In [1]:
```
%%javascript
/*****************************************************************************
**********************
Known Mathjax Issue with Chrome - a rounding issue adds a border to the
 right of mathjax markup
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-equ
ations-with-a-trailing-vertical-line
*****************************************************************************
*********************/

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

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

MIDS - w261 Machine Learning At Scale

Course Lead: Dr James G. Shanahan (**email** Jimi via James.Shanahan *AT* gmail.com)

Assignment - HW9

Name: *Jason Sanchez*
Class: MIDS w261 (Section *Fall 2016 Group 2*)
Email: *jason.sanchez*@iSchool.Berkeley.edu
Due Time: HW9 is due on Tuesday 11/15/2016.

Table of Contents

1 Instructions

Back to Table of Contents

MIDS UC Berkeley, Machine Learning at Scale DATSCIW261 ASSIGNMENT #9

Version 2016-11-01

INSTRUCTIONS for SUBMISSIONS

Please use the following form for HW submission:

https://docs.google.com/forms/d/1ZOr9RnIe_A06AcZDB6K1mJN4vrLeSmS2PD6Xm3eOiis/viewform?usp=send_form (https://docs.google.com/forms/d/1ZOr9RnIe_A06AcZDB6K1mJN4vrLeSmS2PD6Xm3eOiis/viewform?usp=send_form)

IMPORTANT

HW9 can be completed locally on your computer for most part but will require a cluster of computers for the bigger wikipedia dataset.

Documents:

- IPython Notebook, published and viewable online.
- PDF export of IPython Notebook.

2 Useful References

Back to Table of Contents

- See async and live lectures for this week
- Data-intensive text processing with MapReduce. San Rafael, CA: Morgan & Claypool Publishers. Chapter 5.

HW Problems

Back to Table of Contents

HW 9 Dataset

Note that all referenced files are in the enclosing directory. Checkout the Data subdirectory on Dropbox (https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0) or the AWS S3 buckets (details contained each question).

3. HW9.0 Short answer questions

[Back to Table of Contents](#)

What is PageRank and what is it used for in the context of web search? PageRank is an algorithm used to score pages based on the PageRank scores of inbound links. These scores can be used as a component in ranking pages returned by search engines.

What modifications have to be made to the webgraph in order to leverage the machinery of Markov Chains to compute the Steady State Distibution? Stochasticity to resolve dangling edges and teleportation so that any node can be reached by any other node.

OPTIONAL: In topic-specific pagerank, how can we ensure that the irreducible property is satifsied? (HINT: see HW9.4) Drop nodes that have no inlinks.

In [3]:
```
%matplotlib inline
from __future__ import division, print_function
import matplotlib.pyplot as plt
from numpy.random import choice, rand
from collections import defaultdict
from pprint import pprint
import pandas as pd
import numpy as np
```

HW 9.1 Implementation

3. HW9.1 MRJob implementation of basic PageRank

Back to Table of Contents

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 data located here :

In the Data Subfolder for HW7 on Dropbox (same dataset as HW7) with the same file name.

> Dropbox: https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0 (https://www.dropbox.com/sh/2c0k5adwz36lkcw/AAAAKsjQfF9uHfv-X9mCqr9wa?dl=0)

Or on Amazon:

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

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) and 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
```

Here are some simple in memory implementations of PageRank

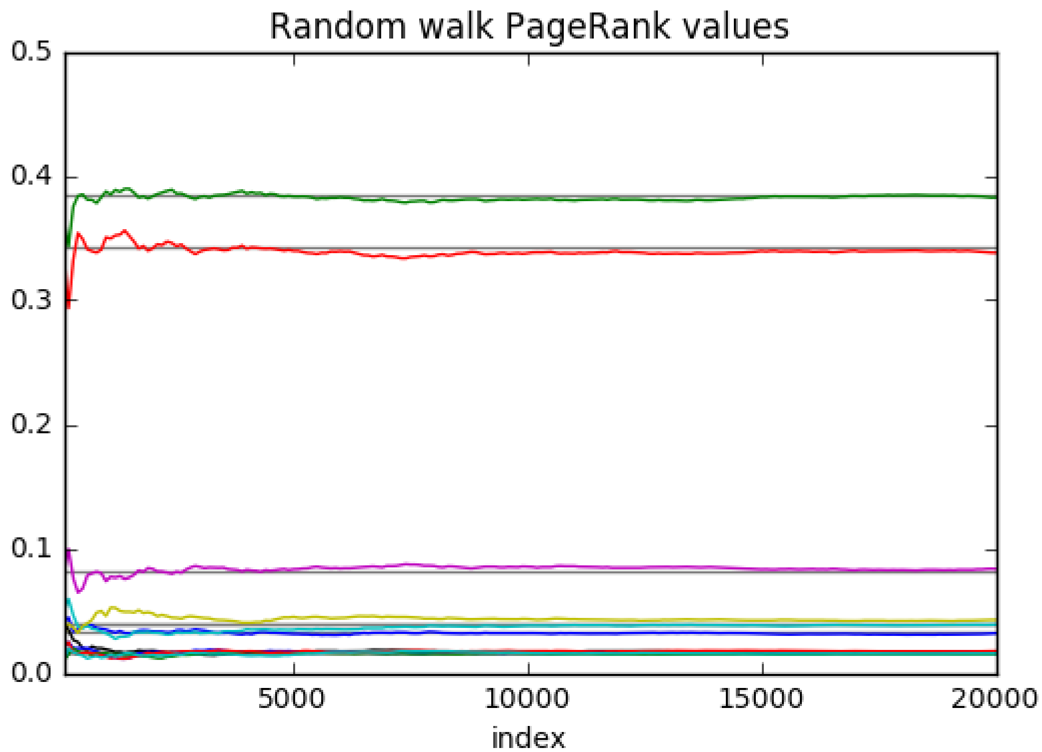
The point of these implementations was for me to deeply understand PageRank.

In [4]:
```
# Here are the correct PR values to compare each implementation against
true_values = [ 0.03278149,  0.38440095,  0.34291029,  0.03908709,  0.08
088569,
                0.03908709,  0.01616948,  0.01616948,  0.01616948,  0.01
616948,
                0.01616948]
```

Perform a simple random walk with adjacency lists and track which pages are visited.

In [5]:
```python
pages = {"B":["C"],
         "C":["B"],
         "D":["A","B"],
         "E":["B","D","F"],
         "F":["B","E"],
         "G":["B","E"],
         "H":["B","E"],
         "I":["B","E"],
         "J":["E"],
         "K":["E"]}

teleport = .15
iterations = 20001
all_nodes = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K"]

iterations_to_plot = 200
page_visits = defaultdict(int)
default_val = 1.0/len(all_nodes)
current_page = pages.keys()[0]
mod = iterations//iterations_to_plot
all_page_visits = []

for i in xrange(iterations):
    if rand() < teleport:
        possible_pages = all_nodes
    else:
        possible_pages = pages.get(current_page, all_nodes)
    current_page = choice(possible_pages)
    page_visits[current_page] += 1
    if i%mod == 0:
        dict_to_save = dict(page_visits)
        dict_to_save["index"] = i
        all_page_visits.append(dict_to_save)

print(dict(page_visits))

total = 0.0
for page, counts in page_visits.items():
    total += counts

for page, counts in page_visits.items():
    print("PageRank for page %s: %f" % (page, counts/total))

data = pd.DataFrame(all_page_visits[1:])
data.index = data.pop("index")
normalized_data = data.div(data.sum(axis=1), axis=0)
normalized_data.plot(legend=False)
plt.ylim(0,.5)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Random walk PageRank values");
```

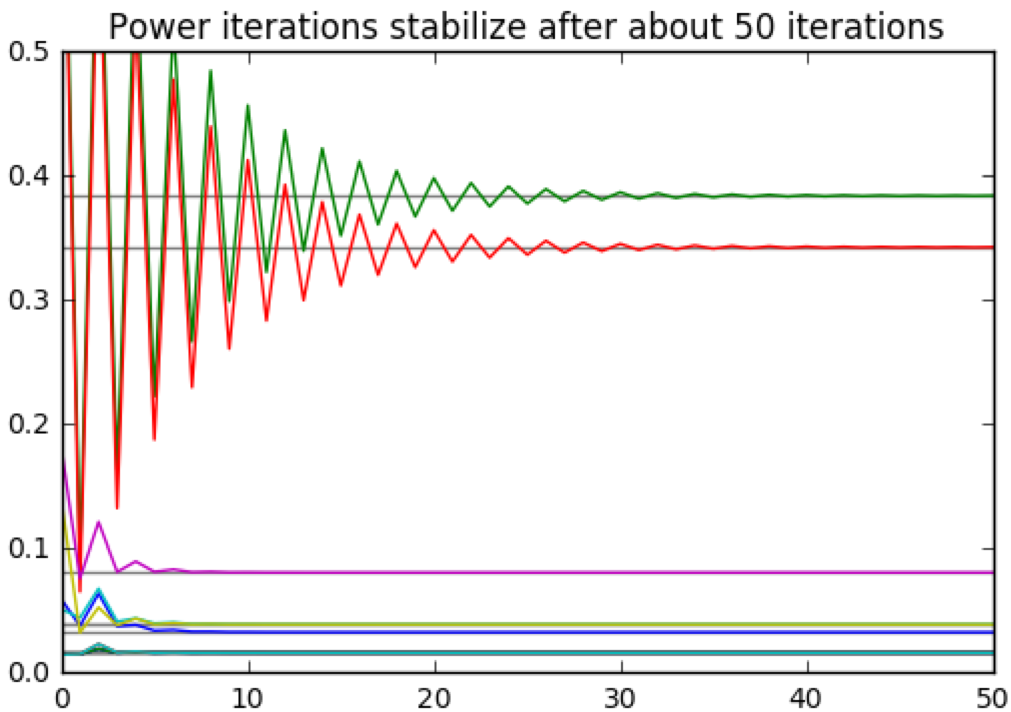
```
{'A': 639, 'C': 6765, 'B': 7657, 'E': 1668, 'D': 780, 'G': 333, 'F': 85
5, 'I': 307, 'H': 330, 'K': 313, 'J': 354}
PageRank for page A: 0.031948
PageRank for page C: 0.338233
PageRank for page B: 0.382831
PageRank for page E: 0.083396
PageRank for page D: 0.038998
PageRank for page G: 0.016649
PageRank for page F: 0.042748
PageRank for page I: 0.015349
PageRank for page H: 0.016499
PageRank for page K: 0.015649
PageRank for page J: 0.017699
```

