DATASCI W261: Machine Learning at Scale

Nick Hamlin (nickhamlin@gmail.com)
Tigi Thomas (tgthomas@berkeley.edu)
Rock Baek (rockb1017@gmail.com)
Hussein Danish (husseindanish@gmail.com)

Time of Submission: 11:20 AM EST, Saturday, March 19, 2016
W261-3, Spring 2016
Week 9 Homework

Submission Notes:

- For each problem, we've included a summary of the question as posed in the instructions. In many cases, we have not included the full text to keep the final submission as uncluttered as possible. For reference, we've included a link to the original instructions in the "Useful Reference" below.
- Some aspects of this notebook don't always render nicely into PDF form. In these situations, please reference the complete rendered notebook on Github (https://github.com/nickhamlin/mids_261_homework/blob/master/HW9/MIDS-W261-2015-HWK-Week09-Hamlin-Thomas-Baek-Danish.ipynb)

Useful References and Notebook Setup:

- **Original Assignment Instructions (https://www.dropbox.com/s/wp4cz1e0bif1k76/HW9-Assignment.txt?dl=0)**
- Raw data on Dropbox (https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0)
- PageRank in Wikipedia (https://en.wikipedia.org/wiki/PageRank)
- Topic-Specific PageRank (http://www-cs-students.stanford.edu/~taherh/papers/topic-sensitive-pagerank.pdf)

In [1]:
```
#Use this to make sure we reload the MrJob code when we make changes
%load_ext autoreload
%autoreload 2
#Render matplotlib charts in notebook
%matplotlib inline

#Import some modules we know we'll use frequently
import numpy as np
import pylab as plt
```

In [86]:
```
#Use this line of code to kick off a persistent cluster
!python -m mrjob.tools.emr.create_job_flow '--conf-path' 'mrjob.conf'
```

```
creating new scratch bucket mrjob-67a1f4bb719f27a3
using s3://mrjob-67a1f4bb719f27a3/tmp/ as our scratch dir on S3
Creating persistent job flow to run several jobs in...
creating tmp directory /var/folders/rz/drh189k95919thyy3gs3tq400000g
n/T/no_script.nicholashamlin.20160318.225015.836477
writing master bootstrap script to /var/folders/rz/drh189k95919thyy3
gs3tq400000gn/T/no_script.nicholashamlin.20160318.225015.836477/b.py
creating S3 bucket 'mrjob-67a1f4bb719f27a3' to use as scratch space
Copying non-input files into s3://mrjob-67a1f4bb719f27a3/tmp/no_scri
pt.nicholashamlin.20160318.225015.836477/files/
Waiting 5.0s for S3 eventual consistency
Creating Elastic MapReduce job flow
Can't access IAM API, trying default instance profile: EMR_EC2_Defau
ltRole
Can't access IAM API, trying default service role: EMR_DefaultRole
Job flow created with ID: j-2BS8O2CWL2MDJ
j-2BS8O2CWL2MDJ
```

HW 9.0

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

PageRank is an algorithm that operates on network graphs by assigning a score to each node of the graph in order to determine their importance. It was named after Larry Page. PageRank is used by Google Search to rank websites in their search engine results. It measures the importance of web pages based on the probability that a user randomly surfing the webgraph lands on a given page. While it is not the only algorithm that is used in web search, it's one of the most famous and widely implemented.

What modifications have to be made to the webgraph in order to leverage the machinery of Markov Chains to compute the steady stade distibuton?

When leveraging the machinery of Markov Chains, pages are viewed as states and the webgraph is viewed as a transition matrix. The modifications required for this are two-fold:

1. Stochasticity adjustment
    - This adjustment is made in order to deal with dangling nodes. In order for a matrix to be stochastic, the rows must sum up to 1. Therefore, instead of using 1 to indicate a transition, a value 1/n is used where n represents the non-zero elements of a row. This adjustment now allows the random surfer to hyperlink to any page randomly after entering a dangling node. From this we now have a stochastic transition matrix H.
2. Primitivity adjustment
    - This adjustment can be thought of as the random surfer getting bored with following the hyperlink structure and sometimes going to an entirely new URL and continuing from there. To achieve this, a damping factor (alpha) is introduced. This is a value between 0 and 1 and represents the probability of making a random jump (or "teleportation"). To achieve our final stochastic transition probability matrix P, we multiply H by (1-alpha) and add to it a teleportation matrix I(1/n) which is multiplied by alpha. Here, n represents the number of nodes in the graph.

After our adjustments we thus have P = (1-alpha) H + alpha I(1/n).

HW 9.1

HW 9.1: Problem Statement

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 ∈ (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. (On Dropbox https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0 (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 (https://en.wikipedia.org/wiki/PageRank):

Here for reference are the corresponding PageRank probabilities:

A,0.033
B,0.384
C,0.343
D,0.039
E,0.081
F,0.039
G,0.016
H,0.016
I,0.016
J,0.016
K,0.016

HW 9.1 - Initial setup job

We'll need to know how many nodes are in the graph to distribute the starting mass, so we can start by recycling our code from HW7 that does this. That said, we do know in advance how many nodes each of the graphs we're going to use in this assignment have, so we can (and do) plug those numbers in manually to save some processing time in subsequent problems. If we needed to scale this implementation to new large datasets with unknown size, we'd need this step.

In [27]:

```
%%writefile mrpagerankinit.py

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

class mrPageRankInit(MRJob):

    def mapper(self, _, line):
        """Emit keyless records (since we don't want to group our resul
        Values are (1,node_degree)"""
        line = line.strip('\n')
        data = line.split("\t")
        nid = data[0]
        N = eval(data[1])
        node_degree = len(N)
        for n in N.iteritems():
            yield _,(n[0],n[1])
        yield _,(nid,0)


    def reducer(self, _, line):
        """Aggregate node counts and degree counts"""
        nodes=set()
        edges=0
        for record in line:
            nodes.add(record[0])
            edges+=record[1]
        yield None, (len(nodes),edges)

    def steps(self):
        return [MRStep(  mapper=self.mapper
                    ,reducer=self.reducer
                )
            ]

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

Overwriting mrpagerankinit.py

HW 9.1 - Main pagerank job

Assuming we know how many nodes are in the graph, we can run our main job. This job accepts a desired number of iterations as a parameter and will (within MRjob) repeat the steps until this value is reached. This is useful because it keeps all the content in the stream, and will not need to interact with a driver during the process.

In [45]:

```
%%writefile mrpagerank.py
from __future__ import division
from mrjob.job import MRJob
from mrjob.job import MRStep
import ast

class mrPageRank(MRJob):

    def configure_options(self):
        super(mrPageRank, self).configure_options()
        self.add_passthrough_option('--d', default=0.85, type=float,
                                    help='dampening factor')
        self.add_passthrough_option('--N', default=None, type=int,
                                    help='total number of nodes')
        self.add_passthrough_option('--iterations', default=2, type=int
                                    help='how many iterations should we


    def mapper_setup(self, nid, nodes_score):
        """
        Ensure that any nodes that are linked to but do not have any ou
        are listed in the full list of nodes
        """
        nodes_score = nodes_score.strip('\n')
        nid, nodes = nodes_score.split('\t')
        #Emit original node
        yield nid, nodes
        nodes=eval(nodes)
        #Emit blank dicts for all linked nodes
        for n,w in nodes.iteritems():
            yield n,'{}'

    def reducer_setup(self, nid, values):
        """
        Aggregate results from mapper evenly distribute starting probab
        """
        nodes={}
        for v in values:
            v=eval(v)
            nodes.update(v)
        score = 1/float(self.options.N)
        yield nid,str(nodes)+"|"+str(score)

    def mapper_distribute_weights(self, nid, nodes_score):
        """
        Main mapper maintains the graph in the stream, identifies dangl
        and distributes each node's mass across its links
        """
        nodes_score = nodes_score.strip('\n')
        nodes,score=nodes_score.split('|')
        nodes=eval(nodes)
        score=float(score)
        # pass along graph structure
        yield nid, ('node', nodes)
```

