

# CS236 Cloud Computing Project

---

KylinChen(陈麒麟), 517030910155, Fall Semester 2019

## 1 | 实验目的

---

- 熟悉云计算平台Spark原理，熟悉编程语言scala和编程环境IDEA。
- 学习使用开源图计算平台GraphX，了解Pagerank算法的实现流程与优化方式。

## 2 | 实验环境

---

- Operating System: Ubuntu 18.04.2 LTS
- Hadoop 3.2.1
- Scala 2.12.8
- Spark include GraphX 2.4.0
- IntelliJ 2019
- Sbt

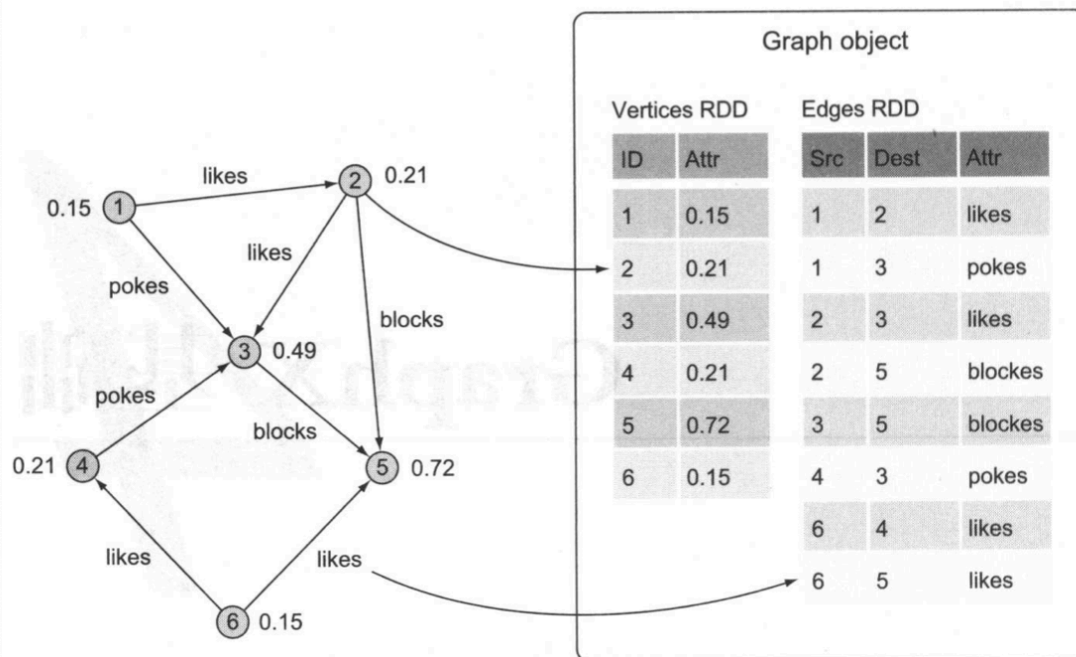
## 3 | 实验分析

---

### 实验1 GraphX API练习

- Spark GraphX API

Spark程序的基础模块是弹性分布式数据集 (RDD)，而图基础类Graph包含两个RDD：边RDD和顶点RDD。



edges ()和 vertices () 是 Graph.RDD对象的两个方法，分别调用图的边集合和顶点集合。另外 Graph.RDD 还提供了 triplets ()方法，返回ERE(Entity-Relation-Entity)类型的数据集合。

GraphX中提供方法的关键在于Map/Reduce。因为在很多图处理任务需要聚集从周围本地连接顶点发出的消息，比如计算图中顶点的入/出度、三角形数统计等。该技术可以通过 aggregateMessages() 方法实现，该方法的 Function prototype 如下：

```
def aggregateMessages[Msg] (
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,
    mergeMsg: (Msg, Msg) => Msg)
: VertexRDD[Msg]
```

sendMsg 以 EdgeContext 作为输入参数，提供 sendToSrc() 和 sendToDst() 两个方法，分别将 Msg消息发给源节点/目的节点。

