Federico Baldassarre

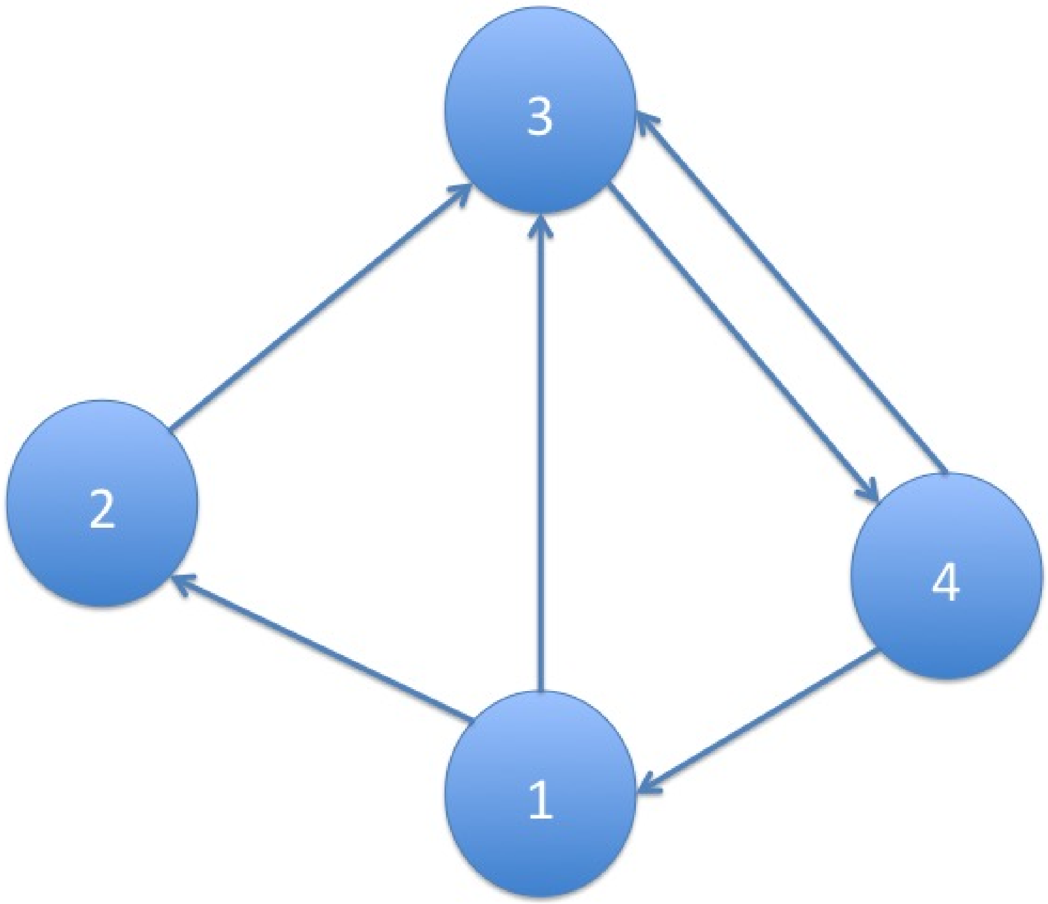
17 October 2017

Iterative PageRank

In short PageRank is a “vote”, by all the other pages on the web, about how important a page is. A link to a page counts as a vote of support. If there is no link, it means there is no support for that page. The PageRank of each page depends on the PageRank of the pages pointing to it. But we won’t know what PR those pages have until the pages pointing to them have their PR calculated and so on.

```
var PR = Array.fill(n)( 1.0 )
val oldPR = Array.fill(n)( 0.0 )
while( max(abs(PR - oldPr)) > tol ) {
  swap(oldPR, PR)
  for( i <- 0 until n if abs(PR[i] - oldPR[i]) > tol )
    PR[i] = alpha + (1 - alpha) * inNbrs[i].map(j => oldPR[j] / outDeg
[j]).sum
}
```

- `alpha` is the random reset probability (typically 0.15)
- `inNbrs[i]` is the set of neighbors which link to `i`
- `outDeg[j]` is the out degree of vertex `j`



In [1]:

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx.GraphLoader
import org.apache.spark.sql.SparkSession
```

In [2]:

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/edges.txt")

// Run Dynamic PageRank
val pageRanks = graph.pageRank(0.0001).vertices

// Join the ranks with the usernames
val users = (sc.textFile("data/nodes.txt")
    .map(_.split(","))
    .map(l => (l(0).toLong, l(1))))

(users.join(pageRanks)
    .sortByKey()
    .map { case (id, (username, rank)) => s"${username} has rank ${rank}" }
    .collect
    .foreach(println))
```

```
user1 has rank 0.732547889234966
user2 has rank 0.4613793054033705
user3 has rank 1.435699152315453
user4 has rank 1.3703736530462107
```

Pregel API

Graphs are inherently recursive data structures: properties of vertices depend on properties of their neighbors which in turn depend on properties of their neighbors. As a consequence many important graph algorithms iteratively recompute the properties of each vertex until a fixed-point condition is reached.

