```
/*
 * 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 de.tuberlin.dima.bdapro.solutions;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.GroupReduceFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import
org.apache.flink.api.java.functions.FunctionAnnotation.ForwardedFiel
ds;
import org.apache.flink.api.java.operators.IterativeDataSet;
import org.apache.flink.api.java.tuple.Tuple1;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.util.Collector;

import java.util.ArrayList;

import static
org.apache.flink.api.java.aggregation.Aggregations.SUM;

/**
 * A basic implementation of the Page Rank algorithm using a bulk
iteration.
 *
 * <p>This implementation requires a set of pages and a set of
directed links as input and works as follows. <br>
 * In each iteration, the rank of every page is evenly distributed
to all pages it points to.
 * Each page collects the partial ranks of all pages that point to
it, sums them up, and applies a dampening factor to the sum.
 * The result is the new rank of the page. A new iteration is
```

```
started with the new ranks of all pages.
 * This implementation terminates after a fixed number of
iterations.<br>
 * This is the Wikipedia entry for the <a href="http://
en.wikipedia.org/wiki/Page_rank">Page Rank algorithm</a>.
 *
 * <p>Input files are plain text files and must be formatted as
follows:
 * <ul>
 * <li>Pages represented as an (long) ID separated by new-line
characters.<br>
 * For example <code>"1\n2\n12\n42\n63"</code> gives five pages with
IDs 1, 2, 12, 42, and 63.
 * <li>Links are represented as pairs of page IDs which are
separated by space
 * characters. Links are separated by new-line characters.<br>
 * For example <code>"1 2\n2 12\n1 12\n42 63"</code> gives four
(directed) links (1)-&gt;(2), (2)-&gt;(12), (1)-&gt;(12), and (42)-
&gt;(63).<br>
 * For this simple implementation it is required that each page has
at least one incoming and one outgoing link (a page can point to
itself).
 * </ul>
 *
 * <p>Usage: <code>PageRankBasic --pages &lt;path&gt; --links
&lt;path&gt; --output &lt;path&gt; --numPages &lt;n&gt; --iterations
&lt;n&gt;</code><br>
 * If no parameters are provided, the program is run with default
data from {@link de.tuberlin.dima.bdapro.solutions.PageRankData} and
10 iterations.
 *
 * <p>This example shows how to use:
 * <ul>
 * <li>Bulk Iterations
 * <li>Default Join
 * <li>Configure user-defined functions using constructor
parameters.
 * </ul>
 */
@SuppressWarnings("serial")
public class PageRank {

    private static final double DAMPENING_FACTOR = 0.85;
    private static final double EPSILON = 0.0001;

    //
*************************************************************************
*****
    //     PROGRAM
    //
*************************************************************************
*****

    public static void main(String[] args) throws Exception {
```

```
        ParameterTool params = ParameterTool.fromArgs(args);

        final int numPages = params.getInt("numPages",
PageRankData.getNumberOfPages());
        final int maxIterations = params.getInt("iterations", 10);

        // set up execution environment
        final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

        // make the parameters available to the web ui
        env.getConfig().setGlobalJobParameters(params);

        // get input data
        DataSet<Long> pagesInput = getPagesDataSet(env, params);
        DataSet<Tuple2<Long, Long>> linksInput =
getLinksDataSet(env, params);

        // assign initial rank to pages
        DataSet<Tuple2<Long, Double>> pagesWithRanks = pagesInput.
                map(new RankAssigner((1.0d / numPages)));

        // build adjacency list from link input
        DataSet<Tuple2<Long, Long[]>> adjacencyListInput =
                linksInput.groupBy(0).reduceGroup(new
BuildOutgoingEdgeList());

        // set iterative data set
        IterativeDataSet<Tuple2<Long, Double>> iteration =
pagesWithRanks.iterate(maxIterations);

        DataSet<Tuple2<Long, Double>> newRanks = iteration
                // join pages with outgoing edges and distribute
rank
                .join(adjacencyListInput).where(0).equalTo(0).flatMa
p(new JoinVertexWithEdgesMatch())
                // collect and sum ranks
                .groupBy(0).aggregate(SUM, 1)
                // apply dampening factor
                .map(new Dampener(DAMPENING_FACTOR, numPages));

        DataSet<Tuple2<Long, Double>> finalPageRanks =
iteration.closeWith(
                newRanks,
                newRanks.join(iteration).where(0).equalTo(0)
                        // termination condition
                        .filter(new EpsilonFilter()));

        // emit result
        if (params.has("output")) {
            finalPageRanks.writeAsCsv(params.get("output"), "\n", "
");
            // execute program
```

```
            env.execute("Basic Page Rank Example");
        } else {
            System.out.println("Printing result to stdout. Use --
output to specify output path.");
            finalPageRanks.print();
        }
    }

    //
*************************************************************************
*****
    //     USER FUNCTIONS
    //
*************************************************************************
*****

    /**
     * A map function that assigns an initial rank to all pages.
