Shortest Paths Example

Shortest Paths Example

We are going to delve into a full Giraph example using the single source shortest paths algorithm. The actual code is part of the examples included in Giraph SimpleShortestPathsVertex. We will also define a very simple input and output format for the graph.

Be aware of the version 0.1 or 0.2 - in the later one the class SimpleShortestPathsVertex does not have a main() method, so you have to run it via the ToolRunner.

First, let’s start with the VertexInputFormat. In this example, we have extended VertexInputFormat to produce our own simple JSON based input format. VertexInputFormat is very similar to the Hadoop InputFormat. The relevant code snippet is below:

```java
public static class SimpleShortestPathsVertexInputFormat extends
    TextVertexInputFormat<LongWritable, DoubleWritable, FloatWritable> {
  @Override
  public VertexReader<LongWritable, DoubleWritable, FloatWritable>
      createVertexReader(InputSplit split, TaskAttemptContext context)
      throws IOException {
    return new SimpleShortestPathsVertexReader(
        textInputFormat.createRecordReader(split, context));
  }
}

public static class SimpleShortestPathsVertexReader extends
    TextVertexReader<LongWritable, DoubleWritable, FloatWritable> {
  public SimpleShortestPathsVertexReader(
      RecordReader<LongWritable, Text> lineRecordReader) {
    super(lineRecordReader);
  }
  
  @Override
  public boolean next(MutableVertex<LongWritable, DoubleWritable, FloatWritable, ?> vertex)
      throws IOException, InterruptedException {
    Text line = getRecordReader().getCurrentValue();
    try {
      JSONArray jsonVertex = new JSONArray(line.toString());
      vertex.setVertexId(
          new LongWritable(jsonVertex.getLong(0)));
      vertex.setVertexValue(
          new DoubleWritable(jsonVertex.getDouble(1)));
      JSONArray jsonEdgeArray = jsonVertex.getJSONArray(2);
      for (int i = 0; i < jsonEdgeArray.length(); ++i) {
        JSONArray jsonEdge = jsonEdgeArray.getJSONArray(i);
        Edge<LongWritable, FloatWritable> edge =
            new Edge<LongWritable, FloatWritable>(
                new LongWritable(jsonEdge.getLong(0)),
                new FloatWritable((float) jsonEdge.getDouble(1)));
        vertex.addEdge(edge);
      }
    } catch (JSONException e) {
      throw new IllegalArgumentException(
          "next: Couldn't get vertex from line " + line, e);
    }
    return true;
  }
}
```
The idea is to split the graph into manageable parts to distribute across the Giraph workers. The first thing that happens is that `getSplits()` is called by the master and then the workers will process the InputSplit objects with the VertexReader to load their portion of the graph into memory. In this example, we use composition to internally use the TextVertexInputFormat to do most work for us of generating one split per input file in the directory. Then, the SimpleShortestPathsVertexReader can read line by line, one vertex per line. We implemented a very simple description of the graph, a json array that contains the vertex id, a vertex value, then a json array of edges (each of which are a json array of destination vertex id and edge value). For example, `[2,100,[3,200]]` specified the vertex id = 2, vertex value = 100, and there is a single edge (destination vertex 3 with edge value of 200). Also, one more thing to note is that we do load the vertex values but will override them in our application on superstep 0.

We also have to have an output format to store our graph. Hence we have the SimpleShortestPathsVertexOutputFormat.

```java
public static class SimpleShortestPathsVertexOutputFormat extends
    TextVertexOutputFormat<LongWritable, DoubleWritable,
    FloatWritable> {

    @Override
    public VertexWriter<LongWritable, DoubleWritable, FloatWritable>
    createVertexWriter(TaskAttemptContext context)
    throws IOException, InterruptedException {
        RecordWriter<Text, Text> recordWriter =
            textOutputFormat.getRecordWriter(context);
        return new SimpleShortestPathsVertexWriter(recordWriter);
    }
}

public static class SimpleShortestPathsVertexWriter extends
    TextVertexWriter<LongWritable, DoubleWritable, FloatWritable> {

    public SimpleShortestPathsVertexWriter(
        RecordWriter<Text, Text> lineRecordWriter) {
        super(lineRecordWriter);
    }

    @Override
    public void writeVertex(BasicVertex<LongWritable, DoubleWritable,
    FloatWritable, ?> vertex)
    throws IOException, InterruptedException {
        try {
            JSONArray jsonVertex = new JSONArray();
            jsonVertex.put(vertex.getVertexId().get());
            jsonVertex.put(vertex.getVertexValue().get());
            JSONArray jsonEdgeArray = new JSONArray();
            for (Edge<LongWritable, FloatWritable> edge :
                vertex.getOutEdgeMap().values()) {
                JSONArray jsonEdge = new JSONArray();
                jsonEdge.put(edge.getDestVertexId().get());
                jsonEdge.put(edge.getEdgeValue().get());
                jsonEdgeArray.put(jsonEdge);
            }
            jsonVertex.put(jsonEdgeArray);
        } catch (JSONException e) { throw new IllegalArgumentException(    
            "writeVertex: Couldn't write vertex ", vertex); }
    getRecordWriter().write(new Text(jsonVertex.toString()), null);
    }
}
```

The output format basically does the inverse of the input format, writing the vertices out as json arrays.

