Parallel and Distributed Computing
Lab4
PageRank

In this lab, you'll practice using the Spark "join" operation along with understanding partitioning/shuffles. You will do this by implementing a version of the PageRank algorithm.

PageRank Overview

PageRank was the original ranking algorithm used at Google, developed by and named after one of Google's co-founders, Larry Page. It works generally as follows:

1. Every page has a number of points: its PageRank score.
2. Every page votes for the pages that it links to, distributing to each page a number of points equal to the linking page's PageRank score divided by the number of pages it links to.
3. Each page then receives a new PageRank score, based on the sum of all the votes it received.
4. The process is repeated until the scores are either within a tolerance or for a fixed number of iterations.

Your Implementation

To implement PageRank you should do the following:

Load in the webpage linking data. Eventually I want you to use the short data from lab3. But to start just hardcode the following with a parallelize. It will make a nice simple test case.

["a b c", "b a a", "c b", "d a"]

That is, A links to B and C. B links to A twice. C link to B. And D links to A.

Next you will need to make 2 RDDs. One for linking information and one for ranking information.

The linking RDD, **links**, holds the neighbor links for each source page. Build this directly from the file (or hardcoded) RDD. The following is what links should look like when collected. Note that duplicate links from the same source page are removed (e.g. "b a a"). Note that once this RDD is built, it will never change, but it will be used numerous times in the iterative step below.

[('a', ['b', 'c']), ('b', ['a']), ('c', ['b']), ('d', ['a'])]

The other RDD, **rankings**, holds the source page ranking information. This will just be pairs connecting source page to its ranking as seen below. Initially every page should be given the same ranking. Note that for our implementation, we want the rankings to always add up to 1 - like a probability distribution.

[('a', 0.25), ('b', 0.25), ('c', 0.25), ('d', 0.25)]

After the initial RDDs are setup, it is time to implement the iterative step. This can be done until the rankings stabilize or for a fixed number of iterations. We will setup our code to simply do a fixed number of iterations. 10 iterations is the number we will be using for the demos.

Each iteration need to figure out the contribution each source page makes to its neighbor links. This was step 2 outlined above. Every page votes for the pages that it links to, distributing to each page a number of points equal to the linking page's PageRank score divided by the number of pages it links to. This sounds like a mapping. And then we need to do step 3. Each page then receives a new PageRank score, based on the sum of all the votes it received. This sounds like a reduction.

However in order to do the mapping, note that you will need both the neighbor links and the page's ranking. Those are in different RDDs. And you can't just pass one RDD as the parameter to the other RDD's mapping function. Thus, you will need to join the RDDs together to form a temporary RDD you can map over. For our demo example, that join looks like:

[('b', (['a'], 0.25)), ('c', (['b'], 0.25)), ('a', (['b', 'c'], 0.25)), ('d', (['a'], 0.25))]

Then after mapping to calculate the neighbor contributions, the results should look like:

[('a', 0.25), ('b', 0.25), ('b', 0.125), ('c', 0.125), ('a', 0.25)]

And finally reducing to combine the individual neighbor contributions:

[('b', 0.375), ('c', 0.125), ('a', 0.5)]

Note that these are our new rankings for the next iteration. We can throw the old ones away and simply use the new ones calculated from the neighbor contributions. Also note that any page that didn't have any links going into it gets dropped from this list. It will still exist as a page in the links RDD and we can just infer its ranking is 0.

As a final step, after the 10 iterations, you should then sort your rankings by ranking, highest to lowest.

Demo

As stated above, it is best if you test this with hardcoded data to understand how it works. It will run much faster if you only display the final results and not the intermediate steps. However, I want to see the intermediate steps in your output. So for the hardcoded demo you code should produce the following:

Initial links: [('a', ['b', 'c']), ('b', ['a']), ('c', ['b']), ('d', ['a'])]
Initial rankings: [('a', 0.25), ('b', 0.25), ('c', 0.25), ('d', 0.25)]
Iteration: 0
Joined RDD: [('b', (['a'], 0.25)), ('c', (['b'], 0.25)), ('a', (['b', 'c'], 0.25)), ('d', (['a'], 0.25))]
Neighbor contributions: [('a', 0.25), ('b', 0.25), ('b', 0.125), ('c', 0.125), ('a', 0.25)]
New rankings: [('b', 0.375), ('c', 0.125), ('a', 0.5)]