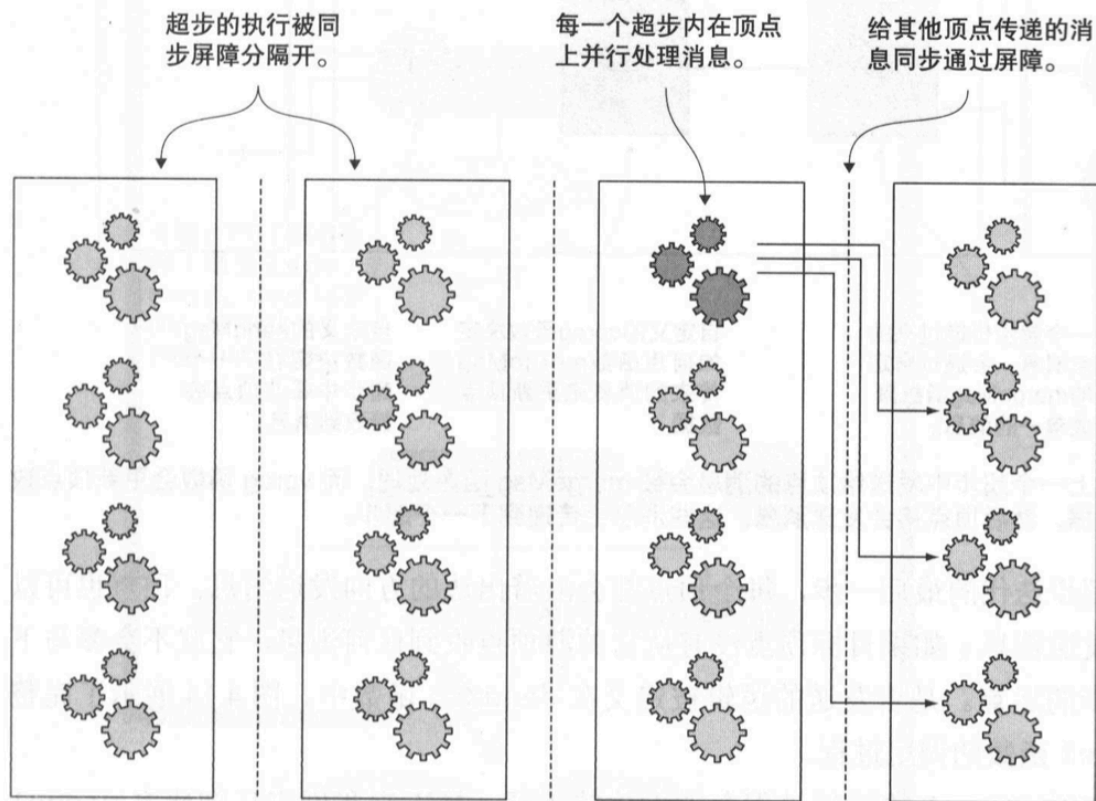
mergeMsg 以接收到的 sendMsg的信息为原数据，进行处理后生成一个 VertexRDD 数据体。

