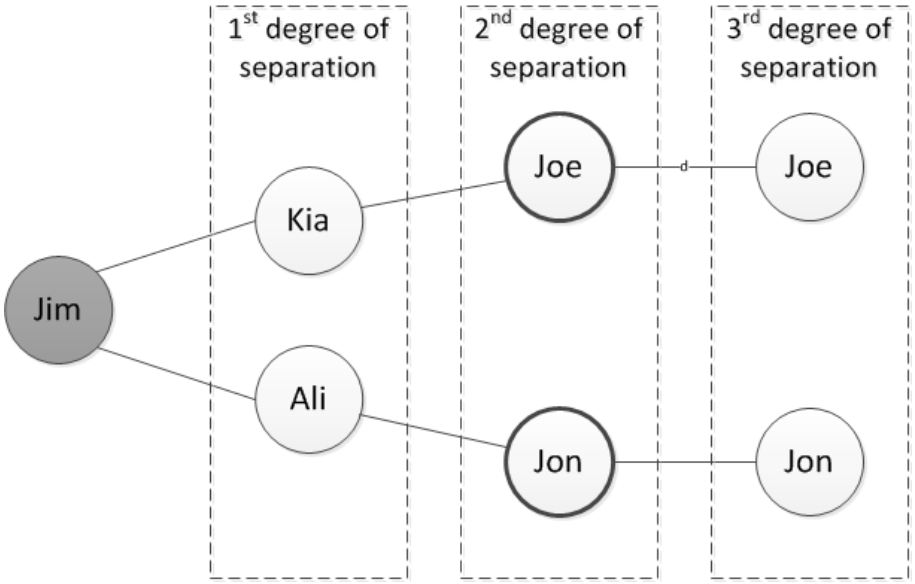
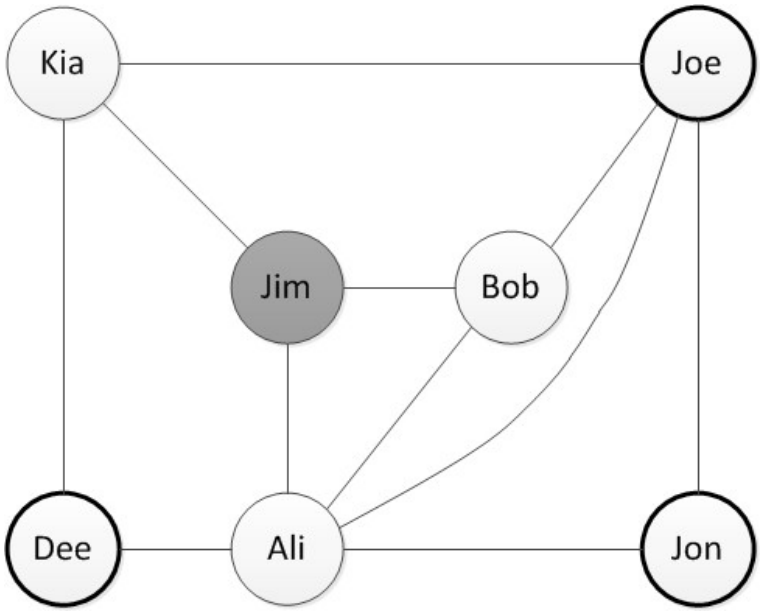
Lab Lecture #5

Introduction

The friends-of-friends (a.k.a. FoF) algorithm suggest friends that a user may know that aren't part of their immediate network. You could consider the FoF to be in the 2nd degree network.



Social networks sites such Facebook and LinkedIn use the FoF algorithm to help users broaden their networks. The trick to success with this approach is to order FoFs by the number of common friends, which increases the chances that the user knows the FoF.
Two MapReduce jobs are required to calculate the FoFs for each user in a social network. The first calculates the FoFs and counts the number of friends in common for each user. The second sorts the commond friends by the number of connections to your friends, producing an ordered list of FoFs by the common friends.
The idea behind this algorithm could be better explained giving a look at the following example social graph:



Suppose we want to suggest

FoF to Jim. They are the users Dee, Joe and Jon. Next to their name there are the number of friends that they have in common with Jim. Our goal here is to determine all the FoFs and order them by the number of friends in common. Therefore, our expected results would have Joe as the first FoF recommendation, followed by Dee and then by Jon.