```
        # pass mass associated with dangling nodes
        if len(nodes)==0:
            yield '*',('score',score)

        else:
            #dispense mass from current node evenly across all linked n
            for n, w in nodes.iteritems():
                yield n, ('score', score*w/len(nodes))

    def reducer_init_main(self):
        """Create a place to store running dangling mass total"""
        self.dangling_score=0

    def reducer_gather_weights(self, nid, values):
        """Aggregate dangling mass and node-by-node scores (not includi
        nodes={}
        total_score = 0

        #Use order inversion to calculate total dangling mass
        if nid =='*':
            for typ, value in values:
                self.dangling_score+=value

        else:
            for typ, value in values:
                if typ == 'node':
                    nodes = value
                elif typ == 'score':
                    total_score += value

            yield nid, str(nodes)+"|"+str(total_score)

    def reducer_final_emit_dangling(self):
        """Emit total dangling mass for the graph"""
        yield '*',self.dangling_score

    def reducer_init_2(self):
        """Initialize dangling mass total on new reducer"""
        self.dangling_mass=0

    def reducer_distribute_dangling_weights(self, nid, nodes_score):
        """Compute final pagerank score for each node, based on
        partial result from the previous step and the (now known)
        total dangling mass"""

        stripe=[v for v in nodes_score][0]
        if nid=='*':
            self.dangling_mass+=stripe
        else:
            nodes,partial_score=stripe.split("|")
            partial_score=eval(partial_score)

            N = self.options.N
```

```
            d = self.options.d

            new_mass=float(self.dangling_mass/self.options.N)
            score = (1-d)/float(N) + d*float(partial_score+new_mass)
            yield nid, str(nodes)+"|"+str(score)

    def steps(self):
        return (
                [MRStep(mapper = self.mapper_setup,
                       reducer=self.reducer_setup)] +

                # These two steps repeat over and over until we've comp
                # the desired number of iterations
                [MRStep(mapper = self.mapper_distribute_weights
                       ,reducer_init=self.reducer_init_main
                       ,reducer = self.reducer_gather_weights
                       ,reducer_final=self.reducer_final_emit_dangling
                       )
                ,
                MRStep(
                    reducer_init=self.reducer_init_2,
                    reducer = self.reducer_distribute_dangling_weights
                       )
                ]*self.options.iterations
        )

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

```
Overwriting mrpagerank.py
```

HW 9.1 - Driver

The driver runs the initial setup job to calculate the number of nodes. The main pagerank job runs within a function (we'll need this later for 9.2) and writes the final output to file.

In [47]:

```
## HW7 - Directed Toy Example, running locally
%reload_ext autoreload
%autoreload 2
from mrpagerank import mrPageRank
from mrpagerankinit import mrPageRankInit
from __future__ import division

num_iterations=40
nodes=0 #initialize number of nodes

input_dir_prefix='PageRank-test'
input_directory=input_dir_prefix+'.txt'
output_directory=input_dir_prefix+'Output.txt'

mr_job = mrPageRankInit(args=[input_directory,'--no-strict-protocols'])


#First init job figures out how many nodes we have
#NOTE: We'll do this here to show how it works, but for subsequent prob
#we know how many nodes we have in advance and can skip this step.
with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        _,count =  mr_job.parse_output_line(line)
        nodes+=count[0]
print "Total Nodes = {}".format(nodes)


def run_jobs(d,output_directory):
#LOCAL VERSION - IN WHICH WE WRITE RESULTS TO A FILE
    mr_job2 = mrPageRank(args=[input_directory,
                              '--no-strict-protocols',
                              '--d',d,
                              '--N',str(nodes),
                              '--iterations',str(num_iterations)])


    total_score=0 #Keep track of our total probability mass to make sur
    with mr_job2.make_runner() as runner2:
        runner2.run()
        #Stream output locally
        with open(output_directory, 'w+') as f:
            for line in runner2.stream_output():
                print line.strip()
                nid,stripe =  mr_job.parse_output_line(line)
                _,score=stripe.split("|")
                total_score+=eval(score)
                output=str(nid)+'\t'+str(stripe)+'\n'
                f.write(output)
        print "TOTAL SCORE: "+str(total_score)
        print ""

    print "ALL DONE"
```

```
run_jobs(0.85,output_directory)
```

```
Total Nodes = 11
"A"     "{}|0.0327814931611"
"B"     "{'C': 1}|0.384242635388"
"C"     "{'B': 1}|0.343068598924"
"D"     "{'A': 1, 'B': 1}|0.039087092102"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.0808856932376"
"F"     "{'B': 1, 'E': 1}|0.039087092102"
"G"     "{'B': 1, 'E': 1}|0.0161694790171"
"H"     "{'B': 1, 'E': 1}|0.0161694790171"
"I"     "{'B': 1, 'E': 1}|0.0161694790171"
"J"     "{'E': 1}|0.0161694790171"
"K"     "{'E': 1}|0.0161694790171"
TOTAL SCORE: 1.0

ALL DONE
```

Sure enough, we are able to replicate the desired result for the test dataset

HW 9.2

HW 9.2 - Problem Statement

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 this (https://en.wikipedia.org/wiki/PageRank#/media/File:PageRanks-Example.svg)

HW 9.2 - Implementation

We've written the driver for 9.1 in terms of a function that accepts different values of alpha, so we can easily run it iteratively for each damping parameter we're interested in.

In [261]:

```
#HW 9.2 - Calculate pagerank for different alphas
factors=[0,0.25,0.5,0.75, 0.85, 1]
for d in factors:
    print "running job for alpha="+str(d)
    output_directory='92d'+str(d)+'.txt'
    run_jobs(d,output_directory)
```

```
running job for alpha=0
"A"     "{}|0.0909090909091"
"B"     "{'C': 1}|0.0909090909091"
"C"     "{'B': 1}|0.0909090909091"
"D"     "{'A': 1, 'B': 1}|0.0909090909091"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.0909090909091"
"F"     "{'B': 1, 'E': 1}|0.0909090909091"
"G"     "{'B': 1, 'E': 1}|0.0909090909091"
"H"     "{'B': 1, 'E': 1}|0.0909090909091"
"I"     "{'B': 1, 'E': 1}|0.0909090909091"
"J"     "{'E': 1}|0.0909090909091"
"K"     "{'E': 1}|0.0909090909091"
TOTAL SCORE: 1.0

ALL DONE
running job for alpha=0.25
"A"     "{}|0.0802296662471"
"B"     "{'C': 1}|0.155730909761"
"C"     "{'B': 1}|0.108937947128"
"D"     "{'A': 1, 'B': 1}|0.0817955724769"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.141484233473"
"F"     "{'B': 1, 'E': 1}|0.0817955724769"
"G"     "{'B': 1, 'E': 1}|0.0700052196874"
"H"     "{'B': 1, 'E': 1}|0.0700052196874"
"I"     "{'B': 1, 'E': 1}|0.0700052196874"
"J"     "{'E': 1}|0.0700052196874"
"K"     "{'E': 1}|0.0700052196874"
TOTAL SCORE: 1.0

ALL DONE
running job for alpha=0.5
"A"     "{}|0.0669478123353"
"B"     "{'C': 1}|0.228430855737"
"C"     "{'B': 1}|0.162713055702"
"D"     "{'A': 1, 'B': 1}|0.0738007380074"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.151818661044"
"F"     "{'B': 1, 'E': 1}|0.0738007380074"
"G"     "{'B': 1, 'E': 1}|0.0484976278334"
"H"     "{'B': 1, 'E': 1}|0.0484976278334"
"I"     "{'B': 1, 'E': 1}|0.0484976278334"
"J"     "{'E': 1}|0.0484976278334"
"K"     "{'E': 1}|0.0484976278334"
TOTAL SCORE: 1.0

ALL DONE
running job for alpha=0.75
"A"     "{}|0.0463014615641"
"B"     "{'C': 1}|0.328715480769"
"C"     "{'B': 1}|0.272422531056"
"D"     "{'A': 1, 'B': 1}|0.0544460560079"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.114247461787"
"F"     "{'B': 1, 'E': 1}|0.0544460560079"
"G"     "{'B': 1, 'E': 1}|0.0258841905612"
"H"     "{'B': 1, 'E': 1}|0.0258841905612"
```

```
"I"     "{'B': 1, 'E': 1}|0.0258841905612"
"J"     "{'E': 1}|0.0258841905612"
"K"     "{'E': 1}|0.0258841905612"
TOTAL SCORE: 0.999999999998

ALL DONE
running job for alpha=0.85
"A"     "{}|0.0327814931611"
"B"     "{'C': 1}|0.384242635388"
"C"     "{'B': 1}|0.343068598924"
"D"     "{'A': 1, 'B': 1}|0.039087092102"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.0808856932376"
"F"     "{'B': 1, 'E': 1}|0.039087092102"
"G"     "{'B': 1, 'E': 1}|0.0161694790171"
"H"     "{'B': 1, 'E': 1}|0.0161694790171"
"I"     "{'B': 1, 'E': 1}|0.0161694790171"
"J"     "{'E': 1}|0.0161694790171"
"K"     "{'E': 1}|0.0161694790171"
TOTAL SCORE: 1.0

ALL DONE
running job for alpha=1
"A"     "{}|1.75543268125e-09"
"B"     "{'C': 1}|0.385321096929"
"C"     "{'B': 1}|0.614678893011"
"D"     "{'A': 1, 'B': 1}|1.94029676792e-09"
"E"     "{'B': 1, 'D': 1, 'F': 1}|3.14304437704e-09"
"F"     "{'B': 1, 'E': 1}|1.94029676792e-09"
"G"     "{'B': 1, 'E': 1}|2.56082324161e-10"
"H"     "{'B': 1, 'E': 1}|2.56082324161e-10"
"I"     "{'B': 1, 'E': 1}|2.56082324161e-10"
"J"     "{'E': 1}|2.56082324161e-10"
"K"     "{'E': 1}|2.56082324161e-10"
TOTAL SCORE: 0.999999999999

ALL DONE
```

