MIDS W261 HW7

Undirected toy network dataset

In an undirected network all links are symmetric, i.e., for a pair of nodes 'A' and 'B,' both of the links:

```
A -> B and B -> A
```

will exist.

The toy data are available in a sparse (stripes) representation:

```
(node) \t (dictionary of links)
```

on AWS/Dropbox via the url:

s3://ucb-mids-mls-networks/undirected_toy.txt On under the Data Subfolder for HW7 on Dropbox with the same file name. The Data folder is in: https://db.tt/Kxu48mL1 (https://db.tt/Kxu48mL1))

In the dictionary, target nodes are keys, link weights are values (here, all weights are 1, i.e., the network is unweighted).

Directed toy network dataset

In a directed network all links are not necessarily symmetric, i.e., for a pair of nodes 'A' and 'B,' it is possible for only one of:

```
A -> B or B -> A
```

to exist.

These toy data are available in a sparse (stripes) representation:

```
(node) \t (dictionary of links)
```

on AWS/Dropbox via the url:

s3://ucb-mids-mls-networks/directed_toy.txt On under the Data Subfolder for HW7 on Dropbox with the same file name

In the dictionary, target nodes are keys, link weights are values (here, all weights are 1, i.e., the network is unweighted).

HW 7.0: Shortest path graph distances (toy networks)

In this part of your assignment you will develop the base of your code for the week.

Write MRJob classes to find shortest path graph distances, as described in the lectures. In addition to finding the distances, your code should also output a distance-minimizing path between the source and target. Work locally for this part of the assignment, and use both of the undirected and directed toy networks.

To proof you code's function, run the following jobs

- shortest path in the undirected network from node 1 to node 4 Solution: 1,5,4. NOTE: There is another shortest path also (HINT: 1->5->4)! Either will suffice (you will find this also in the remaining problems. E.g., 7.2 and 7.4.
- shortest path in the directed network from node 1 to node 5 Solution: 1,2,4,5

and report your output---make sure it is correct!

Undirected Graph Data

```
1    {'2': 1,'5': 1}
2    {'1': 1,'3': 1,'4': 1,'5': 1}
3    {'2': 1, '4': 1}
4    {'2': 1,'3': 1,'5': 1}
5    {'1': 1, '2': 1, '4': 1}
```

In [1]:
```
# Jupyter requires this for MRJob to reload classes properly
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

In [ ]:
```
%%writefile init_data.py

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

class initGraphJob(MRJob):

    def configure_options(self): #configure start options
        super(initGraphJob, self).configure_options()
        self.add_passthrough_option('--startNode', default = '1') #
provide start node as argument

    def mapper(self, _, node):
        nodeID, links = node.split('\t') #split on input tab
        links = eval(links) #make a dictionary

        if nodeID == self.options.startNode:
            yield nodeID, (links.keys(), 0, 'Q', [nodeID]) #sets up
start node
        else:
            yield nodeID, (links.keys(), maxint, 'U', []) #otherwis
e set all nodes to unvisited

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

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

In [ ]:
```
#Runs job and outputs results to newfile, requires changing paramet
ers and probably want to change
from init_data import initGraphJob

mr_job = initGraphJob(args = ['directed_toy.txt'])

with open('newgraph.txt', 'w+') as myfile:
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            myfile.write(line)
```

In [ ]:

```
%%writefile shortestPathJob.py

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

class ShortestPathJob(MRJob):

    def mapper(self, _, line):
        newline = line.strip().split('\t') #split input

        node = eval(newline[0]) #get node

        data = eval(newline[1]) #unpack parameters from input
        neighbors = (data[0])
        distance = int(data[1])
        label = data[2]
        path = data[3]

        if label == 'Q': #if the label is in the queue, move all it
s neighbors into the queue
            for neighbor in neighbors:
                newPath = list(path)
                newPath.append(neighbor)
                yield neighbor, [None, distance + 1, 'Q', newPath]
            yield node, [neighbors, distance, 'V', path] #mark the
node as visited
        else:
            yield node, [neighbors, distance, label, path] #otherwi
se emit the node



    def reducer(self, key, values):
        #By default assume a node is unvisited with an empty list o
f neighbors, makes updating below easier
        neighbors = [] #these are global options
        distance = sys.maxint
        label = 'U'
        path = []

        for value in values: #iterate through list of values

            temp_neighbors = value[0]
            temp_distance = value[1]
            temp_label = value[2]
            temp_path = value[3]

            if temp_label == 'V': #if we're at a visited node set t
he parameters and break out of the loop
                neighbors = temp_neighbors
                distance = temp_distance
                label = temp_label
                path = temp_path
```

```
                break

            elif temp_label == 'Q': #if a node is in the queue chan
ge the label and update the distance and path
                label = temp_label
                distance = temp_distance
                path = temp_path

            elif temp_label == 'U': #update neighbrs based on unvis
ited copies
                neighbors = temp_neighbors

        yield key, [neighbors, distance, label, path]



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

Second way we did this

In [ ]:

