MIDS-W261-2016-HWK-Week09-Lane

July 21, 2016

In [294]:
```
%%javascript
/*********************************************************************************************
Known Mathjax Issue with Chrome - a rounding issue adds a border to the right of mathjax marku
https://github.com/mathjax/MathJax/issues/1300
A quick hack to fix this based on stackoverflow discussions:
http://stackoverflow.com/questions/34277967/chrome-rendering-mathjax-equations-with-a-trailing
*********************************************************************************************

$('.math>span').css("border-left-color","transparent")
```

```
<IPython.core.display.Javascript object>
```

In [295]:
```
%reload_ext autoreload
%autoreload 2
```

1 DATASCI W261 - Machine Learning At Scale

1.1 ## Assignment - Week 09

Name: Jackson Lane
Class: W261 (Section 3)
Email: jelane@ischool.berkeley.edu
Week: 09

HW 9 Dataset

Note that all referenced files life in the enclosing directory. Checkout the Data subdirectory on Dropbox or the AWS S3 buckets (details contained each question).

HW 9.0: Short answer questions

__ What is PageRank and what is it used for in the context of web search?__

PageRank is a graph ranking algorith that models a random surfer visiting visiting pages and following links. PageRank iteratively ranks nodes by the distributing the rank of the neighbors. PageRank was originally developed by Google to imrpove search results relevance. It has since gone through many iterations and resdesigns to improve the algorithm. In this homework's implementation of PageRank, each web page is a node and each link on a webpage is a weighted, directed link on the graph. This homework's implementation also doesn't take into account text similarity to a search query word. Rather, this homework's implementation just ranks each page on the number and weight of in-links.

__ What modifications have to be made to the webgraph in order to leverage the machinery of Markov Chains to compute the Steady State Distibution? __

We need to make to make two adjustsments:

1. Dangling Nodes Adjustment There are an often nodes that have in-links, but no outlinks. These are called dangling nodes and present a problem for PageRank. So rather than let weight accumulate at these dangling nodes after each iteration, the PageRank algorithm should distribute the weight of these dangling nodes to all nodes in the graph, as though the dangling node had outlinks to every node in the

graph, including itself. This can be modeled in the transition matrix by setting the row corresponding to each dangling node to 1/N, where N is the number of nodes in the graph. Technically, this adjustment is not necessary, and you could still make a Markov process out of a PageRank algorithm that lets all the weight accumulate at dangling nodes. It just wouldn't be a very accurate model though for search results relevance.

2. Teleportation adjustment Similar to dangling nodes, there are often connected groups of nodes that may have no outlinks to any other nodes outside of the group. For example, a particular website may only have links to other pages in that website. Rather than let weight accumulate within this group of nodes, the PageRank algorithm gives the random surfer a probability after each iteration to teleport to a random node anywhere in the graph. This can be modeled in the transition matrix by multiplying each node's PageRank by one minus the teleportation probability and then adding the teleportation probability divided by number of nodes.

__ OPTIONAL: In topic-specific pagerank, how can we ensure that the irreducible property is satifsied? (HINT: see HW9.4) __

HW 9.1: MRJob implementation of basic PageRank

Write a basic MRJob implementation of the iterative PageRank algorithm that takes sparse adjacency lists as input (as explored in HW 7).

Make sure that you implementation utilizes teleportation (1-damping/the number of nodes in the network), and further, distributes the mass of dangling nodes with each iteration so that the output of each iteration is correctly normalized (sums to 1).

[NOTE: The PageRank algorithm assumes that a random surfer (walker), starting from a random web page, chooses the next page to which it will move by clicking at random, with probability d,one of the hyperlinks in the current page. This probability is represented by a so-called damping factor d, where d $\in$ (0, 1). Otherwise, with probability (1 − d), the surfer jumps to any web page in the network. If a page is a dangling end, meaning it has no outgoing hyperlinks, the random surfer selects an arbitrary web page from a uniform distribution and "teleports" to that page]

As you build your code, use the test data:

s3://ucb-mids-mls-networks/PageRank-test.txt

Or under the Data Subfolder for HW7 on Dropbox with the same file name. > Dropbox: https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0

with teleportation parameter set to 0.15 (1-d, where d, the damping factor is set to 0.85), and crosscheck your work with the true result, displayed in the first image in the Wikipedia article and here for reference are the corresponding PageRank probabilities:

HW 9.1 Implementation

In [5]: %%writefile MRJob9_1_findnodes.py

```
#This the findnodes code from HW 7.  It will be used in 9.1 and 9.3
#MR Job code to find all the nodes in a graph.  It does this by emitting a key for each node id

from mrjob.job import MRJob
from mrjob.step import MRStep

class MRJob9_1_findnodes(MRJob):

    def steps(self):
        return [MRStep(
                    mapper=self.mapper
                    , combiner=self.reducer
                       , reducer=self.reducer
                ,jobconf =  {
            "mapred.map.tasks":4,
```

```
                    "mapred.reduce.tasks":2,
                                                        'stream.num.map.output.key.fields': '1'

                }
                            )
                        ]

            def mapper(self, _, line):
                node, neighbors = line.strip().split('\t')
                neighbors = eval(neighbors)
                yield str(node), 1
                #Since the graph data file only has nodes with out links, we also need to emit each nei
                # ensure that we also cover nodes that may only have in links but no out links.
                # On Wikipedia in particular, these
                for (node,_) in neighbors.items():
                    yield str(node), 1


            def reducer(self, node, values):
                yield str(node), 1

        if __name__ == '__main__':
            MRJob9_1_findnodes.run()
```

```
Overwriting MRJob9_1_findnodes.py
```

In [435]:
```
%%writefile MRJob9_1_pagerank.py

          #This job runs the page rank algorithm in three steps
          #In the first step, the nodes are initialized to a weight of 1/N,
          # where N is the number of nodes in the graph.  Dangling nodes emit their weights
          # with a special key "*".
          #In the second step, the each node distributes its weight to its neighbors.  Each neighbor
          # then sums up its weights.
          #The third step adds the dangling weights to each node and then applies the damping algorithm
          #The second and third steps will repeat for the specified number of iterations
          #

          from mrjob.job import MRJob
          from mrjob.step import MRStep
          import sys

          class MRJob9_1_pagerank(MRJob):
              def steps(self):
                  step =  [MRStep(
                          mapper=self.mapper,
                          reducer=self.reducer,
                          jobconf =  {
                      "mapred.map.tasks":4,
                      "mapred.reduce.tasks":2,
                      'stream.num.map.output.key.fields': '1'
                  }),
                          MRStep(reducer_init=self.reducer_stochastic_init,
                                 reducer=self.reducer_stochastic,
                                 jobconf =  {
```

```
        "mapred.reduce.tasks":1,
        'stream.num.map.output.key.fields': '1'
    })]
    return [MRStep(mapper=self.mapper_preprocess,reducer=self.reducer_preprocess
        ,jobconf =  {
        "mapred.map.tasks":4,
        "mapred.reduce.tasks":2,
        'stream.num.map.output.key.fields': '1
    })] + step * int(self.options.iterations)



def configure_options(self):
    super(MRJob9_1_pagerank, self).configure_options()
    self.add_passthrough_option('--N', default="1")
    self.add_passthrough_option('--alpha', default=".15")
    self.add_passthrough_option('--iterations', default="4")

def mapper_preprocess(self, _, line):
    fields = line.strip().split('\t')
    #Check if line comes from a graph data file
    # or an index file.  You usually can tell if its
    # a graph data file if the first field is a number.
    neighbors = eval(fields[1])
    if(str(type(neighbors)) == "<type 'dict'>"):
        id = fields[0]
        # if it turns out that this was actually a line from an index file
        # and the node name was just all numeric, then discard
        for neighbor in neighbors:
            yield neighbor, None
        yield id, neighbors
    else:
        # if line is from an index file, yield node id and name with order inversion
        id = fields[1]
        name = fields[0]
        yield id, name

# The reducer here will perform left on graph data with the names from the index file, if
def reducer_preprocess(self,id,values):
    name = id
    id = str(id)
    neighbors = {}
    weight = 1/float(self.options.N)
    for value in values:
    #Again, we can check if an entry came from the graph or from an index file by looking
    # at the type of value
        if(str(type(value)) == "<type 'dict'>"):
            neighbors = value.keys()
        else:
            if value:
                name = value
    yield id, [name,neighbors, weight]
```

```
    def mapper(self, node, values):
        name,neighbors, weight = values
        node = str(node)
        neighbors = map(str,neighbors)
        yield node, [name,neighbors,weight]
        if neighbors:
            #Yield each of node's neighbors.
            #Note that mapper does not have any information on the neighbors
                # except for node id.  So the mapper just emits what it can,
                # and it's up to the reducer to fill in the missing information
            for neighbor in neighbors:
                yield neighbor, ['',None,weight/len(neighbors)]

    def reducer(self,node, values):
        name = ""
        neighbors = {}
        oldweight = 0
        newweight= 0
        for value in values:
            if (value[0] == ''):
                newweight += value[2]
            else:
                name,neighbors,oldweight= value
                if len(neighbors)== 0:
                    yield "*", oldweight
        yield node, [name, neighbors,newweight]


    def reducer_stochastic_init(self):
        self.dangling_weight = float(0)

    def reducer_stochastic(self,node,values):
        #Collect dangling node weights
        if (node == "*"):
            self.dangling_weight += sum(values)/float(self.options.N)
        else:
        #Add dangling nodee weights to nodes and then apply damping
            name,neighbors,oldweight= values.next()
            a = float(self.options.alpha)
            newweight = (1-a)/float(self.options.N) + a*float( self.dangling_weight+oldweight
            yield node, [name, neighbors,newweight]

if __name__ == '__main__':
    MRJob9_1_pagerank.run()
```

```
Overwriting MRJob9_1_pagerank.py
```

```
In [436]: %reload_ext autoreload
          %autoreload 2

          #This is a driver function for the MRJob9_1_pagerank.

          from __future__ import division
          from MRJob9_1_findnodes import MRJob9_1_findnodes
          from MRJob9_1_pagerank import MRJob9_1_pagerank
```

```
        import numpy

        def pageRank(fileName,indexFile,runnerType,iterations,alpha):
            mr_job = MRJob9_1_findnodes(args=[
                fileName, '-r', runnerType ])

            #First task is to count up the number of nodes in the graph.
            #We will do this using the findnodes MRJob.
            nodes = 0

            #Run find nodes MRJob
            with mr_job.make_runner() as runner:
                runner.run()
                for line in runner.stream_output():
                    out = mr_job.parse_output_line(line)
                    #Sum up the results
                    nodes += out[1]

            #Now that we know the number of nodes, we can run the main pagerank MRJob
            #Pass in the value for alpha and the number of iterations, nodes as parameters
            mr_job = MRJob9_1_pagerank(args=[
                fileName,indexFile, '-r', runnerType ,
                '--N',nodes,
                '--iterations',iterations
                ,'--alpha',alpha])

            #Write the results of the pagerank MRJob to a file
            with mr_job.make_runner() as runner:
                runner.run()
                with open(fileName+"_pagerank","w") as myfile:
                    for line in runner.stream_output():
                        out = mr_job.parse_output_line(line)
                        myfile.write(line)

In [399]: pageRank("Data/PageRank-test.txt","","inline",50,.85)
          !cat "Data/PageRank-test.txt_pagerank"

"A"       ["A", [], 0.032781493159347676]
"B"       ["B", ["C"], 0.38436978095287694]
"C"       ["C", ["B"], 0.34294145336903575]
"D"       ["D", ["A", "B"], 0.03908709209997012]
"E"       ["E", ["B", "D", "F"], 0.08088569323450434]
"F"       ["F", ["B", "E"], 0.03908709209997012]
"G"       ["G", ["B", "E"], 0.016169479016858935]
"H"       ["H", ["B", "E"], 0.016169479016858935]
"I"       ["I", ["B", "E"], 0.016169479016858935]
"J"       ["J", ["E"], 0.016169479016858935]
"K"       ["K", ["E"], 0.016169479016858935]
```