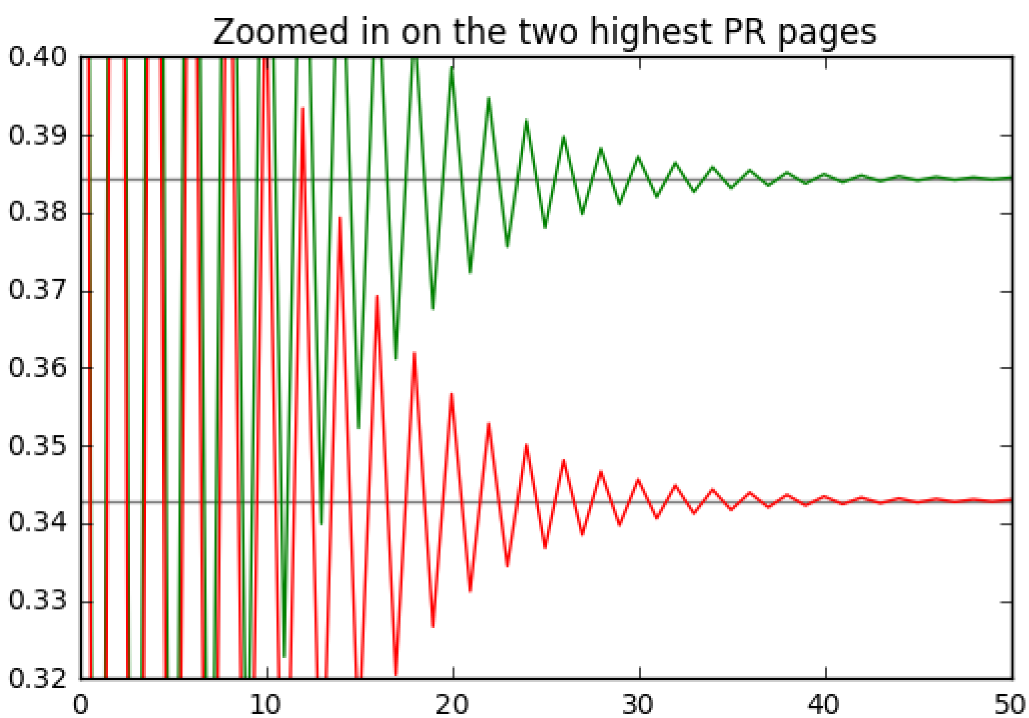


Use power iterations to solve

In [6]:

```
iterations = 51
d = .85

thd = 1/3.0
fll = 1/11.0
T = np.array( [[ fll,  fll,  fll,  fll,  fll,  fll,  fll,  fll,  fll,  f
ll,  fll],
               [ 0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  1. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0.5,  0.5,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  thd,  0. ,  thd,  0. ,  thd,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ]])

teleport = np.ones(T.shape)/T.shape[0]
T = d*T + (1-d)*teleport

stable = T

all_stables = []

for i in xrange(iterations):
    stable = stable.dot(T)
    all_stables.append(stable.diagonal())

plt.plot(all_stables)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Power iterations stabilize after about 50 iterations")
plt.ylim(0,.5)
plt.show()

plt.plot(all_stables)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Zoomed in on the two highest PR pages")
plt.ylim(.32, .4);
```



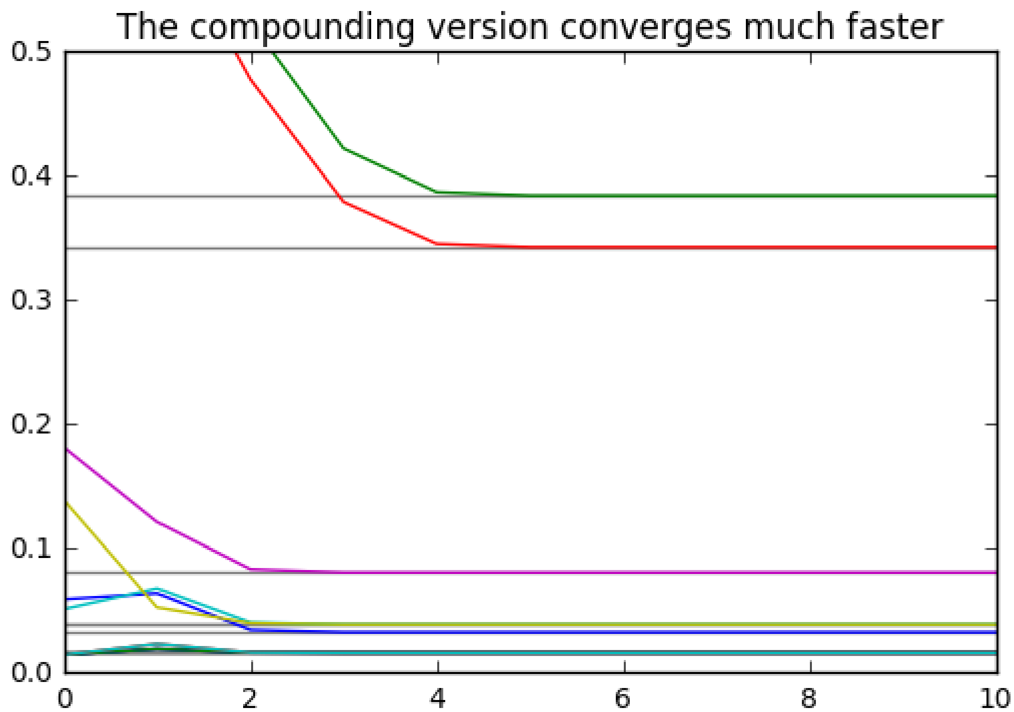


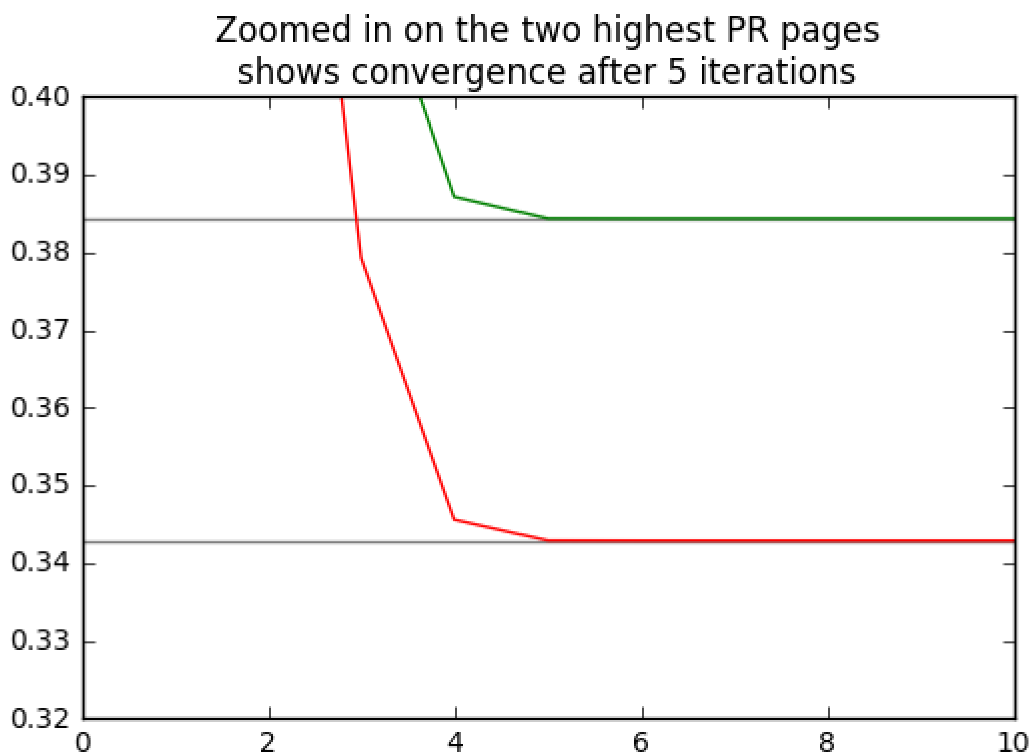
New algo: Power iteration with compounding T

Instead of multiplying the current result by the transition matrix, we can start with the transition matrix and multiply it by itself. This results in much faster convergence.