```
%%writefile MrJobGraph70.py
from mrjob.job import MRJob
from mrjob.step import MRStep
from sys import maxint
import re
#
# Visit all nodes of the graph to calculate the minimum distance of
the graph
#
class MrJobGraph70(MRJob):

    def configure_options(self):
        super(MrJobGraph70, self).configure_options()

    # key = node, value = {},status
    # {} = dictionary of node:distances
    # status = one of Q,V,U
    def mapper(self, _, line):
        q = re.split('\t',line.strip())
        node = q[0].strip('"')
        value = eval(q[1].strip())
        neighbors = value[0]
        weight = int(value[1])
        status = value[2].strip('"')
        path = value[3]
        if status == 'Q':
            for neighbor in neighbors:
                path_to_node = list(path)
                path_to_node.append(neighbor)
                yield neighbor.strip('"'), (None, weight+1, 'Q', pa
th_to_node)
            # don't forget to yield the updated original node recor
d to V
            yield node, (neighbors, weight, 'V', path)
        else:
            yield node, (neighbors, weight, status, path)


    # key = node, value=[[neighbors], weight, node status]
    # the reducer needs to merge the key, value records
    def reducer(self, key, values):
        weight = maxint
        adj_list = []
        path = []
        node_status = 'U'
        for q in values:
            status = q[2].strip()
            if status == 'Q':
                weight = min(q[1], weight)
                node_status = 'Q'
                path = q[3]
            if status == 'U':
                if q[0]:
                    for adj_node in q[0]:
```

```
                        if adj_node not in adj_list:
                            adj_list.append(adj_node)
            if status == 'V':
                node_status = 'V'
                yield key, q
                break  # do not process more
        if node_status != 'V':
            # we've processed the list of values for the key and we
didn't
            # encounter a visited node so emit the combined record
            yield key,(adj_list, weight, node_status, path)


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

if __name__ == '__main__':
    MrJobGraph70.run()
```

In [ ]:
```
%%writefile MrJobTransform.py
from mrjob.job import MRJob
from mrjob.step import MRStep
from sys import maxint
import re
#
# Take an adjancy matrix for a graph and transform it into priority
queue format
#
class MrJobTransform(MRJob):

    def configure_options(self):
        super(MrJobTransform, self).configure_options()
        # we need to know what the start node is so we can mark tha
t node with status Q
        self.add_passthrough_option(
            '--startNode', dest='start_node', default=1, type='st
r',
            help='startNode: label of graph node on which to open t
he frontier')


    # each line is node \t {neighbor: weight}
    def mapper(self, _, line):
        adj_line = re.split('\t',line.strip())
        node = adj_line[0].strip('"')
        neighbors = eval(adj_line[1])
        if node == self.options.start_node:
            path_to_node = [node]
            yield node, (neighbors.keys(), 1, 'Q', path_to_node)
        else:
            yield node, (neighbors.keys(), maxint, 'U', [])


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

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

In [ ]:
```
from shortestPathJob import ShortestPathJob
mr_job = ShortestPathJob(args = ['newgraph.txt'])

with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        print mr_job.parse_output_line(line)
```

