Chapter 2

Graph Processing

2.1 Introduction to Graph Theory

A graph is a structure to represent relations between a collection of elements. An intuitive example would be to think about Facebook as a graph, where people are vertices (or nodes) and friendship relations are edges that connect them. In general, a graph G is identified by its vertex set V, and edge set E. Figure 1 shows a sample graph with four vertices and two edges.

Graphs can be directed or undirected depending on the relationships that the edges represent. You can think Facebook as a very large graph connection people across the world. A friendship connection indicates an undirected relation: if Alice and Bob are friends, this can be represented by a single edge that connects them. On the other hand, some graphs contain additional information about the direction of the relation. For example, a graph that represents the follower information on Twitter is directed. If Alice follows Bob on Twitter, that does not necessarily mean that Bob also follows Alice. An undirected graph can always be converted to a directed graph simply replacing each edge by two directed edges pointing to opposite directions.

A connected graph is a graph where there is at least one path between every pair of vertices. In other words, in a connected graph, it is possible to start at an arbitrary vertex and visit all vertices. In a disconnected graph, this is not possible. As an example, think about a graph that represents each state in the United States as a vertex, and the major freeways as edges that connect them. This graph is disconnected because there is no way to go from Hawai to any other state.

A weighted graph contains weights on its edges. This enables representing more complex information between the vertices. For example, the above mentioned graph can contain weights that indicate the speed limit in each

freeway as the weight. That would be useful for doing more complicated analysis such as finding the fastest route between two states.

2.2 Representing Graphs

There are two common ways to represent graphs in computer science. The first one is adjacency matrix representation. A graph having $|V|$ vertices is represented by a $|V|X|V|$ binary matrix A. If there is an edge with weight w from vertex i to vertex j, then $A[i][j] = w$, else $A[i][j] = 0$. In unweighted graphs, we replace w with 1. Notice that the adjacency matrix representation of undirected graphs results in symmetric matrices. Figure 2 shows a sample graph and its adjacency matrix representation.

Adjacency matrices can be stored in a two dimensional array in computer programs. They enable constant time access to edge queries. However, this is not appropriate for sparse matrices where there are too many vertices loosely connected by a very small number of edges. Such a matrix will result in a very large structure having most of its entries equal to 0. Adjaceny list representation solves this problem by simply listing all neighbors of a node in a list. This only requires storing the actual connections between the vertices and enables saving space, which is more desirable especially for sparse graphs. Figure 3 shows an adjacency list representation of the same sample graph in the previous example.

2.3 Shortest Path Problem

Finding shortest paths between nodes in a graph has been an interesting and common problem studied well in computer science. The most basic form of this problem is to find the length of the shortest path from a given node to all other nodes in the graph. This is known as the single source shortest path problem. A more complex problem asks for finding the lengths of the shortest paths between all pairs of nodes in a graph.

A popular method to solve the single source shortest path problem is to use Dijkstra's Algorithm [ref], which has $O(|V|2)$ time complexity. This can be improved to run in $O(|E|+|V|log|V|)$ using Fibonacci heaps [ref]. All pairs shortest path problem can be solved using Floyd-Warshall Algorithm [ref] with $O(|V|3)$ time complexity. Although these algorithms are outside the scope of this text, we strongly urge curious readers to read the cited references for detailed descriptions and mathematical analysis of these methods.

We now turn to the question of solving the single source shortest path

problem in large graphs in parallel. For simplicity, we assume the graph is non-weighted, but the algorithms covered can easily be extended to operate on weighted graphs. Computing the single source shortest paths in large graphs that do not fit in the memory of a single computer is not feasible using traditional algorithms mentioned above, because they utilize special data structures such as priority queues and associative arrays that are global. We will instead partition the graph and assume each node can be processed independently and concurrently, following a breadth first traversal approach.

The key idea is to propagate the information from a given node to its neighbors at each step using what we know about the graph so far. Initially, all we know is the distance of the source to itself, which is 0. Using this information, we can immediately update the distances of its neighbors to 1 at the next step. Then, we can update their neighbors and continue in this fashion until no more updates can be performed, which implies the algorithm converged.