The next part of the code we focus on is the actual computation executed by every vertex:
private boolean isSource() {
    return (getVertexId().get() ==
            getContext().getConfiguration().getLong(SOURCE_ID,
            SOURCE_ID_DEFAULT));
}

@Override
public void compute(Iterator<DoubleWritable> msgIterator) {
    if (getSuperstep() == 0) {
        setVertexValue(new DoubleWritable(Double.MAX_VALUE));
    }
    double minDist = isSource() ? 0d : Double.MAX_VALUE;
    while (msgIterator.hasNext()) {
        minDist = Math.min(minDist, msgIterator.next().get());
    }
    if (LOG.isDebugEnabled()) {
        LOG.debug("Vertex " + getVertexId() + " got minDist = " + minDist +
                " vertex value = " + getVertexValue());
    }
    if (minDist < getVertexValue().get()) {
        setVertexValue(new DoubleWritable(minDist));
        for (Edge<LongWritable, FloatWritable> edge :
                getOutEdgeMap().values()) {
            if (LOG.isDebugEnabled()) {
                LOG.debug("Vertex " + getVertexId() + " sent to " +
                        edge.getDestVertexId() + " = " +
                        (minDist + edge.getEdgeValue().get()));
            }
            sendMsg(edge.getDestVertexId(),
                    new DoubleWritable(minDist +
                            edge.getEdgeValue().get()));
        }
    }
    voteToHalt();
}

In the superstep 0, all the vertices initialize their vertex values to the maximum value (unreachable). Then, the source vertex will propagate the cost of going to its neighbors. In subsequent supersteps, all vertices will propagate the minimum cost of getting to its neighbors until the application is complete. At this point, all the vertex values reflect the cost of going there from the source vertex.

The final bit of code to discuss here is starting up the job:
@Override
public int run(String[] argArray) throws Exception {
    if (argArray.length != 4) {
        throw new IllegalArgumentException(
            "run: Must have 4 arguments <input path> <output path> " +
            "<source vertex id> <# of workers>";
    }
    GiraphJob job = new GiraphJob(getConf(), getClass().getName());
    job.setVertexClass(getClass());
    job.setVertexInputFormatClass(SimpleShortestPathsVertexInputFormat.class);
    job.setVertexOutputFormatClass(SimpleShortestPathsVertexOutputFormat.class);
    FileInputFormat.addInputPath(job, new Path(argArray[0]));
    FileOutputFormat.setOutputPath(job, new Path(argArray[1]));
    job.getConfiguration().setLong(SimpleShortestPathsVertex.SOURCE_ID,
                                      Long.parseLong(argArray[2]));
    job.setWorkerConfiguration(Integer.parseInt(argArray[3]),
                                Integer.parseInt(argArray[3]),
                                100.0f);
    if (job.run(true) == true) {
        return 0;
    } else {
        return -1;
    }
}

public static void main(String[] args) throws Exception {
    System.exit(ToolRunner.run(new SimpleShortestPathsVertex(), args));
}

Implementing the Tool interface allows us to start Hadoop jobs from ToolRunner. The run method sets up the Giraph job with the appropriate vertex class (computation), vertex input / output formats, workers, etc. These configurations can also be set with generic Hadoop options as well. This is not the only way to start up a Hadoop job, but it’s a simple way.

Great! Now we need some input data. You can create your own, or you can download some example data from shortestPathsInputGraph. Untar the example data with the following command:

tar zxfv shortestPathsInputGraph.tar.gz
x shortestPathsInputGraph/
  x shortestPathsInputGraph/part-m-00001
  x shortestPathsInputGraph/part-m-00002
  x shortestPathsInputGraph/part-m-00003

Then upload the graph to HDFS:

hadoop fs -copyFromLocal shortestPathsInputGraph shortestPathsInputGraph

Running the example on a hadoop instance:
hadoop jar giraph-0.1-jar-with-dependencies.jar org.apache.giraph.examples.SimpleShortestPathsVertex shortestPathsInputGraph shortestPathsOutputGraph 0 3
11/08/02 12:44:40 INFO mapred.JobClient: Running job: job_201108021212_0026
11/08/02 12:44:41 INFO mapred.JobClient:  map 0% reduce 0%
11/08/02 12:44:58 INFO mapred.JobClient:  map 25% reduce 0%
11/08/02 12:45:04 INFO mapred.JobClient:  map 75% reduce 0%
11/08/02 12:45:10 INFO mapred.JobClient:  map 100% reduce 0%
11/08/02 12:45:15 INFO mapred.JobClient: Job complete: job_201108021212_0026
11/08/02 12:45:15 INFO mapred.JobClient: Counters: 41
11/08/02 12:45:15 INFO mapred.JobClient:  Map 0% Reduce 0%
11/08/02 12:45:15 INFO mapred.JobClient:  SLOTS_MILLIS_MAPS=70529
11/08/02 12:45:15 INFO mapred.JobClient:  Total time spent by all maps waiting after reserving slots (ms)=0
11/08/02 12:45:15 INFO mapred.JobClient:  Launched map tasks=4
11/08/02 12:45:15 INFO mapred.JobClient:  SLOTS_MILLIS_REDUCES=0
11/08/02 12:45:15 INFO mapred.JobClient:  Total time spent by all maps waiting after reserving slots (ms)=0
11/08/02 12:45:15 INFO mapred.JobClient:  Total time spent by all reduces waiting after reserving slots (ms)=0
11/08/02 12:45:15 INFO mapred.JobClient:  Setup (milliseconds)=6046
11/08/02 12:45:15 INFO mapred.JobClient:  Superstep 0 (milliseconds)=2104
11/08/02 12:45:15 INFO mapred.JobClient:  Shutdown (milliseconds)=73
11/08/02 12:45:15 INFO mapred.JobClient:  Aggregate finished vertices=15
11/08/02 12:45:15 INFO mapred.JobClient:  Aggregate vertices=15
11/08/02 12:45:15 INFO mapred.JobClient:  Aggregate edges=15
11/08/02 12:45:15 INFO mapred.JobClient:  Current workers=3
11/08/02 12:45:15 INFO mapred.JobClient:  Spilled Records=0
11/08/02 12:45:15 INFO mapred.JobClient:  Map input records=4
11/08/02 12:45:15 INFO mapred.JobClient:  Map output records=0
11/08/02 12:45:15 INFO mapred.JobClient:  GLUE_SYSTEM_COUNTERS=0
11/08/02 12:45:15 INFO mapred.JobClient:  Bytes Written=0
11/08/02 12:45:15 INFO mapred.JobClient:  File output format counters
11/08/02 12:45:15 INFO mapred.JobClient:  File System Counters
11/08/02 12:45:15 INFO mapred.JobClient:  FILE_BYTES_READ=354
11/08/02 12:45:15 INFO mapred.JobClient:  HDFS_BYTES_READ=465
11/08/02 12:45:15 INFO mapred.JobClient:  FILE_BYTES_WRITTEN=86688
11/08/02 12:45:15 INFO mapred.JobClient:  HDFS_BYTES_WRITTEN=8554
11/08/02 12:45:15 INFO mapred.JobClient:  Map-Reduce Framework
11/08/02 12:45:15 INFO mapred.JobClient:  Bytes Read=0
11/08/02 12:45:15 INFO mapred.JobClient:  Bytes Written=0
11/08/02 12:45:15 INFO mapred.JobClient:  Map input records=4
11/08/02 12:45:15 INFO mapred.JobClient:  Map output records=0
11/08/02 12:45:15 INFO mapred.JobClient:  SPLIT_RAW_BYTES=176

Now let’s check the results.
hadoop fs -ls shortestPathsOutputGraph
Found 5 items
-rw-r--r-- 1 aching supergroup 0 2011-08-02 13:20 /user/aching/shortestPathsOutputGraph/_SUCCESS
drwxr-xr-x - aching supergroup 0 2011-08-02 13:20 /user/aching/shortestPathsOutputGraph/_logs
-rw-r--r-- 1 aching supergroup 96 2011-08-02 13:20 /user/aching/shortestPathsOutputGraph/part-m-00001
-rw-r--r-- 1 aching supergroup 109 2011-08-02 13:20 /user/aching/shortestPathsOutputGraph/part-m-00002
-rw-r--r-- 1 aching supergroup 84 2011-08-02 13:20 /user/aching/shortestPathsOutputGraph/part-m-00003

hadoop fs -cat shortestPathsOutputGraph/part*

```json
[[5, 1000, [ [6, 500] ] ],
 [6, 1500, [ [7, 600] ] ],
 [7, 2100, [ [8, 700] ] ],
 [8, 2800, [ [9, 800] ] ],
 [9, 3600, [ [10, 900] ] ],
 [10, 4500, [ [11, 1000] ] ],
 [11, 5500, [ [12, 1100] ] ],
 [12, 6600, [ [13, 1200] ] ],
 [13, 7800, [ [14, 1300] ] ],
 [14, 9100, [ [0, 1400] ] ],
 [0, 0, [ [1, 0] ] ],
 [1, 0, [ [2, 100] ] ],
 [2, 100, [ [3, 200] ] ],
 [3, 300, [ [4, 300] ] ],
 [4, 600, [ [5, 400] ] ]]
```