The text file describing the above social graph is shown here:

```
joe  jon  kia  bob  ali
kia  joe  jim  dee
dee  kia  ali
ali  dee  jim  bob  joe  jon
jon  joe  ali
bob  joe  ali  jim
jim  kia  bob  ali
```

FoF calculation, Job 1

Let's dive into the code in the following listing and look at the first MapReduce job, which calculates the FoFs for each user.

```
public static class Map
      extends Mapper<Text, Text, TextPair,    IntWritable> {

    private TextPair pair = new TextPair();
    private IntWritable one = new IntWritable(1);
    private IntWritable two = new IntWritable(2);

    @Override
    protected void map(Text key, Text value, Context context)
        throws IOException, InterruptedException {

      String[] friends = StringUtils.split(value.toString());

      for (int i = 0; i < friends.length; i++) {

        // they already know each other, so emit the pair with
        // a "1" to indicate this, so that this relationship
        // can be discarded in the reduce phase. The TextPair
        // class lexicographically orders the two names so
        // that for a given pair of userthere will be a single
        // reducer key.

        pair.set(key.toString(), friends[i]);
        context.write(pair, one);

        // go through all the remaining friends in the list
        // and emit the fact that they are 2nd-degree friends

        for (int j = i + 1; j < friends.length; j++) {
          pair.set(friends[i], friends[j]);
          context.write(pair, two);
        }
      }
    }
  }
```

```
  public static class Reduce
      extends Reducer<TextPair, IntWritable, TextPair, IntWritable> {

    private IntWritable friendsInCommon = new IntWritable();

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

      int commonFriends = 0;
      boolean alreadyFriends = false;
      // if the friends know each other then we'll eventually
      // see a value will be a "1", which will cause us to
      // break out of the loop
      for (IntWritable hops : values) {
        if (hops.get() == 1) {
          // ignore this relationship if the users are already friends
          alreadyFriends = true;
          break;
        }

        commonFriends++;
      }

      if (!alreadyFriends) {
        // output the fact that they're FoFs, including a count of
        // commond friends. This also uses the TextPair class to
        // lexicographically order the user names.

        friendsInCommon.set(commonFriends);
        context.write(key, friendsInCommon);
      }
    }
  }
```

The output for executing this job against the example graph above showed is the following:

```
ali     kia     3
bob     dee     1
bob     jon     2
bob     kia     2
dee     jim     2
dee     joe     2
dee     jon     1
jim     joe     3
jim     jon     1
jon     kia     1
```

FoF sorts, Job 2

The job of the second MapReduce job in the following listing is to sort the FoFs such that you see FoFs with a higher number of mutual friends ahead of those that have a smaller number of mutual friends.

```
public final class SortMapReduce {
  public static class Map
      extends Mapper<Text, Text, Person, Person> {

    private Person outputKey = new Person();
    private Person outputValue = new Person();

    @Override
    protected void map(Text key, Text value, Context context)
        throws IOException, InterruptedException {

      String[] parts = StringUtils.split(value.toString());
      String name = parts[0];
      int commonFriends = Integer.valueOf(parts[1]);

      // emit the first half of the relationship
      outputKey.set(name, commonFriends);
      outputValue.set(key.toString(), commonFriends);
      context.write(outputKey, outputValue);

      // emit the second half of the relationship
      outputValue.set(name, commonFriends);
      outputKey.set(key.toString(), commonFriends);
      context.write(outputKey, outputValue);
    }
  }
```

```
public static class Reduce
      extends Reducer<Person, Person, Text, Text> {

    private Text name = new Text();
    private Text potentialFriends = new Text();

    @Override
    public void reduce(Person key, Iterable<Person> values,
                       Context context)
        throws IOException, InterruptedException {

      StringBuilder sb = new StringBuilder();
      // the 2nd-degree friends will be sorted by the number
      // of common friends, so emit the top 10
      int count = 0;
      for (Person potentialFriend : values) {
        if(sb.length() > 0) {
          sb.append(",");
        }
        sb.append(potentialFriend.getName())
            .append(":")
            .append(potentialFriend.getCommonFriends());

        if (++count == 10) {
          break;
        }
      }

      name.set(key.getName());
      potentialFriends.set(sb.toString());
      context.write(name, potentialFriends);
    }
  }
```

The output of this second (and final) job is the following:

```
ali  kia:3
bob  kia:2,jon:2,dee:1
dee  jim:2,joe:2,jon:1,bob:1
jim  joe:3,dee:2,jon:1
joe  jim:3,dee:2
jon  bob:2,kia:1,dee:1,jim:1
kia  ali:3,bob:2,jon:1
```

The results confirm what we are expecting from the FoFs of Jim: Joe, Dee and Jon, ordered by the number of common friends. We wan't show here the whole driver code, but to enable the secondary sort we had to write a few extra classes as well as inform the job to use the classes for partitioning and sorting:

```
job.setPartitionerClass(PersonNamePartitioner.class);
job.setSortComparatorClass(PersonComparator.class);
job.setGroupingComparatorClass(PersonNameComparator.class);
```