In general, suppose the shortest distance from the source to an arbitrary node v_x at step i is given by the distance function $d_i(v_x)$. Assume v_x has in-degree equal to k meaning it has k in-neighbors, v_{x1} to v_{xk}. Figure 4 summarizes the graph representation of these definitions:

The length of the shortest path from the source to v_x at step $i+1$ is then given by:

$$d_{i+1}(v_x) = min[d_i(v_x), min_{j=1,k}[d_i(v_{xj})] + 1] \tag{2.1}$$

That is, we first find the neighbor with the minimum distance and add 1 to it. This is the length of the shortest path that we can reach v_x through its neighbors, represented by $min_{j=1,k}[d_i(v_{xj})] + 1$. Next, we compare that distance with the current distance of v_x that we already know, namely $d_i(v_x)$ and pick the minimum of these. Figure 5 shows a step by step execution of this algorithm with A being the source node. Initially, the distance of A to itself is 0, and the distance of all other nodes to A is set to ∞. At step 1, we update the distances of A's neighbors B and C to 1, since they are 1 hop away from A. At the next step, we continue by updating the distances of D and E to 2 as there are edges from B to D, and C to E respectively. Notice that there is also an edge from B to A, but we do not update A's distance to 2 because we already have a known smaller distance, which is 0. At the final step, we update the distance of F to 3 since we can reach it through D. Again observe that there is also an edge from E to D, but we do not update D's distance because the currently known distance of D is 2, which is less than 3.

2.4 Shortest Path with MapReduce

The above algorithm can easily be parallelized as we can process different nodes in different machines at each step. Mappers send messages containing distance information and reducers can group together all messages targeting the same node, finally picking the minimum. Listing 1 shows the pseudo code for the mappers and reducers.

Algorithm 1 Shortest Path MapReduce

```
1: map(Key K, Value Node)
2: begin
3:   emit(Node.Id,Node)                    //metadata
4:   foreach out-neighbor n in Node.neighbors do
5:     emit(n.Id, Node.distance + 1)
6:   end for
7: end

8: reduce(Key Id, Value Messages[])
9: begin
10:   min ← ∞
11:   meta ← null
12:   foreach Message m in Messages[] do
13:     if m.IsMetadata then
14:       meta ← m
15:     else if m.distance < min then
16:           min ← m.distance
17:         end if
18:     end if
19:   end for
20:   if min < meta.distance then
21:     meta.distance ← min
22:   end if
23:   emit(Id, meta)
24: end
```

Each mapper gets a node metadata which contains the id of the node, its current distance to the source and a list of its out-neighbors. Initially, the current distance of all nodes equals to ∞ except the source. Mappers emit two kinds of messages. The first message is the metadata, which is emitted and passed to the reducer without any changes. The key is the **Id** of the node whose metadata is being emitted. The other messages contain the distance

information. The **foreach** loop at line 4 goes through all neighbors of the node and emits the length of the path from the source to each neighbor passing through itself. Notice that in this case the key is the **Id** of the neighbor that we are emitting a distance value for.

Each reducers receive a metadata and a list of distances for a given node. The reduce function checks the type of the message and saves it if it contains the metadata information. Otherwise, it compares the distance value with the minimum distance observed so far (**min**), and updates **min** if necessary. Observe that a final check is done at line 20, which compares **min** with the distance that comes from the metadata. If **min** is smaller, this indicates that we discovered a shorter path so the metadata is updated accordingly. Finally, the *new* metadata is emitted for the next iteration - potentially having a smaller distance to the source node.

Multiple iterations of this MapReduce program should be invoked on the uptaded metadata until no more updates are performed, which implies that all the shortest paths are discovered and the algorithm converged. We will discuss how to accomplish this efficiently in Hadoop, using a special feature that comes with the framework.

2.5 Hadoop Implementation

We start by describing the input format of the graph. Each line in the input file contains metadata information about a single node with each field being separated by a space character. Figure 6 shows the fields of a single record.

node id	source id	distance	# of neighbors	list of neighbors

The names of the fields are self explanatory, and the only *dynamic* field that is updated as the algorithm makes progress is *distance*. Initially, the distance of all nodes but the source are set to ∞. Figure 7 shows the initial input file for the graph in Figure 5. Observe that for each node, we only maintain a list of *out-neighbors* since the distance information propogates from a node to its out-going edges.