HW 9.1 Analysis
HW 9.2: Exploring PageRank teleportation and network plots

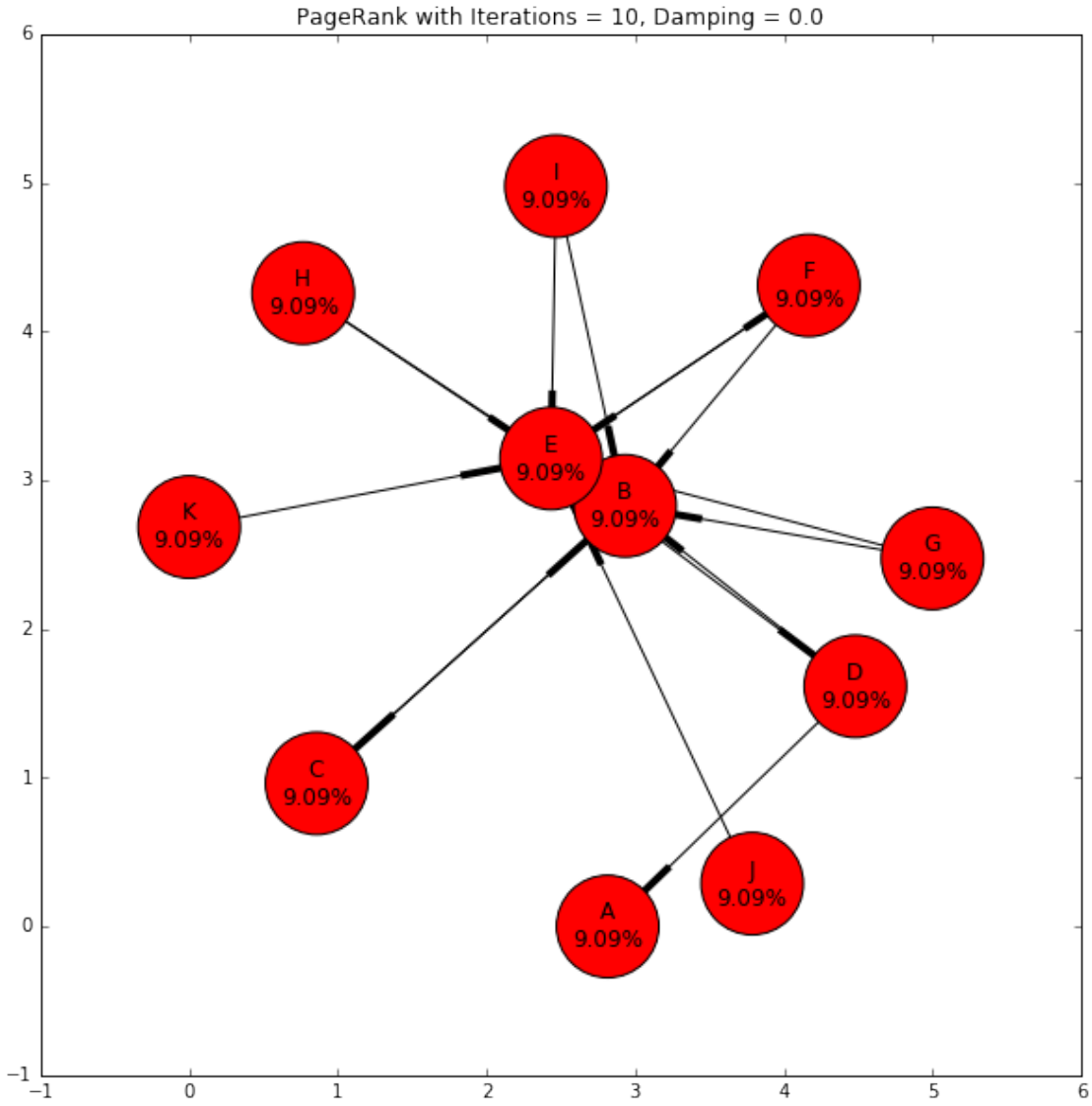
- In order to overcome problems such as disconnected components, the damping factor (a typical value for d is 0.85) can be varied.

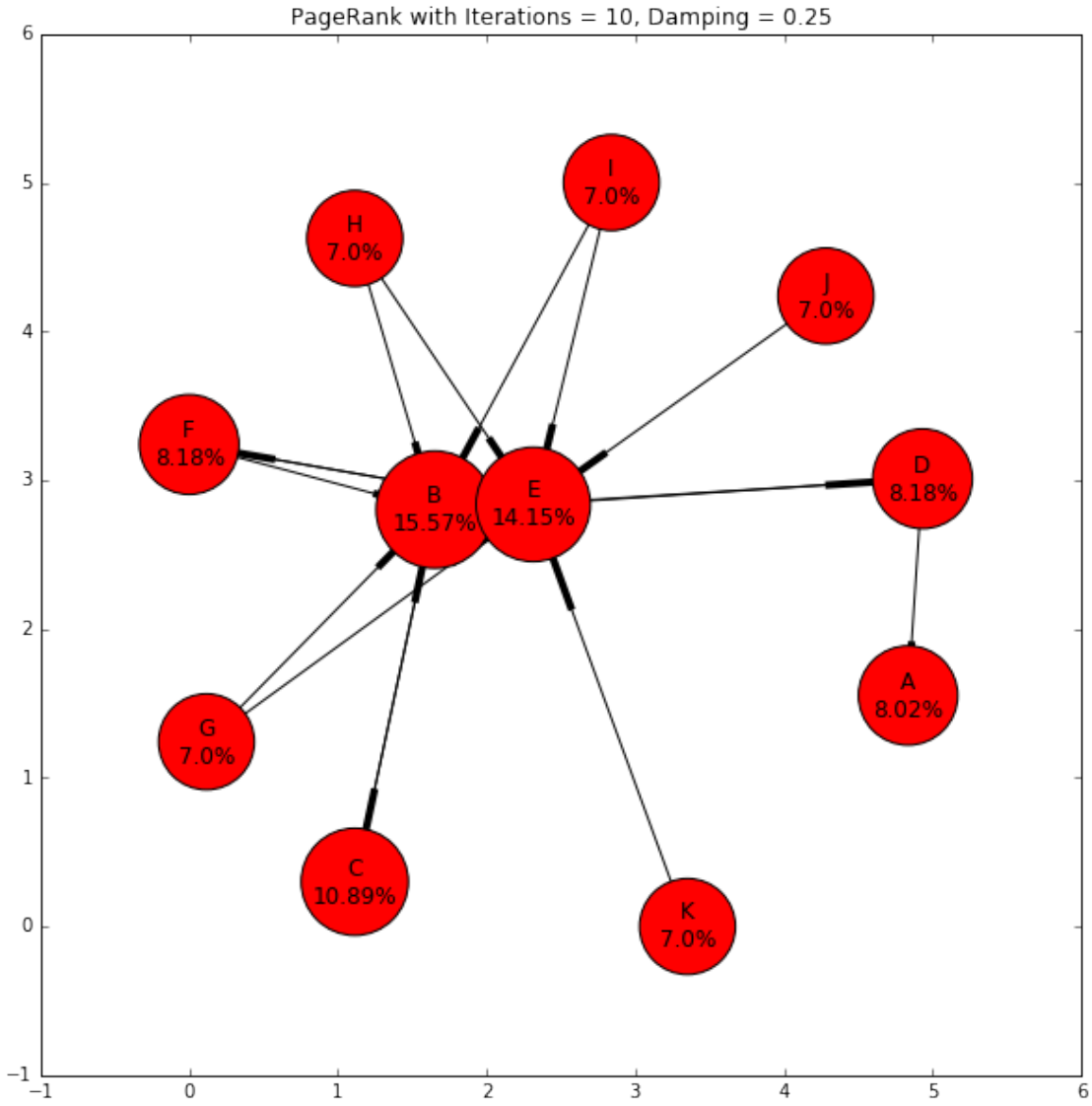
- Using the graph in HW1, plot the test graph (using networkx, https://networkx.github.io/) for several values of the damping parameter alpha, so that each nodes radius is proportional to its PageRank score.
- In particular you should do this for the following damping factors: [0,0.25,0.5,0.75, 0.85, 1].
- Note your plots should look like the following: https://en.wikipedia.org/wiki/PageRank#/media/File:PageRanks-Example.svg