('1', [['2', '6'], 0, 'V', [1]]) ('2', [['1', '3', '4'], 1, 'V', [1, '2']]) ('3', [['2', '4'], 2, 'Q', [1, '2', '3']]) ('4', [['2', '5'], 2, 'Q', [1, '2', '4']]) ('5', [['1', '2', '4'], 9223372036854775807, 'U', []]) ('6', [[], 1, 'V', [1, '6']])

Exactly what we would expect to see

In [ ]:
```
#Driver for iterations
import os
from init_data import initGraphJob
from shortestPathJob import ShortestPathJob

def findShortestPath(filename, startNode, endNode):

    mr_job_init = initGraphJob(args = [filename, '--startNode', sta
rtNode]) #initialize a file and startnode

    with open('working-graph.txt', 'w+') as myfile: #creates a loca
l version called working graph
        with mr_job_init.make_runner() as runner:
            runner.run()
            for line in runner.stream_output():
                myfile.write(line) #write results


    while True:
        with open('newFile.txt', 'w+') as myfile: #now create a new
File to store our iteration output
            mr_job = ShortestPathJob(args = ['working-graph.txt'])
#run shortest path and output results
            with mr_job.make_runner() as runner:
                runner.run()
                for line in runner.stream_output():
                    output = mr_job.parse_output_line(line) #get ou
tput line
                    myfile.write(line)
                    if output[0] == endNode and output[1][2] ==
"V": #our stop condition, note this is inefficient
                        return (output[1][3], output[1][1]) #path
                        break

        os.rename('newFile.txt', 'working-graph.txt')

print 'Shortest path in undirected Graph:'
results = findShortestPath('undirected_toy.txt', '1', '4')
print 'Path: ' + str(results[0]) + ', with distance ' + str(results
[1])

print 'Shortest path in directed Graph:'
results = findShortestPath('directed_toy.txt', '1', '5')
print 'Path: ' + str(results[0]) + ', with distance ' + str(results
[1])
```

Shortest path in undirected Graph: Path: ['1', '5', '4'], with distance 2

Shortest path in directed Graph: Path: ['1', '2', '4', '5'], with distance 3

Main dataset 1: NLTK synonyms

In the next part of this assignment you will explore a network derived from the NLTK synonym database used for evaluation in HW 5. At a high level, this network is undirected, defined so that there exists link between two nodes/words if the pair or words are a synonym. These data may be found at the location:

```
s3://ucb-mids-mls-networks/synNet/synNet.txt
s3://ucb-mids-mls-networks/synNet/indices.txt
On under the Data Subfolder for HW7 on Dropbox with the same file names
```

where synNet.txt contains a sparse representation of the network:

```
(index) \t (dictionary of links)
```

in indexed form, and indices.txt contains a lookup list

```
(word) \t (index)
```

of indices and words. This network is small enough for you to explore and run scripts locally, but will also be good for a systems test (for later) on AWS.

In the dictionary, target nodes are keys, link weights are values (here, all weights are 1, i.e., the network is unweighted).

HW 7.1: Exploratory data analysis (NLTK synonyms)

Using MRJob, explore the synonyms network data. Consider plotting the degree distribution (does it follow a power law?), and determine some of the key features, like:

- number of nodes,
- number links,
- or the average degree (i.e., the average number of links per node),
- etc...

As you develop your code, please be sure to run it locally first (though on the whole dataset). Once you have gotten your code to run locally, deploy it on AWS as a systems test in preparation for our next dataset (which will require AWS).

In [21]:

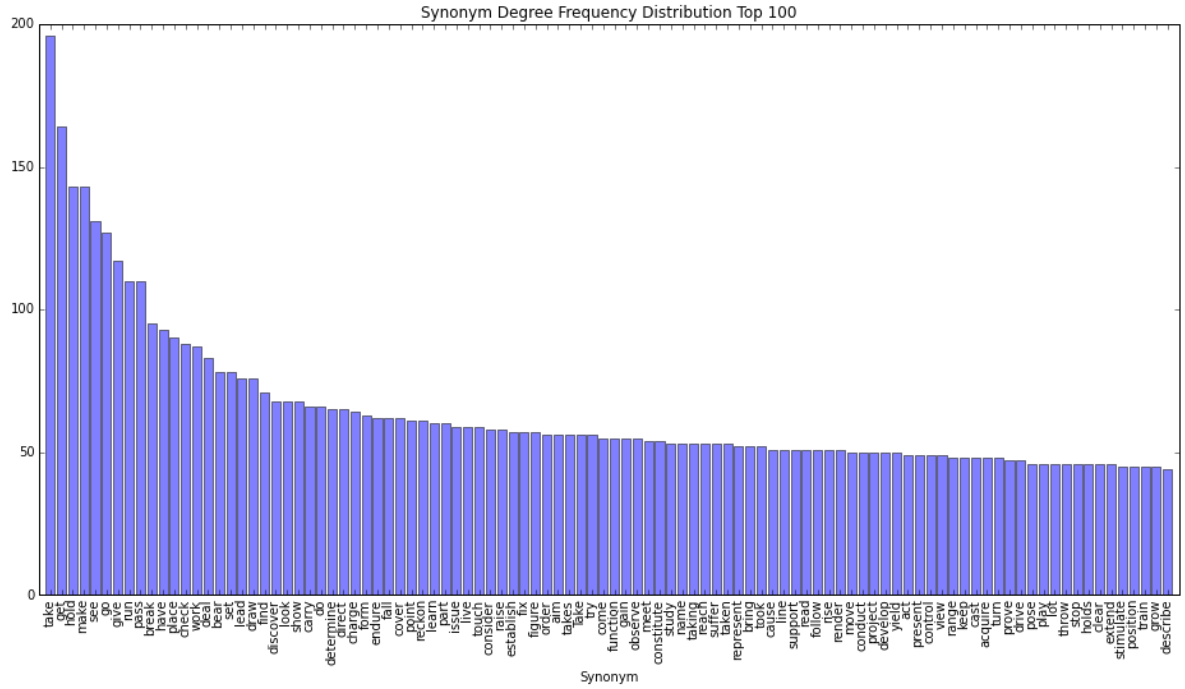
```
%%writefile MrJobSynFreq.py

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

class MrJobSynFreq(MRJob):
    SORT_VALUES = True

    def configure_options(self):
        super(MrJobSynFreq, self).configure_options()
        self.add_passthrough_option(
            '--idxFile', type='string', default=None)

    # line is  node \t dictionary where dictionary entries are {nei
ghbor : weight}
    def mapper(self, _, line):
        node_entry = re.split('\t',line.strip())
        node = node_entry[0]
        neighbors = eval(node_entry[1])
        for neighbor in neighbors:
            yield neighbor, neighbors[neighbor]

    def reducer_init(self):
        self.syn_idx = {}
        with open(self.options.idxFile, 'rU') as idxFile:
            for line in idxFile.readlines():
                idx = re.split('\t',line.strip())
                self.syn_idx[idx[1]]=idx[0]


    # key=node, values=weights
    def reducer(self, key, values):
        yield self.syn_idx[key], sum(values)

    def identity_mapper(self, key, value):
        yield value, key

    def sorting_reducer(self, _, values):
        for value in values:
            yield _, value

    def steps(self):
        return [MRStep(mapper = self.mapper,
                       reducer_init = self.reducer_init,
                       reducer = self.reducer),
                MRStep(mapper = self.identity_mapper,
                       reducer = self.sorting_reducer,
                       jobconf={'mapred.output.key.comparator.clas
s':
                                'org.apache.hadoop.mapred.lib.KeyFi
eldBasedComparator',
                                'mapred.text.key.comparator.option
s': '-k1nr -k2',
```

```
                                    })]

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

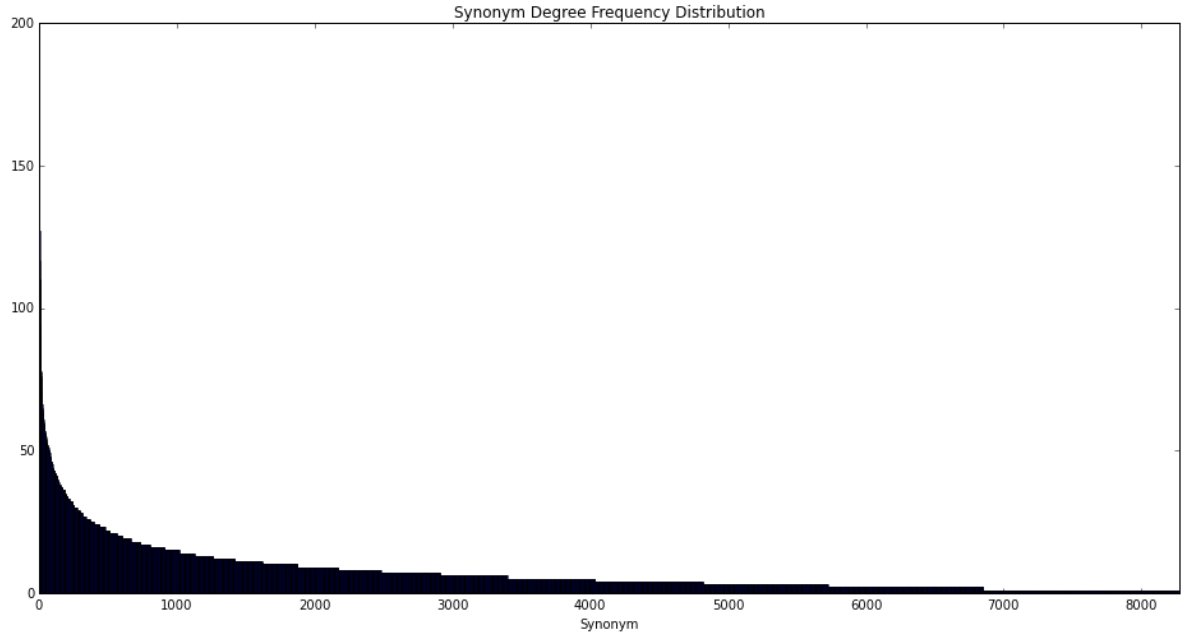
```
Writing MrJobSynFreq.py
```

In [22]:
```
from MrJobSynFreq import MrJobSynFreq