A	A	0	2	B C
B	A	∞	2	A D
C	A	∞	1	E
D	A	∞	1	F
E	A	∞	1	D

There are two important remarks regarding the initial input file. The first one is the inclusion of the source node in each record. Since this is a single source shortest path algorithm, we do not need to include the source node and it might be a good idea to save some space by omitting it. However, we will extend this algorithm to work with multiple sources and make use of the *source* field later in this chapter.

The second one is how we choose to deal with *sink* nodes. A sink node is a node with no out-going edges. Observe that node **F** is a sink node in the graph but we do not have any metadata information for it in the initial input file since there is no immediate need to create metadata for this node. Metadata for sink nodes are only created on demand - when they are discovered the first time. We will explain how to handle sink nodes in more detail as we walk through the implementation of the reducer.

Since this is an iterative MapReduce algorithm, the output of one job goes as input to the next one. Thus, the input and output file formats must be the same to achieve consistent program execution. Figure 8 shows the outputs of the first and last jobs respectively. Notice how the metadata for **F** is created at the final step when it is discovered via the edge from **D**.

A	A	0	2	B C
B	A	1	2	A D
C	A	1	1	E
D	A	∞	1	F
E	A	∞	1	D

A	A	0	2	B C
B	A	1	2	A D
C	A	1	1	E
D	A	2	1	F
E	A	2	1	D
F	A	3	0	

2.5.1 Parsing the Input

We use **org.apache.hadoop.mapreduce.lib.input.TextInputFormat** and **org.apache.hadoop.mapreduce.lib.output.TextOutputFormat** classes which are readily available from the framework. Input to mapper is a single line of text containing metadata for a single node. Lines are parsed and a **Node** object is created for each record as shown in listing 2.5.1.

Listing 2.1: Node.java

```
public class Node {

    public long id;
    public long source;
    public float distance;
    public int degree;
```

```
    public ArrayList<Long> neighbors;

    public Node(String record) {

        String[] tokens = record.split("\\s+");
        id = Long.parseLong(tokens[0]);
        source = Long.parseLong(tokens[1]);
        try {
            distance = Float.parseFloat(tokens[2]);
        } catch(NumberFormatException e) {
            distance = Float.POSITIVE_INFINITY;
        }
        degree = Integer.parseInt(tokens[3]); // assuming
           degree is small, we use an int data type
        // extract the neighbors
        neighbors = new ArrayList<Long>();
        int neighborStart = 4;
        for(int i = 0; i < degree; i++) {
            neighbors.add(Long.parseLong(tokens[neighborStart+
               i]));
        }
    }

    public String getNeighbors() {
        return convertToString(neighbors);
    }

    private static String convertToString(ArrayList<Long> list
       ) {
        StringBuilder buf = new StringBuilder();
        for(int i=0; i<list.size(); i++) {
            buf.append(" ");
            buf.append(list.get(i));
        }
        return buf.toString();
    }
}
```

The class **Node.java** has public members *id*, *source*, *distance*, *degree* and *neighbors* which correspond to fields in the input file. As this implementation aims to process very large graphs, the nodes are represented by the *long* data type. The *distance* field is represented by a *float* to cover both unweighted and weighted graphs with positive non-integer weights. The average out-degree of a node in the graph is assumed to be relatively small and thus, represented by an *integer*. Depending on the size of your graph, you can change these data types to increase memory utilization.

The constructor takes a line of text containing the record and splits at white spaces. Each token in the record string is converted to a primitive data type (long, float or int) and assigned as a class member. The utility method *convertToString()* is used to convert a list of long integers to a string sequence, which is utilized by *getNeighbors()*.

2.5.2 Message Objects

In section 2.4 we gave the pseudo code for the shortest path algorithm and mentioned two types of messages that are passed from mappers to reducers. We represent these messages with an abstract class defined in **Message.java** that implements **org.apache.hadoop.io.Writable** given in listing 2.5.2.