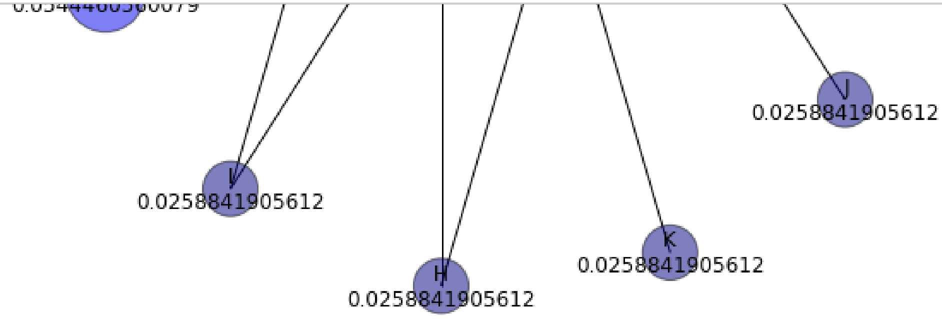
HW 9.2 - Plotting the networks

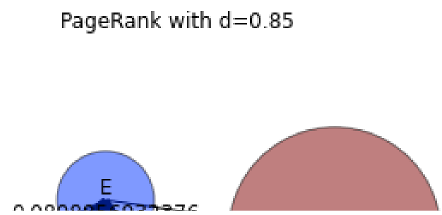
Now that we have our pagerank scores for the different values of alpha, we can plot the results visually.

In [262]:

```
%matplotlib inline
import networkx as nx
import ast
from matplotlib import pyplot as plt

# Draw graphs
def draw(edges, scores, d):
    plt.figure(figsize=(10, 10))

    # initialize directed graph
    DG = nx.DiGraph()

    # add edges
    for edge in edges:
        DG.add_edge(edge[0], edge[1])

    node_size = [scores[n]*40000 for n in DG.nodes()]

    graph_pos = nx.circular_layout(DG)

    # set labels
    labels = {}
    for node in DG.nodes():
        labels[node] = '{}\n{}'.format(node, scores[node])

    # draw graph
    nx.draw_networkx_nodes(DG, graph_pos, node_size = node_size, node_c
    nx.draw_networkx_edges(DG, graph_pos, edge_color = 'black', arrows 
    nx.draw_networkx_labels(DG, graph_pos, labels=labels, font_size = 1

    # show graph
    plt.title("PageRank with d={}".format(d))
    plt.axis('off')
    plt.tight_layout()
    plt.show()

# Take filename, damping factor as input to produce plots by drawing gr
def plot(f, d):
    edges = []
    scores = {}

    for line in open(f).read().strip().split('\n'):
        nid, nodes_score = line.split('\t') # Parse the line into the m

        nid = nid.replace('"', '') # Remove double quotes from node nam

        nodes, score = map(ast.literal_eval, nodes_score.strip('"').spl

        edges.extend([(nid, n) for n in nodes.keys()]) # For each node 
        scores[nid] = score # Set the score for the main node id

    draw(edges, scores, d) # Send our edges and scores and damping fact

D = [0, 0.25, 0.5, 0.75, 0.85, 1]
```

```
for d in D:
    f = '92d{}.txt'.format(d)
    plot(f, d)
```





HW 9.3

HW 9.3 - Problem Statement

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 50 iterations run. Then plot the pagerank values for the same 100 pages that resulted from the 10 iterations run.

HW 9.3 - Implementation details

Unlike the locally-run jobs shown in the previous problems, we need to deal with the problem of multiple reducers here. Instead of relying on all mappers and reducers having access to the running total for the dangling mass the way we did earlier, we'll create a second job that aggregates this total separately. That way, the main job can focus only on streaming the intermediate results to S3 while this second (much smaller) job streams data through the driver. The tradeoff here is that we need a more complex driver, since we can't rely on MRJob itself to deal with the iterations. For brevity, we've also omitted the initial job that calculates how many nodes we have, since we already know that aspect of these datasets. If we didn't know that in advance, running the job to figure it out would be a trivial additional step.

HW 9.3 - First pass job

Add any dangling nodes to the list (with empty dictionaries of associated edges) and evenly distribute starting mass across all nodes

In [48]:
```
%%writefile mrpagerankfirstpass.py
from __future__ import division
from mrjob.job import MRJob
from mrjob.job import MRStep
import ast

class mrPageRankFirstPass(MRJob):

    def configure_options(self):
        super(mrPageRankFirstPass, self).configure_options()
        self.add_passthrough_option('--N', default=None, type=int,
                                    help='total number of nodes')

    def mapper_setup(self, nid, nodes_score):
        nodes_score = nodes_score.strip('\n')
        nid, nodes = nodes_score.split('\t')
        yield str(nid), nodes
        nodes=eval(nodes)
        for n,w in nodes.iteritems():
            yield str(n),'{}'

    def reducer_setup(self, nid, values):
        nodes={}
        for v in values:
            v=eval(v)
            nodes.update(v)
        score = 1/float(self.options.N)
        yield str(nid),str(nodes)+"|"+str(score)


    def steps(self):
        return (
                #Init step - add dangling nodes as separate stripes and
                #starting mass evenly
                [MRStep(mapper = self.mapper_setup,
                        reducer=self.reducer_setup)] )

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

Overwriting mrpagerankfirstpass.py

HW 9.3 - Main pagerank job

This is similar to the code above, but runs one job per iteration rather than treating each iteration as a step within the single job

In [52]:

```
%%writefile mrpagerank.py
from __future__ import division
from mrjob.job import MRJob
from mrjob.job import MRStep
import ast

class mrPageRank(MRJob):

    def configure_options(self):
        super(mrPageRank, self).configure_options()
        self.add_passthrough_option('--d', default=0.85, type=float,
                                    help='dampening factor')
        self.add_passthrough_option('--N', default=None, type=int,
                                    help='total number of nodes')
        self.add_passthrough_option('--dangling', default=0, type=float
                                    help='What dangling mass do we have

    def mapper_distribute_weights(self, _, line):
        """Split each node's mass between linked nodes"""
        line=line.strip('\n')
        nid,nodes_score=line.split('\t')
        #nodes_score=eval(nodes_score) #Comment this out when running l
        nodes,score=nodes_score.split('|')
        #nid=eval(nid)#Comment this out when running locally
        nodes=eval(nodes)
        score=float(score)
        # pass along graph structure
        yield str(nid), ('node', nodes)