aggregateMessages() 方法保证了 GraphX 中数据的并行处理，只要多次重复/递归执行该方法，就可以构建常用的图算法。

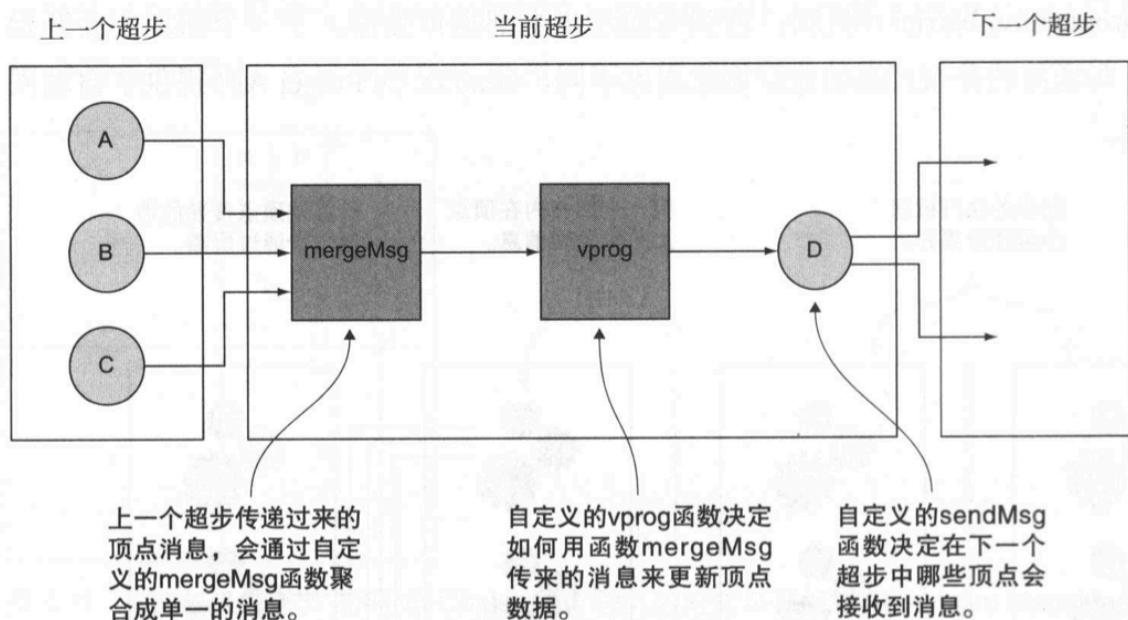
- Pregel API

Pregel API 是 GraphX 基于谷歌 Pregel 的迭代算法，与多次重复执行构建算法的 aggregateMessages() 方法不同，Pregel API 的算法执行只需要一次调用。而且相比基于 aggregateMessages() 的算法设计，它无需考虑数据集缓存(caching)和释放缓存(uncaching)操作来提升性能。

Pregel API利用整体同步并行计算模型(Bulk Synchronous Parallel, BSP)的理念。如图，算法被实现细节分为一系列超步(superstep)，即进行一系列相互隔离、存在mutex的迭代。



相对于 `aggregateMessages()` 方法，Pregel API 无需具体编写每个节点间的信息传递模型，开发者只需负责每个超步的处理机制的编写。而超步之间由于同步屏障(Synchronization Barrier)机制的存在，保证了超步之间是可交换、可替代的，所以也无需关心超步的迭代问题。其具体机制如下图所示，其中的 `mergeMsg()`、`vprog()`、`sendMsg()` 就是 Pregel API 中重要的三个参数。



Pregel 函数的 Prototype 如下所示，相比于 `aggregateMessages()` 返回一个 `VertexRDD` 对象，`Pregel()` 直接返回一个 `graph` 对象，可见它是一个更为高层的封装函数。

```
def pregel[A]
  (initialMsg: A,
   maxIter: Int = Int.MaxValue,
   activeDir: EdgeDirection = EdgeDirection.Out)
  (vprog: (VertexId, VD, A) => VD,
   sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
   mergeMsg: (A, A) => A)
  : Graph[VD, ED]
```

对参数列表进行简单的解释：

initialMsg表示迭代的初始值，即初始信息。maxIter定义了超步的最大数量(最大迭代次数)，防止不收敛情况的循环运行。EdgeDirection定义了点之间信息的传播方向。

在第二个参数表内，sendMsg 与 mergeMsg 的功能与 aggregateMessages() 一致，而增加的 vprog (vertex program) 是用来写每一个超步的附加逻辑的，比如如何更新 graph 的问题等。