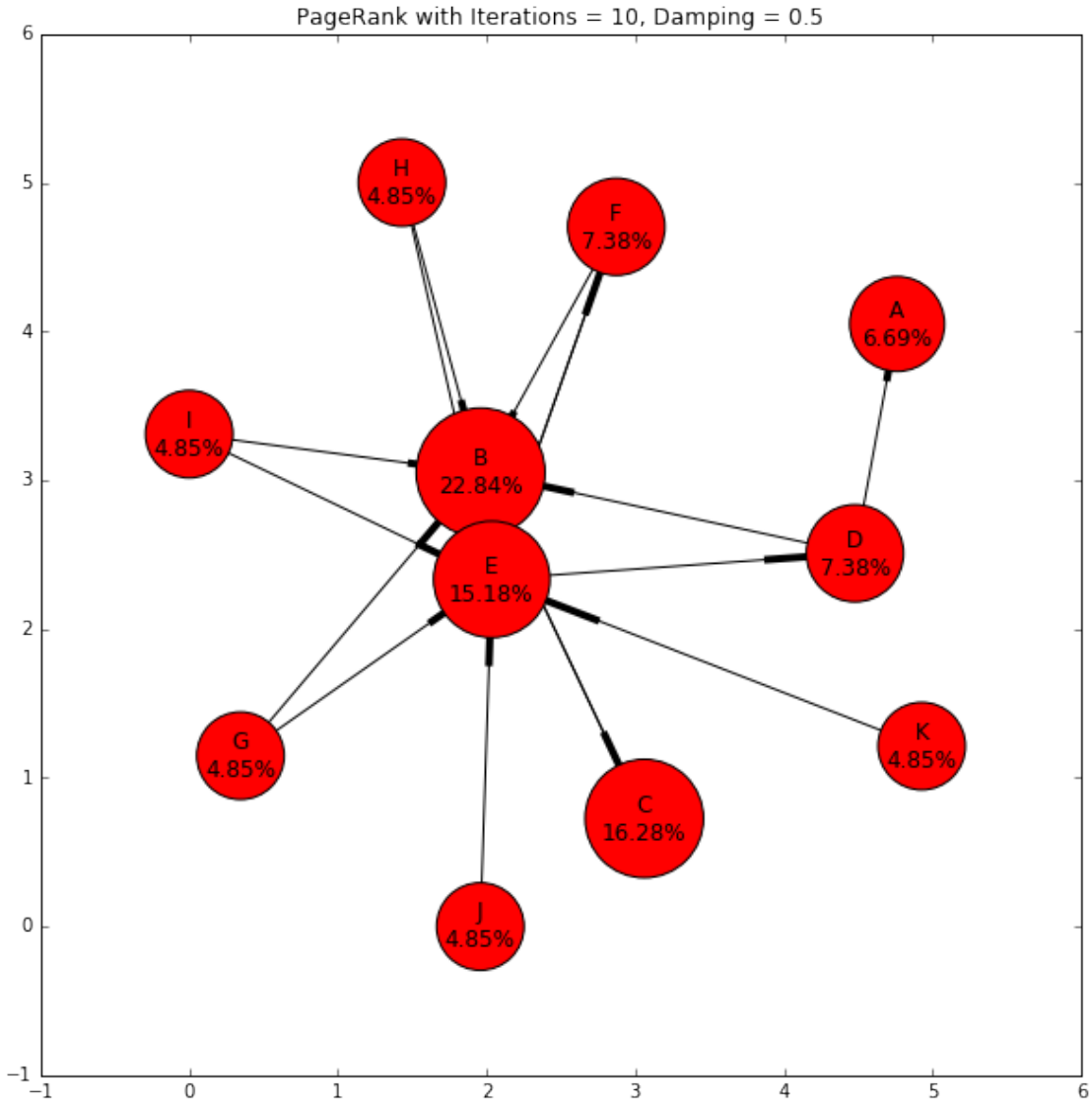
HW 9.2 Implementation

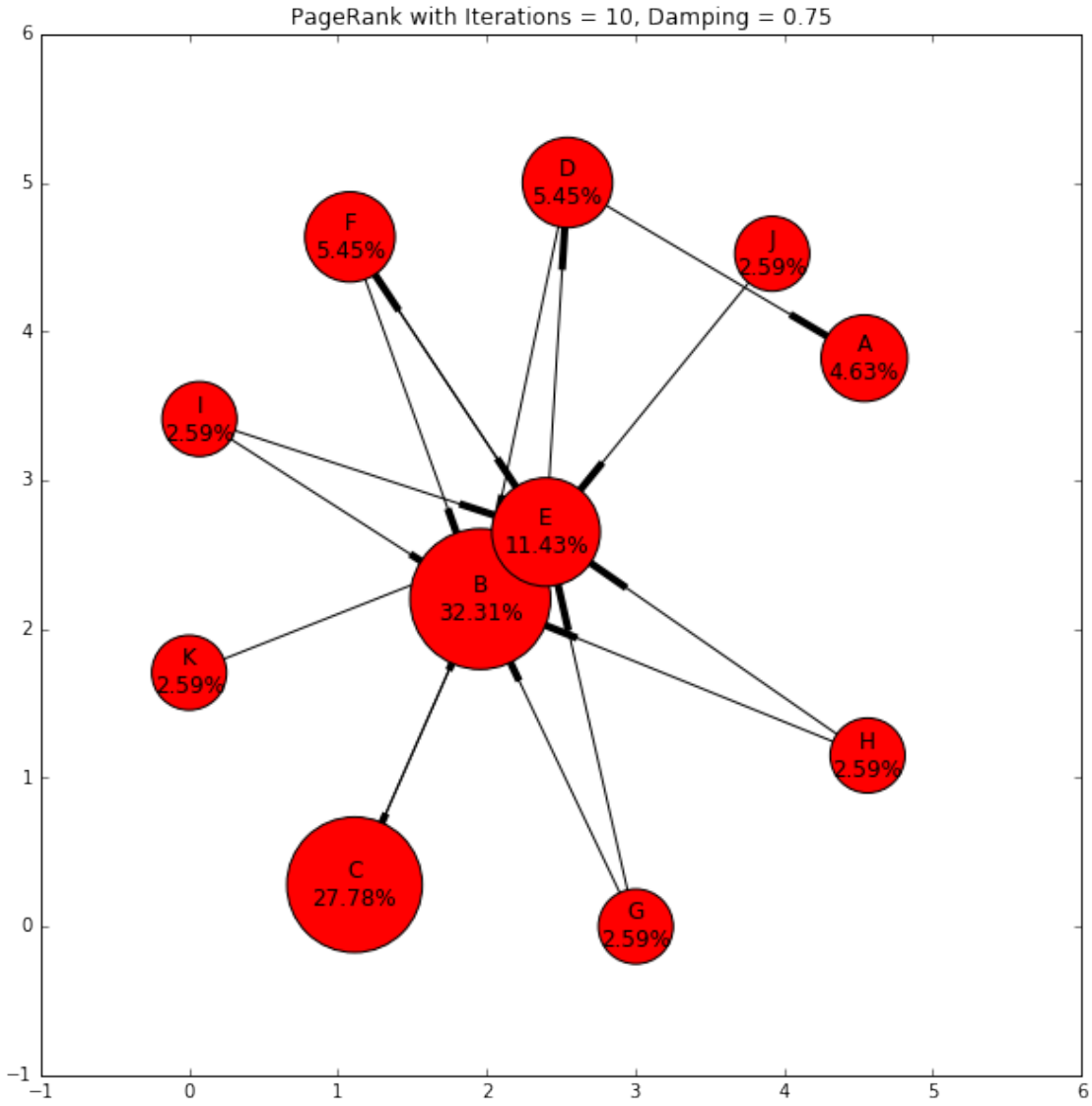
In [58]:
```
import networkx as nx
import ast
import matplotlib.pyplot as plt
%matplotlib inline
#This codes builds weighted network graphs for the PageRank results on the toy data set


# The values of Alpha we want to build plots for
alphas = [0.0,0.25,0.5,0.75,.85,1.0]

def makePageRankGraph(fileName,runnerType,iterations):
    for i,alpha in enumerate(alphas):
        DG = nx.DiGraph()
        node_sizes=[]
        labels={}

        pageRank(fileName,"",runnerType,iterations,alpha)
        with open(fileName + "_pagerank","r") as myfile:
            for line in myfile.readlines():

                #get all node information
                node, values = line.strip().split("\t")
                node = node.strip('\"')
                [name,neighbors,score] = eval(values)
                #insert into networkx graph structure
                DG.add_node(node,size=(score**.25+5)*10000)
                if neighbors:
                    for neighbor in neighbors:
                        DG.add_edge(node,neighbor)
                labels[node]= str(node) + "\n" + str(round(score*100,2))+"%"
        pos=nx.spring_layout(DG,scale=5)
        plt.figure(i+1,figsize=(10,10))
        nx.draw_networkx(DG, pos,
                         node_size=[n[1]["size"] for n in DG.nodes(True)],
                         labels=labels, with_labels=True,
                         arrows=True)

        plt.title('PageRank with Iterations = '+str(iterations) + ', Damping = '+str(alpha))

    plt.show()
```