        #dispense mass from current node evenly across all linked nodes
        for n, w in nodes.iteritems():
            yield str(n), ('score', score*w/len(nodes))

    def reducer_gather_weights(self, nid, values):
        """Aggregate new mass associated with each node"""
        nodes={}
        partial_score = 0

        for typ, value in values:
            if typ == 'node':
                nodes = value
            elif typ == 'score':
                partial_score += value

        N = self.options.N
        d = self.options.d
        mass=self.options.dangling
        new_mass=float(mass/N)

        score = (1-d)/float(N) + d*float(partial_score+new_mass)

        yield str(nid), str(nodes)+"|"+str(score)
```

```
    def steps(self):
        return (
                #Main step - redistribute and gather weights
                [MRStep(mapper = self.mapper_distribute_weights
                       ,reducer = self.reducer_gather_weights
                       )
                ]
        )

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

```
Overwriting mrpagerank.py
```

HW 9.3 - Add up dangling mass

This job iterates through the results of the previous job and totals the mass associated with dangling nodes. This can then be passed back to the main pagerank job as a parameter for the subsequent iteration.

In [53]:

```
%%writefile mrpageranksumweight.py
from __future__ import division
from mrjob.job import MRJob
from mrjob.job import MRStep

class mrPageRankSumWeight(MRJob):

    def mapper(self, _, line):
        """Scan for dangling nodes and emit associated mass"""
        line=line.strip('\n')
        nid,nodes_score=line.split('\t')
        #nodes_score=eval(nodes_score) #comment this out when running l
        nodes,score=nodes_score.split('|')
        nodes=eval(nodes)
        score=float(score)

        # pass mass associated with dangling nodes
        if len(nodes)==0:
            yield _,score


    def reducer(self, nid, values):
        """
        Aggregate total dangling mass
        The final sum is calculated in the driver
        to enable this to work on multiple reducers
        """
        mass = sum([i for i in values])

        #Emit total dangling mass
        yield None, str(mass)


    def steps(self):
        return (
                #Main step - redistribute and gather weights
                [MRStep(mapper = self.mapper
                       ,reducer = self.reducer
                       )
                ]
        )

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

```
Overwriting mrpageranksumweight.py
```

HW 9.3 - System Test Local Driver

Before we run our code on the full dataset, we run it again on the test dataset from 9.1 in EMR for a few iterations to ensure everything is working. In addition, we'll print out some intermediate results to compare to our upcoming EMR version.

In [55]:

```
## HW7 - Directed Toy Example, running locally
%reload_ext autoreload
%autoreload 2
from mrpagerank import mrPageRank
from mrpagerankfirstpass import mrPageRankFirstPass
from mrpageranksumweight import mrPageRankSumWeight
from __future__ import division

num_iterations=40
input_dir_prefix='PageRank-test'
nodes=11 #We already know this, but we could run our init job to calcul
dangling_mass=0
d=0.85
current_iteration=1
input_directory=input_dir_prefix+'.txt'
output_directory=input_dir_prefix+'Output{0}.txt'.format(str(current_it

mr_job = mrPageRankFirstPass(args=[input_directory,'--no-strict-protoco

#First job only runs once, expands graph, and evenly distributes starti
total_score=0
with mr_job.make_runner() as runner:
    runner.run()
    #Stream output locally
    with open(output_directory, 'w+') as f:
        for line in runner.stream_output():
            nid,stripe =  mr_job.parse_output_line(line)
            _,score=stripe.split("|")
            total_score+=eval(score)
            output=str(nid)+'\t'+str(stripe)+'\n'
            f.write(output)

dangling_mass=0
mr_job3 = mrPageRankSumWeight(args=[output_directory,'--no-strict-proto

with mr_job3.make_runner() as runner3:
    runner3.run()
    for line in runner3.stream_output():
        _,partial_mass =  mr_job.parse_output_line(line)
        dangling_mass+=eval(partial_mass)

#Start main job loop
current_iteration+=1
while current_iteration<=num_iterations:
    input_directory=input_dir_prefix+'Output{0}.txt'.format(str(current
    output_directory=input_dir_prefix+'Output{0}.txt'.format(str(curren

    #LOCAL VERSION - IN WHICH WE WRITE RESULTS TO A FILE
    #Second job does the main pagerank calculation
    mr_job2 = mrPageRank(args=[input_directory,
                               '--no-strict-protocols',
                               '--d',d,
                               '--N',str(nodes),
                               '--dangling',str(dangling_mass)])
```

```
    total_score=0
    with mr_job2.make_runner() as runner2:
        runner2.run()
        #Stream output locally
        with open(output_directory, 'w+') as f:
            if current_iteration in [2,3,40]:
                print "Iteration {0}".format(str(current_iteration))
            for line in runner2.stream_output():
                if current_iteration in [2,3,40]:
                    #pass
                    print line.strip()
                nid,stripe =  mr_job.parse_output_line(line)
                _,score=stripe.split("|")
                total_score+=eval(score)
                output=str(nid)+'\t'+str(stripe)+'\n'
                f.write(output)
            if current_iteration in [2,3,40]:
                print ""

    #Third job aggregates the dangling mass
    dangling_mass=0
    mr_job3 = mrPageRankSumWeight(args=[output_directory,
                        '--no-strict-protocols'])

    with mr_job3.make_runner() as runner3:
        runner3.run()
        for line in runner3.stream_output():
            _,partial_mass =  mr_job.parse_output_line(line)
            dangling_mass+=eval(partial_mass)

    current_iteration+=1

print "ALL DONE"
```

```
Iteration 2
"A"     "{}|0.0592975206612"
"B"     "{'C': 1}|0.316873278237"
"C"     "{'B': 1}|0.0979338842975"
"D"     "{'A': 1, 'B': 1}|0.0464187327824"
"E"     "{'B': 1, 'D': 1, 'F': 1}|0.329752066116"
"F"     "{'B': 1, 'E': 1}|0.0464187327824"
"G"     "{'B': 1, 'E': 1}|0.0206611570248"
"H"     "{'B': 1, 'E': 1}|0.0206611570248"
"I"     "{'B': 1, 'E': 1}|0.0206611570248"
"J"     "{'E': 1}|0.0206611570248"
"K"     "{'E': 1}|0.0206611570248"

Iteration 3
"A"     "{}|0.0379464062109"
"B"     "{'C': 1}|0.260690896569"
"C"     "{'B': 1}|0.28756073128"
```

```
"D"      "{'A': 1, 'B': 1}|0.111648196845"
"E"      "{'B': 1, 'D': 1, 'F': 1}|0.0994133483597"
"F"      "{'B': 1, 'E': 1}|0.111648196845"
"G"      "{'B': 1, 'E': 1}|0.0182184447784"
"H"      "{'B': 1, 'E': 1}|0.0182184447784"
"I"      "{'B': 1, 'E': 1}|0.0182184447784"
"J"      "{'E': 1}|0.0182184447784"
"K"      "{'E': 1}|0.0182184447784"

Iteration 40
"A"      "{}|0.0327814931627"
"B"      "{'C': 1}|0.384587199891"
"C"      "{'B': 1}|0.342724034413"
"D"      "{'A': 1, 'B': 1}|0.0390870921036"
"E"      "{'B': 1, 'D': 1, 'F': 1}|0.0808856932407"
"F"      "{'B': 1, 'E': 1}|0.0390870921036"
"G"      "{'B': 1, 'E': 1}|0.0161694790173"
"H"      "{'B': 1, 'E': 1}|0.0161694790173"
"I"      "{'B': 1, 'E': 1}|0.0161694790173"
"J"      "{'E': 1}|0.0161694790173"
"K"      "{'E': 1}|0.0161694790173"

