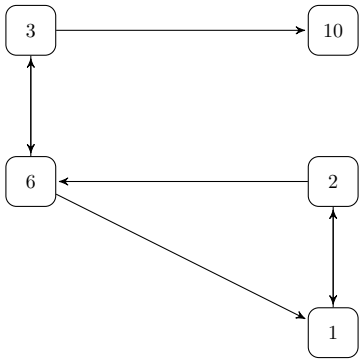
Ejercicios Scala

Instrucciones

Para simplificar su trabajo, puede escribir su código en un archivo de texto plano e invocarlo desde `spark-shell` usando `:load archivo.scala`. Para la entrega debe subir un documento en texto plano que contenga su código y una breve descripción de cada respuesta.

1. Ejercicio 1: Calcular valor mínimo con Pregel

En este ejercicio aplicarán los conceptos básicos de Pregel para calcular el valor mínimo entre los nodos de un grafo. Considere el siguiente grafo:



Representado de la siguiente manera en GraphX:

```
val vertices: RDD[(VertexId, Int)] =
sc.parallelize(Array(
    (1L, 3),
    (2L, 6),
    (3L, 2),
    (4L, 1),
    (5L, 10)
  )
)

val relationships: RDD[Edge[Boolean]] =
sc.parallelize(Array(
    Edge(1L, 2L, true),
    Edge(2L, 1L, true),
    Edge(2L, 4L, true),
    Edge(3L, 2L, true),
    Edge(3L, 4L, true),
    Edge(4L, 3L, true),
    Edge(1L, 5L, true)
  )
)

val graph = Graph(vertices, relationships)
```

Se pide que:

1. Use la función `mapVertices` para instanciar el grafo inicial. Este grafo debería mantener las aristas, pero a cada vértice debería ser agregado un valor auxiliar que represente el valor en la iteración anterior
2. Defina las funciones necesarias para que el grafo inicial pueda llamar a la función `pregel`.
   - `vprog`: guarda en el vértice el mínimo entre el valor actual y el valor recibido.
   - `sendMsg`: si está activo, enviar el valor a los vecinos; si no ha cambiado, entonces no debe enviar nada.
   - `mergeMsg`: de todos los valores recibidos, dejar el mínimo.
3. Ejecute la función `pregel` e imprima el grafo final.

Puede asumir que todos los nodos son menores que 9999, por lo que este valor puede ser su mensaje inicial.

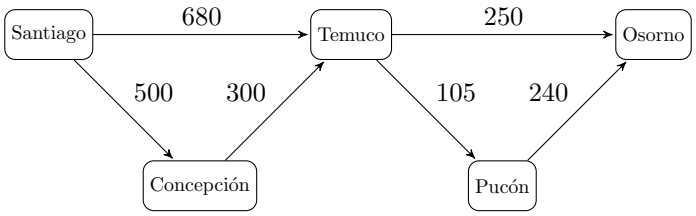
1.1. Solución

```
1  import org.apache.spark.graphx._
2  import org.apache.spark.rdd.RDD
3  import org.apache.spark.{SparkConf, SparkContext}
4  import scala.math._
5
6  object MinPregelExample extends App {
7
8    val conf = new SparkConf().setAppName("TestApp").setMaster("local")
9    val sc = new SparkContext(conf)
10
11   val vertices: RDD[(VertexId, Int)] =
12     sc.parallelize(Array(
13         (1L, 3),
14         (2L, 6),
15         (3L, 2),
16         (4L, 1),
17         (5L, 10)
18       )
19     )
20
21     val relationships: RDD[Edge[Boolean]] =
22     sc.parallelize(Array(
23         Edge(1L, 2L, true),
24         Edge(2L, 1L, true),
25         Edge(2L, 4L, true),
26         Edge(3L, 2L, true),
27         Edge(3L, 4L, true),
28         Edge(4L, 3L, true),
29         Edge(1L, 5L, true)
30       )
31     )
32
33     val graph = Graph(vertices, relationships)
34     val initial_graph = graph.mapVertices((id, attr) => (attr, -1))
35
36     initial_graph.vertices.collect.foreach(println)
37
38     val initialMsg = 9999
39
40     def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {
41       if (message == initialMsg) {
42           value
43       }
44       else {
45           (min(message, value._1), value._1)
46       }
47     }
48
49     def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]): Iterator[(VertexId, Int)] = {
50       val sourceVertex = triplet.srcAttr
51       if (sourceVertex._1 == sourceVertex._2) {
52           Iterator.empty
53       }
54       else {
```

```
55            Iterator((triplet.dstId, sourceVertex._1))
56          }
57        }
58
59        def mergeMsg(msg1: Int, msg2: Int): Int = min(msg1, msg2)
60
61        val maxGraph = initial_graph.pregel(initialMsg,
62          Int.MaxValue,
63          EdgeDirection.Out)(
64          vprog,
65          sendMsg,
66          mergeMsg)
67
68      maxGraph.vertices.collect.foreach(v => println(s"${v}"))
69    }
```