infile = '/Users/rcordell/Documents/MIDS/W261/week07/HW7/Data/synNe
t/synNet.txt'
outfile = '/Users/rcordell/Documents/MIDS/W261/week07/HW7/node_coun
ts.txt'
idxFile = '/Users/rcordell/Documents/MIDS/W261/week07/HW7/Data/synN
et/indices.txt'

mr_job = MrJobSynFreq(args = [infile,
                              '-r', 'hadoop',
                              '--idxFile='+idxFile])

with open(outfile, 'w') as countFile:
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            mr_job.parse_output_line(line)
            countFile.write(line)
```

In [19]:
```
import matplotlib.pyplot as plt
import re
import numpy as np

words = []
counts = []
with open('node_counts.txt','r') as cntFile:
    for line in cntFile.readlines():
        count, word = re.split('\t',line.strip())
        counts.append(int(count))
        words.append(word.strip('"'))

y_pos = np.arange(100)
fig = plt.figure(figsize=(16,8))
plt.bar(y_pos, counts[:100], align='center', alpha=0.5)
plt.xlim(-1,100)
plt.xticks(y_pos, words[:100], rotation=90)
plt.xlabel('Synonym')
plt.title('Synonym Degree Frequency Distribution Top 100')
plt.show()
```



In [20]:
```
y_pos = np.arange(len(words))
fig = plt.figure(figsize=(16,8))
plt.bar(y_pos, counts, align='center', alpha=0.5)
plt.xlim(-1,len(words))
#plt.xticks(y_pos, words[:100], rotation=90)
plt.xlabel('Synonym')
plt.title('Synonym Degree Frequency Distribution')
plt.show()
```



In [25]:
```
print "Number of Nodes: {0}".format(len(words))
print "Number of Links: {0}".format(sum(counts))
print "Average Degree:  {0}".format(float(sum(counts))/len(words))
```