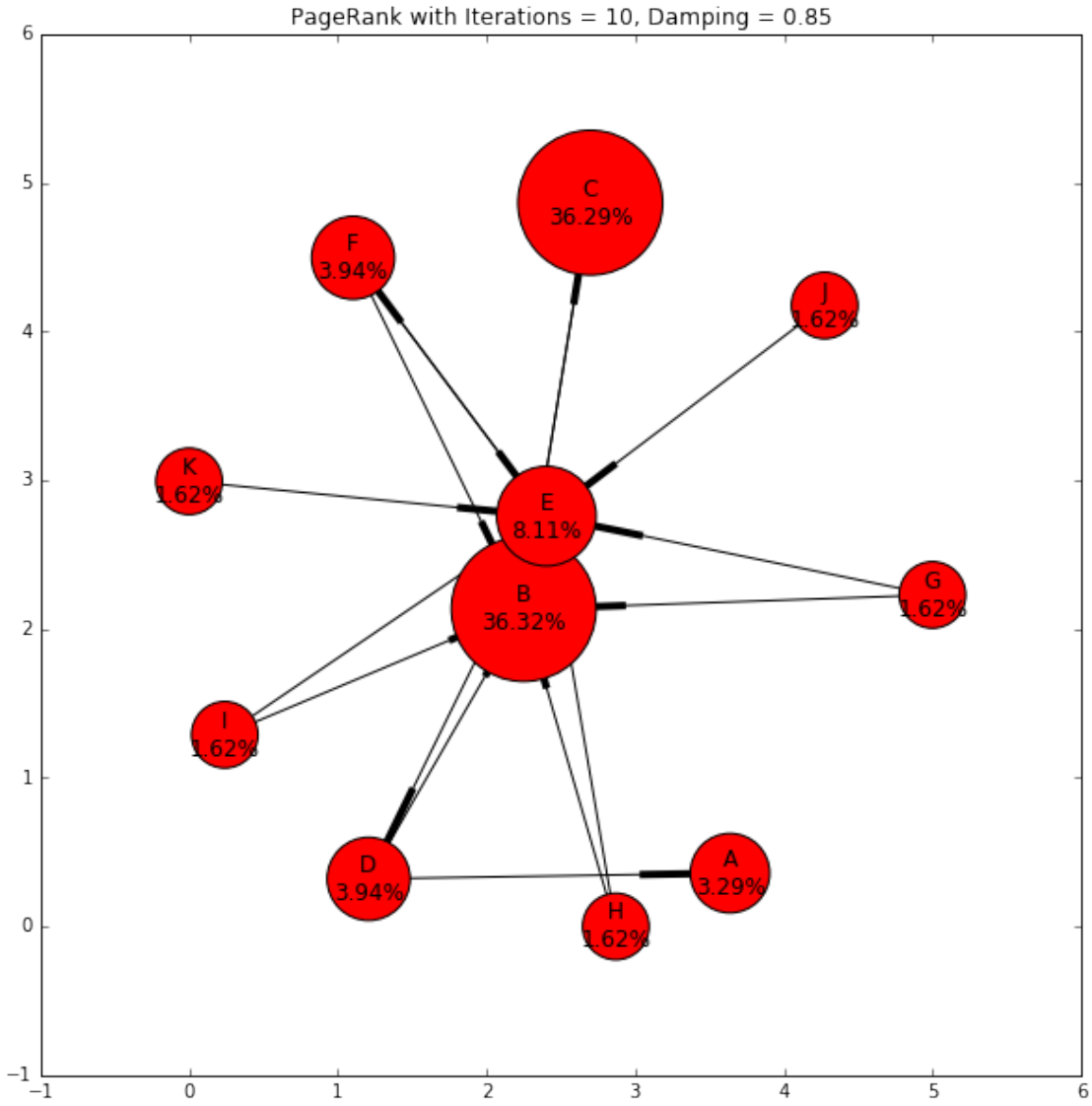
In [256]:
```
makePageRankGraph("Data/PageRank-test.txt","inline",10)
```

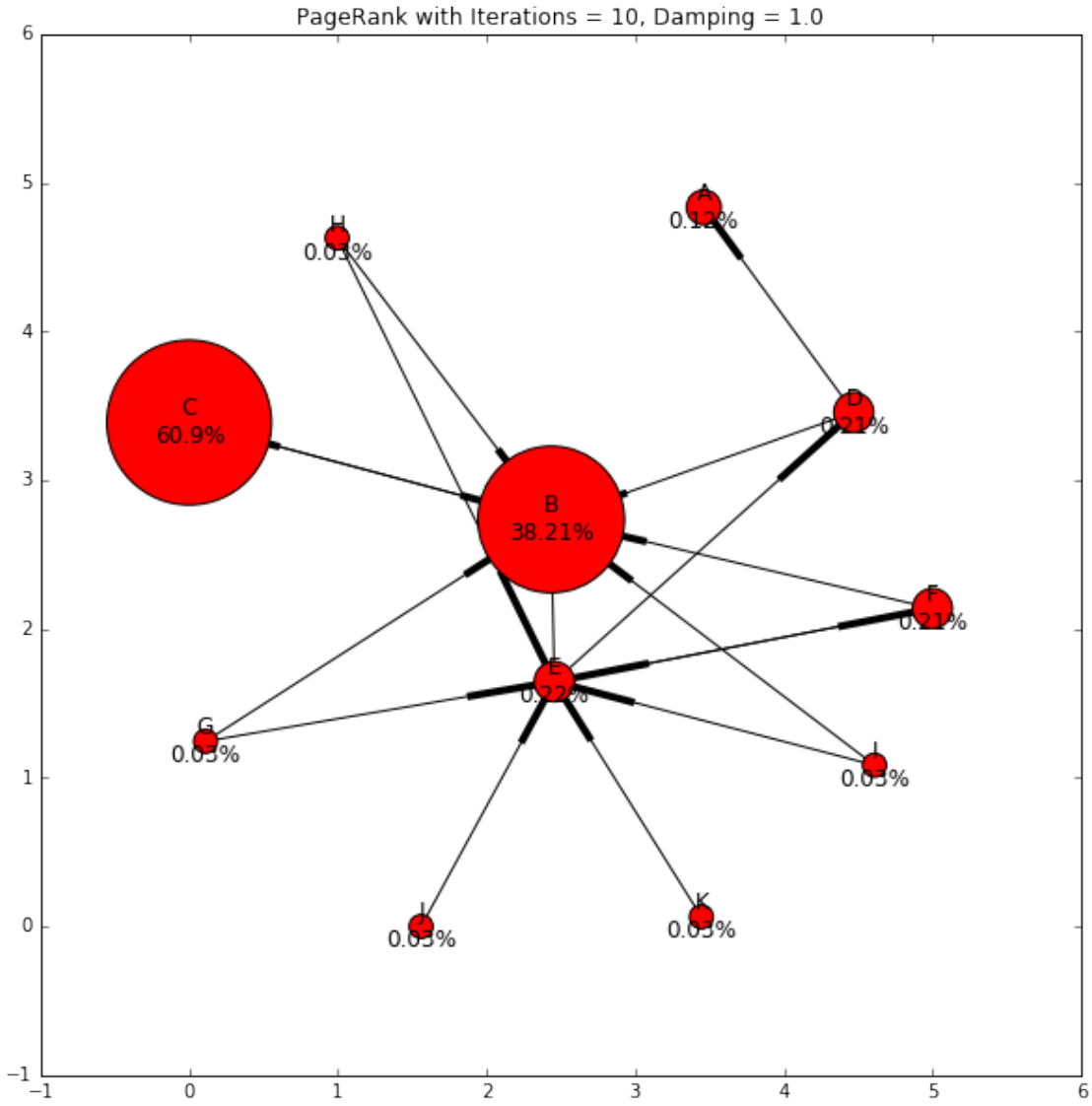

PageRank with Iterations = 10, Damping = 0.0
I
9.09%
H
9.09%
F
9.09%
E
9.09%
B
9.09%
K
9.09%
G
9.09%
D
9.09%
C
9.09%
J
9.09%
A
9.09%
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6