2. Ejercicio 2: Shortest path en Pregel

En este ejercicio calcularemos las rutas más cortas desde un nodo inicial en un grafo utilizando Pregel. Considere el siguiente grafo:



Representado de la siguiente manera en GraphX:

```
val vertexArray = Array(
  (1L, ("Santiago")),
  (2L, ("Concepción")),
  (3L, ("Temuco")),
  (4L, ("Pucón")),
  (5L, ("Osorno"))
)
val edgeArray = Array(
  Edge(1L, 2L, 500),
  Edge(1L, 3L, 680),
  Edge(2L, 3L, 300),
  Edge(3L, 4L, 105),
  Edge(3L, 5L, 250),
  Edge(4L, 5L, 240)
)
```

Se pide que genere un grafo inicial sobre el que debe ejecutar la función Pregel. Este grafo debe contener para cada ciudad la distancia mínima para llegar a cada nodo partiendo desde Santiago. Para esto debe:

1. Instanciar el grafo inicial. Este grafo mantiene las aristas, pero para cada vértice debe guardar una tupla con el nombre de la ciudad y el valor del costo acumulado para llegar al vértice. Para la ciudad inicial debe ser 0, y para todas las demás infinito.
2. Defina las funciones necesarias para que el grafo inicial pueda llamar a la función `pregel`.
   - `vprog`: guarda en el vértice el mínimo entre la distancia acumulada hasta ahora, o la recibida desde los otros vértices.
   - `sendMsg`: si su valor acumulado sumado con el valor de la arista es menor que el valor acumulado en el nodo de destino, entonces enviar el nuevo valor acumulado; en otro caso vaciar el iterador.
   - `mergeMsg`: de todas las distancias acumuladas enviadas, dejar el mínimo.
3. Ejecute la función `pregel` e imprima el grafo final.

Como ayuda adicional se mostrará la firma de los métodos de la función pregel:

```scala
def vprog(id: VertexId, attr: (String, Double), newDistance: Double): (String, Double) = {
  ...
}

def sendMsg(triplet: EdgeTriplet[(String, Double), Int]): Iterator[(VertexId, Double)] = {
  ...
}

def mergeMsg(a: Double, b: Double): Double = ...
```