     */
    public static final class RankAssigner implements
MapFunction<Long, Tuple2<Long, Double>> {
        Tuple2<Long, Double> outPageWithRank;

        public RankAssigner(double rank) {
            this.outPageWithRank = new Tuple2<Long, Double>(-1L,
rank);
        }

        @Override
        public Tuple2<Long, Double> map(Long page) {
            outPageWithRank.f0 = page;
            return outPageWithRank;
        }
    }

    /**
     * A reduce function that takes a sequence of edges and builds
the adjacency list for the vertex where the edges
     * originate. Run as a pre-processing step.
     */
    @ForwardedFields("0")
    public static final class BuildOutgoingEdgeList implements
GroupReduceFunction<Tuple2<Long, Long>, Tuple2<Long, Long[]>> {

        private final ArrayList<Long> neighbors = new
ArrayList<Long>();

        @Override
        public void reduce(Iterable<Tuple2<Long, Long>> values,
Collector<Tuple2<Long, Long[]>> out) {
            neighbors.clear();
            Long id = 0L;

            for (Tuple2<Long, Long> n : values) {
```

```
                id = n.f0;
                neighbors.add(n.f1);
            }
            out.collect(new Tuple2<Long, Long[]>(id,
neighbors.toArray(new Long[neighbors.size()])));
        }
    }

    /**
     * Join function that distributes a fraction of a vertex's rank
to all neighbors.
     */
    public static final class JoinVertexWithEdgesMatch implements
FlatMapFunction<Tuple2<Tuple2<Long, Double>, Tuple2<Long, Long[]>>,
Tuple2<Long, Double>> {

        @Override
        public void flatMap(Tuple2<Tuple2<Long, Double>,
Tuple2<Long, Long[]>> value, Collector<Tuple2<Long, Double>> out){
            Long[] neighbors = value.f1.f1;
            double rank = value.f0.f1;
            double rankToDistribute = rank / ((double)
neighbors.length);

            for (Long neighbor: neighbors) {
                out.collect(new Tuple2<Long, Double>(neighbor,
rankToDistribute));
            }
        }
    }

    /**
     * The function that applies the page rank dampening formula.
     */
    @ForwardedFields("0")
    public static final class Dampener implements
MapFunction<Tuple2<Long, Double>, Tuple2<Long, Double>> {

        private final double dampening;
        private final double randomJump;

        public Dampener(double dampening, double numVertices) {
            this.dampening = dampening;
            this.randomJump = (1 - dampening) / numVertices;
        }

        @Override
        public Tuple2<Long, Double> map(Tuple2<Long, Double> value)
{
            value.f1 = (value.f1 * dampening) + randomJump;
            return value;
        }
    }
```

```
    /**
     * Filter that filters vertices where the rank difference is
below a threshold.
     */
    public static final class EpsilonFilter implements
FilterFunction<Tuple2<Tuple2<Long, Double>, Tuple2<Long, Double>>> {

        @Override
        public boolean filter(Tuple2<Tuple2<Long, Double>,
Tuple2<Long, Double>> value) {
            return Math.abs(value.f0.f1 - value.f1.f1) > EPSILON;
        }
    }

    //
*************************************************************************
*****
    //     UTIL METHODS
    //
*************************************************************************
*****

    private static DataSet<Long>
getPagesDataSet(ExecutionEnvironment env, ParameterTool params) {
        if (params.has("pages")) {
            return env.readCsvFile(params.get("pages"))
                    .fieldDelimiter(" ")
                    .types(Long.class)
                    .map(new MapFunction<Tuple1<Long>, Long>() {
                        @Override
                        public Long map(Tuple1<Long> v) {
                            return v.f0;
                        }
                    });
        } else {
            System.out.println("Executing PageRank example with
default pages data set.");
            System.out.println("Use --pages to specify file
input.");
            return PageRankData.getDefaultPagesDataSet(env);
        }
    }

    private static DataSet<Tuple2<Long, Long>>
getLinksDataSet(ExecutionEnvironment env, ParameterTool params) {
        if (params.has("links")) {
            return env.readCsvFile(params.get("links"))
                    .fieldDelimiter(" ")
                    .types(Long.class, Long.class);
        } else {
            System.out.println("Executing PageRank example with
default links data set.");
            System.out.println("Use --links to specify file
input.");
```

```
            return PageRankData.getDefaultEdgeDataSet(env);
        }
    }
}
```