Listing 2.2: Message.java

```
public abstract class Message implements Writable {

  public Message() { }

  public void writeList(ArrayList<Long> list, DataOutput out)
    throws IOException{

    WritableUtils.writeVInt(out, list.size());
    for (long n : list) {
      WritableUtils.writeVLong(out,n);
    }
  }

  public ArrayList<Long> readList(DataInput in) throws
     IOException{

    int len = WritableUtils.readVInt(in);
    ArrayList<Long> list = new ArrayList<Long>(len);

    for (int i = 0; i < len; i++) {
      list.add(WritableUtils.readVLong(in));
    }
    return list;
  }

  public ArrayList<Long> copyList(ArrayList<Long> list) {
    ArrayList<Long> newList = new ArrayList<Long>(list.size())
       ;
    for(long n : list) {
      newList.add(n);
    }
    return newList;
  }
```

```
    public abstract void readFields(DataInput in) throws
        IOException;
    public abstract void write(DataOutput out) throws
        IOException;

  }
```

This class uses **org.apache.hadoop.io.WritableUtils** to serialize and deserialize variable length integers and longs to save disk space. The intuition is to represent smaller numbers with less bytes and larger numbers with more, so for instance the node id's 1 and 1000000 do not occupy the same amount of space when they go in the wire.

There is a default constructor and three utility methods: *writeList()* serializes a list of longs into a data output stream, whereas *readList()* does the reverse operation by reading it from a data input stream. The *copyList()* method creates a duplicate of a list of longs. The last two abstract methods *readFileds()* and *write()* are part of the **Writable** interface and child classes that extend **Message** must implement them to enable serialization.

Next, we define a child class to be used in the shortest path algorithm which extends **Message**. Listing 2.5.2 shows the implementation of **MessageSP.java**.

Listing 2.3: MessageSP.java

```
public class MessageSP extends Message {

  long source;
  float distance;
  int degree;
  ArrayList<Long> neighbors;
  boolean isMetadata;

  public MessageSP() {
    super();
  }

  public MessageSP(long source, float distance, int degree,
      ArrayList<Long> neighbors) {

    this.isMetadata = true;
    this.source = source;
    this.distance = distance;
    this.degree = degree;
    this.neighbors = neighbors; // This is a shallow copy.
  }
```

```
    public MessageSP(float distance) {
      this.isMetadata = false;
      this.distance = distance;
    }

    public MessageSP createCopy() {
      MessageSP c = new MessageSP();
      c.isMetadata = isMetadata;
      if (isMetadata) {
        c.source = source;
        c.degree = degree;
        c.neighbors = copyList(neighbors);
      }
      c.distance = distance;
      return c;
    }

    public void write(DataOutput out) throws IOException {

      out.writeBoolean(isMetadata);

      if (isMetadata) {
        WritableUtils.writeVLong(out, source);
        out.writeFloat(distance);
        WritableUtils.writeVInt(out, degree);
        writeList(neighbors, out);
      } else {
        out.writeFloat(distance);
      }
    }

    public void readFields(DataInput in) throws IOException {

      isMetadata = in.readBoolean();

      if (isMetadata) {
        source = WritableUtils.readVLong(in);
        distance = in.readFloat();
        degree = WritableUtils.readVInt(in);
        neighbors = readList(in);
      } else {
        distance = in.readFloat();
      }
    }
  }
```

MessageSP has five private members: *source*, *distance*, *degree* and *neighbors* are self explanatory and correspond to the fields in Figure 6 whereas *isMetadata* is used to distinguish between two different message types: metadata message and distance message.

The default constructor is empty and simply calls the constructor of its super class. The next constructor is used to create a metadata message that contains the most up-to-date information about a node. It initializes all class members and sets *isMetadata* to true. The last constructor is used to create a message that contains a *distance* value. Notice that it sets *isMetadata* to false.

The *createCopy()* method duplicates a **MessageSP** object by creating a deep copy of it. The last two methods *write()* and *readFields()* are used for object serialization.

Having explained the utility classes for input parsing and messages, we now start describing the main components of the implementation.

2.5.3 Mapper

Listing 2.5.3 shows the code for the mapper. The input value type is **org.apache.hadoop.io.Text** and it contains a single line from the input file. The output $< key, value >$ types are **org.apache.hadoop.io.LongWritable** and **MessageSP** respectively.

