breadth-first graph search using an iterative map-reduce algorithm

posted june 18th, 2009 by cailin in hadoop, mapreduce, social-graph, tech

i've noticed two trending topics in the tech world today: social graph manipulation and map-reduce algorithms. in the last blog, i gave a quickie guide to setting up hadoop, an open-source map-reduce implementation and an example of how to use hive - a sql-like database layer on top of that. while this is one reasonable use of map-reduce, this time we'll explore it's more algorithmic uses, while taking a glimpse at both of these trendy topics!

for a great introduction to graph theory, start here. (note though - in code examples that follow i use the term "node" instead of "vertex"). these days, on of the most common uses of a graph is the "social graph" - e.g. your network of friends, as represented on a social network such as linkedin or facebook. one way to store a graph is using an adjacency list. in an adjacency list, each "node on the graph" (e.g. each person) is stored with a link to a list of the "edges emanating from that node" (e.g. their list of friends). for example :

```
frank -> {mary, jill}
jill -> {frank, bob, james}
mary -> {william, joe, erin}
```

or, in the machine world we might represent those people by their integer ids and wind up with something like

```
0-> {1, 2}
2-> {3, 4, 5}
1-> {6, 7, 8}
```

single threaded breadth-first search

one common task involving a social graph is to use it to construct a tree, starting from a particular node. e.g. to construct the tree of mary's friends and the mary's friends of friends, etc. the simplest way to do this is to perform what is called a breadth-first search. you can read all about that here. this reference also includes a pseudo-code implementation. below follows a Java implementation. (the Node class is a simple bean. you can download Node.java here and Graph.java here.)

```
package demo.algorithm.sort;

import java.util.*;

public class Graph {

  private Map<Integer, Node> nodes;

  public Graph() {
    this.nodes = new HashMap<Integer, Node>();
  }

  public void breadthFirstSearch(int source) {

    // Set the initial conditions for the source node
    Node snode = nodes.get(source);
    snode.setColor(Node.Color.GRAY);
```

Got Bonus?

Interested in setting up an peer recognition program?

bonus.ly can manage your peer-to-peer bonus program.

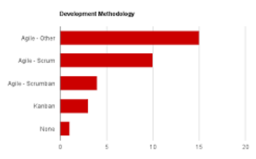
If you're interested in being an alpha user, sign-up!

Interviewing software developers?

Need help coming up with the best possible software interview questions?

Check out theReq's selection of questions for Java, Ruby, PHP, and Python interviews.

thereq



Software Engineering Tool Set - A Survey

tech.thereq.com
the way we do the things we do

Now Blogging at tech.thereq.com

estimating the true cost of a new feature

fitting the kiva redesign into a two week release cycle

kiva engineering: tools of the agile trade

kiva engineering: innovation iteration

kiva engineering: a year in review

```
    snode.setDistance(0);

    Queue<Integer> q = new LinkedList<Integer>();
    q.add(source);

    while (!q.isEmpty()) {
      Node unode = nodes.get(q.poll());

      for (int v : unode.getEdges()) {
        Node vnode = nodes.get(v);
        if (vnode.getColor() == Node.Color.WHITE) {
          vnode.setColor(Node.Color.GRAY);
          vnode.setDistance(unode.getDistance() + 1);
          vnode.setParent(unode.getId());
          q.add(v);
        }
      }
      unode.setColor(Node.Color.BLACK);
    }

  }

  public void addNode(int id, int[] edges) {

    // A couple lines of hacky code to transform our
    // input integer arrays (which are most comprehensible
    // write out in our main method) into List<Integer>
    List<Integer> list = new ArrayList<Integer>();
    for (int edge : edges)
      list.add(edge);

    Node node = new Node(id);
    node.setEdges(list);
    nodes.put(id, node);
  }

  public void print() {
    for (int v : nodes.keySet()) {
      Node vnode = nodes.get(v);
      System.out.printf("v = %2d parent = %2d distance = %2d \n", vnode.getId(),
vnode.getParent(),
          vnode.getDistance());
    }
  }

  public static void main(String[] args) {

    Graph graph = new Graph();
    graph.addNode(1, new int[] { 2, 5 });
    graph.addNode(2, new int[] { 1, 5, 3, 4 });
    graph.addNode(3, new int[] { 2, 4 });
    graph.addNode(4, new int[] { 2, 5, 3 });
    graph.addNode(5, new int[] { 4, 1, 2 });

    graph.breadthFirstSearch(1);
    graph.print();
  }

}
```

parallel breadth-first search using hadoop

okay, that's nifty and all, but what if your graph is really really big? this algorithm marches slowly down the tree on level at a time. once your past the first level there will be many many nodes whose edges need to be examined, and in the code above this happens sequentially. how can we modify this to work for a huge graph and run the algorithm in parallel? enter hadoop and map-reduce!