Iteration: 1
Joined RDD: [('b', (['a'], 0.375)), ('a', (['b', 'c'], 0.5)), ('c', (['b'], 0.125))]
Neighbor contributions: [('a', 0.375), ('b', 0.25), ('c', 0.25), ('b', 0.125)]
New rankings: [('b', 0.375), ('a', 0.375), ('c', 0.25)]
Iteration: 2
Joined RDD: [('b', (['a'], 0.375)), ('a', (['b', 'c'], 0.375)), ('c', (['b'], 0.25))]
Neighbor contributions: [('a', 0.375), ('b', 0.1875), ('c', 0.1875), ('b', 0.25)]
New rankings: [('b', 0.4375), ('a', 0.375), ('c', 0.1875)]
Iteration: 3
Joined RDD: [('b', (['a'], 0.4375)), ('a', (['b', 'c'], 0.375)), ('c', (['b'], 0.1875))]
Neighbor contributions: [('a', 0.4375), ('b', 0.1875), ('c', 0.1875), ('b', 0.1875)]
New rankings: [('b', 0.375), ('a', 0.4375), ('c', 0.1875)]
Iteration: 4
Joined RDD: [('b', (['a'], 0.375)), ('c', (['b'], 0.1875)), ('a', (['b', 'c'], 0.4375))]
Neighbor contributions: [('a', 0.375), ('b', 0.1875), ('b', 0.21875), ('c', 0.21875)]
New rankings: [('b', 0.40625), ('c', 0.21875), ('a', 0.375)]
Iteration: 5
Joined RDD: [('c', (['b'], 0.21875)), ('a', (['b', 'c'], 0.375)), ('b', (['a'], 0.40625))]
Neighbor contributions: [('b', 0.21875), ('b', 0.1875), ('c', 0.1875), ('a', 0.40625)]
New rankings: [('c', 0.1875), ('a', 0.40625), ('b', 0.40625)]
Iteration: 6
Joined RDD: [('a', (['b', 'c'], 0.40625)), ('c', (['b'], 0.1875)), ('b', (['a'], 0.40625))]
Neighbor contributions: [('b', 0.203125), ('c', 0.203125), ('b', 0.1875), ('a', 0.40625)]
New rankings: [('a', 0.40625), ('c', 0.203125), ('b', 0.390625)]
Iteration: 7
Joined RDD: [('c', (['b'], 0.203125)), ('a', (['b', 'c'], 0.40625)), ('b', (['a'], 0.390625))]
Neighbor contributions: [('b', 0.203125), ('b', 0.203125), ('c', 0.203125), ('a', 0.390625)]
New rankings: [('c', 0.203125), ('a', 0.390625), ('b', 0.40625)]
Iteration: 8
Joined RDD: [('b', (['a'], 0.40625)), ('a', (['b', 'c'], 0.390625)), ('c', (['b'], 0.203125))]
Neighbor contributions: [('a', 0.40625), ('b', 0.1953125), ('c', 0.1953125), ('b', 0.203125)]
New rankings: [('b', 0.3984375), ('a', 0.40625), ('c', 0.1953125)]
Iteration: 9
Joined RDD: [('a', (['b', 'c'], 0.40625)), ('c', (['b'], 0.1953125)), ('b', (['a'], 0.3984375))]
Neighbor contributions: [('b', 0.203125), ('c', 0.203125), ('b', 0.1953125), ('a', 0.3984375)]
New rankings: [('a', 0.3984375), ('c', 0.203125), ('b', 0.3984375)]
Final sorted rankings:
a has rank: 0.3984375
b has rank: 0.3984375
c has rank: 0.203125

Lab3short

You should also do a test run on lab3short. For this run, you can comment out the intermediate prints and just display the final sorted rankings.

Submission

After completing the lab, upload your .py and .dbc files to Canvas. You should also upload an example run showing the results you got running on the demo test data.