Cloud Computing 实验报告

吕艺 517021910745

实验目的

1. 熟悉云计算平台Spark原理，熟悉编程语言scala和编程环境IDEA。
2. 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。

实验要求

实验1: 使用GraphX API，将源数据集数据抽象为图数据格式

实验2： 使用pagerank算法解决Wikipedia投票选举问题

实验环境要求

- JAVA开发环境
- hadoop， Spark
- GraphX

实验数据说明

- 数据集来源：**SNAP networks**，分成 `wiki-Vote.txt` 和 `web-Google.txt`
- 实验一中分别使用了 `wiki-Vote.txt` 和 `web-Google.txt`
- 实验二中使用了 `web-Google.txt`

实验流程

实验一 基于Pregel实现SSSP (single source shortest path)

1. 利用GraphLoader读入数据集，任意规定一个起始点，初始化每条边的值。在这里我用节点的属性来表示他们到起始点的距离。

```
// Create the graph with  wiki-Vote.txt
val graph: Graph[Int, Int] = GraphLoader.edgeListFile(sc,dataPath)
// The ultimate source
val sourceId: VertexId = 32
// Initialize the graph such that all vertices except the root have distance
infinity.
val initialGraph = graph.mapVertices((id, _) =>
  if (id == sourceId) 0.0 else Double.PositiveInfinity)
```

2. 基于Pregel实现单源最短路算法。在这里Pregel API有两个参数，第一个是节点属性的初始值，第二个参数是具体Message Forward的定义。

   Message Forward具体分为三个部分

   a. 取节点收到消息中距离最小的值，更新节点

   b. 发送消息，在这里如果到某节点的距离减小了，就将新的距离消息发出去，如果没有减少，就不发出去

   c. 整合消息，相当于Map Reduce中的Reduce，比较两条消息中的距离，取较小的一个

```
// Start forwarding the messages and calculate the  minumun distance
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
    (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
     triplet => {  // Send Message
      if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
          Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
        } else {
          Iterator.empty
        }
      },
      (a, b) => math.min(a, b) // Merge Message
)
```

3. 打印得到的距离

```
// Display Result
println(sssp.vertices.collect.mkString("\n"))
```

试验结果

显示了部分节点到节点32的值

```
.............
(7119,4.0)
(4409,Infinity)
(4561,4.0)
(875,3.0)
(2569,Infinity)
(7949,Infinity)
(6625,Infinity)
(2307,4.0)
(531,Infinity)
(1071,Infinity)
.............
```

实验二 使用PageRank解决Wikipedia投票选举问题

1. 根据数据集创建图

```
// Create the graph with  wiki-Vote.txt
val graph: Graph[Int, Int] = GraphLoader.edgeListFile(sc, dataPath)
```

2. 根据PageRank的定义，每个页面的得分取决于自身的基本分和进入该节点边的权值。而入度节点的权值等于边起始节点的分数除以该节点的出度数。在这里，我们假设每个页面的起始分数都是1。为了更方便的计算每条边的权重，我们先将每个点的属性设为该点的出度值，之后再将每条边的权值设为1/出度值，最后我们用每个顶点的属性代表它的得分。

```
// Create the graph with  wiki-Vote.txt
val graph: Graph[Int, Int] = GraphLoader.edgeListFile(sc, dataPath)

// Given a graph where the vertex property is the out degree
val inputGraph = graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) =>
degOpt.getOrElse(0))

// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
var pageRankGraph = inputGraph.mapTriplets(triplet => 1.0 /
triplet.srcAttr).mapVertices((id, _) => 1.0)
```

3. 迭代计算每个页面的取值, 这里设定的轮数为100轮

   在这里使用了aggregateMessage API，它有三个参数

   a. 计算发出去的消息，在这里发出去的消息是该节点的得分除以它的出度

   b. 将得到消息进行累加，整合消息

   c. 声明需要访问到原节点的哪部分参数，在这里设定的事原节点的得分

```
// Repeat PageRank calculation for 100 times
for (iteration <- 1 to 100) {
    pageRankGraph.cache()
    val rankUpdates = pageRankGraph.aggregateMessages[Double] (
        triplet => {
                triplet.sendToDst(triplet.attr *  triplet.srcAttr);
            },
            _ + _,
            TripletFields.Src  // Fields to access
          )

    // Update Graph with
    pageRankGraph = pageRankGraph.outerJoinVertices(rankUpdates) {
       (id, oldPageRank, msgSumUpdates) => (1 - q) / vNum + q *
msgSumUpdates.getOrElse(0.0)
    }

    // Display Progress Information
    println(s"PageRank Iterations $iteration.")
}
```

4. 打印出最受欢迎的20个页面

```
4037
15
6634
2625
2398
2470
2237
4191
7553
5254
2328
1186
1297
4335
7620
5412
7632
4875
6946
3352
```