有了这些背景知识之后，我们就可以来用Pregel完成SSSP (Single Source Shortest Path)的算法实现，由于发现 Pregel API 更适合使用 Dijkstra 算法计算单源最短路的几个原因：

- 1) 每个节点到 Source 的最短路是相对独立的步骤，即使一开始未达到最优，也可以不断进行迭代更新。适合于 Pregel 的特征。
- 2) Dijkstra 算法中需要对每一个节点的计算与之前的状态进行比较来获取最优，也就是说 vprog 正好给了这个比较逻辑的接口。

现在给出算法伪代码如下：

---

### Algorithm 1: Dijkstra's Algorithm

---

```
1 foreach  $u \in V$  do
2    $\text{INSERT}(Q, u)$ ;
3 while  $Q \neq \emptyset$  do
4    $u \leftarrow \text{EXTRACT-MIN}(Q)$ ;
5    $S \leftarrow S \cup \{u\}$ ;
6   foreach  $v \in \text{Adj}[u]$  do
7     if  $d[v] > d[u] + w(u, v)$  then
8        $d[v] \leftarrow d[u] + w(u, v)$ ; /* Relaxation Step */
9        $\text{DECREASE-KEY}(Q, v)$ ;
```

---

在第四部分，将给出 scala 代码实现细节。

## 实验2 使用 PageRank 解决 Wikipedia 投票选举问题

首先对 Pagerank 的算法做总结, 简单来说, Pagerank 算法就是利用链入节点的权重来不断更新当前节点权重的算法, 其最早用于 Google 的网页排名。

从算法上来说, 2003年改进后的Pagerank算法可以用一个表达式表示:

$$PageRank(p_i) = \frac{1-q}{N} + q \sum_{p_i} \frac{PageRank(p_i)}{L(p_i)}$$

公式对于每一个页面  $p_i$ , 在每一个轮次中进行pagerank值(即  $PageRank(p_i)$  值的更新)。公式中  $q$  是一个权重比, 用于控制每一次来自入度节点的更新程度, 即为一个调优参数(tuning parameter)。  $N$  是网络内所有节点的数量(在网页计算中, 即页面节点的数量)。  $L(p_i)$  表示页面节点的链出数量(即每个节点出度的大小), 这个值在静态图模型中是始终不变的。

决定结束时间的是另一个公式:

$$Count(|PageRank_{i-1} - PageRank_i| > Threshold) = 0$$

此处  $Count(Para)$  是统计满足  $Para$  条件的个数的函数,  $PageRank_i$  表示第  $i$  轮的 PageRank 矩阵, 即每一个页面的 PageRank 值组成的列向量。  $Threshold$  是人为设定的阈值, 表明 PageRank 的收敛范围。这个公式表明了 PageRank 的停止条件即为给一个 PageRank 值都处于稳定不变的状态( $Threshold$  的范围内)。

在 Wikipedia 投票选举问题中, 我们的目标在获得有向网络内最大的影响力的节点, 而这与 pagerank 的算法目标显然是一样的(最初的 Pagerank 算法目的在于获取节点影响力的排名作为 Google 搜索引擎在相同关联度下的链接排序)。

根据要求, 我们将从基础构建 PageRank 函数, 将使用到 `aggregateMessages()` 函数。在实验1中, 我们已经给出了 `aggregateMessages()` 的函数 Prototype 如下:

```
def aggregateMessages[Msg] (
  sendMsg: EdgeContext[VD, ED, Msg] => Unit,
  mergeMsg: (Msg, Msg) => Msg)
: VertexRDD[Msg]
```

容易知道 `aggregateMessages()` 的实质是在于节点间信息的发送和聚合, 那么在 Pagerank 的实现过程中, 我们可以把 `sendMsg` 过程定义为把当前节点的 Pagerank 权重传递给出度的节点, 而聚合过程作为利用调优参数(tuning parameter)和入度节点的权重进行当前节点 Pagerank 值的更新。而利用 Scala 脚本控制迭代结束(Pagerank 更新范围小于  $Threshold$ )。