In [89]:

```
iterations = 11
d = .85

thd = 1/3.0
fll = 1/11.0
T = np.array( [[ fll,  fll,  fll,  fll,  fll,  fll,  fll,  fll,  fll,  f
ll,  fll],
               [ 0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  1. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0.5,  0.5,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  thd,  0. ,  thd,  0. ,  thd,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ],
               [ 0. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,  0. ,
0. ,  0. ]])

teleport = np.ones(T.shape)/T.shape[0]
T = d*T + (1-d)*teleport

all_stables = []

for i in xrange(iterations):
    T = T.dot(T)
    all_stables.append(T.diagonal())

plt.plot(all_stables);
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("The compounding version converges much faster")
plt.ylim(0,.5)
plt.show()

plt.plot(all_stables)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Zoomed in on the two highest PR pages\nshows convergence afte
r 5 iterations")
plt.ylim(.32, .4);

print()
```





My suspicion is that a four iteration solution is possible, but that is probably the limit given smart default PageRanks are not set for each node.

Here is a one-stage solution to the problem that uses a single reducer

In [8]:

```
%%writefile PageRank.py

from __future__ import print_function, division
from mrjob.job import MRJob
from mrjob.job import MRStep
from mrjob.protocol import JSONProtocol
from sys import stderr

class PageRank(MRJob):
    INPUT_PROTOCOL = JSONProtocol

    def configure_options(self):
        super(PageRank,
              self).configure_options()

        self.add_passthrough_option(
            '--n_nodes',
            dest='n_nodes',
            type='float',
            help="""number of nodes
            that have outlinks. You can
            guess at this because the
            exact number will be
            updated after the first
            iteration.""")

    def mapper(self, key, lines):
        # Handles special keys
        # Calculate new Total PR
        # each iteration
        if key in ["****Total PR"]:
            raise StopIteration
        if key in ["**Distribute", "***n_nodes"]:
            # !!! This is where the special
            # hash to the same reducer code
            # will need to go.
            yield (key, lines)
            raise StopIteration
        # Handles the first time the
        # mapper is called. The lists
        # are converted to dictionaries
        # with default PR values.
        if isinstance(lines, list):
            n_nodes = self.options.n_nodes
            default_PR = 1/n_nodes
            lines = {"links":lines,
                     "PR": default_PR}
            # Also perform a node count
            yield ("***n_nodes", 1.0)
        PR = lines["PR"]
        links = lines["links"]
        n_links = len(links)
        # Pass node onward
        yield (key, lines)
        # Track total PR in system
        yield ("****Total PR", PR)
        # If it is not a dangling node
```

```
        # distribute its PR to the
        # other links.
        if n_links:
            PR_to_send = PR/n_links
            for link in links:
                yield (link, PR_to_send)
        else:
            # !!! This is also where the special
            # hash must go.
            yield ("**Distribute", PR)

    def reducer_init(self):
        self.to_distribute = None
        self.n_nodes = None
        self.total_pr = None

    def reducer(self, key, values):
        total = 0
        node_info = None

        for val in values:
            if isinstance(val, float):
                total += val
            else:
                node_info = val

        if node_info:
            distribute = self.to_distribute or 0
            pr = total + distribute
            decayed_pr = .85 * pr
            teleport_pr = .15/self.n_nodes
            new_pr = decayed_pr + teleport_pr
            node_info["PR"] = new_pr
            yield (key, node_info)
        elif key == "****Total PR":
            self.total_pr = total
            yield (key, total)
        elif key == "***n_nodes":
            self.n_nodes = total
            yield (key, total)
        elif key == "**Distribute":
            extra_mass = total
            # Because the node_count and
            # the mass distribution are
            # eventually consistent, a
            # simple correction for any early
            # discrepancies is a good fix
            excess_pr = self.total_pr - 1
            weight = extra_mass - excess_pr
            self.to_distribute = weight/self.n_nodes
        else:
            # The only time this should run
            # is when dangling nodes are
            # discovered during the first
            # iteration. By making them
            # explicitly tracked, the mapper
            # can handle them from now on.
```

```
            yield ("**Distribute", total)
            yield ("***n_nodes", 1.0)
            yield (key, {"PR": total,
                         "links": []})

    def steps(self):
        mr_steps = [MRStep(mapper=self.mapper,
                           reducer_init=self.reducer_init,
                           reducer=self.reducer)]*50
        return mr_steps

if __name__ == "__main__":
    PageRank.run()
```

```
Overwriting PageRank.py
```

In [9]:
```
%reload_ext autoreload
%autoreload 2
from PageRank import PageRank

mr_job = PageRank(args=["data/PageRank-test.txt",
                        "--n_nodes=11",
                        "--jobconf=mapred.reduce.tasks=1"])
with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        print(mr_job.parse_output_line(line))
```

```
(u'****Total PR', 1.0)
(u'***n_nodes', 11.0)
(u'A', {u'PR': 0.03278149315934761, u'links': []})
(u'B', {u'PR': 0.3843611835646984, u'links': [u'C']})
(u'C', {u'PR': 0.34295005075721485, u'links': [u'B']})
(u'D', {u'PR': 0.039087092099970085, u'links': [u'A', u'B']})
(u'E', {u'PR': 0.08088569323450426, u'links': [u'B', u'D', u'F']})
(u'F', {u'PR': 0.039087092099970085, u'links': [u'B', u'E']})
(u'G', {u'PR': 0.016169479016858924, u'links': [u'B', u'E']})
(u'H', {u'PR': 0.016169479016858924, u'links': [u'B', u'E']})
(u'I', {u'PR': 0.016169479016858924, u'links': [u'B', u'E']})
(u'J', {u'PR': 0.016169479016858924, u'links': [u'E']})
(u'K', {u'PR': 0.016169479016858924, u'links': [u'E']})
```

Here is a one-stage solution that uses multiple reducers

In [87]:

```
%%writefile PageRank.py
from __future__ import print_function, division
import itertools
from mrjob.job import MRJob
from mrjob.job import MRStep
from mrjob.protocol import JSONProtocol
from sys import stderr
from random import random

class PageRank(MRJob):
    INPUT_PROTOCOL = JSONProtocol

    def configure_options(self):
        super(PageRank,
              self).configure_options()

        self.add_passthrough_option(
            '--n_nodes',
            dest='n_nodes',
            type='float',
            help="""number of nodes
            that have outlinks. You can
            guess at this because the
            exact number will be
            updated after the first
            iteration.""")

        self.add_passthrough_option(
            '--reduce.tasks',
            dest='reducers',
            type='int',
            help="""number of reducers
            to use. Controls the hash
            space of the custom
            partitioner""")

        self.add_passthrough_option(
            '--iterations',
            dest='iterations',
            type='int',
            help="""number of iterations
            to perform.""")

        self.add_passthrough_option(
            '--damping_factor',
            dest='d',
            default=.85,
            type='float',
            help="""Is the damping
            factor. Must be between
            0 and 1.""")

        self.add_passthrough_option(
            '--smart_updating',
            dest='smart_updating',
            type='str',
            default="False",
```

```
        help="""Can be True or
        False. If True, all updates
        to the new PR will take into
        account the value of the old
        PR.""")

def mapper_init(self):
    self.values = {"****Total PR": 0.0,
                   "***n_nodes": 0.0,
                   "**Distribute": 0.0}
    self.n_reducers = self.options.reducers

def mapper(self, key, lines):
    n_reducers = self.n_reducers
    key_hash = hash(key)%n_reducers
    # Handles special keys
    # Calculate new Total PR
    # each iteration
    if key in ["****Total PR"]:
        raise StopIteration
    if key in ["**Distribute"]:
        self.values[key] += lines
        raise StopIteration
    if key in ["***n_nodes"]:
        self.values[key] += lines
        raise StopIteration
    # Handles the first time the
    # mapper is called. The lists
    # are converted to dictionaries
    # with default PR values.
    if isinstance(lines, list):
        n_nodes = self.options.n_nodes
        default_PR = 1/n_nodes
        lines = {"links":lines,
                 "PR": default_PR}
    # Perform a node count each time
    self.values["***n_nodes"] += 1.0
    PR = lines["PR"]
    links = lines["links"]
    n_links = len(links)
    # Pass node onward
    yield (key_hash, (key, lines))
    # Track total PR in system
    self.values["****Total PR"] += PR
    # If it is not a dangling node
    # distribute its PR to the
    # other links.
    if n_links:
        PR_to_send = PR/n_links
        for link in links:
            link_hash = hash(link)%n_reducers
            yield (link_hash, (link, PR_to_send))
    else:
        self.values["**Distribute"] = PR

def mapper_final(self):
    for key, value in self.values.items():
```

```
            for k in range(self.n_reducers):
                yield (k, (key, value))

    def reducer_init(self):
        self.d = self.options.d
        smart = self.options.smart_updating
        if smart == "True":
            self.smart = True
        elif smart == "False":
            self.smart = False
        else:
            msg = """--smart_updating should
                       be True or False"""
            raise Exception(msg)
        self.to_distribute = None
        self.n_nodes = None
        self.total_pr = None

    def reducer(self, hash_key, combo_values):
        gen_values = itertools.groupby(combo_values,
                                       key=lambda x:x[0])
        for key, values in gen_values:
            total = 0
            node_info = None

            for key, val in values:
                if isinstance(val, float):
                    total += val
                else:
                    node_info = val

            if node_info:
                old_pr = node_info["PR"]
                distribute = self.to_distribute or 0
                pr = total + distribute
                decayed_pr = self.d * pr
                teleport_pr = (1-self.d)/self.n_nodes
                new_pr = decayed_pr + teleport_pr
                if self.smart:
                    # If the new value is less than
                    # 30% different than the old
                    # value, set the new PR to be
                    # 80% of the new value and 20%
                    # of the old value.
                    diff = abs(new_pr - old_pr)
                    percent_diff = diff/old_pr
                    if percent_diff < .3:
                        new_pr = .8*new_pr + .2*old_pr
                node_info["PR"] = new_pr
                yield (key, node_info)
            elif key == "****Total PR":
                self.total_pr = total
            elif key == "***n_nodes":
                self.n_nodes = total
            elif key == "**Distribute":
                extra_mass = total
                # Because the node_count and
```

```
                # the mass distribution are
                # eventually consistent, a
                # simple correction for any early
                # discrepancies is a good fix
                excess_pr = self.total_pr - 1
                weight = extra_mass - excess_pr
                self.to_distribute = weight/self.n_nodes
            else:
                # The only time this should run
                # is when dangling nodes are
                # discovered during the first
                # iteration. By making them
                # explicitly tracked, the mapper
                # can handle them from now on.
                yield ("**Distribute", total)
                yield ("***n_nodes", 1.0)
                yield (key, {"PR": total,
                             "links": []})

    def reducer_final(self):
        print_info = False
        if print_info:
            print("Total PageRank", self.total_pr)

    def steps(self):
        iterations = self.options.iterations
        mr_steps = [MRStep(mapper_init=self.mapper_init,
                           mapper=self.mapper,
                           mapper_final=self.mapper_final,
                           reducer_init=self.reducer_init,
                           reducer=self.reducer,
                           reducer_final=self.reducer_final)]
        return mr_steps*iterations


if __name__ == "__main__":
    PageRank.run()
```

```
Overwriting PageRank.py
```

