package com.twitter.graph.batch.job.tweepcred

import com.twitter.scalding.\_

/\*\*

\* weighted page rank for the given graph, start from the given pagerank,

\* perform one iteration, test for convergence, if not yet, clone itself

\* and start the next page rank job with updated pagerank as input;

\* if converged, start ExtractTweepcred job instead

\*

\* Options:

\* --pwd: working directory, will read/generate the following files there

\* numnodes: total number of nodes

\* nodes: nodes file <'src\_id, 'dst\_ids, 'weights, 'mass\_prior>

\* pagerank: the page rank file eg pagerank\_0, pagerank\_1 etc

\* totaldiff: the current max pagerank delta

\* Optional arguments:

\* --weighted: do weighted pagerank, default false

\* --curiteration: what is the current iteration, default 0

\* --maxiterations: how many iterations to run. Default is 20

\* --jumpprob: probability of a random jump, default is 0.1

\* --threshold: total difference before finishing early, default 0.001

\*

\* plus the following options for ExtractTweepcred:

\* --user\_mass: user mass tsv file, generated by twadoop user\_mass job

\* --output\_pagerank: where to put pagerank file

\* --output\_tweepcred: where to put tweepcred file

\* Optional:

\* --post\_adjust: whether to do post adjust, default true

\*

\*/

class WeightedPageRank(args: Args) extends Job(args) {

val ROW\_TYPE\_1 = 1

val ROW\_TYPE\_2 = 2

val PWD = args("pwd")

val ALPHA = args.getOrElse("jumpprob", "0.1").toDouble

val WEIGHTED = args.getOrElse("weighted", "false").toBoolean

val THRESHOLD = args.getOrElse("threshold", "0.001").toDouble

val MAXITERATIONS = args.getOrElse("maxiterations", "20").toInt

val CURITERATION = args.getOrElse("curiteration", "0").toInt

// 'size

val numNodes = getNumNodes(PWD + "/numnodes")

// 'src\_id, 'dst\_ids, 'weights, 'mass\_prior

val nodes = getNodes(PWD + "/nodes")

// 'src\_id\_input, 'mass\_input

val inputPagerank = getInputPagerank(PWD + "/pagerank\_" + CURITERATION)

// one iteration of pagerank

val outputPagerank = doPageRank(nodes, inputPagerank)

val outputFileName = PWD + "/pagerank\_" + (CURITERATION + 1)

outputPagerank

.project('src\_id, 'mass\_n)

.write(Tsv(outputFileName))

// detect convergence

val totalDiff = outputPagerank

.mapTo(('mass\_input, 'mass\_n) -> 'mass\_diff) { args: (Double, Double) =>

scala.math.abs(args.\_1 - args.\_2)

}

.groupAll { \_.sum[Double]('mass\_diff) }

.write(Tsv(PWD + "/totaldiff"))

/\*\*

\* test convergence, if not yet, kick off the next iteration

\*/

override def next = {

// the max diff generated above

val totalDiff = Tsv(PWD + "/totaldiff").readAtSubmitter[Double].head

if (CURITERATION < MAXITERATIONS - 1 && totalDiff > THRESHOLD) {

val newArgs = args + ("curiteration", Some((CURITERATION + 1).toString))

Some(clone(newArgs))

} else {

val newArgs = args + ("input\_pagerank", Some(outputFileName))

Some(new ExtractTweepcred(newArgs))

}

}

def getInputPagerank(fileName: String) = {

Tsv(fileName).read

.mapTo((0, 1) -> ('src\_id\_input, 'mass\_input)) { input: (Long, Double) =>

input

}

}

/\*\*

\* read the pregenerated nodes file <'src\_id, 'dst\_ids, 'weights, 'mass\_prior>

\*/

def getNodes(fileName: String) = {

mode match {

case Hdfs(\_, conf) => {

SequenceFile(fileName).read

.mapTo((0, 1, 2, 3) -> ('src\_id, 'dst\_ids, 'weights, 'mass\_prior)) {

input: (Long, Array[Long], Array[Float], Double) =>

input

}

}

case \_ => {

Tsv(fileName).read

.mapTo((0, 1, 2, 3) -> ('src\_id, 'dst\_ids, 'weights, 'mass\_prior)) {

input: (Long, String, String, Double) =>

{

(

input.\_1,

// convert string to int array

if (input.\_2 != null && input.\_2.length > 0) {

input.\_2.split(",").map { \_.toLong }

} else {

Array[Long]()