PageRank with Iterations = 10, Damping = 0.25
I
7.0%
H
7.0%
J
7.0%
F
8.18%
B
15.57%
E
14.15%
D
8.18%
A
8.02%
G
7.0%
C
10.89%
K
7.0%
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6


PageRank with Iterations = 10, Damping = 0.5
H
4.85%
F
7.38%
A
6.69%
I
4.85%
B
22.84%
D
7.38%
E
15.18%
G
4.85%
K
4.85%
C
16.28%
J
4.85%


PageRank with Iterations = 10, Damping = 0.75
D
5.45%
F
5.45%
J
2.59%
A
4.63%
I
2.59%
E
11.43%
B
32.31%
K
2.59%
H
2.59%
C
27.78%
G
2.59%


PageRank with Iterations = 10, Damping = 0.85
C
36.29%
F
3.94%
J
1.62%
K
1.62%
E
8.11%
B
36.32%
G
1.62%
I
1.62%
D
3.94%
A
3.29%
H
1.62%



HW 9.2 Analysis
HW 9.3: Applying PageRank to the Wikipedia hyperlinks network

- Run your PageRank implementation on the Wikipedia dataset for 5 iterations, and display the top 100 ranked nodes (with alpha = 0.85).
- Run your PageRank implementation on the Wikipedia dataset for 10 iterations, and display the top 100 ranked nodes (with teleportation factor of 0.15).
- Have the top 100 ranked pages changed? Comment on your findings.
- Plot the pagerank values for the top 100 pages resulting from the 5 iterations run. Then plot the pagerank values for the same 100 pages that resulted from the 10 iterations run.

HW 9.3 Implementation

```
In [102]: %%writefile MRJob9_3_pagerank.py

          #The wikipedia data is indexed, so I'm going to write a new MRJob to take advantage of that.
```

```
#Instead of using a separate MRJob to distribute dangling weights, I'll just emit to every noo
#The wikipedia data is also much larger, and the amount of connections is exponentially highe
#So to reduce through-put, I've also changed the job to only emit dangling nodes at the end o
# rather than as soon as they come in.
#Finally, this reducer emits the log of each node's PageRank score, rather than the PageRank s
#This is to ensure that python doesn't convert the score to scientific notatioon, which throws
# MRJob sorting


from mrjob.job import MRJob
from mrjob.step import MRStep
from math import log, exp
import sys

class MRJob9_3_pagerank(MRJob):
    def steps(self):
        step =  [MRStep(
                mapper_init = self.mapper_init,
                mapper=self.mapper,
                mapper_final = self.mapper_final,
                reducer=self.reducer
                ,jobconf =  {
            "mapred.map.tasks":20,
            "mapred.reduce.tasks":10,
            'stream.num.map.output.key.fields': '1'

        }
                    )
                ]
        sortstep = [MRStep(mapper=self.mapper_sort_filter,
                    reducer_init=self.reducer_sort_filter_init,
                    reducer=self.reducer_sort_filter
                    ,jobconf =  {
            "mapred.reduce.tasks":1,
            'stream.num.map.output.key.fields': '1'

        })]
        return [MRStep(mapper=self.mapper_preprocess,reducer=self.reducer_preprocess
            ,jobconf =  {
            "mapred.map.tasks":20,
            "mapred.reduce.tasks":10,
            'stream.num.map.output.key.fields': '1'

        })] + step * int(self.options.iterations) + sortstep



    def configure_options(self):
        super(MRJob9_3_pagerank, self).configure_options()
        self.add_passthrough_option('--N', default="1")
        self.add_passthrough_option('--alpha', default=".15")
        self.add_passthrough_option('--iterations', default="4")
        self.add_passthrough_option('--sortlimit', default="4")
```

```
#Here we create a list of all the nodes and join nodes with their names from
# an index file if one is present.
def mapper_preprocess(self, _, line):
    fields = line.strip().split('\t')
    #Check if line comes from a graph data file
    # or an index file.  You can tell if the second value in the line
    # is a dictionary or a number
    neighbors = eval(fields[1])
    if(str(type(neighbors)) == "<type 'dict'>"):
        id = fields[0]
        # if it turns out that this was actually a line from an index file
        # and the node name was just all numeric, then discard
        for neighbor in neighbors:
            yield neighbor, None
        yield id, neighbors
    else:
        # if line is from an index file, yield node id and name
        id = fields[1]
        name = fields[0]
        yield id, name


#This reducer mergers the node information with the naame information from the index file
#The reducer also initializes the weights for each node to 1/N
def reducer_preprocess(self,id,values):
    name = id
    id = str(id)
    neighbors = {}
    weight = 1/float(self.options.N)
    for value in values:
        if(str(type(value)) == "<type 'dict'>"):
            neighbors = value
        else:
            if(value):
                name = value
    #Yield complete node at end.
    #Note that we use log(weight) here to prevent python from converting the
    # low decimals into scientific notation.
    yield id, [name,neighbors, log(weight)]

#Rather than emit dangling nodes each time we come across them,
# this mapper accummulates the dangling mass and emits the total at the end.
#This reduces the number key value pairs the job needs to process
def mapper_init(self):
    self.dangling = 0

def mapper(self, node, values):
    name,neighbors, weight = values
    #Convert log weights back into regular weights with exp function
    weight = exp(weight)
    node = str(node)
    yield node, [name,neighbors,log(weight)]
    if len(neighbors) > 0:
        total_weights = sum([int(w) for _, w in neighbors.items()])
```

```
                for neighbor,link_weight in neighbors.items():
                    yield neighbor, ['',None,log(float(link_weight)*weight/float(total_weights))]
            else:
                self.dangling += weight

        #Yield total dangling mass
        def mapper_final(self):
            if(self.dangling > 0):
                #Since we know that data is indexed, we can emit to every node by just ierating fi
                for i in range(int(self.options.N)):
                        yield str(i+1),['',None,log(self.dangling/float(self.options.N))]

        #This reducer sums up the weights for each node, including the dangling masss,
        # and then applies the damping adjustments.
        #The reducer also merges weight information with node information like name and neighbors
        def reducer(self,node, values):
            name = ""
            neighbors = {}
            newweight= float(0)
            for value in values:
                if (value[0] == ''):
                    newweight += exp(value[2])
                else:
                    name,neighbors,_= value
            a = float(self.options.alpha)
            newweight = (1-a)/float(self.options.N) + a*newweight
            yield node, [name, neighbors,log(newweight)]


        #These next three functions will get the top 100 (or more) values.
        #This type of filtering technically isn't functionally neccessary,
        # but it makes the results file smaller and more managable.
        def mapper_sort_filter(self,node,value):
            name,neighbors, weight = value
            # Multiply weight by negative one to get highest to lowest ordering
            yield weight*-1, name

        #Initalize the limit to the number of items we want to return
        #This is why we have to use a single reducer for this task
        def reducer_sort_filter_init(self):
            self.limit = int(self.options.sortlimit)

        def reducer_sort_filter(self,weight,names):
            weight = exp(weight*-1)
            for name in names:
                if(self.limit > 0):
                    yield name.encode('utf-8').strip(),weight
                    self.limit -= 1

    if __name__ == '__main__':
        MRJob9_3_pagerank.run()
```

```
Overwriting MRJob9_3_pagerank.py
```

```
In [109]: %reload_ext autoreload
```

```
%autoreload 2
from __future__ import division
from MRJob9_1_findnodes import MRJob9_1_findnodes

from MRJob9_3_pagerank import MRJob9_3_pagerank
import numpy,sys

def pageRank(fileName,indexFile,runnerType,iterations,alpha,sortlimit):

    if False:
        mr_job = MRJob9_1_findnodes(args=[
            fileName, '-r', runnerType
                    ,'--cleanup','ALL'
    ])
        nodes = 0

        #Run find nodes MRJob
        with mr_job.make_runner() as runner:
            runner.run()
            for line in runner.stream_output():
                out = mr_job.parse_output_line(line)
                nodes += out[1]
    mr_job = MRJob9_3_pagerank(args=[
        fileName,indexFile, '-r', runnerType ,
        '--N',str(15192277),
        '--iterations',str(iterations)
        ,'--alpha',str(alpha),
        '--sortlimit',str(sortlimit)
        ,'--cleanup','ALL'
        ,'--output-dir','results/9.3/'+str(iterations)
        ])

    with mr_job.make_runner() as runner:
        runner.run()
        with open(fileName+"_pagerank"+str(iterations),"w") as myfile:
            total_score  = 0
            for line in runner.stream_output():
                myfile.write(line.replace('\"',''))
```

First run a unit test using PageRank-test_indexed. First create an index file for the toy dataset.

```
In [104]: %%writefile toy_indices.txt
          A        1
          B        2
          C        3
          D        4
          E        5
          F        6
          G        7
          H        8
          I        9
          J        10
          K        11

Overwriting toy_indices.txt
```

```
In [105]: !hdfs dfs -rm -r -f -skipTrash results/9.3/20
          !hdfs dfs -mkdir -p results/9.3/20
          pageRank("Data/PageRank-test_indexed.txt","toy_indices.txt","inline",20,.85,10)
          !cat "Data/PageRank-test_indexed.txt_pagerank20"

Deleted results/9.3/20
B       0.38031552249156747
C       0.3469932932917439
E       0.08088637369406643
D       0.03908760216841475
F       0.03908760216841475
A       0.03278187703400854
G       0.01616954583035677
H       0.01616954583035677
I       0.01616954583035677
J       0.01616954583035677
```