This implementation converges to the correct answer. Fifty iterations takes about 2.5 seconds.

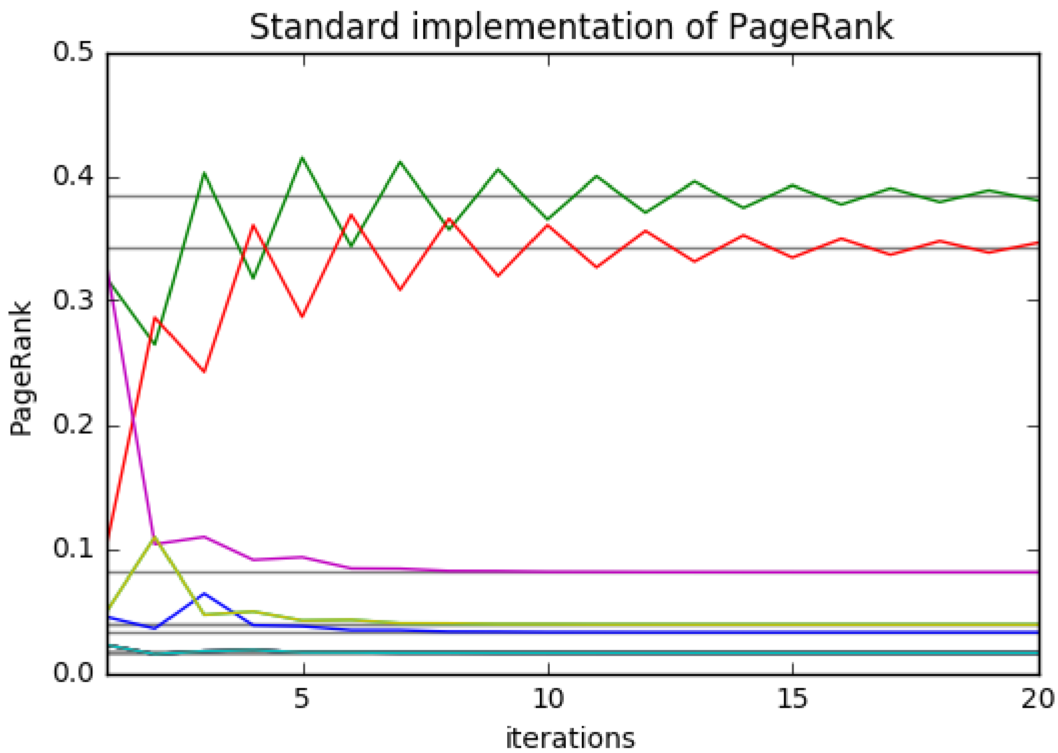
In [79]:
```
%%time
%reload_ext autoreload
%autoreload 2
from PageRank import PageRank

mr_job = PageRank(args=["data/PageRank-test.txt",
                        "--iterations=50",
                        "--n_nodes=11",
                        "--damping_factor=.85",
                        "--jobconf=mapred.reduce.tasks=5",
                        "--reduce.tasks=5"])

results = {}
with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        result = mr_job.parse_output_line(line)
        results[result[0]] = result[1]["PR"]
pprint(results)
```

```
{u'A': 0.03278149315934773,
 u'B': 0.3843730253341818,
 u'C': 0.342938208987731,
 u'D': 0.03908709209997017,
 u'E': 0.08088569323450442,
 u'F': 0.03908709209997017,
 u'G': 0.016169479016858956,
 u'H': 0.016169479016858956,
 u'I': 0.016169479016858956,
 u'J': 0.016169479016858956,
 u'K': 0.016169479016858956}
CPU times: user 1.81 s, sys: 508 ms, total: 2.31 s
Wall time: 2.58 s
```

The chart below investigates how the PageRank parameters evolve as a function of the number of iterations in the standard algorithm.

In [86]:

```
%reload_ext autoreload
%autoreload 2
from PageRank import PageRank

all_results = []

for iteration in range(1, 21):
    mr_job = PageRank(args=["data/PageRank-test.txt",
                            "--iterations=%d" % iteration,
                            "--n_nodes=11",
                            "--damping_factor=.85",
                            "--jobconf=mapred.reduce.tasks=5",
                            "--reduce.tasks=5"])

    results = {}
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            result = mr_job.parse_output_line(line)
            try:
                results[result[0]] = result[1]["PR"]
            except:
                pass
        results["index"] = iteration
    all_results.append(results)

data = pd.DataFrame(all_results)
data.index = data.pop("index")
data.plot(kind="line", legend=False)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Standard implementation of PageRank")
plt.xlabel("iterations")
plt.ylabel("PageRank")
plt.ylim(0,.5)
plt.show()
```



Notice the oscillation in the scores above. This is likely because there is a feedback loop between the two most highly ranked pages. This oscillation makes sense because B and C are only linked to each other and they both have very high PageRank scores.

In order to fix this and increase the speed of convergence, I added a new PageRank update rule that can be turned on using the `--smart_updating=True` argument. This update rule does the following:

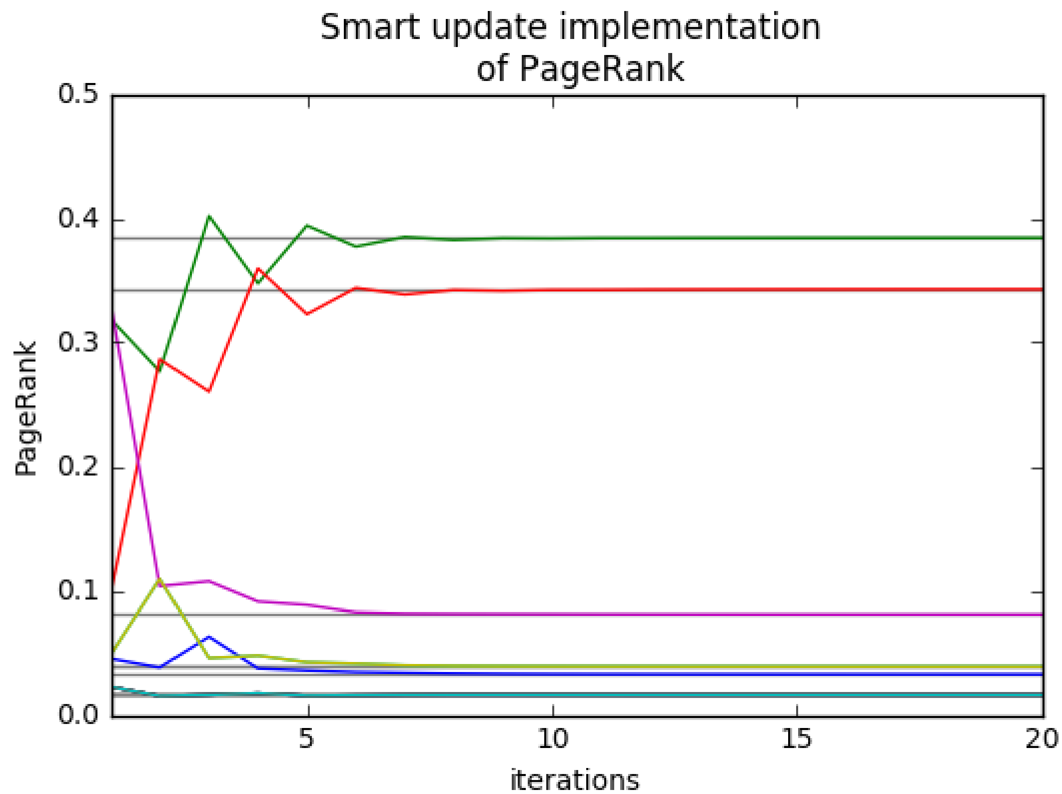
- Compare the old and new PageRank for a node
- If the percent difference is less than 30%, the actual PageRank value assigned to the node is 75% of the new value plus 25% of the old value.