start here for a decent introduction to graph algorithms on map-reduce. on again though, this resource gives some tips, but no actual code.

let's talk through how we go about this, and actually write a little code! basically, the idea is this - every Map iteration "makes a mess" and every Reduce iteration "cleans up the mess". let's say we by representing a node in the following text format

```
ID    EDGES|DISTANCE_FROM_SOURCE|COLOR|
```

where `EDGES` is a comma delimited list of the ids of the nodes that are connected to this node. in the beginning, we do not know the distance and will use `Integer.MAX_VALUE` for marking "unknown". the color tells us whether or not we've seen the node before, so this starts off as white.

suppose we start with the following input graph, in which we've stated that node #1 is the source (starting point) for the search, and as such have marked this one special node with distance 0 and color GRAY.

```
1       2,5|0|GRAY|
2       1,3,4,5|Integer.MAX_VALUE|WHITE|
3       2,4|Integer.MAX_VALUE|WHITE|
4       2,3,5|Integer.MAX_VALUE|WHITE|
5       1,2,4|Integer.MAX_VALUE|WHITE|
```

the mappers are responsible for "exploding" all gray nodes - e.g. for exploding all nodes that live at our current depth in the tree. for each gray node, the mappers emit a new gray node, with distance = distance + 1. they also then emit the input gray node, but colored black. (once a node has been exploded, we're done with it.) mappers also emit all non-gray nodes, with no change. so, the output of the first map iteration would be

```
1       2,5|0|BLACK|
2       NULL|1|GRAY|
5       NULL|1|GRAY|
2       1,3,4,5|Integer.MAX_VALUE|WHITE|
3       2,4|Integer.MAX_VALUE|WHITE|
4       2,3,5|Integer.MAX_VALUE|WHITE|
5       1,2,4|Integer.MAX_VALUE|WHITE|
```

note that when the mappers "explode" the gray nodes and create a new node for each edge, they do not know what to write for the edges of this new node - so they leave it blank.

the reducers, of course, receive all data for a given key - in this case it means that they receive the data for all "copies" of each node. for example, the reducer that receives the data for key = 2 gets the following list of values :

```
2       NULL|1|GRAY|
2       1,3,4,5|Integer.MAX_VALUE|WHITE|
```

the reducers job is to take all this data and construct a new node using

- the non-null list of edges
- the minimum distance
- the darkest color

using this logic the output from our first iteration will be :

```
1       2,5,|0|BLACK
2       1,3,4,5,|1|GRAY
3       2,4,|Integer.MAX_VALUE|WHITE
```

```
4       2,3,5,|Integer.MAX_VALUE|WHITE
5       1,2,4,|1|GRAY
```

the second iteration uses this as the input and outputs :

```
1       2,5,|0|BLACK
2       1,3,4,5,|1|BLACK
3       2,4,|2|GRAY
4       2,3,5,|2|GRAY
5       1,2,4,|1|BLACK
```

and the third iteration outputs:

```
1       2,5,|0|BLACK
2       1,3,4,5,|1|BLACK
3       2,4,|2|BLACK
4       2,3,5,|2|BLACK
5       1,2,4,|1|BLACK
```

subsequent iterations will continue to print out the same output.

how do you know when you're done? you are done when there are no output nodes that are colored gray. note - if not all nodes in your input are actually connected to your source, you may have final output nodes that are still white.