1.1.1 Now test on Wikipedia data

```
In [111]: #Code to run driver and get top Wikipedia page ranks after 5 iterations.

          after5 = []
          !hdfs dfs -rm -r -f -skipTrash results/9.3/5
          !hdfs dfs -mkdir -p results/9.3/

          pageRank("Data/wikipedia/all-pages-indexed-out.txt","Data/wikipedia/indices.txt",
                   "hadoop",1,.85,100)
          with open("Data/wikipedia/all-pages-indexed-out.txt_pagerank5") as myfile:
              for line in myfile.readlines():
                  name, value = line.strip().split("\t")
                  value = float(value)
                  #round value to 6 decimal places for display
                  value = round(value,6)
                  after5.append([name,value])


Deleted results/9.3/5

In [112]: #Code to run driver and get top Wikipedia page ranks after 10 iterations.
          after10 = []
          !hdfs dfs -rm -r -f -skipTrash results/9.3/10
          !hdfs dfs -mkdir -p results/9.3/

          #Since we need to do a join with the 5 iterations data, we need to make sure that all the
          # pages in the top 100 after 5 iterations are present in the data from the page rank after 10
          # So we'll need to get more than just the top 100 results from the 10 iterations.
          # There are 15 million pages on Wikipedia, but PageRank is also pretty quick to converge,
          # so I think I can get away with just taking the top 500 pages.

          pageRank("Data/wikipedia/all-pages-indexed-out.txt","Data/wikipedia/indices.txt",
                   "hadoop",10,.85,500)
          with open("Data/wikipedia/all-pages-indexed-out.txt_pagerank10") as myfile:
              for line in myfile.readlines():
                  name, value = line.strip().split("\t")
                  value = float(value)
```

```
                value = round(value,6)
                after10.append([name,value])
```

```
Deleted results/9.3/10
```

1.1.2 Graphs and charts

```
In [113]: import matplotlib.pyplot as plt
          %matplotlib inline

          rowLabels=list(range(1, len(after10[:100])+1))
          colLabels = ["Label","Score"]
          plt.rcParams["figure.figsize"] = (20,45)
          plt.axis('off')
          myTable = plt.table(cellText=after5[:101], rowLabels = rowLabels, colLabels=colLabels, loc='c
          plt.title('PageRank with 5 Iterations',fontsize=23)
          myTable.auto_set_font_size(False)
          myTable.set_fontsize(16)
          myTable.scale(1, 2)
```

PageRank with 5 Iterations

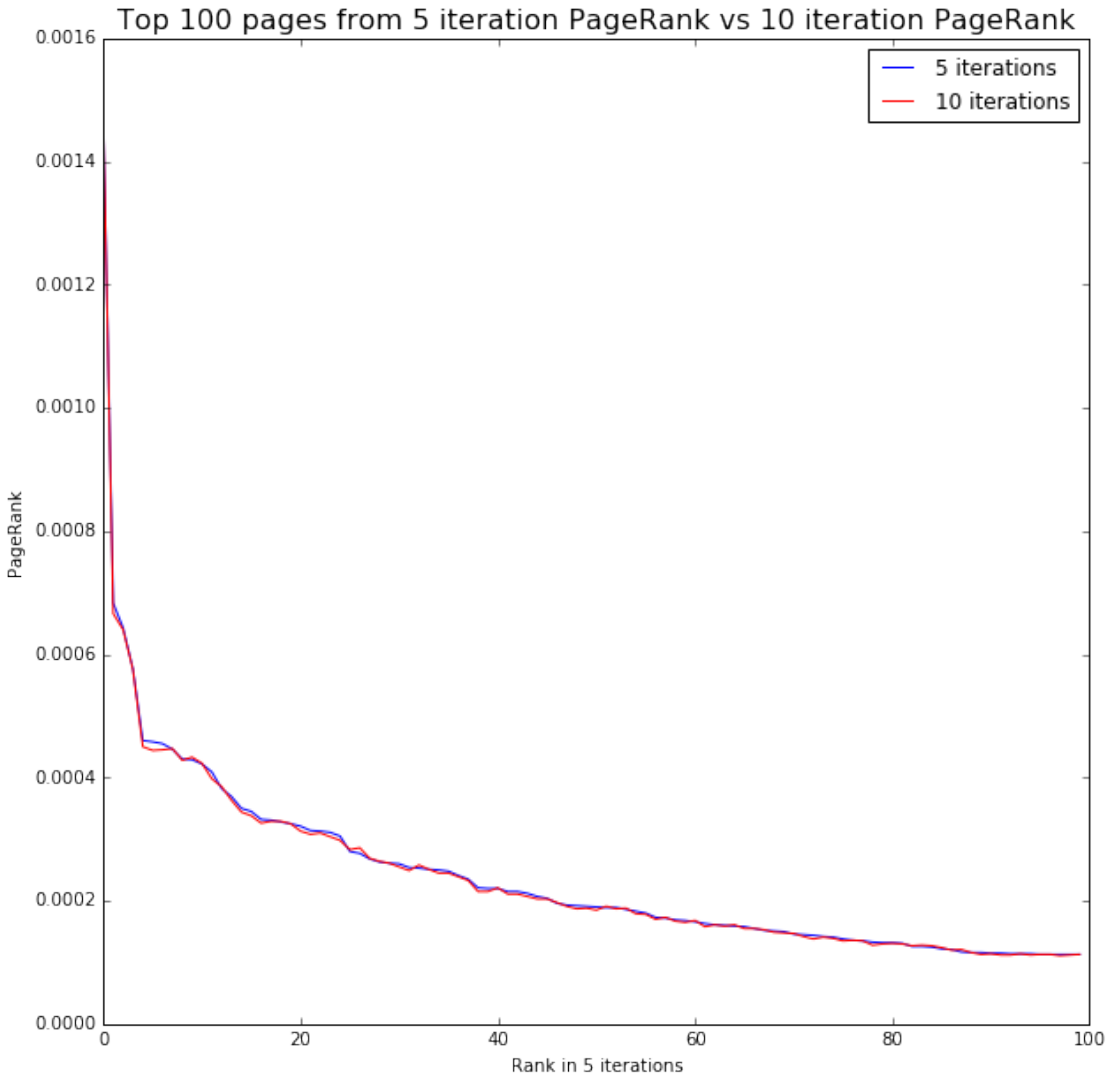
	Label	Score
1	United States	0.001461
2	Animal	0.000683
3	France	0.000643
4	Germany	0.000576
5	Arthropod	0.00046
6	List of sovereign states	0.000458
7	Insect	0.000455
8	Canada	0.000446
9	India	0.00043
10	United Kingdom	0.000429
11	England	0.000422
12	Iran	0.000409
13	World War II	0.000382
14	Poland	0.000369
15	village	0.00035
16	Countries of the world	0.000345
17	List of countries	0.000332
18	Japan	0.000331
19	Italy	0.000328
20	Australia	0.000325
21	Voivodeships of Poland	0.000321
22	Lepidoptera	0.000314
23	National Register of Historic Places	0.000313
24	Powiat	0.000311
25	Gmina	0.000305
26	London	0.00028
27	The New York Times	0.000277
28	English language	0.000268
29	China	0.000263
30	Russia	0.000261
31	Departments of France	0.00026
32	Communes of France	0.000254
33	New York City	0.000253
34	Spain	0.000251
35	moth	0.00025
36	Brazil	0.000248
37	Association football	0.000241
38	association football	0.000235
39	Counties of Iran	0.000221
40	Provinces of Iran	0.00022
41	California	0.00022
42	Romania	0.000215
43	Central European Time	0.000215
44	Bakhsh	0.000212
45	Rural Districts of Iran	0.000207
46	Sweden	0.000204
47	Netherlands	0.000196
48	Private Use Areas	0.000193
49	Iran Standard Time	0.000192
50	Central European Summer Time	0.000191
51	AllMusic	0.00019
52	World War I	0.000189
53	Mexico	0.000189
54	New York	0.000186
55	Iran Daylight Time	0.000183
56	Hangul	0.00018
57	gene	0.000173
58	Scotland	0.000172
59	Norway	0.000169
60	Allmusic	0.000168
61	Soviet Union	0.000166
62	Plant	0.000163
63	New Zealand	0.000161
64	Turkey	0.00016
65	Paris	0.000159
66	Geographic Names Information System	0.000158
67	Switzerland	0.000155
68	Los Angeles	0.000152
69	Romanize	0.000151
70	United States Census Bureau	0.00015
71	Europe	0.000146
72	Angiosperms	0.000145
73	Flowering plant	0.000144
74	South Africa	0.000142
75	census	0.000141
76	protein	0.000138
77	Austria	0.000136
78	U.S. state	0.000135
79	Chordate	0.000133
80	Political divisions of the United States	0.000132
81	Argentina	0.000132
82	population density	0.000131
83	Belgium	0.000126
84	Catholic Church	0.000126
85	BBC	0.000125
86	Chicago	0.000122
87	Pakistan	0.000121
88	Washington, D.C.	0.000117
89	Finland	0.000116
90	genus	0.000116
91	Czech Republic	0.000115
92	species	0.000115
93	Eastern European Time	0.000114
94	Ontario	0.000114
95	football (soccer)	0.000114
96	Philippines	0.000113
97	Denmark	0.000113
98	Eudicots	0.000113
99	Hungary	0.000113
100	Greece	0.000113