If there is a big change between the old and new PageRank values (common during the first iterations of the algorithm), the actual PageRank value used is the standard value used. This allows each page to rapidly get to its approximately correct place.

If there is not a big change, oscillations are removed by smoothing the new PageRank value with the past PageRank value.

In [88]:

```
%reload_ext autoreload
%autoreload 2
from PageRank import PageRank

all_results = []

for iteration in range(1, 21):
    mr_job = PageRank(args=["data/PageRank-test.txt",
                            "--iterations=%d" % iteration,
                            "--n_nodes=11",
                            "--damping_factor=.85",
                            "--jobconf=mapred.reduce.tasks=5",
                            "--reduce.tasks=5",
                            "--smart_updating=True"])

    results = {}
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            result = mr_job.parse_output_line(line)
            try:
                results[result[0]] = result[1]["PR"]
            except:
                pass
        results["index"] = iteration
    all_results.append(results)

data = pd.DataFrame(all_results)
data.index = data.pop("index")
data.plot(kind="line", legend=False)
plt.hlines(true_values,0,iterations-1, colors="grey")
plt.title("Smart update implementation \n of PageRank")
plt.xlabel("iterations")
plt.ylabel("PageRank")
plt.ylim(0,.5)
plt.show()
```



The updated algorithm converges much faster on the dataset and the oscillations are removed.

HW 9.1 Analysis

In the lectures, it was said that a one-stage PageRank algorithm was not possible. This is a working, fully distributed, one-stage PageRank algorithm.

Also, we found that we could adjust the update process to lead to significantly faster convergences on this dataset.

3. HW9.2: Exploring PageRank teleportation and network plots

Back to Table of Contents

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

HW 9.2 Implementation

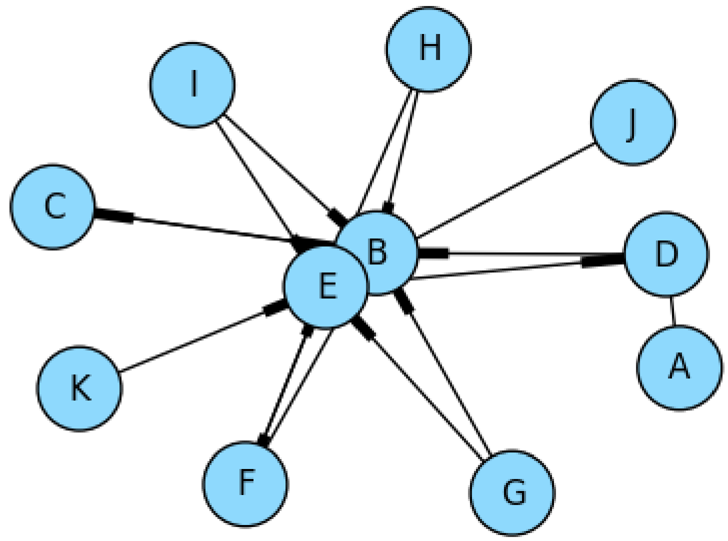
In [91]:
```
import networkx as nx
```

In [164]:

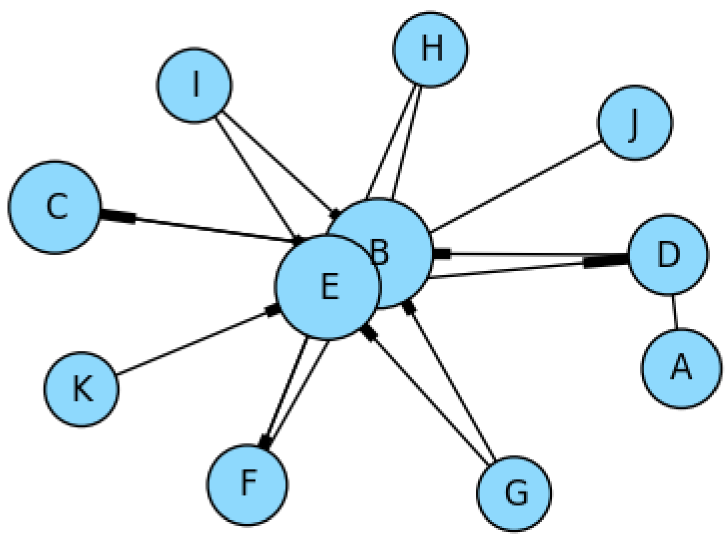
```
def display_graph(edges, PageRanks, title, node_scaling=10000,
pos=None):
    DG = nx.DiGraph()
    DG.add_edges_from(edges)

    w = []
    for node in DG.nodes():
        weight = PageRanks[node]
        w.append(weight*node_scaling)

    nx.draw_networkx(DG,
                     node_size=w,
                     node_color="#8ED9FD",
                     pos=pos)
    plt.title(title)
    plt.axis('off')
    plt.show()


pages = {"B":["C"],
         "C":["B"],
         "D":["A","B"],
         "E":["B","D","F"],
         "F":["B","E"],
         "G":["B","E"],
         "H":["B","E"],
         "I":["B","E"],
         "J":["E"],
         "K":["E"]}

edges = []

for page, links in pages.items():
    for link in links:
        edges.append([page, link])

# Get constant positions
DG = nx.DiGraph()
DG.add_edges_from(edges)
pos = nx.layout.spring_layout(DG)

for damping_factor in [0,0.25,0.5,0.75, 0.85, 1]:
    mr_job = PageRank(args=["data/PageRank-test.txt",
                            "--iterations=20",
                            "--n_nodes=11",
                            "--damping_factor=%f" % damping_factor,
                            "--jobconf=mapred.reduce.tasks=5",
                            "--reduce.tasks=5",
                            "--smart_updating=True"])
    results = {}
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            result = mr_job.parse_output_line(line)
            try:
                results[result[0]] = result[1]["PR"]
            except:
```

```
                pass

    display_graph(edges, results, "Damping factor: %.2f" % damping_facto
r, pos=pos)
```