在第四部分, 将给出 scala 代码实现细节。

## 4 | 实验过程

---

Caution: 项目中的代码均使用 SBT 构建（这一点会在第五部分具体说明），所以在 Scala 脚本中均使用的名为 helloworld 的 object，如要在自己的环境下运行，请首先把 scala 代码重命名为 helloworld.scala

## 实验1

- 利用GraphLoader.edgeListFile进行数据预处理，把txt数据文件转化为Vertex RDD 和 Edge RDD。

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.graphx.util.GraphGenerators

object helloworld {
  def main(args: Array[String]) = {
    val sc = new SparkContext(new
SparkConf().setMaster("local").setAppName("helloworld"))
    // A graph with edge attributes containing distances
    val graph = GraphLoader.edgeListFile(sc, "/home/hadoop/Documents/web-
Google.txt")
    graph.triplets.collect.foreach(a=>print(a+"\n"))
    // output triplets
  }
}
```

该代码利用GraphLoader.edgeListFile方法读入web-Google.txt(附件提供)，并利用 triplets 输出了 节点-边-节点(ERE) 模型。

- 使用Pregel实现SSSP(single source shortest path): 参考Pregel API, GraphOps

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.graphx.util.GraphGenerators

object helloworld {
  def main(args: Array[String]) = {
    val sc = new SparkContext(new
SparkConf().setMaster("local").setAppName("helloworld"))
    // A graph with edge attributes containing distances
```

```

val graph = GraphLoader.edgeListFile(sc,
"/home/hadoop/Documents/sssp_data.txt").mapEdges(e => e.attr.toDouble)
graph.edges.foreach(println)
val sourceId: VertexId = 0 // The ultimate source
val initialGraph : Graph[(Double, List[VertexId]), Double] =
graph.mapVertices((id, _) => if (id == sourceId) (0.0, List[VertexId]
(sourceId)) else (Double.PositiveInfinity, List[VertexId]()))
// pregel API
val sssp = initialGraph.pregel((Double.PositiveInfinity, List[VertexId]
()), Int.MaxValue, EdgeDirection.Out)(

// Vertex Program
(id, dist, newDist) => if (dist._1 < newDist._1) dist else newDist,

// Send Message
triplet => {
    if (triplet.srcAttr._1 < triplet.dstAttr._1 - triplet.attr ) {
        Iterator((triplet.dstId, (triplet.srcAttr._1 + triplet.attr ,
triplet.srcAttr._2 :+ triplet.dstId)))
    } else {
        Iterator.empty
    }
},
//Merge Message
(a, b) => if (a._1 < b._1) a else b)
println(sssp.vertices.collect.mkString("\n"))
}
}

```

在这里我们给出了一个10个节点的有向图文件 sssp\_data.txt (与web-Google.txt等数据文件格式相同):

```

# Directed graph (random generate by GraphX RandomGenerator): sssp_data.txt
# KylinChen, www.kylinchen.top, 2019
# Nodes: 10
# FromNodeId    ToNodeId
0 3
0 3
0 3
0 5
0 6
0 7
0 7
0 7
1 0
1 4
1 5
1 5
1 5
1 8
1 9
1 9
2 0
2 1
2 1
2 2
2 3

