离小于当前距离，更新最短距离，并向邻居顶点发射信息。
   3. 收到信息的邻居节点更新最短距离信息，并向后序邻居节点发射信息，形成更新最短距离的波阵面。
3. 若每个顶点的对应值不再发生变化停止迭代。







SimpleShortestPathsVertex.java源码解析：

/\*

\* Licensed to the Apache Software Foundation (ASF) under one

\* or more contributor license agreements. See the NOTICE file

\* distributed with this work for additional information

\* regarding copyright ownership. The ASF licenses this file

\* to you under the Apache License, Version 2.0 (the

\* "License"); you may not use this file except in compliance

\* with the License. You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

package org.apache.giraph.examples;

import org.apache.giraph.conf.LongConfOption;

import org.apache.giraph.edge.Edge;

import org.apache.giraph.graph.Vertex;

import org.apache.hadoop.io.DoubleWritable;

import org.apache.hadoop.io.FloatWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.log4j.Logger;

/\*\*

\* Demonstrates the basic Pregel shortest paths implementation.

\*/

@Algorithm(

name = "Shortest paths",

description = "Finds all shortest paths from a selected vertex"

)

public class SimpleShortestPathsVertex extends

Vertex<LongWritable, DoubleWritable,

FloatWritable, DoubleWritable> {

/\*\* 最短路径 id \*/

public static final LongConfOption SOURCE\_ID =

new LongConfOption("SimpleShortestPathsVertex.sourceId", 1);

/\*\* Class logger \*/

private static final Logger LOG =

Logger.getLogger(SimpleShortestPathsVertex.class);

/\*\*

\* 顶点是否是源顶点

\*

\* @如果是原定点返回true

\*/

private boolean isSource() {

return getId().get() == SOURCE\_ID.get(getConf());

}

@Override

public void compute(Iterable<DoubleWritable> messages) {

/\*在超级歩为0的时候，即初始情况下，将每个点的对应值初始化为最大值（不可达到）\*/

if (getSuperstep() == 0) {

setValue(new DoubleWritable(Double.MAX\_VALUE));

}

/\*如果是源顶点则最小距离更新为0，否则为最大值\*/

double minDist = isSource() ? 0d : Double.MAX\_VALUE;

/\*得到最小更新值：messages为邻居节点得到的距离消息，若值小于minDist则将其更新为消息中的最小距离值\*/

for (DoubleWritable message : messages) {

minDist = Math.min(minDist, message.get());

}

/\*debug环境下输出日志\*/

if (LOG.isDebugEnabled()) {

LOG.debug("Vertex " + getId() + " got minDist = " + minDist +

" vertex value = " + getValue());

}

/\*如果minDist小于顶点关联值则更新其为minDist\*/

if (minDist < getValue().get()) {

setValue(new DoubleWritable(minDist));

/\*发送顶点关联值和每一个出边关联值给每一个邻居\*/

for (Edge<LongWritable, FloatWritable> edge : getEdges()) {

double distance = minDist + edge.getValue().get();

/\*debug环境下输出日志：当前顶点（id）发给邻居顶点（id）待更新距离\*/

if (LOG.isDebugEnabled()) {

LOG.debug("Vertex " + getId() + " sent to " +

edge.getTargetVertexId() + " = " + distance);

}

sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));

}

}

/\*把顶点置为InActive状态\*/

voteToHalt();

}

}

**Writable**：

当要在进程间传递对象或持久化对象的时候，就需要序列化对象成字节流，反之当要将接收到或从磁盘读取的字节流转换为对象，就要进行反序列化。Writable是Hadoop的序列化格式，Hadoop定义了这样一个Writable接口。

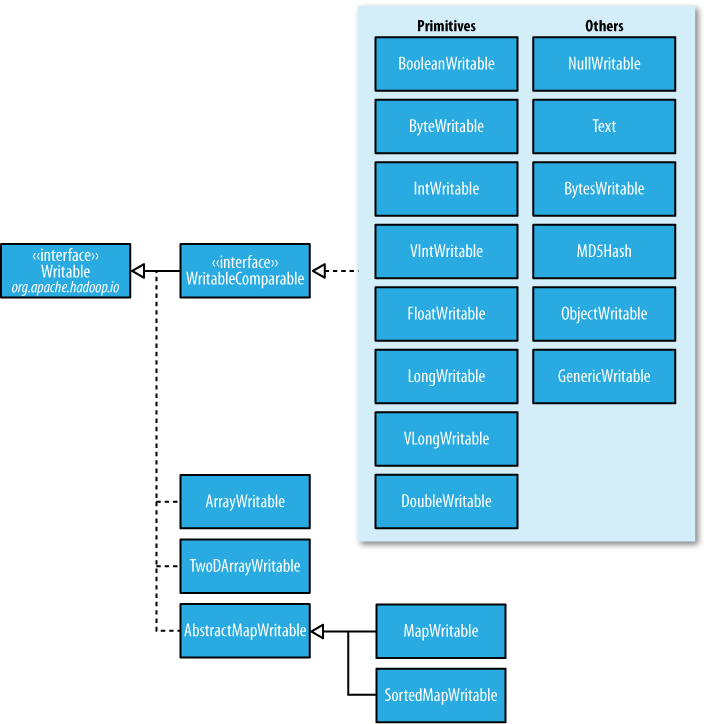
public interface Writable {

void write(DataOutput out) throws IOException;