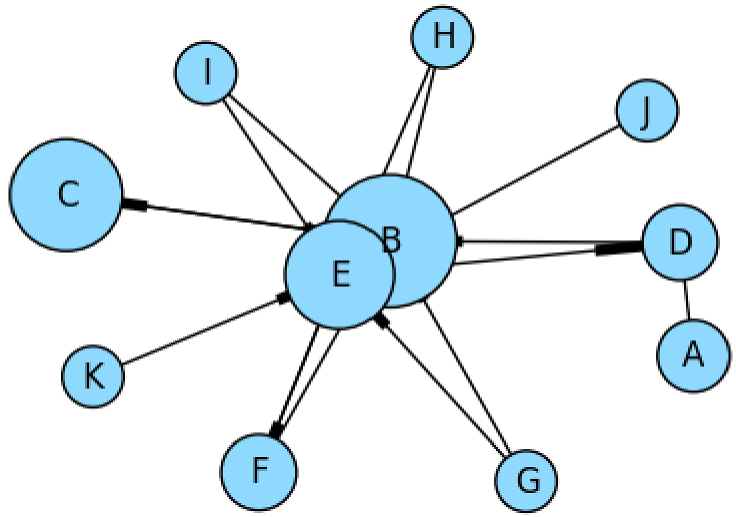
Damping factor: 0.00



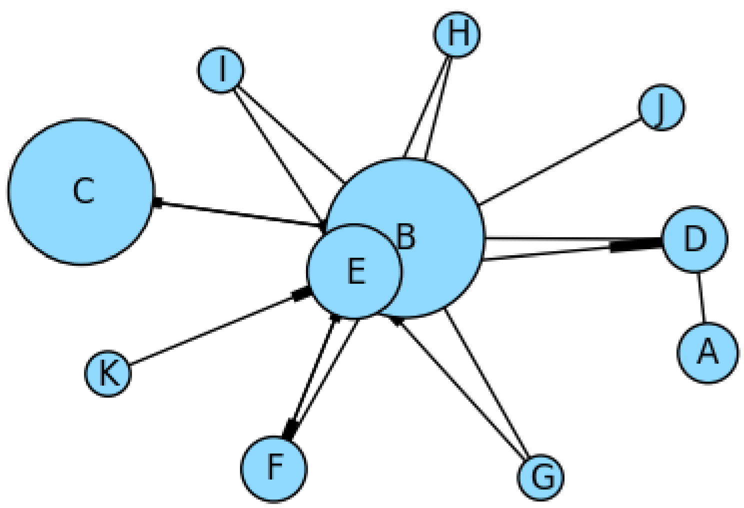
Damping factor: 0.25



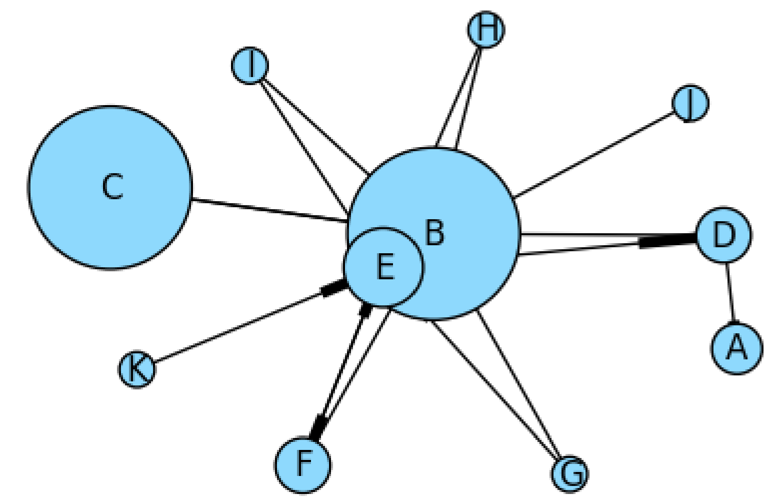
Damping factor: 0.50



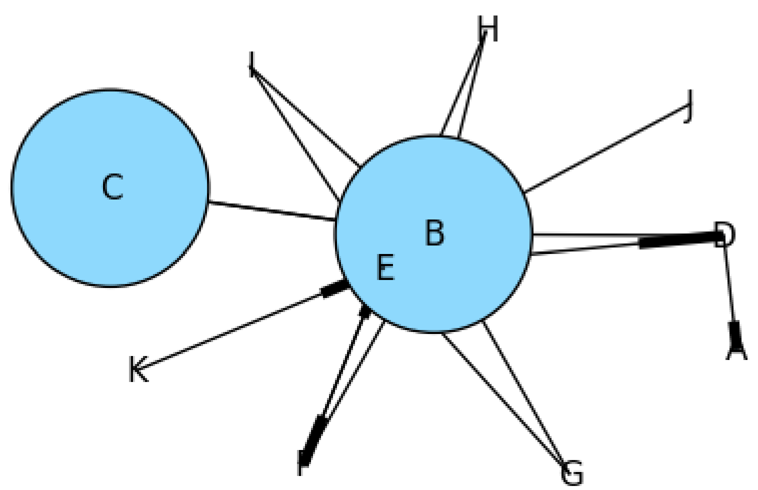
Damping factor: 0.75



Damping factor: 0.85



Damping factor: 1.00



3. HW9.3: Applying PageRank to the Wikipedia hyperlinks network

Back to Table of Contents

<

- 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 [8]:
```
!head -n 100 Temp_data/all-pages-indexed-out.txt > Temp_data/wiki_test.t
xt
```

In [9]:
```
!head Temp_data/wiki_test.txt
```

```
73      {'14417532': 1}
299     {'4214575': 1}
2552    {'15043376': 1, '13430968': 1, '13451035': 1, '7263397': 1, '13
001625': 1, '13443575': 1, '13451269': 1, '13432316': 1, '11623371': 1,
 '15028971': 1, '13425865': 1, '15042703': 1, '5051368': 1, '9854998':
 2, '13442976': 1, '13315025': 1, '2992307': 1, '1054486': 1, '132232
5': 1, '13450983': 1}
2570    {'983991': 1}
2616    {'9045350': 1}
2711    {'752887': 1}
2818    {'3534183': 1}
2847    {'3797918': 1}
2892    {'2893': 1}
2921    {'5158607': 1, '6007184': 1, '14773825': 1, '11777840': 2, '928
5165': 1, '6420484': 1, '14670682': 1, '7316613': 1, '7125893': 1, '149
65920': 1, '14229952': 1, '9447742': 2, '1425342': 1, '11390944': 2, '5
141': 1, '14928135': 2, '13636570': 3, '14687433': 1, '15105458': 1, '1
1656072': 1, '6420027': 1, '10898196': 1, '6416278': 1, '11497740': 2}
```

I ran this code at least 20 times and consistently made errors with the config settings. I will continue to work on this section of the code. The current version works on a sample of the Wikipedia dataset (i.e. first 10000 lines). It also works locally in the -r local mode.

My goal is to figure out how to spin up a hundred node cluster and compute the answer in a very fast amount of time.

In [16]:
```
%%writefile WikiPageRank.py
from __future__ import print_function, division
import itertools
from mrjob.job import MRJob
from mrjob.job import MRStep
from mrjob.protocol import JSONProtocol
from sys import stderr
from random import random
import json

class WikiPageRank(MRJob):
    def configure_options(self):
        super(WikiPageRank,
              self).configure_options()

        self.add_passthrough_option(
            '--n_nodes',
            dest='n_nodes',
            type='float',
            help="""number of nodes
            that have outlinks. You can
            guess at this because the
            exact number will be
            updated after the first
            iteration.""")

        self.add_passthrough_option(
            '--reduce.tasks',
            dest='reducers',
            type='int',
            help="""number of reducers
            to use. Controls the hash
            space of the custom
            partitioner""")

        self.add_passthrough_option(
            '--iterations',
            dest='iterations',
            type='int',
            help="""number of iterations
            to perform.""")

        self.add_passthrough_option(
            '--damping_factor',
            dest='d',
            default=.85,
            type='float',
            help="""Is the damping
            factor. Must be between
            0 and 1.""")

        self.add_passthrough_option(
            '--smart_updating',
            dest='smart_updating',
            type='str',
            default="False",
            help="""Can be True or
```

```
            False. If True, all updates
            to the new PR will take into
            account the value of the old
            PR.""")

    def clean_data(self, _, lines):
        key, value = lines.split("\t")
        value = json.loads(value.replace("'", '"'))
        values = value.keys()
        yield (str(key), values)

    def mapper_init(self):
        self.values = {"****Total PR": 0.0,
                       "***n_nodes": 0.0,
                       "**Distribute": 0.0}
        self.n_reducers = self.options.reducers

    def mapper(self, key, lines):
        n_reducers = self.n_reducers
        key_hash = hash(key)%n_reducers
        # Handles special keys
        # Calculate new Total PR
        # each iteration
        if key in ["****Total PR"]:
            raise StopIteration
        if key in ["**Distribute"]:
            self.values[key] += lines
            raise StopIteration
        if key in ["***n_nodes"]:
            self.values[key] += lines
            raise StopIteration
        # Handles the first time the
        # mapper is called. The lists
        # are converted to dictionaries
        # with default PR values.
        if isinstance(lines, list):
            n_nodes = self.options.n_nodes
            default_PR = 1/n_nodes
            lines = {"links":lines,
                     "PR": default_PR}
        # Perform a node count each time
        self.values["***n_nodes"] += 1.0
        PR = lines["PR"]
        links = lines["links"]
        n_links = len(links)
        # Pass node onward
        yield (key_hash, (key, lines))
        # Track total PR in system
        self.values["****Total PR"] += PR
        # If it is not a dangling node
        # distribute its PR to the
        # other links.
        if n_links:
            PR_to_send = PR/n_links
            for link in links:
                link_hash = hash(link)%n_reducers
                yield (link_hash, (link, PR_to_send))
```

```
        else:
            self.values["**Distribute"] = PR

    def mapper_final(self):
        for key, value in self.values.items():
            for k in range(self.n_reducers):
                yield (k, (key, value))

    def reducer_init(self):
        self.d = self.options.d
        smart = self.options.smart_updating
        if smart == "True":
            self.smart = True
        elif smart == "False":
            self.smart = False
        else:
            msg = """--smart_updating should
                     be True or False"""
            raise Exception(msg)
        self.to_distribute = None
        self.n_nodes = None
        self.total_pr = None

    def reducer(self, hash_key, combo_values):
        gen_values = itertools.groupby(combo_values,
                                       key=lambda x:x[0])
        for key, values in gen_values:
            total = 0
            node_info = None

            for key, val in values:
                if isinstance(val, float):
                    total += val
                else:
                    node_info = val

            if node_info:
                old_pr = node_info["PR"]
                distribute = self.to_distribute or 0
                pr = total + distribute
                decayed_pr = self.d * pr
                teleport_pr = (1-self.d)/self.n_nodes
                new_pr = decayed_pr + teleport_pr
                if self.smart:
                    # If the new value is less than
                    # 30% different than the old
                    # value, set the new PR to be
                    # 80% of the new value and 20%
                    # of the old value.
                    diff = abs(new_pr - old_pr)
                    percent_diff = diff/old_pr
                    if percent_diff < .3:
                        new_pr = .8*new_pr + .2*old_pr
                if new_pr < 0:
                    new_pr = 0
                node_info["PR"] = new_pr
                yield (key, node_info)
```

```
            elif key == "****Total PR":
                self.total_pr = total
            elif key == "***n_nodes":
                self.n_nodes = total
            elif key == "**Distribute":
                extra_mass = total
                # Because the node_count and
                # the mass distribution are
                # eventually consistent, a
                # simple correction for any early
                # discrepancies is a good fix
                excess_pr = self.total_pr - 1
                weight = extra_mass - excess_pr
                self.to_distribute = weight/self.n_nodes
            else:
                # The only time this should run
                # is when dangling nodes are
                # discovered during the first
                # iteration. By making them
                # explicitly tracked, the mapper
                # can handle them from now on.
                yield ("**Distribute", total)
                yield ("***n_nodes", 1.0)
                yield (key, {"PR": total,
                             "links": []})

    def reducer_final(self):
        print_info = False
        if print_info:
            print("Total PageRank", self.total_pr)

    def steps(self):
        iterations = self.options.iterations
        mr_steps = ([MRStep(mapper=self.clean_data)]
                     +
                     [MRStep(
                            mapper_init=self.mapper_init,
                            mapper=self.mapper,
                            mapper_final=self.mapper_final,
                            reducer_init=self.reducer_init,
                            reducer=self.reducer,
                            reducer_final=self.reducer_final
                             )]*iterations
                    )
        return mr_steps


if __name__ == "__main__":
    WikiPageRank.run()
```

```
Overwriting WikiPageRank.py
```