```



该图文件由 GraphX 的随机图生成器产生，所以可能会存在重复的边，但是我们只做为一条边读入，所以不影响实验结果。

代码的关键部分在于 Pregel API 参数的指定：

```
val sssp = initialGraph.pregel((Double.PositiveInfinity, List[VertexId]()),
    Int.MaxValue, EdgeDirection.Out)(

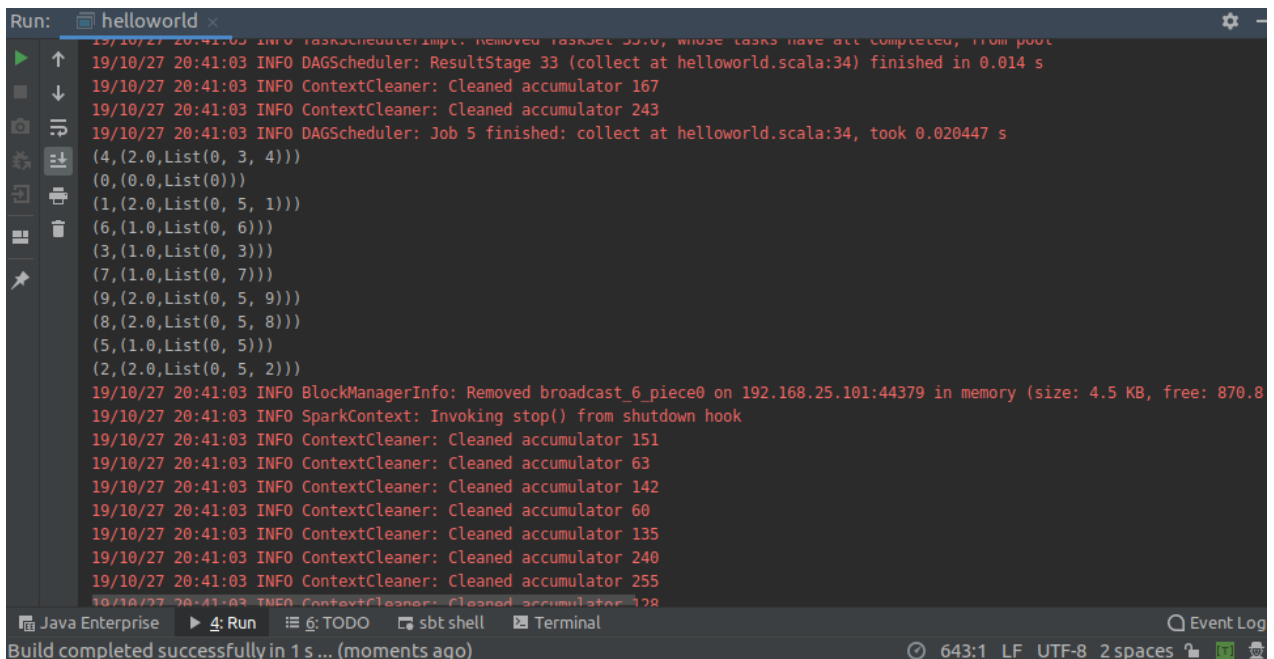
    // Vertex Program
    (id, dist, newDist) => if (dist._1 < newDist._1) dist else newDist,

    // Send Message
    triplet => {
        if (triplet.srcAttr._1 < triplet.dstAttr._1 - triplet.attr) {
            Iterator((triplet.dstId, (triplet.srcAttr._1 + triplet.attr,
triplet.srcAttr._2 :+ triplet.dstId)))
        } else {
            Iterator.empty
        }
    },
    //Merge Message
    (a, b) => if (a._1 < b._1) a else b)
```

对照第三部分给出的 Pregel 函数Prototype，我们可以发现

- 1) 第一个参数表指定了Msg始终为正浮点数(Double.PositiveInfinity)，最大迭代次数不加以限制(Int.MaxValue)，寻路方向始终为出度方向(EdgeDirection.Out)
- 2) 第二个参数表指定的 Dijkstra 算法中数据更新的方式，如比较上一次记录的最短路与这一次的值，并更新路径。