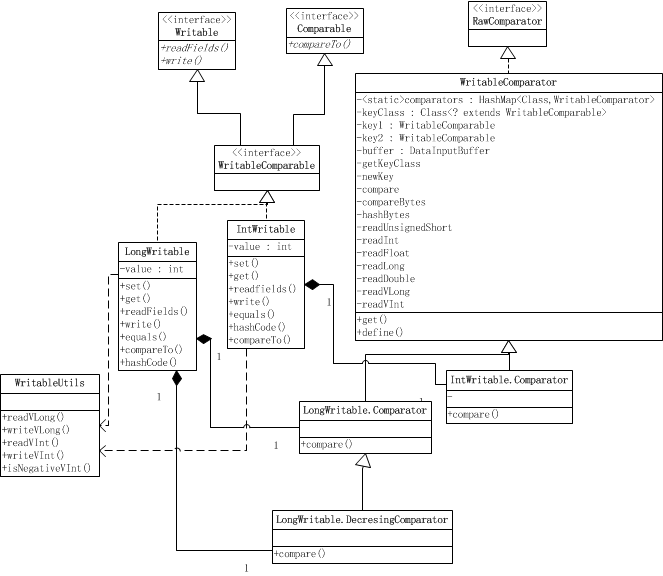
void readFields(DataInput in) throws IOException;

}

一个类要支持可序列化只需实现这个接口即可。下面是Writable类得层次结构，借用了<<Hadoop:The Definitive Guide>>的图。



下面我们一点一点来看，先是IntWritable和LongWritable



WritableComparable接口扩展了Writable和Comparable接口，以支持比较。正如层次图中看到，IntWritable、LongWritable、ByteWritable等基本类型都实现了这个接口。IntWritable和LongWritable的readFields()都直接从实现了DataInput接口的输入流中读取二进制数据并分别重构成int型和long型，而write()则直接将int型数据和long型数据直接转换成二进制流。IntWritable和LongWritable都含有相应的Comparator内部类，这是用来支持对在不反序列化为对象的情况下对数据流中的数据单位进行直接的，这是一个优化，因为无需创建对象。看下面IntWritable的代码片段：

public class IntWritable implements WritableComparable {

private int value;

//…… other methods

public static class Comparator extends WritableComparator {

public Comparator() {

super(IntWritable.class);

}

public int compare(byte[] b1, int s1, int l1,

byte[] b2, int s2, int l2) {

int thisValue = readInt(b1, s1);

int thatValue = readInt(b2, s2);

return (thisValue<thatValue ? -1 : (thisValue==thatValue ? 0 : 1));

}

}

static { // register this comparator

WritableComparator.define(IntWritable.class, new Comparator());

}

}

代码中static块调用WritableComparator的static方法define()用来注册上面这个Comparator，就是将其加入WritableComparator的comparators成员中，comparators是HashMap类型且是static的。这样，就告诉WritableComparator，当我使用WritableComparator.get（IntWritable.class）方法的时候，你返回我注册的这个Comparator给我[对IntWritable来说就是IntWritable.Comparator]，然后我就可以使用comparator.compare(byte[] b1, int s1, int l1,byte[] b2, int s2, int l2)来比较b1和b2，而不需要将它们反序列化成对象[像下面代码中]。comparator.compare(byte[] b1, int s1, int l1,byte[] b2, int s2, int l2)中的readInt()是从WritableComparator继承来的，它将IntWritable的value从byte数组中通过移位转换出来。

//params byte[] b1, byte[] b2

RawComparator<IntWritable> comparator = WritableComparator.get(IntWritable.class);

comparator.compare(b1,0,b1.length,b2,0,b2.length);

当comparators中没有注册要比较的类的Comparator，则会返回一个默认的Comparator，然后使用这个默认Comparator的compare(byte[] b1, int s1, int l1,byte[] b2, int s2, int l2)方法比较b1、b2的时候还是要序列化成对象的，详见后面细讲WritableComparator。

LongWritable的方法基本和IntWritable一样，区别就是LongWritable的值是long型，且多了一个额外的LongWritable.DecresingComparator，它继承于LongWritable.Comparator，只是它的比较方法返回值与使用LongWritable.Comparator比较相反[取负]，这个应当是为降序排序准备的。

public class LongWritable implements WritableComparable {

private long value;

//……others

/\*\* A decreasing Comparator optimized for LongWritable. \*/

public static class DecreasingComparator extends Comparator {

public int compare(WritableComparable a, WritableComparable b) {

return -super.compare(a, b);

}

public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {

return -super.compare(b1, s1, l1, b2, s2, l2);

}

}

static { // register default comparator

WritableComparator.define(LongWritable.class, new Comparator());

}

}

**getValue()：返回顶点对应值（存储在顶点中的值）**

**getId(): 返回顶点ID**

**setValue(V value):设定顶点对应值（立即生效）**

**voteToHalt(): 调用这个函数后对于当前顶点compute函数 将不再被调用。**

**getEdges():得到当前顶点的出边（仅可读）**

**compute():是Vertex类中的虚函数，用户覆写Vertex类的虚函数Compute()，该函数会在每一个超级步中对每一个顶点进行调用。预定义的Vertex类方法允许Compute()方法查询当前顶点及其边的信息，以及发送消息到其他的顶点。Compute()方法可以通过调用GetValue()方法来得到当前顶点的值，或者通过调用setValue()方法来修改当前顶点的值。同时还可以通过由出边的迭代器提供的方法来查看修改出边对应的值。这种状态的修改是立时可见的。由于这种可见性仅限于被修改的那个顶点，所以不同顶点并发进行的数据访问是不存在竞争关系的。**