A range of graph-parallel abstractions have been proposed to express these iterative algorithms.

In [3]:

```
import scala.reflect.ClassTag

def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED], numIter: Int = 20, res
etProb: Double = 0.15) {
    val outdegreeGraph: Graph[Double, Double] = graph
        // Associate the degree with each vertex
        .outerJoinVertices(graph.outDegrees) { (vid, vdata, deg) => deg.getOrEls
e(0) }
        // Set the weight on the edges based on the degree
        .mapTriplets( e => 1.0 / e.srcAttr )
        // Set the vertex attributes to the initial pagerank values
        .mapVertices( (id, attr) => 1.0 )
        .cache()

    outdegreeGraph.triplets
        .map(triplet => s"Message ${triplet.srcId} -> ${triplet.dstId} : ${tripl
et.attr}")
        .collect
        .foreach(println)


    def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
        resetProb + (1.0 - resetProb) * msgSum

    def sendMessage(edge: EdgeTriplet[Double, Double]) =
        Iterator((edge.dstId, edge.srcAttr * edge.attr))

    def messageCombiner(a: Double, b: Double): Double = a + b

    val initialMessage = 0.0

    val pageRankGraph = outdegreeGraph.pregel(initialMessage, numIter, activeDir
ection = EdgeDirection.Out)(
      vertexProgram, sendMessage, messageCombiner)

    pageRankGraph.vertices
        .sortByKey()
        .map(v => s"${v._1} has pagerank= ${v._2}")
        .collect
        .foreach {println}
}

val graph = GraphLoader.edgeListFile(sc, "data/edges.txt")
val outDegrees: VertexRDD[Int] = graph.outDegrees

outDegrees.sortByKey().map(v => s"Node ${v._1} is connected to ${v._2} nodes").c
ollect.foreach(println)
run(graph, 20, 0.15)
```

```
Node 1 is connected to 2 nodes
Node 2 is connected to 1 nodes
Node 3 is connected to 1 nodes
Node 4 is connected to 2 nodes
Message 1 -> 2 : 0.5
Message 1 -> 3 : 0.5
Message 2 -> 3 : 1.0
Message 3 -> 4 : 1.0
Message 4 -> 1 : 0.5
Message 4 -> 3 : 0.5
1 has pagerank= 0.7084771144966524
2 has pagerank= 0.44930930552703563
3 has pagerank= 1.3878977176859082
4 has pagerank= 1.3225334566030538
```

Wiki Data

The file `wiki-Vote.txt` contains a tab-separated list of who `voted -> who got the vote` pairs. Those can be seen as the edges of the graph.

In [4]:

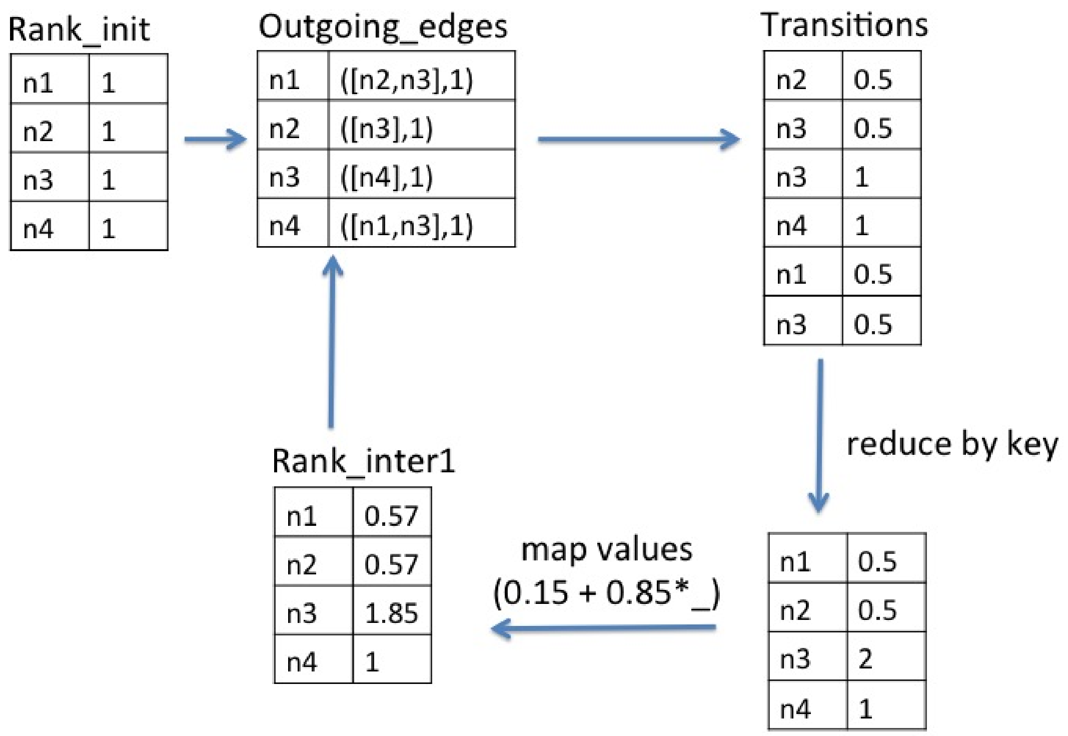
```
val wiki = GraphLoader.edgeListFile(sc, "data/wiki-Vote.txt")
println("Vertices (unique users): " + wiki.vertices.count())
println("Edges (votes cast): " + wiki.edges.count())
```

```
Vertices (unique users): 7115
Edges (votes cast): 103689
```

Iterative PageRank

Now you need to impelement the iterative version of pagerank using RDDs directly. The figure below show the steps you need to implement.