Listing 2.4: MapperClass.java

```
public class MapClass extends Mapper<LongWritable, Text,
    LongWritable, MessageSP> {

  LongWritable key = new LongWritable();
  MessageSP value = null;

  public void map(LongWritable K, Text V, Context context)
    throws IOException, InterruptedException {

    String record = V.toString();
    Node N = new Node(record);
    key.set(N.id);

    // emit metadata
    value = new MessageSP(N.source, N.distance, N.degree, N.
       neighbors);
    context.write(key, value);

    // for each neighbor emit <neighbor, dist+1> as the
       distance from root to neighbor
    value = new MessageSP(N.distance + 1);
```

```
      for (int i = 0; i < N.degree; i++) {
        key.set(N.neighbors.get(i));
        context.write(key, value);
      }
    }
  }
```

A **Node** object is created by the single parameter constructor which parses the line of text, and the key is set to the id of the node that is currently being processed - **N.id**. The mapper first emits the metadata for the node by creating a new **MessageSP** object with the relevant parameters. Next, a simple for loop goes through all neighbors and propogates the distance information. In this case, the key is set to the id of the neighbor that the message is aimed at - **N.neighbors.get(i)** - and the value is set to contain the distance of the current node we are processing *plus one* for the edge that goes from this node to its neighbor. Notice that we create the **MessageSP** object containing distance before the for loop and emit it multiple times, ie. once for each neighbor.

2.5.4 Reducer

The reducer implementation is long but fairly straightforward. There are a few important considerations required for handling the sink nodes and updating the metadata. The code for the reducer is given in listing 2.5.4.

Listing 2.5: ReducerClass.java

```
public class ReduceClass extends Reducer<LongWritable,
   MessageSP,Text,NullWritable>{

    public void reduce(LongWritable key, Iterable<MessageSP>
       values, Context context) throws
    IOException, InterruptedException {

        MessageSP meta = null;
        float minDist = Float.POSITIVE_INFINITY;

        for(MessageSP M : values) {
            if(M.isMetadata) {
                meta = (MessageSP) M.createCopy();
            } else {
                if(M.distance < minDist) {
                    minDist = M.distance;
                }
            }
        }
```

```
            String result = null;
            if(meta != null)
            {
                if(minDist < meta.distance)
                {
                    // Update the metadata.
                    meta.distance = minDist;
                    context.getCounter(Records.UPDATED).increment
                       (1);
                }
                result = String.format("%s %s %s %s%s",
                        key.get(), meta.source, meta.distance,
                           meta.degree, convToStr(meta.neighbors)
                           );
            } else {
                // This is a sink node.
                result = String.format("%s %s %s %s%s", key.get(),
                    "-1", minDist, 0, "");
            }

            context.write(new Text(result), NullWritable.get());
        }

      /**
       * Converts an array list to string representation.
       */
      private String convToStr(ArrayList<Long> list) {
        StringBuilder buf = new StringBuilder();
        for (int i = 0; i < list.size(); i++) {
          buf.append(" ");
          buf.append(list.get(i));
        }
        return buf.toString();
      }
    }
```

ReduceClass has a private utility method called *convToStr()* which converts a list of longs to a string sequence. The output $< key, value >$ types for the reducer are **org.apache.hadoop.io.Text** and **org.apache.hadoop.io.NullWritable**. Output records are stored inside the key and the value is null. The for loop in the reducer iterates through all the messages targeted for the current node specified by the key. If the message contains metadata, it is saved in the **meta** variable. Otherwise, the distance is compared to the minimum distance observed so far and **minDist** is updated if necessary.

The outer if clause checks whether there exists metadata for this node or not. In the former case, **minDist** is compared to the distance value in

the metadata and **meta.distance** is updated if **minDist** is smaller, indicating a shorter path has been discovered. Otherwise, the original value of **meta.distance** that comes from the input file is preserved.

If there is no metadata, this implies the reducer is processing a sink node with no out-going edges. Remember node **F** in the sample graph shown in Figure 5 is a sink node and there is no metadata for it in the initial input file given in Figure 7. When a sink node is discovered, the reducer creates new metadata with the following code:

```
result = String.format("%s %s %s %s%s",
key.get(), "-1", minDist, 0, "");
```

Note that a sink node has 0 neighbors represented by the empty string. We insert -1 at the *source* field to distinguish sink nodes in the final output file although this is not required for program correctness since this field is not used in the single source implementation.

The final *result* is converted to a **Text** object and emitted via *context.write()* at the end of the reduce method.

2.5.5 Counters

Notice that whenever the metadata is updated with a smaller distance, a global counter with the name $Records.UPDATED$ is incremented to keep track of the number of nodes whose metadata changed. Counters in Hadoop are of type *enum* and usually defined as *public* since different classes in a MapReduce program should be able to access them. We define a single counter in a separate file **Counters.Java** shown in listing **??**.