In [17]:
```
import heapq

class TopList(list):
    def __init__(self, max_size, num_position=0):
        """
        Just like a list, except the append method adds the new value to
 the
        list only if it is larger than the smallest value (or if the siz
e of
        the list is less than max_size).

        If each element of the list is an int or float, uses that value
 for
        comparison. If the elements in the list are lists or tuples, use
s the
        list_position element of the list or tuple for the comparison.
        """
        self.max_size = max_size
        self.pos = num_position

    def _get_key(self, x):
        return x[self.pos] if isinstance(x, (list, tuple)) else x

    def append(self, val):
        if len(self) < self.max_size:
            heapq.heappush(self, val)
        elif self._get_key(self[0]) < self._get_key(val):
            heapq.heapreplace(self, val)

    def final_sort(self):
        return sorted(self, key=self._get_key, reverse=True)
```

In [ ]:

```
# %%time
%reload_ext autoreload
%autoreload 2
from WikiPageRank import WikiPageRank

mr_job = WikiPageRank(args=["Temp_data/wiki_test.txt",
                            "--iterations=1",
                            "--n_nodes=11",
                            "--damping_factor=.85",
#                             "--jobconf=mapred.reduce.tasks=5",
                            "--reduce.tasks=5",
#                            "--pool-clusters",
#                             "--instance-type", "m4.xlarge",
#                             "--num-core-instances", "4",
#                             "--ec2-core-instance-bid-price", "0.1",
                            "-r", "emr"])

results = TopList(max_size=100, num_position=1)

with mr_job.make_runner() as runner:
    runner.run()
    i=0
    for line in runner.stream_output():
        result = mr_job.parse_output_line(line)
        try:
            results.append((result[0], result[1]["PR"]))
        except:
            i += 1
            if i < 100:
                print(result)
```

In [13]:

```
!mrjob create-cluster --max-hours-idle 1 --master-instance-type=m3.xlarg
e
```

```
Using configs in /Users/BlueOwl1/.mrjob.conf
Using s3://mrjob-3d3e189cec521ef3/tmp/ as our temp dir on S3
Creating persistent cluster to run several jobs in...
Creating temp directory /var/folders/sz/4k2bbjts7x5fmg9sn7kh6hlw0000gn/
T/no_script.Jason.20161116.104414.914466
Copying local files to s3://mrjob-3d3e189cec521ef3/tmp/no_script.Jason.
20161116.104414.914466/files/...
j-NSRHH0RMLJ7
```

In [ ]:
```
!python WikiPageRank.py -r emr --cluster-id j-NSRHH0RMLJ7 --iterations=1
--master-instance-type=m3.xlarge --n_nodes=11 --damping_factor=.85 --red
uce.tasks=5 Temp_data/wiki_test.txt
```

```
Using configs in /Users/BlueOwl1/.mrjob.conf
Using s3://mrjob-3d3e189cec521ef3/tmp/ as our temp dir on S3
Creating temp directory /var/folders/sz/4k2bbjts7x5fmg9sn7kh6hlw0000gn/
T/WikiPageRank.Jason.20161116.110241.924886
Copying local files to s3://mrjob-3d3e189cec521ef3/tmp/WikiPageRank.Jas
on.20161116.110241.924886/files/...
Adding our job to existing cluster j-NSRHH0RMLJ7
  Opening ssh tunnel to resource manager...
  Connect to resource manager at: http://localhost:40743/cluster
Waiting for step 1 of 2 (s-3A0EOXM11UUFM) to complete...
  RUNNING for 28.0s
     5.0% complete
```

In [ ]:
```
results = results.final_sort()
```

In [16]:
```
%%time
labels = {}

with open("Temp_data/indices.txt") as label_data:
    for line in label_data:
        data = line.strip().split("\t")
        text = data[0]
        position = data[1]
        labels[position] = text
```

```
CPU times: user 20.3 s, sys: 1.56 s, total: 21.9 s
Wall time: 22.1 s
```

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

sorted_top_100 = [(labels[number].decode('utf-8')[:30], pr) for (number,
 pr) in results]
index, values = zip(*sorted_top_100)
data = pd.Series(data=values, index=index).sort_values()
data.plot(kind="barh", figsize=(2,18))
plt.locator_params(axis='x',nbins=4)
plt.xticks(rotation=90);
```

In [ ]:
```
## Code goes here
```

In [ ]:
```
## Drivers & Runners
```

In [ ]:
```
## Run Scripts, S3 Sync
```

HW 9.3 Analysis

3. HW9.4: Topic-specific PageRank implementation using MRJob

Back to Table of Contents

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 (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 [ ]:
```
## Code goes here
```

In [87]:

```
%%writefile TopicPageRank.py
from __future__ import print_function, division
import itertools
from mrjob.job import MRJob
from mrjob.job import MRStep
from mrjob.protocol import JSONProtocol
from sys import stderr
from random import random

class TopicPageRank(MRJob):
    INPUT_PROTOCOL = JSONProtocol

    def configure_options(self):
        super(PageRank,
              self).configure_options()

        self.add_passthrough_option(
            '--n_nodes',
            dest='n_nodes',
            type='float',
            help="""number of nodes
            that have outlinks. You can
            guess at this because the
            exact number will be
            updated after the first
            iteration.""")

        self.add_passthrough_option(
            '--reduce.tasks',
            dest='reducers',
            type='int',
            help="""number of reducers
            to use. Controls the hash
            space of the custom
            partitioner""")

        self.add_passthrough_option(
            '--iterations',
            dest='iterations',
            type='int',
            help="""number of iterations
            to perform.""")

        self.add_passthrough_option(
            '--damping_factor',
            dest='d',
            default=.85,
            type='float',
            help="""Is the damping
            factor. Must be between
            0 and 1.""")

        self.add_passthrough_option(
            '--smart_updating',
            dest='smart_updating',
            type='str',
            default="False",
```

```
            help="""Can be True or
            False. If True, all updates
            to the new PR will take into
            account the value of the old
            PR.""")

    def mapper_init(self):
        self.values = {"****Total PR": 0.0,
                       "***n_nodes": 0.0,
                       "**Distribute": 0.0} ### Distribute should be a d
ictionary where keys
                                            ### are topics and values ar
e PR
        self.n_reducers = self.options.reducers

    def mapper(self, key, lines):
        n_reducers = self.n_reducers
        key_hash = hash(key)%n_reducers
        # Handles special keys
        # Calculate new Total PR
        # each iteration
        if key == "****Total PR":
            raise StopIteration
        if key == "**Distribute":
            self.values[key] += lines
            raise StopIteration
        if key == "***n_nodes":
            self.values[key] += lines
            raise StopIteration
        # Handles the first time the
        # mapper is called. The lists
        # are converted to dictionaries
        # with default PR values.
        if isinstance(lines, list):
            n_nodes = self.options.n_nodes
            default_PR = 1/n_nodes
            lines = {"links":lines,
                     "PR": default_PR}
        # Perform a node count each time
        self.values["***n_nodes"] += 1.0
        PR = lines["PR"]
        links = lines["links"]
        n_links = len(links)
        # Pass node onward
        yield (key_hash, (key, lines))
        # Track total PR in system
        self.values["****Total PR"] += PR
        # If it is not a dangling node
        # distribute its PR to the
        # other links.
        if n_links:
            PR_to_send = PR/n_links
            for link in links:
                link_hash = hash(link)%n_reducers
                yield (link_hash, (link, PR_to_send))
        else:
            self.values["**Distribute"] = PR
```

```
    def mapper_final(self):
        for key, value in self.values.items():
            for k in range(self.n_reducers):
                yield (k, (key, value))

    def reducer_init(self):
        self.d = self.options.d
        smart = self.options.smart_updating
        if smart == "True":
            self.smart = True
        elif smart == "False":
            self.smart = False
        else:
            msg = """--smart_updating should
                       be True or False"""
            raise Exception(msg)
        self.to_distribute = None
        self.n_nodes = None
        self.total_pr = None

    def reducer(self, hash_key, combo_values):
        gen_values = itertools.groupby(combo_values,
                                       key=lambda x:x[0])
        for key, values in gen_values:
            total = 0
            node_info = None

            for key, val in values:
                if isinstance(val, float):
                    total += val
                else:
                    node_info = val

            if node_info:
                old_pr = node_info["PR"]
                distribute = self.to_distribute or 0
                pr = total + distribute
                decayed_pr = self.d * pr
                teleport_pr = (1-self.d)/self.n_nodes
                new_pr = decayed_pr + teleport_pr
                if self.smart:
                    # If the new value is less than
                    # 30% different than the old
                    # value, set the new PR to be
                    # 80% of the new value and 20%
                    # of the old value.
                    diff = abs(new_pr - old_pr)
                    percent_diff = diff/old_pr
                    if percent_diff < .3:
                        new_pr = .8*new_pr + .2*old_pr
                node_info["PR"] = new_pr
                yield (key, node_info)
            elif key == "****Total PR":
                self.total_pr = total
            elif key == "***n_nodes":
                self.n_nodes = total
```

