Web Data Management Assignment 3 Writeup

Alex Simes (4415299) and Peter van Buul (1512269)

Introduction

For this assignment, we built two solutions to the problem of finding triangles inside graphs, one with Hadoop and one with Apache Giraph. For the first solution we used Apache Giraph (which uses Hadoop) to count triangles. Next, we used regular Hadoop map-reduce to solve the problem. Both the solutions are written in Java using maven to handle dependancies, with maven both solutions can be packaged in a single jar file. This file is used to call either solution.

In addition to these two solutions, we also built a script that automates the running of both solutions on a generated graph. The script also runs a brute-force php solution we implemented to check the output.

Apache Giraph

Brief Explanation

The Apache Giraph solution works in four steps during which messages are sent along the edges of a triangle.

1. In the first step, a message containing the ID of the sending vertex is sent along all edges where the neighboring vertex has a higher ID. If this vertex is part of a triangle, then this step is sending a message along the AB edge of the ABC triangle.
2. In the second step, this message is forwarded to again to all edges of which the neighboring vertex has an higher ID. If this vertex is part of the ABC triangle it now forwards this message along the BC edge.
3. In the third step the message is forwarded to all neighbors, if this vertex is part of the ABC triangle it now forwards this message along the BC edge.
4. In the last step, the IDs of the messages are compared with the ID of the current vertex. If the message ID matches the ID of the vertex, the message in this message has traveled along three edges in a triangle. Thus, a triangle is counted.

Pseudo Code

```
//step 1 send message over AB edge

foreach neighbor {
```

```
  if (neighborId > ownId) {
    send selfId to neighbor
  }
}

//step 2 forward message over BC edge
foreach message {
  foreach neighbor {
    if (neighborId > ownId) {
      send message to neighbor
    }
  }
}

//step 3 forward message over CA edge
foreach message {
  foreach neighbor {
    send message to neighbor
  }
}

//step 4 count number of triangles
numberOfTriangles = 0
foreach message {
```

```
  if (messagePayload == ownId) {
    numberOfTriangles++
  }
}

ownValue = numberOfTriangles
```

Hadoop Map-Reduce

Brief Explanation

The second algorithm that was implemented is a 3-way join using the map-reduce algorithm presented in the lecture series. The algorithm first sends the edges corresponding reducers, to determine which reducers a edge should be send to a hash is created based on the id's of the vertices in the edge and the number of buckets (in our case 3), this way triangles are stored in the same reducer. The reducers now determines for the received vertices if they can form triangles, if so a triangle is counted.

Pseudo Code

```
map {
  foreach edge startingFrom startVertex {
    if (startVertexId < neighborVertexId) {
      startVertexModulo = mod(startVertexId)
      neighborVertexModulo = mod(neighborVertexId)

      foreach bucketNumber from buckets {
        // send edge AB to reducer with hash(A, B, j)
        reducerNumber = hash(startVertexModulo) + hash(neighborVertexModulo) +
```

```
bucketNumber

        send(reducerNumber, edge(startVertexId, neighborVertexId, AB)



        // send edge BC to reducer with hash(j, B, C)

        reducerNumber = hash(bucketNumber) + hash(startVertexModulo) +
neighborVertexModulo

        send(reducerNumber, edge(startVertexId, neighborVertexId, BC)



        // send edge AC to reducer with hash(A, j, C)

        reducerNumber = hash(startVertexModulo) + hash(bucketNumber) +
neighborVertexModulo

        send(reducerNumber, edge(startVertexId, neighborVertexId, AC)

      }

    }

  }

}



reduce {

  numberOfTriangles = 0

  foreach edgeAB from recievedEdges {

    foreach edgeBC from recievedEdges {

      if (recievedEdgesOfTypeAC.contains(edge(edgeAB.A, edgeBC.C))) {

        numberOfTriangles++

      }

    }
```

```
    }


    reducerValue = numberOfTriangles
  }
```

Testing Script

Graph Generator and Solver

We built a graph generating php script that takes a number of nodes as argument and spits out a graph. Our generated graphs always have bi-directional connections. Meaning that if vertex 0 is connected to vertex 3, vertex 3 is connected to vertex 0. The generator can be called using the script generate_graph.sh and passing it a number of nodes.

The brute force solver was also implemented in PHP. This was used to check our solutions in map-reduce and Giraph. The script simply takes each edge in the graph (A,B) and sees if it can find any node C with edges (C,A) and (C,B).

Automation Script

The automation script takes a graph as input. The script resets the Hadoop filesystem and puts the given graph in the correct directories. It then calls both the Map-Reduce solution and the Giraph solution in turn through our jar file. Since these two programs don't output a plain number of triangles found, we used a simple php script to get a single number of triangles found out of the output. Finally, the script runs our brute force algorithm so we can be sure of the correctness.