- **Initialization**:
  1. The rank of each node is initialized with `1.0`
  2. For each node, a list of the reachable nodes is computed
- **Iteration**: for as many times as needed:
  1. **Sending**: Each node *spreads* its `current rank` across all the nodes reachable from there (join between the current rank table and the reachable nodes table), in other words each connected node is sent `1 / # sender neighborhood` of the sender's `current rank`
  2. **Receiving**: Each node adds the values it has received by the other nodes (reduce by the message table receiver id)
  3. **Simulate Reset**: The `current rank` is set to be the weighted average between `1.0` and the previously computed sum



Initialization

Assign an initial rank of `1.0` to every page

In [5]:

```
val initialRanks = wiki.vertices.mapValues(v => 1.0)
initialRanks.map(v => s"${v._1} has rank ${v._2}").take(3).foreach(println)
```

```
4904 has rank 1.0
1084 has rank 1.0
7942 has rank 1.0
```

For each node, list the votes cast to other users

In [6]:

```
val votesCast = wiki.collectNeighborIds(EdgeDirection.Out).cache()
votesCast.map(v => v._1 + " voted for: " + v._2.mkString(",
")).take(3).foreach(println)
```

```
4904 voted for: 15, 4037
1084 voted for: 271, 338, 626, 1080, 2210
7942 voted for: 7021
```

Iteration

In [7]:

```
val messages = (votesCast
    .join(initialRanks)
    .flatMap {
        case (senderId, (receiverIds, rank)) => {
            // the sender id is retained only for clarity
            receiverIds.map(receiverId => (senderId, receiverId, rank / receiver
Ids.size))
        }
    })
messages.map(v => s"Message ${v._1} -> ${v._2}: ${v._3}").take(7).foreach(printl
n)
```

```
Message 4904 -> 15: 0.5
Message 4904 -> 4037: 0.5
Message 1084 -> 271: 0.2
Message 1084 -> 338: 0.2
Message 1084 -> 626: 0.2
Message 1084 -> 1080: 0.2
Message 1084 -> 2210: 0.2
```

In [8]:

```
val receivedMessageSum = (messages
    .map(v => (v._2, v._3))
    .reduceByKey(_+_))
receivedMessageSum.map(v => s"${v._1} received
${v._2}").take(3).foreach(println)
```

```
3456 received 9.107574739671758
6400 received 8.110271151778324
2354 received 7.873685268401263
```

In [9]:

```
var ranks = receivedMessageSum.mapValues(r => 0.15 + 0.85 * r)
ranks.map(v => s"${v._1} has rank ${v._2}").take(3).foreach(println)
```

```
3456 has rank 7.891438528720995
6400 has rank 7.043730479011575
2354 has rank 6.8426324781410734
```

In [10]:

```
for (i <- 1 to 20) {
  ranks = (votesCast
    .join(ranks)
    .flatMap {
        case (senderId, (receiverIds, rank)) => {
            receiverIds.map(receiverId => (receiverId, rank / receiverIds.size))
        }
    })
    .reduceByKey(_+_)
    .mapValues(r => 0.15 + 0.85 * r)
}
```

Results

Using the hand-made solution

In [11]:

```
ranks.map(_.swap).top(10).map(v => s"${v._2}\thas rank
${v._1}").foreach(println)
```

```
6634	has rank 2.8568613485821817
2625	has rank 2.6209719145033232
15	has rank 2.021597259314939
2398	has rank 1.8866008332445308
4037	has rank 1.8480070766999845
5412	has rank 1.7450392479347159
4335	has rank 1.6818711124420087
1297	has rank 1.5683586599765378
2066	has rank 1.4985393148691863
7553	has rank 1.4954999297505818
```

Using the built-in solution

In [12]:

```
val pageRanks = wiki.pageRank(0.0001).vertices
pageRanks.map(_.swap).top(10).map(v => s"${v._2}\thas rank ${v._1}").foreach(pri
ntln)
```

```
4037	has rank 32.78074239389385
15	has rank 26.18174657476919
6634	has rank 25.518550140728546
2625	has rank 23.361004685170897
2398	has rank 18.559437057563535
2470	has rank 17.957604768297593
2237	has rank 17.76401205997604
4191	has rank 16.135404511533686
7553	has rank 15.436932186579376
5254	has rank 15.297497713729927
```