ALL DONE
```

HW 9.3 - System Test EMR Driver

Now that we've confirmed this new archtecture reproduces our results from 9.1 and we know what our intermediate results should look like, we can run the same job again on EMR. This requires a modified driver.

In [21]:

```
## HW9.3 - Test dataset, running in EMR
%reload_ext autoreload
%autoreload 2
from mrpagerank import mrPageRank
from mrpagerankfirstpass import mrPageRankFirstPass
from mrpageranksumweight import mrPageRankSumWeight
from __future__ import division

num_iterations=3
input_dir_prefix='PageRank-test'
nodes=11 #We already know this, but we could run our init job to calcul
dangling_mass=0
d=0.85
current_iteration=1

input_dir_prefix='PageRank-test'
input_directory='s3://hamlin-mids-261/'+input_dir_prefix+'.txt'
output_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0}'.f
cluster='j-1K47D3ANROPM1'

mr_job = mrPageRankFirstPass(args=[
        '-r','emr',
        input_directory,
        '--no-strict-protocols',
        '--output-dir',output_directory,
        '--emr-job-flow-id', cluster,
        '--no-output',
        '--N',str(nodes)
        ])

#First job only runs once, expands graph, and evenly distributes starti
with mr_job.make_runner() as runner:
    runner.run()

dangling_mass=0
mr_job3 = mrPageRankSumWeight(args=[
        '-r','emr',
        output_directory+'/',
        '--no-strict-protocols',
        '--no-output',
        '--emr-job-flow-id', cluster])

with mr_job3.make_runner() as runner3:
    runner3.run()
    for line in runner3.stream_output():
        _,partial_mass =  mr_job.parse_output_line(line)
        dangling_mass+=eval(partial_mass)

#Start main job loop
current_iteration+=1
while current_iteration<=num_iterations:
    print current_iteration
    input_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0}
    output_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0
```

```
    #Second job does the main pagerank calculation
    mr_job2 = mrPageRank(args=['-r','emr',
                                input_directory,
                                '--no-strict-protocols',
                                '--d',str(d),
                                '--N',str(nodes),
                                '--dangling',str(dangling_mass),
                                '--output-dir',output_directory,
                                '--emr-job-flow-id', cluster,
                                '--no-output'])


    with mr_job2.make_runner() as runner2:
        runner2.run()

    #Third job aggregates the dangling mass
    dangling_mass=0
    mr_job3 = mrPageRankSumWeight(args=[
        '-r','emr',
        output_directory+'/',
        '--no-strict-protocols',
        '--no-output',
        '--emr-job-flow-id', cluster])

    with mr_job3.make_runner() as runner3:
        runner3.run()
        for line in runner3.stream_output():
            _,partial_mass =  mr_job3.parse_output_line(line)
            dangling_mass+=eval(partial_mass)

    current_iteration+=1

print "ALL DONE"
```

```
2
3
ALL DONE
```

For brevity, we haven't included all the results of the system test here. Manually spot-checking the output files confirms that this EMR version of the implementation is working, so now we can run it on the full dataset.

HW 9.3 - Running the full job

In [23]:

```
## HW9.3 - Full dataset, running in EMR
%reload_ext autoreload
%autoreload 2
from mrpagerank import mrPageRank
from mrpagerankfirstpass import mrPageRankFirstPass
from mrpageranksumweight import mrPageRankSumWeight
from __future__ import division

num_iterations=10
input_dir_prefix='all-pages-indexed-out'
nodes=5781290 #We already know this, but we could run our init job to c
dangling_mass=0
d=0.85
current_iteration=1

input_directory='s3://hamlin-mids-261/'+input_dir_prefix+'.txt'
output_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0}'.f
cluster='j-QDJ8C8U3MWWU'

mr_job = mrPageRankFirstPass(args=[
        '-r','emr',
        input_directory,
        '--no-strict-protocols',
        '--output-dir',output_directory,
        '--emr-job-flow-id', cluster,
        '--no-output',
        '--N',str(nodes)
        ])

#First job only runs once, expands graph, and evenly distributes starti
with mr_job.make_runner() as runner:
    runner.run()

mr_job3 = mrPageRankSumWeight(args=[
        '-r','emr',
        output_directory+'/',
        '--no-strict-protocols',
        '--no-output',
        '--emr-job-flow-id', cluster])

with mr_job3.make_runner() as runner3:
    runner3.run()
    for line in runner3.stream_output():
        _,partial_mass =  mr_job.parse_output_line(line)
        dangling_mass+=eval(partial_mass)

current_iteration+=1
while current_iteration<=num_iterations:
    print current_iteration
    input_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0}
    output_directory='s3://hamlin-mids-261/'+input_dir_prefix+'Output{0

    #Second job does the main pagerank calculation
    mr_job2 = mrPageRank(args=['-r','emr',
```

```
                                   input_directory,
                                   '--no-strict-protocols',
                                   '--d',str(d),
                                   '--N',str(nodes),
                                   '--dangling',str(dangling_mass),
                                   '--output-dir',output_directory,
                                   '--emr-job-flow-id', cluster,
                                   '--no-output'])


    with mr_job2.make_runner() as runner2:
        runner2.run()

    #Third job aggregates the dangling mass
    dangling_mass=0
    mr_job3 = mrPageRankSumWeight(args=[
        '-r','emr',
        output_directory+'/',
        '--no-strict-protocols',
        '--no-output',
        '--emr-job-flow-id', cluster])

    with mr_job3.make_runner() as runner3:
        runner3.run()
        for line in runner3.stream_output():
            _,partial_mass =  mr_job3.parse_output_line(line)
            dangling_mass+=eval(partial_mass)

    current_iteration+=1

print "ALL DONE"
```

```
2
3
4
5
6
7
8
9
10
ALL DONE
```

HW9.3 - Organize the results

In theory, the most scalable way to handle the final sorting would be to create another mapreduce job. In the interest of time and EMR costs, we've instead done this locally.

In [125]:
```
#HW9.3 - Download results
! mkdir ./wiki_5_iterations
! aws s3 cp --recursive s3://hamlin-mids-261/all-pages-indexed-outOutpu
! mkdir ./wiki_10_iterations
! aws s3 cp --recursive s3://hamlin-mids-261/all-pages-indexed-outOutpu
```

```
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/_SUCCES
S to wiki_10_iterations/_SUCCESS
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
001 to wiki_10_iterations/part-00001
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
000 to wiki_10_iterations/part-00000
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
002 to wiki_10_iterations/part-00002
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
003 to wiki_10_iterations/part-00003
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
004 to wiki_10_iterations/part-00004
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
005 to wiki_10_iterations/part-00005
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
006 to wiki_10_iterations/part-00006
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
008 to wiki_10_iterations/part-00008
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
007 to wiki_10_iterations/part-00007
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
009 to wiki_10_iterations/part-00009
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
010 to wiki_10_iterations/part-00010
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
011 to wiki_10_iterations/part-00011
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
013 to wiki_10_iterations/part-00013
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
012 to wiki_10_iterations/part-00012
download: s3://hamlin-mids-261/all-pages-indexed-outOutput10/part-00
014 to wiki_10_iterations/part-00014
```

In [126]:

```
#HW 9.3 - Sort wikipedia results and extract top 100 pages for each
import os

def top_100(folder):
    files=os.listdir('./'+folder)[1:]
    files=['./'+folder+'/'+i for i in files]

    num_results=100
    output=[]
    for result in files:
        with open(result,'r') as f:
            for line in f.readlines():
                #print line
                line=line.strip('\n')
                nid,nodes_score=line.split('\t')
                nid=eval(nid)
                nodes_score=eval(nodes_score)
                nodes,score=nodes_score.split('|')
                score=float(score)
                output.append((score,nid))
                if len(output)>num_results:
                    output.sort(key=lambda x: -float(x[0]))
                    output=output[:num_results]

    output.sort(key=lambda x: -float(x[0]))

    with open(folder+'SortResults.txt','w') as f:
        for i in output:
            f.writelines(str(i)+'\n')
            #print i

top_100('wiki_5_iterations')
top_100('wiki_10_iterations')
```

In [128]:

```
# HW 9.3 - Load index into memory so we can look things up
word_dict={}
node_dict={}
with open('indices.txt') as f:
    for line in f.readlines():
        word,node_id,_,_=line.strip().split('\t')
        node_dict[node_id]=word #Enables us to find words by ID
        word_dict[word]=node_id #Enables us to find IDs by word
```

In [145]:

```
# HW 9.3 - Prettify and display results for top pages after 5 iteration
scores_5=[]
print "Top 100 pages after 5 iterations"
print "SCORE    |  ID - TITLE"
print "----------------------------------"
with open('wiki_5_iterationsSortResults.txt','r') as f:
    for line in f.readlines():
        score,nid=eval(line)
        scores_5.append(score)
        print '{0:3.6f} |  {1} - {2}  '.format(score,int(nid),node_dict
```

```
Top 100 pages after 5 iterations
SCORE    |  ID - TITLE
----------------------------------
0.026193 |  13455888 - United States
0.011438 |  1184351 - Animal
0.011243 |  4695850 - France
0.010776 |  5051368 - Germany
0.008472 |  6076759 - India
0.008276 |  4196067 - England
0.008261 |  1384888 - Arthropod
0.008075 |  6113490 - Insect
0.008068 |  2437837 - Canada
0.007765 |  6172466 - Iran
0.006547 |  13425865 - United Kingdom
0.006294 |  6416278 - Japan
0.006212 |  6237129 - Italy
0.006165 |  10390714 - Poland
0.005860 |  1516699 - Australia
0.005850 |  7835160 - List of countries
0.005846 |  14112583 - World War II
0.005772 |  7576704 - Lepidoptera
0.005754 |  15164193 - village
0.005687 |  13432150 - United States Census Bureau
0.005609 |  9276255 - National Register of Historic Places
0.005541 |  7902219 - List of sovereign states
0.005382 |  2155467 - Brazil
0.005365 |  3191491 - Countries of the world
0.005131 |  11147327 - Romania
0.004944 |  12074312 - Spain
0.004940 |  13725487 - Voivodeships of Poland
0.004878 |  7990491 - London
0.004739 |  10469541 - Powiat
0.004733 |  11253108 - Russia
0.004673 |  5154210 - Gmina
0.004556 |  14881689 - moth
0.004542 |  11245362 - Rural Districts of Iran
0.004358 |  12836211 - The New York Times
0.004255 |  2396749 - California
0.004240 |  9386580 - New York City
0.004230 |  12430985 - Sweden
0.004128 |  2797855 - China
0.004076 |  3191268 - Counties of Iran
0.004044 |  3603527 - Departments of France
0.004003 |  10566120 - Provinces of Iran
0.003971 |  9355455 - Netherlands
0.003962 |  4198751 - English language
0.003911 |  3069099 - Communes of France
0.003876 |  1637982 - Bakhsh
0.003875 |  14503460 - association football
0.003832 |  1441065 - Association football
0.003816 |  10527224 - Private Use Areas
0.003783 |  8697871 - Mexico
0.003622 |  994890 - Allmusic
0.003544 |  5490435 - Hangul
```

0.003480	6172167 - Iran Standard Time
0.003464	9562547 - Norway
0.003364	9391762 - New York
0.003318	6171937 - Iran Daylight Time
0.003311	10728264 - Race (United States Census)
0.003277	2614581 - Central European Time
0.003201	11582765 - Scotland
0.003022	13280859 - Turkey
0.003018	9394907 - New Zealand
0.002893	981395 - AllMusic
0.002887	2614578 - Central European Summer Time
0.002840	14112408 - World War I
0.002808	11148415 - Romanize
0.002770	3577363 - Democratic Party (United States)
0.002729	9997298 - Paris
0.002708	12067030 - Soviet Union
0.002698	12447593 - Switzerland
0.002682	14725161 - gene
0.002627	1332806 - Argentina
0.002616	12038331 - South Africa
0.002592	10917716 - Republican Party (United States)
0.002456	1947095 - Billboard (magazine)
0.002452	4978429 - Geographic Names Information System
0.002428	14565507 - census
0.002419	8641167 - Member of Parliament
0.002417	4568647 - Finland
0.002413	9742161 - Ontario
0.002398	1523975 - Austria
0.002397	9924814 - Pakistan
0.002360	1813634 - Belgium
0.002356	8019937 - Los Angeles
0.002336	12048800 - South Korea
0.002325	1175360 - Angiosperms
0.002211	10246542 - Philippines
0.002177	14963657 - population density
0.002168	14981725 - protein
0.002147	5908108 - Hungary
0.002122	10399499 - Political divisions of the United States
0.002114	12685893 - Texas
0.002087	3591832 - Denmark
0.002052	1575979 - BBC
0.002045	4344962 - Europe
0.002032	5274313 - Greece
0.002012	10345830 - Plant
0.001980	13328060 - U.S. state
0.001934	2778099 - Chicago
0.001932	14727077 - genus
0.001926	3328327 - Czech Republic
0.001907	15070394 - species

In [146]:

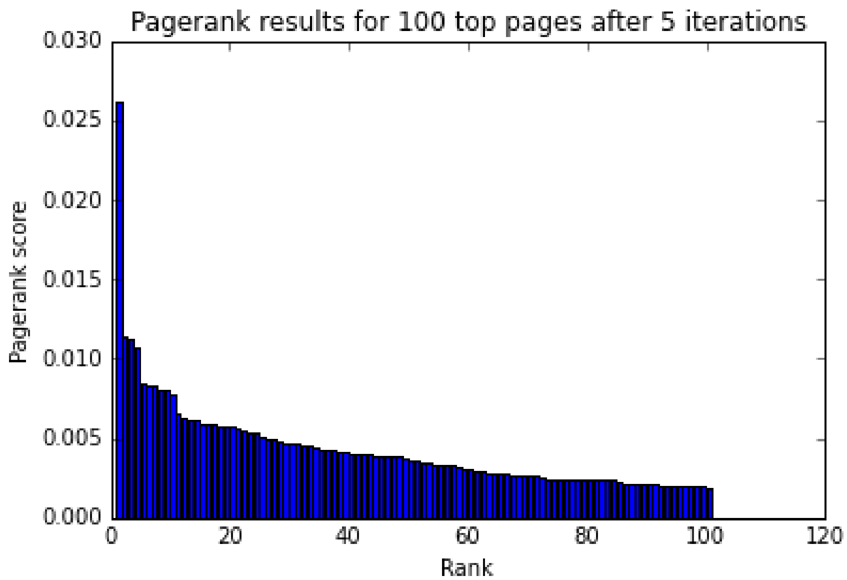
```
# HW 9.3 - Prettify and display results for top pages after 10 iteratic
scores_10=[]
print "Top 100 pages after 10 iterations"
print "SCORE    |  ID - TITLE"
print "-----------------------------------"
with open('wiki_10_iterationsSortResults.txt','r') as f:
    for line in f.readlines():
        score,nid=eval(line)
        scores_10.append(score)
        print '{0:3.6f} |  {1} - {2}  '.format(score,int(nid),node_dict
```

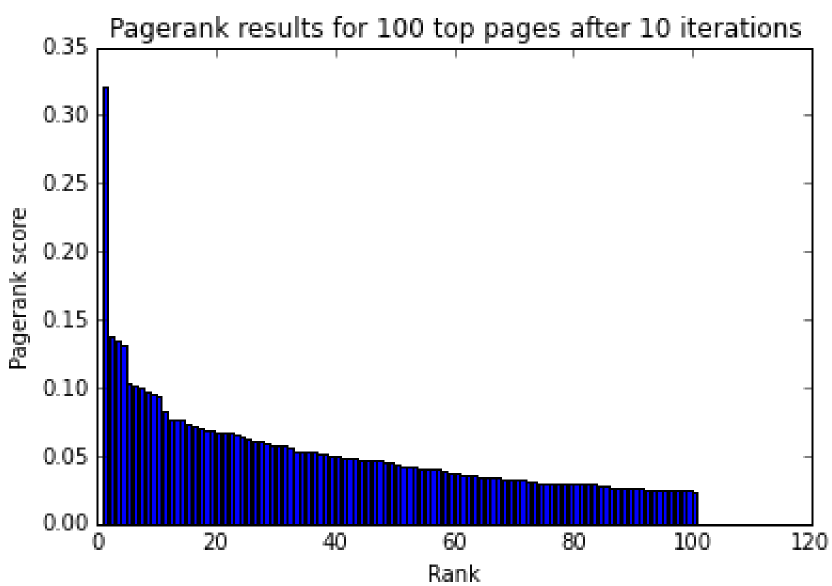
```
Top 100 pages after 10 iterations
SCORE    |  ID - TITLE
----------------------------------
0.321667 |  13455888 - United States
0.137224 |  4695850 - France
0.134542 |  1184351 - Animal
0.131372 |  5051368 - Germany
0.102965 |  6076759 - India
0.101892 |  4196067 - England
0.100011 |  2437837 - Canada
0.097023 |  1384888 - Arthropod
0.094844 |  6113490 - Insect
0.093708 |  6172466 - Iran
0.082219 |  13425865 - United Kingdom
0.076869 |  6416278 - Japan
0.076689 |  6237129 - Italy
0.076177 |  10390714 - Poland
0.073394 |  14112583 - World War II
0.072464 |  1516699 - Australia
0.070358 |  13432150 - United States Census Bureau
0.069385 |  7835160 - List of countries
0.068384 |  15164193 - village
0.067796 |  7576704 - Lepidoptera
0.067129 |  7902219 - List of sovereign states
0.066590 |  9276255 - National Register of Historic Places
0.065716 |  2155467 - Brazil
0.064025 |  3191491 - Countries of the world
0.061658 |  11147327 - Romania
0.061143 |  12074312 - Spain
0.061063 |  7990491 - London
0.059780 |  13725487 - Voivodeships of Poland
0.058412 |  11253108 - Russia
0.057185 |  10469541 - Powiat
0.056957 |  12836211 - The New York Times
0.056333 |  5154210 - Gmina
0.053704 |  9386580 - New York City
0.053554 |  11245362 - Rural Districts of Iran
0.053498 |  14881689 - moth
0.052505 |  2396749 - California
0.052147 |  12430985 - Sweden
0.051374 |  2797855 - China
0.049526 |  4198751 - English language
0.049384 |  9355455 - Netherlands
0.048204 |  3191268 - Counties of Iran
0.048085 |  3603527 - Departments of France
0.048043 |  14503460 - association football
0.047473 |  1441065 - Association football
0.047322 |  10566120 - Provinces of Iran
0.046434 |  3069099 - Communes of France
0.046179 |  8697871 - Mexico
0.045771 |  1637982 - Bakhsh
0.044714 |  10527224 - Private Use Areas
0.043458 |  994890 - Allmusic
0.042478 |  9562547 - Norway
```