In [114]:
```
import matplotlib.pyplot as plt
%matplotlib inline

rowLabels=list(range(1, len(after10[:100])+1))
colLabels = ["Label","Score"]
plt.rcParams["figure.figsize"] = (20,45)
plt.axis('off')
myTable = plt.table(cellText=after10[:100], rowLabels = rowLabels, colLabels=colLabels, loc='
plt.title('PageRank with 10 Iterations',fontsize=23)
myTable.auto_set_font_size(False)
myTable.set_fontsize(16)
myTable.scale(1, 2)
```

PageRank with 10 Iterations

	Label	Score
1	United States	0.001461
2	Animal	0.000666
3	France	0.00064
4	Germany	0.000575
5	Arthropod	0.00045
6	Canada	0.000447
7	Insect	0.000445
8	List of sovereign states	0.000444
9	United Kingdom	0.000433
10	India	0.000428
11	England	0.000423
12	Iran	0.000398
13	World War II	0.000385
14	Poland	0.000363
15	village	0.000344
16	Countries of the world	0.000338
17	Japan	0.000329
18	Italy	0.000329
19	List of countries	0.000326
20	Australia	0.000325
21	Voivodeships of Poland	0.000313
22	National Register of Historic Places	0.00031
23	Lepidoptera	0.000308
24	Powiat	0.000304
25	Gmina	0.000298
26	The New York Times	0.000286
27	London	0.000283
28	English language	0.000269
29	China	0.000264
30	Russia	0.000261
31	New York City	0.000258
32	Departments of France	0.000255
33	Spain	0.000251
34	Communes of France	0.000249
35	moth	0.000245
36	Brazil	0.000245
37	Association football	0.000239
38	association football	0.000233
39	California	0.000221
40	Counties of Iran	0.000215
41	Provinces of Iran	0.000215
42	Central European Time	0.000211
43	Romania	0.000211
44	Bakhsh	0.000207
45	Sweden	0.000203
46	Rural Districts of Iran	0.000203
47	Netherlands	0.000197
48	Private Use Areas	0.000191
49	World War I	0.000191
50	Central European Summer Time	0.000188
51	New York	0.000188
52	Mexico	0.000187
53	Iran Standard Time	0.000187
54	AllMusic	0.000185
55	Iran Daylight Time	0.000179
56	Hangul	0.000178
57	Scotland	0.000173
58	gene	0.00017
59	Soviet Union	0.000168
60	Norway	0.000167
61	Allmusic	0.000165
62	Paris	0.000161
63	New Zealand	0.000161
64	Turkey	0.000159
65	Plant	0.000158
66	Geographic Names Information System	0.000155
67	Switzerland	0.000155
68	Los Angeles	0.000153
69	Romanize	0.000149
70	United States Census Bureau	0.000148
71	Europe	0.000147
72	Angiosperms	0.000142
73	South Africa	0.000141
74	census	0.000139
75	Flowering plant	0.000138
76	Austria	0.000136
77	protein	0.000135
78	U.S. state	0.000135
79	Argentina	0.000131
80	Political divisions of the United States	0.00013
81	population density	0.00013
82	Catholic Church	0.000128
83	Chordate	0.000128
84	BBC	0.000127
85	Belgium	0.000127
86	Chicago	0.000124
87	Washington, D.C.	0.000121
88	Pakistan	0.00012
89	Finland	0.000116
90	The Guardian	0.000114
91	Latin	0.000114
92	Ontario	0.000114
93	Czech Republic	0.000114
94	Philippines	0.000113
95	Denmark	0.000113
96	Greece	0.000113
97	genus	0.000113
98	football (soccer)	0.000112
99	Hungary	0.000112
100	Eastern European Time	0.000112

In [117]:
```
import matplotlib.pyplot as plt

def find(l, i):
    for row in l:
        if row[0] == i:
            return row[1]
    print i
    return 0

label = [row[0] for row in after5[:101]]
plt.rcParams["figure.figsize"] = (10,10)

scoreAfter5= [find(after5,l) for l in label]
scoreAfter10 =[find(after10,l) for l in label]
line5, = plt.plot(scoreAfter5, 'b')
line10, = plt.plot(scoreAfter10, 'r')
plt.legend((line5, line10), ('5 iterations', '10 iterations'))
plt.ylabel("PageRank")

plt.xlabel("Rank in 5 iterations")
plt.title('Top 100 pages from 5 iteration PageRank vs 10 iteration PageRank',fontsize=16)
x = list(range(1,101))


plt.show()
```



HW 9.3 Analysis

From the graph, you can see that there is actually not much change in the rankings of each node. After just 5 iterations, it looks like the algorithm has figured out the relative rank of each node. However, the exact scores of each node seem to change. In particular, the higher ranked nodes get higher scores and the lower ranked nodes get slightly lower scores.

HW 9.4: Topic-specific PageRank implementation using MRJob

Modify your PageRank implementation to produce a topic specific PageRank implementation, as described in:

http://www-cs-students.stanford.edu/~taherh/papers/topic-sensitive-pagerank.pdf

Note in this article that there is a special caveat to ensure that the transition matrix is irreducible. This caveat lies in footnote 3 on page 3:

```
A minor caveat: to ensure that M is irreducible when p
contains any 0 entries, nodes not reachable from nonzero
nodes in p should be removed. In practice this is not problematic.
```

and must be adhered to for convergence to be guaranteed.

Run topic specific PageRank on the following randomly generated network of 100 nodes:

s3://ucb-mids-mls-networks/randNet.txt (also available on Dropbox)

which are organized into ten topics, as described in the file:

s3://ucb-mids-mls-networks/randNet_topics.txt (also available on Dropbox)

Since there are 10 topics, your result should be 11 PageRank vectors (one for the vanilla PageRank implementation in 9.1, and one for each topic with the topic specific implementation). Print out the top ten ranking nodes and their topics for each of the 11 versions, and comment on your result. Assume a teleportation factor of 0.15 in all your analyses.

One final and important comment here: please consider the requirements for irreducibility with topic-specific PageRank. In particular, the literature ensures irreducibility by requiring that nodes not reachable from in-topic nodes be removed from the network.

This is not a small task, especially as it it must be performed separately for each of the (10) topics.

So, instead of using this method for irreducibility, please comment on why the literature's method is difficult to implement, and what what extra computation it will require.

Then for your code, please use the alternative, non-uniform damping vector:

```
vji = beta*(1/|Tj|); if node i lies in topic Tj

vji = (1-beta)*(1/(N - |Tj|)); if node i lies outside of topic Tj
```

for beta in (0,1) close to 1.

With this approach, you will not have to delete any nodes. If beta > 0.5, PageRank is topic-sensitive, and if beta < 0.5, the PageRank is anti-topic-sensitive. For any value of beta irreducibility should hold, so please try beta=0.99, and perhaps some other values locally, on the smaller networks.