Listing 2.6: Counters.java

```
public enum Counters { UPDATED }
```

Counters are global, meaning any mapper or reducer can increment them. Local counter values coming from individual map or reduce tasks are automatically aggregated and summed up at the end of the job to collect statistical information. We will explain how to make use of *Counters.UPDATED* to decide whether or not the graph converged when we describe the main program that submits and monitors the MapReduce job.

2.5.6 Combiner

The reducer goes through all distance messages sent to a particular node and finds the minimum. This procedure is preceded by sending the messages from

mappers to reducers over the network, partitioning and sorting them. The number of messages that a node receives is equal to its in-degree, ie. number of edges pointing to it. Running this algorithm on very large input files such as the web graph [**?**] where some nodes have thousands of edges pointing to them might result in long execution times and suboptimal performance. We can optimize it by adding a combiner and performing a local reduction in each mapper to reduce the amount of information that goes through the network by orders of magnitue - depending on the structure of the graph. Listing 2.5.6 shows the implementation of the combiner.

Listing 2.7: CombineClass.java

```
public class CombineClass extends Reducer<LongWritable,
   MessageSP,LongWritable,MessageSP>{

    public void reduce(LongWritable key, Iterable<MessageSP>
       values, Context context) throws
    IOException, InterruptedException {

        float minDist = Float.POSITIVE_INFINITY;
        MessageSP result = null;

        for(MessageSP M : values) {
            if(M.isMetadata) {
                // send this to the reducer
                context.write(key, M);
            } else {
                if(M.distance < minDist) {
                    minDist = M.distance;
                    result = (MessageSP) M.createCopy();
                }
            }
        }

        if(result != null) {
            context.write(key, result);
        }
    }
}
```

The combiner iterates through all messages aiming a node in the graph, and keeps track of the minimum distance in **minDist**. If it receives message containing metadata, it simply emits it without making any changes. This optimization enables detecting *local minimum* distances that are potentially good candidates for the shortest path length in a given iteration.

2.5.7 ShortestPath - Driver Program

Single source shortest path in MapReduce is an iterative algorithm. At the end of the first run, we discover all nodes that are one *hop* away from the source. At the end of the second run, we discover all nodes that are two *hops* away from the source and this goes on until no more distance updates are performed which implies all shortest paths are known.

We perform each iteration by running a separate MapReduce job. Hadoop takes directory names as input and output locations for each job. In this implementation, the output directory of the first iteration becomes the input directory of the next one. Once all iterations complete, the output directory of the last job contains the shortest path lengths for the converged graph.

Listing 2.5.7 shows the main body of the driver program.

Listing 2.8: ShortestPath.java - Main body

```
public class ShortestPath {

  private static List<String> parseArguments(String args[])
  {

    List<String> argList = new ArrayList<String>();
    int numReducers = 1;

    for (int i = 0; i < args.length; ++i)
    {
      try {
        if ("-r".equals(args[i])) {
          numReducers = Integer.parseInt(args[++i]);
        } else {
          argList.add(args[i]);
        }
            } catch (NumberFormatException except) {
        System.out.println("ERROR: Integer expected instead of
            " + args[i]);
        printUsage();
      } catch (ArrayIndexOutOfBoundsException except) {
        System.out.println("ERROR: Required parameter missing
           from " + args[i - 1]);
        printUsage();
      }
    }

    argList.add(new Integer(numReducers).toString());
        return argList;
  }

  public static int main(String[] args) throws Exception
```

```
    {
      List<String> other_args = parseArguments(args);
      String inputPath = other_args.get(0);
      String outputPrefix = other_args.get(1);
      int numReducers = Integer.parseInt(other_args.get(2));

      String outDir = run(inputPath, outputPrefix, numReducers);
      return outDir == null ? -1 : 0;
    }

    static void printUsage()
    {
      System.out.println("ShortestPath [-r <reduces>] <input> <
          output>");
      System.exit(-1);
    }



    ...
    ...


  }
```

Three arguments are required to run the main program: *numReducers* specifies the number of reducers, *inputPath* contains the initial input file for the first iteration and *outputPrefix* specifies the common prefix that all jobs share as part of their output paths. The *parseArguments()* method takes program arguments specified by the user and returns them in a list. Inside the main body the run() method is called to start a series of iterations, finally returning the output of the last iteration which contains the results. Listing 2.5.7 shows the implementation of *run()*.