```
0.042164 |  9391762 - New York
0.041675 |  5490435 - Hangul
0.041145 |  6172167 - Iran Standard Time
0.041068 |  10728264 - Race (United States Census)
0.039943 |  2614581 - Central European Time
0.039754 |  11582765 - Scotland
0.039140 |  6171937 - Iran Daylight Time
0.037147 |  13280859 - Turkey
0.036990 |  9394907 - New Zealand
0.035849 |  981395 - AllMusic
0.035827 |  14112408 - World War I
0.035161 |  2614578 - Central European Summer Time
0.034602 |  3577363 - Democratic Party (United States)
0.034587 |  12067030 - Soviet Union
0.034182 |  9997298 - Paris
0.033613 |  12447593 - Switzerland
0.033028 |  11148415 - Romanize
0.032439 |  1332806 - Argentina
0.032303 |  10917716 - Republican Party (United States)
0.031991 |  12038331 - South Africa
0.031497 |  14725161 - gene
0.030371 |  1947095 - Billboard (magazine)
0.030196 |  9742161 - Ontario
0.029911 |  4568647 - Finland
0.029848 |  8019937 - Los Angeles
0.029806 |  8641167 - Member of Parliament
0.029696 |  1523975 - Austria
0.029655 |  4978429 - Geographic Names Information System
0.029589 |  14565507 - census
0.029393 |  1813634 - Belgium
0.029298 |  9924814 - Pakistan
0.028705 |  12048800 - South Korea
0.027468 |  1175360 - Angiosperms
0.027398 |  10246542 - Philippines
0.026806 |  14963657 - population density
0.026638 |  1575979 - BBC
0.026502 |  5908108 - Hungary
0.026154 |  12685893 - Texas
0.025857 |  4344962 - Europe
0.025857 |  3591832 - Denmark
0.025480 |  14981725 - protein
0.025446 |  10399499 - Political divisions of the United States
0.025211 |  5274313 - Greece
0.024522 |  2778099 - Chicago
0.024450 |  13328060 - U.S. state
0.024358 |  13853369 - Washington, D.C.
0.024227 |  12785678 - The Guardian
0.024022 |  3328327 - Czech Republic
0.023924 |  10345830 - Plant
```

Looking at these results, it appears that while the scores shift with the additional iterations, the top 100 results remain generally unchanged. That said, the specific rankings of individual pages do move slightly from 5 iterations to 10, which makes sense given that we saw the same general behavior in the test set.

In [153]:
```
# HW 9.3 - Plot top pagerank scores for both versions
x=range(1,101)
plt.bar(x,scores_5)
plt.xlabel('Rank')
plt.ylabel('Pagerank score')
plt.title('Pagerank results for 100 top pages after 5 iterations')
plt.show()

plt.bar(x,scores_10)
plt.xlabel('Rank')
plt.ylabel('Pagerank score')
plt.title('Pagerank results for 100 top pages after 10 iterations')
plt.show()
```





As shown, the distribution of the top scores doesn't really change much with the additional iterations. However, we do see that the values of the top scores are slightly higher after 10 iterations than after 5.

HW 9.4

9.4 - Problem Statement

Modify your PageRank implementation to produce a topic specific PageRank implementation, as described here (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 problema
tic.
```

and must be adhered to for convergence to be guaranteed.

Run topic specific PageRank on the randomly generated network of 100 nodes (called randNet.txt) which are organized into ten topics, as described in the file randNet_topics.txt

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.

9.4 - Implementation: literature vs assignment

The literature's method is difficult to implement because it would require us to first implement something similar to our distributed shortest-path algorithm in HW7 to determine which nodes were unreachable from in-topic nodes. As we've seen, this would require a large-scale BFS or DFS approach that would need to traverse a significant section of the graph before identifying the unreachable nodes (it's possible that we'd need to traverse the entire graph, but ideally we'd be able to stop early if we came to a decision about each node). Only once we've done this can we make another pass through the dataset (via another job) to remove the unreachable nodes from the graph and evenly distribute the probability mass across the remaining nodes. By using the assignment's approach instead of the literature's, we can approximate the entire first step by dramatically underweighting the out-of-topic nodes. This allows them to still have non-zero values (thus maintaining irreducibility), but ones that are essentially zero.

HW 9.4 - Main Job

Key differences between this job and the version used in 9.1 and 9.2 are that the initial step now has a reducer_init step to load the list of in-topic nodes into memory. This enables the calculation of the topic weight in the setup reducer. The weights must be propagated through the subsequent iterations, which requires some changes to the KV pair structure in this job as well.