},

// convert string to float array

if (input.\_3 != null && input.\_3.length > 0) {

input.\_3.split(",").map { \_.toFloat }

} else {

Array[Float]()

},

input.\_4

)

}

}

}

}

}

/\*\*

\* the total number of nodes, single line file

\*/

def getNumNodes(fileName: String) = {

Tsv(fileName).read

.mapTo(0 -> 'size) { input: Long =>

input

}

}

/\*\*

\* one iteration of pagerank

\* inputPagerank: <'src\_id\_input, 'mass\_input>

\* return <'src\_id, 'mass\_n, 'mass\_input>

\*

\* Here is a highlevel view of the unweighted algorithm:

\* let

\* N: number of nodes

\* inputPagerank(N\_i): prob of walking to node i,

\* d(N\_j): N\_j's out degree

\* then

\* pagerankNext(N\_i) = (\sum\_{j points to i} inputPagerank(N\_j) / d\_j)

\* deadPagerank = (1 - \sum\_{i} pagerankNext(N\_i)) / N

\* randomPagerank(N\_i) = userMass(N\_i) \* ALPHA + deadPagerank \* (1-ALPHA)

\* pagerankOutput(N\_i) = randomPagerank(N\_i) + pagerankNext(N\_i) \* (1-ALPHA)

\*

\* For weighted algorithm:

\* let

\* w(N\_j, N\_i): weight from N\_j to N\_i

\* tw(N\_j): N\_j's total out weights

\* then

\* pagerankNext(N\_i) = (\sum\_{j points to i} inputPagerank(N\_j) \* w(N\_j, N\_i) / tw(N\_j))

\*

\*/

def doPageRank(nodeRows: RichPipe, inputPagerank: RichPipe): RichPipe = {

// 'src\_id, 'dst\_ids, 'weights, 'mass\_prior, 'mass\_input

val nodeJoined = nodeRows

.joinWithSmaller('src\_id -> 'src\_id\_input, inputPagerank)

.discard('src\_id\_input)

// 'src\_id, 'mass\_n

val pagerankNext = nodeJoined

.flatMapTo(('dst\_ids, 'weights, 'mass\_input) -> ('src\_id, 'mass\_n)) {

args: (Array[Long], Array[Float], Double) =>

{

if (args.\_1.length > 0) {

if (WEIGHTED) {

// weighted distribution

val total: Double = args.\_2.sum

(args.\_1 zip args.\_2).map { idWeight: (Long, Float) =>

(idWeight.\_1, args.\_3 \* idWeight.\_2 / total)

}

} else {

// equal distribution

val dist: Double = args.\_3 / args.\_1.length

args.\_1.map { id: Long =>

(id, dist)

}

}

} else {

//Here is a node that points to no other nodes (dangling)

Nil

}

}

}

.groupBy('src\_id) {

\_.sum[Double]('mass\_n)

}

// 'sum\_mass

val sumPagerankNext = pagerankNext.groupAll { \_.sum[Double]('mass\_n -> 'sum\_mass) }

// 'deadMass

// single row jobs

// the dead page rank equally distributed to every node

val deadPagerank = sumPagerankNext

.crossWithTiny(numNodes)

.map(('sum\_mass, 'size) -> 'deadMass) { input: (Double, Long) =>

(1.0 - input.\_1) / input.\_2

}

.discard('size, 'sum\_mass)

// 'src\_id\_r, 'mass\_n\_r

// random jump probability plus dead page rank

val randomPagerank = nodeJoined

.crossWithTiny(deadPagerank)

.mapTo(('src\_id, 'mass\_prior, 'deadMass, 'mass\_input) -> ('src\_id, 'mass\_n, 'mass\_input)) {

ranks: (Long, Double, Double, Double) =>

(ranks.\_1, ranks.\_2 \* ALPHA + ranks.\_3 \* (1 - ALPHA), ranks.\_4)

}

// 'src\_id, 'mass\_n

// scale next page rank to 1-ALPHA

val pagerankNextScaled = pagerankNext

.map('mass\_n -> ('mass\_n, 'mass\_input)) { m: Double =>

((1 - ALPHA) \* m, 0.0)

}

// 'src\_id, 'mass\_n, 'mass\_input

// random probability + next probability

(randomPagerank ++ pagerankNextScaled)

.groupBy('src\_id) {

\_.sum[Double]('mass\_input) // keep the input pagerank

.sum[Double]('mass\_n) // take the sum

}

}

}