Listing 2.9: ShortestPath.java - run()

```
  public static String run(String inputDir, String outputPrefix,
      int numReducers) throws Exception {

    Date startTime = new Date();
    long exitCode = 0, itr = 0;
    String outputDir = "";

    while(true)
    {
      // set the output path for the current iteration
      outputDir = outputPrefix + "/" + itr;
      exitCode = RunIteration(inputDir, outputDir, numReducers);
```

```
      if(exitCode > 0)
      {
        // set the input path for the next iteration
        inputDir = outputDir;
        itr++;
      } else {
        break;
      }
    }

    if(exitCode == 0)
    {
      Date end_time = new Date();
      System.out.println("Graph Converged!");
      System.out.println("Number of iterations until convergence
          : " + itr);
      System.out.println("Total convergence time " + (end_time.
          getTime() - startTime.getTime()) /1000 + " seconds.");
      System.out.println("See the final output at " + outputDir)
          ;
      return outputDir;
    } else {
      System.out.println("ShortestPath failed!");
      return null;
    }
  }
```

The *run()* method runs iterations in a while loop until the graph converges. The *itr* variable keeps track of the iteration count and is initially set to 0. Inside the loop, *outputDir* is set to point to a new directory which will contain the output of the next job to be submitted by *runIteration()*.

When runIteration() submits the actual MapReduce job, program control blocks until the job completes. Its return value represents the number of nodes whose distances were updated in the last iteration. A return value of 0 indicates the graph converged and the while loop terminates. Any value that is greater than 0 means there are still be some nodes whose distances were just updated so we need to run at least one more iteration. In this case, *itr* is incremented and *inputDir* is set to point to the value of *outputDir* to ensure the next MapReduce job will work on the output of the previous one. A negative value implies there was an error with the last MapReduce job.

The code for *runIteration()* is given in listing 2.5.7.

Listing 2.10: ShortestPath.java - runIteration()

```
  private static long runIteration(String inputDir, String
      outputDir, int numReducers)  throws Exception
```

```
{
  Configuration conf = new Configuration();

  Job job = new Job(conf);
  job.setJobName("ShortestPath");
  job.setNumReduceTasks(numReducers);

  job.setMapOutputKeyClass(LongWritable.class);
  job.setMapOutputValueClass(MessageSP.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(NullWritable.class);

  job.setMapperClass(MapClass.class);
  job.setCombinerClass(CombineClass.class);
  job.setReducerClass(ReduceClass.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);
  TextInputFormat.setInputPaths(job, inputDir);
  TextOutputFormat.setOutputPath(job, new Path(outputDir));

  job.setJarByClass(ShortestPath.class);

  Date startTime = new Date();
  System.out.println("ShortestPath iteration started: " +
     startTime);
  System.out.println("Input path " + inputDir);
  System.out.println("Output path " + outputDir);
  int exitCode = job.waitForCompletion(true) ? 0 : -1;

  System.out.println("Records.UPDATED " + job.getCounters().
     findCounter(Counters.UPDATED).getValue());

  if( exitCode == 0)
  {
    Date end_time = new Date();
    System.out.println("\nShortestPath iteration ended: " +
       end_time);
    System.out.println("ShortestPath iteration took " + (
       end_time.getTime() - startTime.getTime()) /1000 + "
       seconds.");
    return job.getCounters().findCounter(Counters.UPDATED).
       getValue();
  } else {
    System.out.println("ShortestPath iteration failed!");
    return -1;
  }
}
```

A new instance of the class **org.apache.hadoop.mapreduce.Job** is created to specify the mapper, combiner and reducer classes along with the types of the output $< key, value >$ pairs and the number of reducers. When **job.waitforCompletion()** exits, the number of nodes whoes distances are updated in this iteration is returned by accesing the global counter via:

```
job.getCounters().findCounter(Counters.UPDATED).getValue()
```

In case of an error, the job does not succeed and the method returns an exit code of -1.

2.5.8 Running the program

2.5.9 TODO

TODO: Add a sub section to run the code on the sample graph, include the input file and command line arguments. Make sure you tell the readers that this algorithm works for a connected graph. Talk about how to extend it to weighted graphs. Write some stuff about how it is possible to optimize using binary objects rather than lines of text as input and output. Mentioan sample input files at Stanford's web site.

Bibliography