```
            elif key == "**Distribute":
                extra_mass = total
                # Because the node_count and
                # the mass distribution are
                # eventually consistent, a
                # simple correction for any early
                # discrepancies is a good fix
                excess_pr = self.total_pr - 1
                weight = extra_mass - excess_pr
                self.to_distribute = weight/self.n_nodes
            else:
                # The only time this should run
                # is when dangling nodes are
                # discovered during the first
                # iteration. By making them
                # explicitly tracked, the mapper
                # can handle them from now on.
                yield ("**Distribute", total)
                yield ("***n_nodes", 1.0)
                yield (key, {"PR": total,
                             "links": []})

    def reducer_final(self):
        print_info = False
        if print_info:
            print("Total PageRank", self.total_pr)

    def steps(self):
        iterations = self.options.iterations
        mr_steps = [MRStep(mapper_init=self.mapper_init,
                           mapper=self.mapper,
                           mapper_final=self.mapper_final,
                           reducer_init=self.reducer_init,
                           reducer=self.reducer,
                           reducer_final=self.reducer_final)]
        return mr_steps*iterations


if __name__ == "__main__":
    PageRank.run()
```

```
Overwriting PageRank.py
```

In [87]:

```
%%writefile PageRank.py
from __future__ import print_function, division
import itertools
from mrjob.job import MRJob
from mrjob.job import MRStep
from mrjob.protocol import JSONProtocol
from sys import stderr
from random import random

class PageRank(MRJob):
    INPUT_PROTOCOL = JSONProtocol

    def configure_options(self):
        super(PageRank,
              self).configure_options()

        self.add_passthrough_option(
            '--n_nodes',
            dest='n_nodes',
            type='float',
            help="""number of nodes
            that have outlinks. You can
            guess at this because the
            exact number will be
            updated after the first
            iteration.""")

        self.add_passthrough_option(
            '--reduce.tasks',
            dest='reducers',
            type='int',
            help="""number of reducers
            to use. Controls the hash
            space of the custom
            partitioner""")

        self.add_passthrough_option(
            '--iterations',
            dest='iterations',
            type='int',
            help="""number of iterations
            to perform.""")

        self.add_passthrough_option(
            '--damping_factor',
            dest='d',
            default=.85,
            type='float',
            help="""Is the damping
            factor. Must be between
            0 and 1.""")

        self.add_passthrough_option(
            '--smart_updating',
            dest='smart_updating',
            type='str',
            default="False",
```

```
            help="""Can be True or
            False. If True, all updates
            to the new PR will take into
            account the value of the old
            PR.""")

    def mapper_init(self):
        self.values = {"****Total PR": 0.0,
                       "***n_nodes": 0.0,
                       "**Distribute": 0.0}
        self.n_reducers = self.options.reducers

    def mapper(self, key, lines):
        n_reducers = self.n_reducers
        key_hash = hash(key)%n_reducers
        # Handles special keys
        # Calculate new Total PR
        # each iteration
        if key in ["****Total PR"]:
            raise StopIteration
        if key in ["**Distribute"]:
            self.values[key] += lines
            raise StopIteration
        if key in ["***n_nodes"]:
            self.values[key] += lines
            raise StopIteration
        # Handles the first time the
        # mapper is called. The lists
        # are converted to dictionaries
        # with default PR values.
        if isinstance(lines, list):
            n_nodes = self.options.n_nodes
            default_PR = 1/n_nodes
            lines = {"links":lines,
                     "PR": default_PR}
        # Perform a node count each time
        self.values["***n_nodes"] += 1.0
        PR = lines["PR"]
        links = lines["links"]
        n_links = len(links)
        # Pass node onward
        yield (key_hash, (key, lines))
        # Track total PR in system
        self.values["****Total PR"] += PR
        # If it is not a dangling node
        # distribute its PR to the
        # other links.
        if n_links:
            PR_to_send = PR/n_links
            for link in links:
                link_hash = hash(link)%n_reducers
                yield (link_hash, (link, PR_to_send))
        else:
            self.values["**Distribute"] = PR

    def mapper_final(self):
        for key, value in self.values.items():
```

```
            for k in range(self.n_reducers):
                yield (k, (key, value))

    def reducer_init(self):
        self.d = self.options.d
        smart = self.options.smart_updating
        if smart == "True":
            self.smart = True
        elif smart == "False":
            self.smart = False
        else:
            msg = """--smart_updating should
                       be True or False"""
            raise Exception(msg)
        self.to_distribute = None
        self.n_nodes = None
        self.total_pr = None

    def reducer(self, hash_key, combo_values):
        gen_values = itertools.groupby(combo_values,
                                       key=lambda x:x[0])
        for key, values in gen_values:
            total = 0
            node_info = None

            for key, val in values:
                if isinstance(val, float):
                    total += val
                else:
                    node_info = val

            if node_info:
                old_pr = node_info["PR"]
                distribute = self.to_distribute or 0
                pr = total + distribute
                decayed_pr = self.d * pr
                teleport_pr = (1-self.d)/self.n_nodes
                new_pr = decayed_pr + teleport_pr
                if self.smart:
                    # If the new value is less than
                    # 30% different than the old
                    # value, set the new PR to be
                    # 80% of the new value and 20%
                    # of the old value.
                    diff = abs(new_pr - old_pr)
                    percent_diff = diff/old_pr
                    if percent_diff < .3:
                        new_pr = .8*new_pr + .2*old_pr
                node_info["PR"] = new_pr
                yield (key, node_info)
            elif key == "****Total PR":
                self.total_pr = total
            elif key == "***n_nodes":
                self.n_nodes = total
            elif key == "**Distribute":
                extra_mass = total
                # Because the node_count and
```

```
                # the mass distribution are
                # eventually consistent, a
                # simple correction for any early
                # discrepancies is a good fix
                excess_pr = self.total_pr - 1
                weight = extra_mass - excess_pr
                self.to_distribute = weight/self.n_nodes
            else:
                # The only time this should run
                # is when dangling nodes are
                # discovered during the first
                # iteration. By making them
                # explicitly tracked, the mapper
                # can handle them from now on.
                yield ("**Distribute", total)
                yield ("***n_nodes", 1.0)
                yield (key, {"PR": total,
                             "links": []})

    def reducer_final(self):
        print_info = False
        if print_info:
            print("Total PageRank", self.total_pr)

    def steps(self):
        iterations = self.options.iterations
        mr_steps = [MRStep(mapper_init=self.mapper_init,
                           mapper=self.mapper,
                           mapper_final=self.mapper_final,
                           reducer_init=self.reducer_init,
                           reducer=self.reducer,
                           reducer_final=self.reducer_final)]
        return mr_steps*iterations


if __name__ == "__main__":
    PageRank.run()
```

```
Overwriting PageRank.py
```

In [ ]:

```
## Drivers & Runners
```

In [ ]:

```
## Run Scripts, S3 Sync
```

HW 9.4 Analysis

--------- OPTIONAL QUESTIONS SECTION --------

3. HW9.5: (OPTIONAL) Applying topic-specific PageRank to Wikipedia

Back to Table of Contents

Here you will apply your topic-specific PageRank implementation to Wikipedia, defining topics (very arbitrarily) for each page by the length (number of characters) of the name of the article mod 10, so that there are 10 topics.

- Once again, 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. Run for 10 iterations.
- Plot the pagerank values for the top 100 pages resulting from the 5 iterations run in HW 9.3.
- Then plot the pagerank values for the same 100 pages that result from the topic specific pagerank after 10 iterations run.
- Comment on your findings.

HW 9.5 Implementation

In [ ]:
```
## Code goes here
```

In [ ]:
```
## Drivers & Runners
```

In [ ]:
```
## Run Scripts, S3 Sync
```

HW 9.5 Analysis

3. HW9.6: (OPTIONAL) TextRank

Back to Table of Contents

- What is TextRank? Describe the main steps in the algorithm. Why does TextRank work?
- Implement TextRank in MrJob for keyword phrases (not just unigrams) extraction using co-occurrence based similarity measure with with sizes of N = 2 and 3. And evaluate your code using the following example using precision, recall, and FBeta (Beta=1):

```
"Compatibility of systems of linear constraints over the set of natura
l numbers
Criteria of compatibility of a system of linear Diophantine equations,
 strict
inequations, and nonstrict inequations are considered. Upper bounds fo
r
components of a minimal set of solutions and algorithms of constructio
n of
minimal generating sets of solutions for all types of systems are give
n.
These criteria and the corresponding algorithms for constructing a min
imal
supporting set of solutions can be used in solving all the considered
 types of
systems and systems of mixed types."
```

- The extracted keywords should in the following set:

```
linear constraints, linear diophantine equations, natural numbers, non
-strict inequations, strict inequations, upper bounds
```

HW 9.6 Implementation

In [ ]:
```
## Code goes here
```

In [ ]:
```
## Drivers & Runners
```

In [ ]:
```
## Run Scripts, S3 Sync
```

HW 9.6 Analysis