In [234]:

```
%%writefile mrtopicpagerank.py
from __future__ import division
from mrjob.job import MRJob
from mrjob.job import MRStep
import ast

class mrTopicPageRank(MRJob):

    def configure_options(self):
        super(mrTopicPageRank, self).configure_options()
        self.add_passthrough_option('--d', default=0.85, type=float,
                                    help='dampening factor')
        self.add_passthrough_option('--N', default=None, type=int,
                                    help='total number of nodes')
        self.add_passthrough_option('--iterations', default=2, type=int
                                    help='how many iterations should we
        self.add_passthrough_option('--B', default=0.99, type=float,
                                    help='weighting for in-topic nodes'
        self.add_passthrough_option('--topic', default=1, type=int,
                                    help='which topic do we care about?


    def mapper_setup(self, nid, nodes_score):
        """Expand dangling nodes"""
        nodes_score = nodes_score.strip('\n')
        nid, nodes = nodes_score.split('\t')
        yield nid, nodes
        nodes=eval(nodes)
        for n,w in nodes.iteritems():
            yield n,'{}'

    def reducer_setup_init(self):
        self.in_topic_nodes=[]
        with open('randNet_topics.txt', 'r') as f:
            for line in f.readlines():
                line=line.split('\t')
                nid = line[0]
                topic = int(line[1])
                if topic == self.options.topic:
                    self.in_topic_nodes.append(nid)
        #print self.in_topic_nodes

    def reducer_setup(self, nid, values):
        """Evenly distribute probability mass across all nodes (danglin

        #Aggregate nodes from mapper
        nodes={}
        for v in values:
            v=eval(v)
            nodes.update(v)

        #Starting score is the same as in regular pagerank
        score = 1/float(self.options.N)
```

```
        #Weight for in-topic nodes
        if nid in self.in_topic_nodes:
            weight=self.options.B/len(self.in_topic_nodes)

        #Weight for out-of-topic nodes
        else:
            weight=(1-self.options.B)*(1/(self.options.N-len(self.in_to

        #Emit result
        yield nid,(score,weight,nodes)



    def mapper_distribute_weights(self, nid, nodes_score):
        """Distribute score evenly across all linked nodes"""
        #print nid, nodes_score
        score,weight,nodes=nodes_score
        #print score,weight,nodes

        # pass along graph structure (NOW WITH WEIGHT ADDED!)
        yield nid, ('node', nodes ,weight)

        # pass mass associated with dangling nodes
        if len(nodes)==0:
            yield '*',('score',score, None)

        else:
            #dispense mass from current node evenly across all linked n
            for n, w in nodes.iteritems():
                yield n, ('score', score*w/len(nodes), None)

    def reducer_init_main(self):
        """Create a place to track dangling mass total"""
        self.dangling_score=0

    def reducer_gather_weights(self, nid, values):
        """Collect scores by node, let topic weight persist through"""
        nodes={}
        total_score = 0
        weight=0

        if nid =='*':
            for typ, value,_ in values:
                self.dangling_score+=value
        else:
            for typ, value,temp_weight in values:
                if typ == 'node':
                    nodes = value
                    weight=temp_weight
                elif typ == 'score':
                    total_score += value
            yield nid,(total_score,weight,nodes)

    def reducer_final_emit_dangling(self):
```

```
        """emit total dangling mass"""
        yield '*',self.dangling_score

    def reducer_init_2(self):
        """Initialize dangling mass total in next step"""
        self.dangling_mass=0

    def reducer_distribute_dangling_weights(self, nid, nodes_score):
        stripe=[v for v in nodes_score][0]

        #Order inversion is our friend here
        if nid=='*':
            self.dangling_mass+=stripe
        else:
            partial_score,weight,nodes=stripe
            d = self.options.d
            new_mass=float(self.dangling_mass/self.options.N)

            #Note that this version is different now with the topic wei
            score = (1-d)*weight + d*float(partial_score+new_mass)

            yield nid,(score,weight,nodes)

    def steps(self):
        return (
                [MRStep(mapper = self.mapper_setup,
                        reducer_init=self.reducer_setup_init,
                       reducer=self.reducer_setup)]
                     +

                 [MRStep(mapper = self.mapper_distribute_weights
                        ,reducer_init=self.reducer_init_main
                        ,reducer = self.reducer_gather_weights
                       ,reducer_final=self.reducer_final_emit_dangling
                       )
                ,
                MRStep(
                    reducer_init=self.reducer_init_2,
                    reducer = self.reducer_distribute_dangling_weights
                        )
                 ]*self.options.iterations
        )

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

```
Overwriting mrtopicpagerank.py
```

HW 9.4 - Driver

In [247]:

```
## HW9.4 - RandNet topic-sensitive pagerank, running locally
%reload_ext autoreload
%autoreload 2
from mrtopicpagerank import mrTopicPageRank
from __future__ import division


def run_jobs(d,topic,output_directory):
    """Driver function to run topic sensitive pagerank for a given topi

    mr_job = mrTopicPageRank(args=[input_directory,
                                  '--no-strict-protocols',
                                  '--d',d,
                                  '--file','randNet_topics.txt',
                                  '--N',str(nodes),
                                  '--topic',str(topic),
                                  '--B',str(beta),
                                  '--iterations',str(num_iterations)])

    with mr_job.make_runner() as runner:
        runner.run()
        #Stream output locally
        with open(output_directory, 'w+') as f:
            for line in runner.stream_output():
                nid,result_stripe =  mr_job.parse_output_line(line)
                score,weight,node_list=result_stripe
                f.write(line)

######## Run the jobs ############

input_dir_prefix='randNet'
input_directory=input_dir_prefix+'.txt'

#We know there are 100 nodes based on the problem description.
#We could easily check this with a separate job if we wanted to scale t
nodes=100
num_iterations=10
beta=0.99

#Calculate pagerank for each topi
for i in range(1,11):
    print "Running jobs for topic {0}".format(str(i))
    output_directory=input_dir_prefix+'Topic{0}Output.txt'.format(str(i
    run_jobs(0.85,i,output_directory)
print "DONE"
```

```
Running jobs for topic 1
Running jobs for topic 2
Running jobs for topic 3
Running jobs for topic 4
Running jobs for topic 5
Running jobs for topic 6
Running jobs for topic 7
Running jobs for topic 8
Running jobs for topic 9
Running jobs for topic 10
DONE
```

In [239]:

```python
#HW 9.4 - Function for examining results

def get_topic_results(result):
    """Extract topic-sensitive pagerank from files, sort, and return to

    num_results=10
    output=[]
    with open(result,'r') as f:
        for line in f.readlines():
            line=line.strip('\n')
            nid,nodes_score=line.split('\t')
            nid=eval(nid)
            nodes_score=eval(nodes_score)
            score,weight,nodes=nodes_score[:]
            score=float(score)
            output.append((score,nid))
            if len(output)>num_results:
                output.sort(key=lambda x: -float(x[0]))
                output=output[:num_results]

    output.sort(key=lambda x: -float(x[0]))
    return output
```

In [238]:

```python
# HW 9.4 - Load index of topics into memory so we can look things up
topic_dict={}
with open('randNet_topics.txt') as f:
    for line in f.readlines():
        node_id,topic=line.strip().split('\t')
        topic_dict[node_id]=topic #Enables us to find topics by node ID
```

In [246]:
```
# HW 9.4 - Prettify and display final output!

for i in range(1,11):
    output_directory=input_dir_prefix+'Topic{0}Output.txt'.format(str(i
    print "Results for Topic {0}".format(str(i))
    print "SCORE    |  ID | TOPIC"
    print "-----------------------------"
    results=get_topic_results(output_directory)
    for row in results:
        score,nid=row
        print '{0:3.6f} |  {1:<4} | {2}  '.format(score,int(nid),topic_
    print ""
```

```
0.019529 |  92   | 1
0.018566 |  10   | 1
0.018523 |  27   | 1
0.017841 |  85   | 7
0.017692 |  98   | 1
0.017514 |  46   | 1
0.016028 |  74   | 10

Results for Topic 2
SCORE    |  ID | TOPIC
-----------------------------
0.030847 |  58   | 2
0.029665 |  71   | 2
0.029297 |  9    | 2
0.028915 |  73   | 2
0.026889 |  12   | 2
0.025800 |  59   | 2
0.024850 |  75   | 2
0.022858 |  82   | 2
0.016322 |  52   | 1
```

In general, it looks like our top topic-sensitive pagerank results seem to correspond to the actual topics of the nodes. Interestingly though, certain off-topic nodes still maintain their high ranking in other topic's lists. This is likely because these nodes are so highly linked that a random surfer is likely to land on them EVEN when the topic weights are applied. For example, many nodes link to node 74 (the highest ranked node for topic 10), which appears in bottom of the top 10 results for topics 1,2,3,6, and 9.

End of Submission