由于在代码中指定了0号节点为单源起始点(可以自行修改)，最终对于 sssp\_data.txt，我们可以得到如下输出：



```
Run: helloworld x
19/10/27 20:41:03 INFO DAGScheduler: ResultStage 33 (collect at helloworld.scala:34) finished in 0.014 s
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 167
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 243
19/10/27 20:41:03 INFO DAGScheduler: Job 5 finished: collect at helloworld.scala:34, took 0.020447 s
(4,(2.0,List(0, 3, 4)))
(0,(0.0,List(0)))
(1,(2.0,List(0, 5, 1)))
(6,(1.0,List(0, 6)))
(3,(1.0,List(0, 3)))
(7,(1.0,List(0, 7)))
(9,(2.0,List(0, 5, 9)))
(8,(2.0,List(0, 5, 8)))
(5,(1.0,List(0, 5)))
(2,(2.0,List(0, 5, 2)))
19/10/27 20:41:03 INFO BlockManagerInfo: Removed broadcast_6_piece0 on 192.168.25.101:44379 in memory (size: 4.5 KB, free: 870.8
19/10/27 20:41:03 INFO SparkContext: Invoking stop() from shutdown hook
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 151
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 63
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 142
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 60
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 135
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 240
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 255
19/10/27 20:41:03 INFO ContextCleaner: Cleaned accumulator 128
Java Enterprise 4: Run TODO sbt shell Terminal Event Log
Build completed successfully in 1 s ... (moments ago) 643:1 LF UTF-8 2 spaces
```



我们可以看到，对于每一个节点(0~9)，结果给出了0号节点到他们的最短距离与最短路径：

例如，(4,(2.0, List(0,3,4))) 表示0号节点到4号节点到最短距离为2.0，最短路径为0->3->4.

打开 sssp\_data.txt 利用python进行验证，结果准确。

## 实验2

- 使用aggregateMessages, org.apache.spark.graphx.lib 实现Pagerank，选出声望最高的20位候选人

按照第三部分的分析，声望最高可以用网络中的重要度衡量，即可以用 Pagerank 计算得出前20个高Pagerank权重的节点作为 top20 的候选人。

由于第三部分已经对aggregateMessages函数和Pagerank算法进行了深度解析，因此这里直接给出scala代码：

```
import java.io.PrintWriter

import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.graphx._

import scala.util.control.Breaks._
import org.apache.spark.rdd.RDD
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.graphx.util.GraphGenerators

object helloworld {
  def main(args: Array[String]) = {
    val sc = new SparkContext(new
SparkConf().setMaster("local").setAppName("helloworld"))
    // A graph with edge attributes containing distances
    var graph = GraphLoader.edgeListFile(sc, "/home/hadoop/Documents/Wiki-
Vote.txt").mapVertices((id, _) => 1.0)

    // some parameter for pagerank
    var q = 0.85
    var page_number = 7115
    var vertices = graph.vertices
    var threshold = 0.001
    var max_cycle = 50

    // some datastructure before iterate
    var outDegree = graph.aggregateMessages[Int](_.sendToSrc(1), _+_ )
    var rank = graph.vertices.mapValues( v => 1.00)
    var outable_rank = rank
```

```

// iterator section
breakable{
  for (i <- 0 until max_cycle) {
    var rank_previous = rank
    var rank_vertices = outDegree.join(rank)
    var rank_edges = graph.edges
    var rank_graph = Graph(rank_vertices, rank_edges)

    rank_vertices = rank_graph.vertices.mapValues(x=>{
      if(x==null){
        (0,1.0)
      }else{
        x
      }
    })
    rank_graph = Graph(rank_vertices, rank_edges)

    rank = rank_graph.aggregateMessages[Double](
      triplet => {
        triplet.sendToDst(triplet.srcAttr._2*(1/triplet.srcAttr._1))
      },
      _+_
    )
    rank = rank.mapValues((1-q)/page_number+q*_ )
    outable_rank = rank
    var none_indegree = vertices.mapValues(v=>1.00).minus(rank)
    var tmp_rank: VertexRDD[Double] = VertexRDD(none_indegree.union(rank))
    rank = tmp_rank

    // check when the pagerank stop
    var varice = rank_previous.join(rank).mapValues((x$2)=>{
      if(x$2._1-x$2._2<0){
        x$2._2-x$2._1
      }
      else{
        x$2._1-x$2._2
      }
    })
    if (varice.filter(_._2>threshold).count==0){
      println("terminate in "+i+" cycle.\n")
      break;
    }
  }
}

// sort the rank, output the result in out file and terminal
val result=outable_rank.sortBy(_._2, false)
println(result.take(20).mkString("\n"))
}