HW 9.4 Implementation

In [111]:
```
%%writefile MRJob9_4_topiccounts.py

#MR Job code to count the number of nodes by topic

from mrjob.job import MRJob
from mrjob.step import MRStep

class MRJob9_4_topiccounts(MRJob):

    def steps(self):
        return [MRStep(
                    mapper=self.mapper
                    , combiner=self.reducer
                       , reducer=self.reducer
                   ,jobconf =  {
                "mapred.map.tasks":4,
                "mapred.reduce.tasks":2,
                'stream.num.map.output.key.fields': '1'
            })]

    def mapper(self, _, line):
        node,topic = line.strip().split('\t')
        yield topic, 1


    def reducer(self, topic, values):
        yield topic, sum(values)
```

```
if __name__ == '__main__':
    MRJob9_4_topiccounts.run()
```

```
Writing MRJob9_4_topiccounts.py
```

```
In [66]: %%writefile MRJob9_4_pagerank.py

#Topic sensitive version of the PageRank MRJob from 9.3.
#This job uses the pairs approach, emitting a key for each combination of topic and node.
#

from mrjob.job import MRJob
from mrjob.step import MRStep
from math import log, exp
import sys

class MRJob9_4_pagerank(MRJob):
    MRJob.SORT_VALUES = True
    SORT_VALUES = True
    def steps(self):
        step =  [MRStep( mapper_init=self.mapper_init,mapper=self.mapper
                    , reducer_init=self.reducer_init, reducer=self.reducer
                ,jobconf =  {
            "mapred.map.tasks":4,
            "mapred.reduce.tasks":2,

        })]

        # The sort step here is now parallelizable because we need to do a sort for each topic
        sortstep = [ MRStep(mapper=self.mapper_sort_filter,
                    reducer=self.reducer_sort_filter,
                    jobconf =  {
            "mapred.reduce.tasks":2,
            'stream.num.map.output.key.fields': '1'

        })]
        return [MRStep(mapper=self.mapper_preprocess,reducer_init=self.reducer_preprocess_init
            ,jobconf =  {
            "mapred.map.tasks":4,
            "mapred.reduce.tasks":2,
            'stream.num.map.output.key.fields': '1'
        })] + step * int(self.options.iterations) + sortstep



    def configure_options(self):
        super(MRJob9_4_pagerank, self).configure_options()
        self.add_passthrough_option('--countString', default="1")
        self.add_passthrough_option('--alpha', default=".15")
        self.add_passthrough_option('--beta', default=".99")
        self.add_passthrough_option('--iterations', default="4")


    # Join records with a topics file instead of an index file
```

```
def mapper_preprocess(self, _, line):
    fields = line.strip().split('\t')
    neighbors = eval(fields[1])
    if(str(type(neighbors)) == "<type 'dict'>"):
        id = fields[0]
        yield id, neighbors
    else:
        id = fields[0]
        topic = fields[1]
        yield id, topic


#Convert the topic_counts string into an array
# and then sum up topic counts to get total number of nodes.
#This init step will appear several times in the MRJob as we need count information
# often in the PageRank algorithm.
def reducer_preprocess_init(self):
    self.topic_counts = map(int,self.options.countString.split(","))
    self.N = sum(self.topic_counts)

#The reducer here will join the topic information with the node information.
#Then, the reducer will emit each combination of node and topic
def reducer_preprocess(self,id,values):
    id = str(id)
    topic = 0
    neighbors = {}
    weight = 1/float(self.N)
    for value in values:
        if(str(type(value)) == "<type 'dict'>"):
            neighbors = value.keys()
        else:
            topic = value
    for t,_ in enumerate(self.topic_counts):
        yield str(t+1)+"\t"+str(id), [topic,neighbors, log(weight)]
    yield "0\t"+str(id), [topic,neighbors, log(weight)]


def mapper_init(self):
    self.topic_counts = map(int,self.options.countString.split(","))
    self.N = sum(self.topic_counts)

def mapper(self, node, values):
    topic,neighbors, weight = values
    weight = exp(weight)
    node = str(node)
    neighbors = map(str,neighbors)
    yield node, [topic,neighbors,log(weight)]
    t,id = node.split("\t")
    if neighbors:
        for neighbor in neighbors:
            yield t+"\t"+neighbor, ['',None,
                                    log(weight/len(neighbors))]
    else:
        for i in range(int(self.N)):
```

```
            yield t+"\t"+str(i+1), ['',None,
                                    log(weight/float(self.N))]


def reducer_init(self):
    self.topic_counts = map(int,self.options.countString.split(","))
    self.N = sum(self.topic_counts)


#The reducer here is similar to the one from 9.3, but uses a different weighting
# algorithm if the topic in the composite key matches the node topic.
def reducer(self,node, values):
    topic = ""
    t,id = node.split("\t")
    neighbors = {}
    newweight= float(0)
    for value in values:
        if (value[0] == ''):
            newweight += exp(value[2])
        else:
            topic,neighbors,_= value

    a = float(self.options.alpha)
    b = float(self.options.beta)

    #Use different damping formula depending on whether node matches topic or
    # whether topic is null (0) for generic PageRank
    if (t == topic):
        newweight = b*(1-a)/float(self.topic_counts[int(topic)-1]) + a*float( newweight)
    elif t=="0":
        newweight = (1-a)/float(self.N) + a*float( newweight)
    else:
        newweight = (1-b)*(1-a)/float(self.N - self.topic_counts[int(t)-1]) + a*float( new

    yield node, [topic, neighbors,log(newweight)]



def mapper_sort_filter(self,node,value):
    t, id = node.split("\t")
    topic,neighbors, weight = value
    #Invert order so get highest values instead of lowest values
    yield t, (weight,id)

def reducer_sort_filter(self,t,values):

    # This part is not generalizable nor scalable, but it saves me
    # from writing a special table or charting function if I just have the reducer
    # yield proper titles and spacing and hardcode a limit of 10
    yield "",""
    if (t =="0"):
        yield "Top 10 No Topic:", ""
    else:
        yield "Top 10 in Topic: " + t, ""
```

```
                self.limit = 10
                for value in values:
                    if self.limit > 0:
                        self.limit -= 1
                        weight, name = value
                        weight = exp(weight)
        if __name__ == '__main__':
            MRJob9_4_pagerank.run()
```

```
Overwriting MRJob9_4_pagerank.py
```

In [67]:
```
%reload_ext autoreload
%autoreload 2
from __future__ import division
from MRJob9_4_topiccounts import MRJob9_4_topiccounts

from MRJob9_4_pagerank import MRJob9_4_pagerank
import numpy

#Driver code for topic sensitive PageRank

def pageRank(fileName,topicsFile,runnerType,iterations,alpha):
    mr_job = MRJob9_4_topiccounts(args=[
        topicsFile, '-r', runnerType ])
    topic_counts =[0] * 10
    #Run topic counts MRJob
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            out = mr_job.parse_output_line(line)
            topic_counts[int(out[0])-1]=  str(out[1])


    topic_counts = ",".join(topic_counts)
    mr_job = MRJob9_4_pagerank(args=[
        fileName,topicsFile, '-r', runnerType ,
        '--countString',str(topic_counts),
        '--iterations',str(iterations)
        ,'--alpha',str(alpha),
        '--beta',str(.99)
        ,'--cleanup','ALL'

        ])

    #Write results to file
    with mr_job.make_runner() as runner:
        runner.run()
        with open(fileName+"_pagerank","w") as myfile:
            total_score  = 0
            for line in runner.stream_output():
                myfile.write(line.replace('\"',''))
```

In [68]:
```
pageRank("Data/randNet.txt","Data/randNet_topics.txt","inline",10,.85)
!cat "Data/randNet.txt_pagerank"
```

```
Top 10 in Topic: 5
99        0.028963266500299605
90        0.028344984671495925
88        0.027168703255116916
51        0.02683079533282123
45        0.025553308477928555
5        0.02391995502823426
34        0.023909744584958766
4        0.023363274382490386
80        0.022839620864129398
100        0.016741485269746205

Top 10 in Topic: 6
13        0.03457157087753228
56        0.03285390671450889
37        0.031777155876364656
11        0.03133964819851934
69        0.03012079634490478
23        0.028348902674012147
15        0.017242580581008003
85        0.016989438344123805
52        0.01660216963107064
74        0.015461058978408833

Top 10 in Topic: 7
85        0.026794227495348943
25        0.026608778613501596
28        0.024827245457770576
53        0.024767654194382913
35        0.024200828678767147
97        0.02339501397933647
47        0.022865013665705175
55        0.0225608540639878
30        0.022134839131015587
50        0.02008908019623937

Top 10 in Topic: 8
100        0.03286603046093977
61        0.027858570822448362
39        0.027195521505354085
8        0.027153099743645284
62        0.025346793563654338
87        0.025298416024840103
6        0.02350685265428699
54        0.02289173824898396
18        0.020622660245957835
9        0.015379799633741314

Top 10 in Topic: 9
94        0.030198889934084216
14        0.02949519768424192
42        0.029197862681776883
21        0.028399569111395757
57        0.027461793981146128
```

```
96        0.02626094967931082
24        0.025773074359528945
63        0.01716149206692867
61        0.016377328880887897
74        0.014277712975620789

Top 10 No Topic:
15        0.016356322834491864
74        0.015969185600654787
63        0.01577094193547124
100        0.01537650804799067
85        0.015178520144363121
9        0.015032514700360494
58        0.014828157341893347
71        0.014490864725493345
61        0.014407046165411214
52        0.014311040206523361

Top 10 in Topic: 1
32        0.020645898325225
77        0.02054756962678735
52        0.019754313100732442
92        0.01952923824626395
10        0.018565525448276273
27        0.01852253982706892
85        0.017840510571832245
98        0.017692389508390933
46        0.017514128674978233
74        0.016028121317860765

Top 10 in Topic: 10
74        0.026331832014712158
17        0.023590536112239317
49        0.02357426772879787
95        0.02062934049608113
7        0.019913805171583924
43        0.01936417492785354
68        0.019038271865623197
48        0.019005045680706458
1        0.018998083854615256
3        0.018639556040770725

Top 10 in Topic: 2
58        0.03084746002637323
71        0.0296652433249979
9        0.029296846892829638
73        0.028914805418738844
12        0.026888935387018635
59        0.025799681887661265
75        0.024849600531229584
82        0.022858211891270384
52        0.016322098538761664
17        0.015158672198490158
```

```
Top 10 in Topic: 3
15        0.03152906839734724
70        0.027076597693858763
86        0.026527964044297938
91        0.024463316416756184
66        0.02414852788471486
2        0.02370508724370705
31        0.02276711141145176
40        0.022178501156581275
20        0.019745051424447248
74        0.015899972928926757

Top 10 in Topic: 4
63        0.026202043207444136
83        0.021760079633412525
65        0.020623779777070753
78        0.02021010115222984
41        0.01990845975051759
84        0.019519880071332197
79        0.01842865056593551
38        0.017515429366835658
15        0.01675216353937529
72        0.016694729499179148
```

HW 9.4 Analysis

The literature's method would be difficult to implement because of the irreducibility requirement to delete nodes that aren't reachable from in-topic nodes. This means you'll need to perform the PageRank with a different graph for each topic.

Here is how to implement this type of algorithm using the the pairs type of MRJob used in 9.4. In the preprocessing step for each topic T, initialize only the nodes that are in topic T. Then in the main pagerank step, these initial nodes will end up distributing their weights to all reachable nodes after enough iterations. Since the weights are not initialized uniformly, you'll need many more iterations to reach convergence.

However, the teleportation and damping steps would be more difficult as the algorithm would not be able to take advantage of indexed data and would need to use a special, 1 reducer task job for damping and teleportation, similar to what was used in 9.1

——- END OF HWK 9 ———