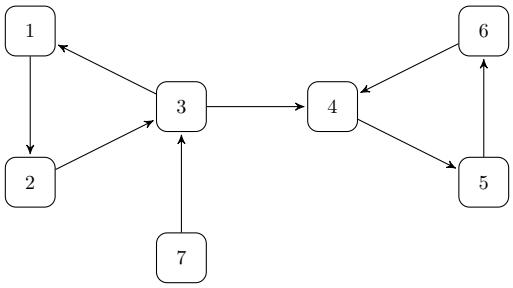
2.1. Solución

```scala
import org.apache.spark.graphx._
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD

object ShortestPathExample extends App {
  val vertexArray = Array(
    (1L, ("Santiago")),
    (2L, ("Concepción")),
    (3L, ("Temuco")),
    (4L, ("Pucón")),
    (5L, ("Osorno"))
  )
  val edgeArray = Array(
    Edge(1L, 2L, 500),
    Edge(1L, 3L, 680),
    Edge(2L, 3L, 300),
    Edge(3L, 4L, 105),
    Edge(3L, 5L, 250),
    Edge(4L, 5L, 240)
  )

  val conf = new SparkConf().setAppName("TestApp").setMaster("local")
  val sc = new SparkContext(conf)

  val vertexRDD: RDD[(Long, String)] = sc.parallelize(vertexArray)
  val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

  val graph: Graph[String, Int] = Graph(vertexRDD, edgeRDD)

  val initialNode: VertexId = 1
  val initialGraph = graph.mapVertices((id, attr) =>
    if (id == initialNode) {
      (attr, 0.0)
    } else {
      (attr, Double.PositiveInfinity)
    }
  )

  for (triplet <- initialGraph.triplets.collect) {
    println(s"${triplet.srcAttr} -" +
      s" ${triplet.attr} -> " + "" +
      s"${triplet.dstAttr}")
```

```
43    }
44
45    def vprog(id: VertexId, attr: (String, Double), newDistance: Double): (String, Double) = {
46      (attr._1, math.min(attr._2, newDistance))
47    }
48
49    def sendMsg(triplet: EdgeTriplet[(String, Double), Int]): Iterator[(VertexId, Double)] = {
50      if (triplet.srcAttr._2 + triplet.attr < triplet.dstAttr._2) {
51        Iterator((triplet.dstId, triplet.srcAttr._2 + triplet.attr))
52      } else {
53        Iterator.empty
54      }
55    }
56
57    def mergeMsg(a: Double, b: Double): Double = math.min(a, b)
58
59    val shortestPathGraph = initialGraph.pregel(Double.PositiveInfinity, Int.MaxValue, EdgeDirection
60      vprog,
61      sendMsg,
62      mergeMsg
63    )
64
65    for (triplet <- shortestPathGraph.triplets.collect) {
66      println(s"${triplet.srcAttr} -" +
67        s" ${triplet.attr} -> " + "" +
68        s"${triplet.dstAttr}")
69    }
70
71  }
```

3. Ejercicio 3: Contar triángulos con Pregel

En este ejercicio se contarán triángulos utilizando Pregel. Considere un grafo:



Representado de la siguiente manera en GraphX:

```
val vertexArray = Array(
  (1L, 1),
  (2L, 2),
  (3L, 3),
  (4L, 4),
  (5L, 5),
  (6L, 6),
  (7L, 7)
)
val edgeArray = Array(
  Edge(1L, 2L, true),
  Edge(2L, 3L, true),
  Edge(3L, 1L, true),
  Edge(3L, 4L, true),
  Edge(4L, 5L, true),
  Edge(5L, 6L, true),
  Edge(6L, 4L, true),
  Edge(7L, 3L, true)
)
```

Se pide que cuente el número de triángulos (ciclos conformado por exactamente tres aristas) en el grafo. Para esto se aconseja que calcule con Pregel todos los caminos de largo 3 que entran en cada nodo, y luego contar cuáles empiezan y terminan en el mismo lugar. Para esto debe:

1. Instanciar el grafo inicial. Este grafo mantiene las aristas, pero para cada vértice se le agrega una lista de tuplas de tres valores (`List((-1, -1, -1))`) y un valor adicional que cuenta las iteraciones, ya que el algoritmo se detiene después de dar tres saltos.
2. Cada nodo debe enviar la lista de tuplas que representan los caminos recorridos hasta ahora. Cada tupla va a representar un camino de largo 3, siendo el primer elemento el nodo desde donde parte el camino, el segundo elemento el nodo al que se llego en el primer salto y el tercer valor el nodo al que se llegó en el segundo salto.
3. El valor adicional debe ir representando la iteración en la que están los nodos. Se debe parar cuando se hayan guardado los caminos de largo 3 en las tuplas (esto pasa cuando los tres valores dejan de ser -1) y se haya enviado a los nodos que están a un tercer salto.
4. Al mezclar los mensajes se debe hacer un merge entre todas las listas recibidas. Este es el motivo por el que son listas, ya que en cada iteración me pueden llegar registros de varios caminos.

5. Finalmente, cada nodo debe comprobar cuantos elementos posee cuyo primer elemento sea igual a ellos mismos. Debemos contar todas estas apariciones y dividir el total por tres.