```

```
}
```

可以看到，我们首先使用了实验1的方法，读入了 Wiki-Vote.txt 文件。然后指定了Pagerank的几个重要参数(参考第三部分):

```
// some parameter for pagerank
var q = 0.85
var page_number = 7115
var vertices = graph.vertices
var threshold = 0.001
var max_cycle = 50
```

这部分可以根据需求改变，q指定了每次更新来自入度节点的权重，page\_number是读入数据节点的个数，threshold可以控制目标收敛域，max\_cycle指定了最大迭代数(不收敛也强制结束)，当然正常情况Pagerank会在10轮之内收敛，这也是该算法优越性之一。

使用了aggregateMessages 函数的主要是两个部分:

1) 计算图中每一个节点的出度(静态图是始终不变的):

```
var outDegree = graph.aggregateMessages[Int](_.sendToSrc(1), _+_)
```

2) 在迭代中传递权重(公式中  $\frac{PageRank(p_i)}{L(p_i)}$ ，参考第三部分Pagerank介绍)

```
rank = rank_graph.aggregateMessages[Double](
  triplet => {
    triplet.sendToDst(triplet.srcAttr._2*(1/triplet.srcAttr._1))
  },
  _+_
)
```

最后，对rank进行降序排序:

```
val result=outable_rank.sortBy(_._2, false)
```

我们可以在输出中得到排行前20的节点号以及其Pagerank值,如下:

```
Run: helloworld x
19/10/27 20:34:41 INFO DAGScheduler: Job 22 finished: take at helloworld.scala:16, took 0.179675 s
(4037,32.78602298921606)
(15,26.177077003395272)
(6634,25.50094071590467)
(2625,23.343561844283258)
(2398,18.550550208669968)
(2470,17.96612688865309)
(2237,17.76900867350178)
(4191,16.131803406845794)
(7553,15.430518449108009)
(5254,15.294830096487432)
(2328,14.501578422760804)
(1186,14.487095542181647)
(1297,13.834941819157882)
(4335,13.76737092260509)
(7620,13.741015139154745)
(5412,13.638757316196367)
(7632,13.566953471898199)
(4875,13.324175558455822)
(6946,12.857383143006583)
(3352,12.685298233522314)
19/10/27 20:34:41 INFO SparkContext: Invoking stop() from shutdown hook
19/10/27 20:34:41 INFO SparkUI: Stopped Spark web UI at http://192.168.25.101:4040
19/10/27 20:34:41 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
19/10/27 20:34:41 INFO MemoryStore: MemoryStore cleared
```

图为声望排行前20对候选人，另外，附件中的 pagerank\_all.txt 给出了所有节点的Pagerank值，可以用于验证

经对Wiki-Vote.txt的统计规律验证，此结果是符合预期的。

## 5 | 代码及环境安装

- 实验环境参考第二部分的环境包。经验证，所提供的环境包版本在 Ubuntu 18.04下是可以编译的。
- 实验利用IntelliJ内的SBT进行虚拟环境搭建，可以在IntelliJ内安装scala插件，现给出built.sbt文件(注意要把scala的版本修改为本地环境的)

```
name := "sbt1"
version := "0.1"

scalaVersion := "2.12.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "2.4.0"
```

- 部分学习代码放于 <https://github.com/KylinC/GraphX-In-Action.git>