here's an actual implementation in hadoop. once again, Node is little more than a simple bean, plus some ugly string processing nonsense. you can download Node.java here and GraphSearch.java here.

```
package org.apache.hadoop.examples;

import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

/**
* This is an example Hadoop Map/Reduce application.
*
* It inputs a map in adjacency list format, and performs a breadth-first search.
* The input format is
* ID   EDGES|DISTANCE|COLOR
* where
* ID = the unique identifier for a node (assumed to be an int here)
* EDGES = the list of edges emanating from the node (e.g. 3,8,9,12)
* DISTANCE = the to be determined distance of the node from the source
* COLOR = a simple status tracking field to keep track of when we're finished with a node
* It assumes that the source node (the node from which to start the search) has
* been marked with distance 0 and color GRAY in the original input.  All other
* nodes will have input distance Integer.MAX_VALUE and color WHITE.
*/
public class GraphSearch extends Configured implements Tool {
```

```
public static final Log LOG = LogFactory.getLog("org.apache.hadoop.examples.GraphSearch");

/**
 * Nodes that are Color.WHITE or Color.BLACK are emitted, as is. For every
 * edge of a Color.GRAY node, we emit a new Node with distance incremented by
 * one. The Color.GRAY node is then colored black and is also emitted.
 */
public static class MapClass extends MapReduceBase implements
    Mapper<LongWritable, Text, IntWritable, Text> {

  public void map(LongWritable key, Text value, OutputCollector<IntWritable, Text> output,
      Reporter reporter) throws IOException {

    Node node = new Node(value.toString());

    // For each GRAY node, emit each of the edges as a new node (also GRAY)
    if (node.getColor() == Node.Color.GRAY) {
      for (int v : node.getEdges()) {
        Node vnode = new Node(v);
        vnode.setDistance(node.getDistance() + 1);
        vnode.setColor(Node.Color.GRAY);
        output.collect(new IntWritable(vnode.getId()), vnode.getLine());
      }
      // We're done with this node now, color it BLACK
      node.setColor(Node.Color.BLACK);
    }

    // No matter what, we emit the input node
    // If the node came into this method GRAY, it will be output as BLACK
    output.collect(new IntWritable(node.getId()), node.getLine())

  }
}

/**
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce extends MapReduceBase implements
    Reducer<IntWritable, Text, IntWritable, Text> {

  /**
   * Make a new node which combines all information for this single node id.
   * The new node should have
   * - The full list of edges
   * - The minimum distance
   * - The darkest Color
   */
  public void reduce(IntWritable key, Iterator<Text> values,
      OutputCollector<IntWritable, Text> output, Reporter reporter) throws IOException {

    List<Integer> edges = null;
    int distance = Integer.MAX_VALUE;
    Node.Color color = Node.Color.WHITE;

    while (values.hasNext()) {
      Text value = values.next();

      Node u = new Node(key.get() + "\t" + value.toString());

      // One (and only one) copy of the node will be the fully expanded
      // version, which includes the edges
      if (u.getEdges().size() > 0) {
        edges = u.getEdges();
      }

      // Save the minimum distance
      if (u.getDistance() < distance) {
        distance = u.getDistance();
      }

      // Save the darkest color
      if (u.getColor().ordinal() > color.ordinal()) {
        color = u.getColor();
      }

    }

    Node n = new Node(key.get());
    n.setDistance(distance);
    n.setEdges(edges);
    n.setColor(color);
    output.collect(key, new Text(n.getLine()));

  }
}

static int printUsage() {
  System.out.println("graphsearch [-m <num mappers>] [-r <num reducers>]");
  ToolRunner.printGenericCommandUsage(System.out);
  return -1;
}

private JobConf getJobConf(String[] args) {
  JobConf conf = new JobConf(getConf(), GraphSearch.class);
  conf.setJobName("graphsearch");

  // the keys are the unique identifiers for a Node (ints in this case).
  conf.setOutputKeyClass(IntWritable.class);
  // the values are the string representation of a Node
  conf.setOutputValueClass(Text.class);

  conf.setMapperClass(MapClass.class);
  conf.setReducerClass(Reduce.class);

  for (int i = 0; i < args.length; ++i) {
    if ("-m".equals(args[i])) {
      conf.setNumMapTasks(Integer.parseInt(args[++i]));
    } else if ("-r".equals(args[i])) {
      conf.setNumReduceTasks(Integer.parseInt(args[++i]));
    }
  }

  return conf;
}

/**
 * The main driver for word count map/reduce program. Invoke this method to
 * submit the map/reduce job.
 *
 * @throws IOException
 *           When there is communication problems with the job tracker.
```

```
   */
  public int run(String[] args) throws Exception {

    int iterationCount = 0;

    while (keepGoing(iterationCount)) {

      String input;
      if (iterationCount == 0)
        input = "input-graph";
      else
        input = "output-graph-" + iterationCount;

      String output = "output-graph-" + (iterationCount + 1);

      JobConf conf = getJobConf(args);
      FileInputFormat.setInputPaths(conf, new Path(input));
      FileOutputFormat.setOutputPath(conf, new Path(output));
      RunningJob job = JobClient.runJob(conf);

      iterationCount++;
    }

    return 0;
  }

  private boolean keepGoing(int iterationCount) {
    if(iterationCount >= 4) {
      return false;
    }

    return true;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new GraphSearch(), args);
    System.exit(res);
  }

}
```

the remotely astute reader may notice that the `keepGoing` method is not actually checking to see if there are any remaining gray node, but rather just stops after the 4th iteration. this is because there is no easy way to communicate this information in hadoop. what we want to do is the following :

1. at the beginning of each iteration, set a global flag `keepGoing = false`
2. as the reducer outputs each node, if the is outputting a gray node set `keepGoing = true`

unfortunately, hadoop provides no framework for setting such a global variable. this would need to be managed using an external semaphore server of some sort. this is left as an exercise for the lucky reader. ;)

cailin's blog

employee recognition