6. Imprimimos el valor anterior para conocer el número de triángulos.

3.1. Solución

3.2. Solución

```
1  import org.apache.spark.graphx._
2  import org.apache.spark.rdd.RDD
3  import org.apache.spark.{SparkConf, SparkContext}
4
5  object TrianglePregel extends App {
6    val vertexArray = Array(
7      (1L, 1),
8      (2L, 2),
9      (3L, 3),
10     (4L, 4),
11     (5L, 5),
12     (6L, 6),
13     (7L, 7)
14   )
15   val edgeArray = Array(
16     Edge(1L, 2L, true),
17     Edge(2L, 3L, true),
18     Edge(3L, 1L, true),
19     Edge(3L, 4L, true),
20     Edge(4L, 5L, true),
21     Edge(5L, 6L, true),
22     Edge(6L, 4L, true),
23     Edge(7L, 3L, true)
24   )
25
26   val conf = new SparkConf().setAppName("TestApp").setMaster("local")
27   val sc = new SparkContext(conf)
28
29   val vertexRDD: RDD[(Long, Int)] = sc.parallelize(vertexArray)
30   val edgeRDD: RDD[Edge[Boolean]] = sc.parallelize(edgeArray)
31
32   val graph: Graph[Int, Boolean] = Graph(vertexRDD, edgeRDD)
33
34   val initialGraph = graph.mapVertices((id, attr) =>
35     (attr, List((-1, -1, -1)), 1)
36   )
37
38   for (triplet <- initialGraph.triplets.collect) {
39     println(s"${triplet.srcAttr} -" +
40       s" ${triplet.attr} -> " + "" +
41       s"${triplet.dstAttr}")
42   }
43
44   def vprog(id: VertexId, attr: (Int, List[(Int, Int, Int)], Int),
45             neighbors: List[(Int, Int, Int)]): (Int, List[(Int, Int, Int)], Int) = {
46     if (neighbors.equals(List((-1, -1, -1)))) {
47       (attr._1, attr._2, attr._3)
```

```
48      } else {
49        (attr._1, neighbors, attr._3 + 1)
50      }
51    }
52
53    def sendMsg(triplet: EdgeTriplet[(Int, List[(Int, Int, Int)], Int), Boolean]):
54        Iterator[(VertexId, List[(Int, Int, Int)])] = {
55      var neighbors = List[(Int, Int, Int)]()
56      if (triplet.srcAttr._3 == 1) {
57        for(el <- triplet.srcAttr._2) {
58          neighbors = neighbors ++ List((triplet.srcAttr._1, el._2, el._3))
59        }
60        Iterator((triplet.dstId, neighbors))
61      } else if (triplet.srcAttr._3 == 2) {
62        for(el <- triplet.srcAttr._2) {
63          neighbors = neighbors ++ List((el._1, triplet.srcAttr._1, el._3))
64        }
65        Iterator((triplet.dstId, neighbors))
66      } else if (triplet.srcAttr._3 == 3) {
67        for(el <- triplet.srcAttr._2) {
68          neighbors = neighbors ++ List((el._1, el._2, triplet.srcAttr._1))
69        }
70        Iterator((triplet.dstId, neighbors))
71      } else {
72        Iterator.empty
73      }
74    }
75
76    def mergeMsg(a: List[(Int, Int, Int)], b: List[(Int, Int, Int)]): List[(Int, Int, Int)] = a ++ b
77
78    val triangleGraph = initialGraph.pregel(List((-1, -1, -1)), Int.MaxValue, EdgeDirection.Out)(
79      vprog,
80      sendMsg,
81      mergeMsg
82    )
83
84    for (triplet <- triangleGraph.triplets.collect) {
85      println(s"${triplet.srcAttr} -" +
86        s" ${triplet.attr} -> " + "" +
87        s"${triplet.dstAttr}")
88    }
89
90    var numberOfTriangles = 0
91    for(v <- triangleGraph.vertices.collect()) {
92      for (p <- v._2._2) {
93        if (p._1 == v._1) {
94          numberOfTriangles += 1
95        }
96      }
97    }
98
99    println(s"El número de triángulos es igual a ${numberOfTriangles / 3}")
100
101 }
```