```
Number of Nodes: 8271
Number of Links: 61134
Average Degree:  7.39136742836
```

HW 7.2: Shortest path graph distances (NLTK synonyms)

Write (reuse your code from 7.0) an MRJob class to find shortest path graph distances, and apply it to the NLTK synonyms network dataset.

Proof your code's function by running the job:

- shortest path starting at "walk" (index=7827) and ending at "make" (index=536),

and showing you code's output. Once again, your output should include the path and the distance.

As you develop your code, please be sure to run it locally first (though on the whole dataset). Once you have gotten your code to run locally, deploy it on AWS as a systems test in preparation for our next dataset (which will require AWS).

Main dataset 2: English Wikipedia

For the remainder of this assignment you will explore the English Wikipedia hyperlink network. The dataset is built from the Sept. 2015 XML snapshot of English Wikipedia. For this directed network, a link between articles:

```
A -> B
```

is defined by the existence of a hyperlink in A pointing to B. This network also exists in the indexed format:

```
Data: s3://ucb-mids-mls-networks/wikipedia/all-pages-indexed-out.txt
Data: s3://ucb-mids-mls-networks/wikipedia/all-pages-indexed-in.txt
Data: s3://ucb-mids-mls-networks/wikipedia/indices.txt
On under the Data Subfolder for HW7 on Dropbox with the same file names
```

but has an index with more detailed data:

```
(article name) \t (index) \t (in degree) \t (out degree)
```

In the dictionary, target nodes are keys, link weights are values. Here, a weight indicates the number of time a page links to another. However, for the sake of this assignment, treat this an unweighted network, and set all weights to 1 upon data input.

In [ ]:
```
#Test run locally
results = findShortestPath('synNet/synNet.txt', '7827', '536')
print 'Path: ' + str(results[0]) + ', with distance ' + str(results
[1])
```

Path: ['7827', '4655', '631', '536'], with distance 3

In [ ]:
```
from init_data import initGraphJob
from shortestPathJob import ShortestPathJob

def findShortestPath2(filename, startNode, endNode, clusterID):

    counter = 0 #keeps track and used to store output

    #provide initial configuration
    mr_job_init = initGraphJob(args = [filename, '--startNode', sta
rtNode,
                                       '--no-strict-protocols', '-
r', 'emr',
                                       '--emr-job-flow-id', clusterI
D,
                                       '--output-dir', 's3://dunmire
g/HW7/output' + str(counter)])


    with mr_job_init.make_runner() as runner: #run the init job to
make the adjacency list
        runner.run()


    iterate = True #continue looping until this is set to False
    while iterate:
        counter += 1 #run new job, note output is stored
        mr_job = ShortestPathJob(args = ['s3://dunmireg/HW7/output'
+ str(counter - 1) + '/',
                                         '--no-strict-protocols', '-
r', 'emr',
                                         '--emr-job-flow-id', cluste
rID,
                                         '--output-dir', 's3://dunmi
reg/HW7/output' + str(counter)])

        with mr_job.make_runner() as runner:
            runner.run()
            for line in runner.stream_output(): #this streams the r
esults, probably want to change that
                output = mr_job.parse_output_line(line)
                if output[0] == endNode and output[1][2] == 'V': #i
f we hit our endNode and V spit out results
                    print "The path is: " + str(output[1][3])
                    print "In a distance of: " + str(output[1][1])
#path
                    iterate = False
                if output[0] == endNode: #if at endNode regardless
of state, break out
                    break
```

In [ ]:
```
!python -m mrjob.tools.emr.create_job_flow '--conf-path' 'mrjob.con
f' #make persistent cluster
#Note using 4 xlarge nodes
```

In [ ]:
```
#infinite loop problem
findShortestPath2('s3://dunmireg/Input/directed_toy.txt', '1', '5',
'j-34BN2T9A3U67T')
```

The path is: ['1', '2', '4', '5'] In a distance of: 3

In [ ]:
```
findShortestPath2('s3://dunmireg/Input/undirected_toy.txt', '1',
'4', 'j-34BN2T9A3U67T')
```

The path is: ['1', '2', '4'] In a distance of: 2

In [ ]:
```
findShortestPath2('s3://dunmireg/Input/synNet.txt', '7827', '536',
'j-34BN2T9A3U67T')
```

The path is: ['7827', '1426', '3555', '536'] In a distance of: 3

NB: the resulting path is different than what we got running locally. We believe this is due to some of the configuration setup and how the file is split. This path however is verified to be one of several shortest paths in this data set

HW 7.3: Exploratory data analysis (Wikipedia)

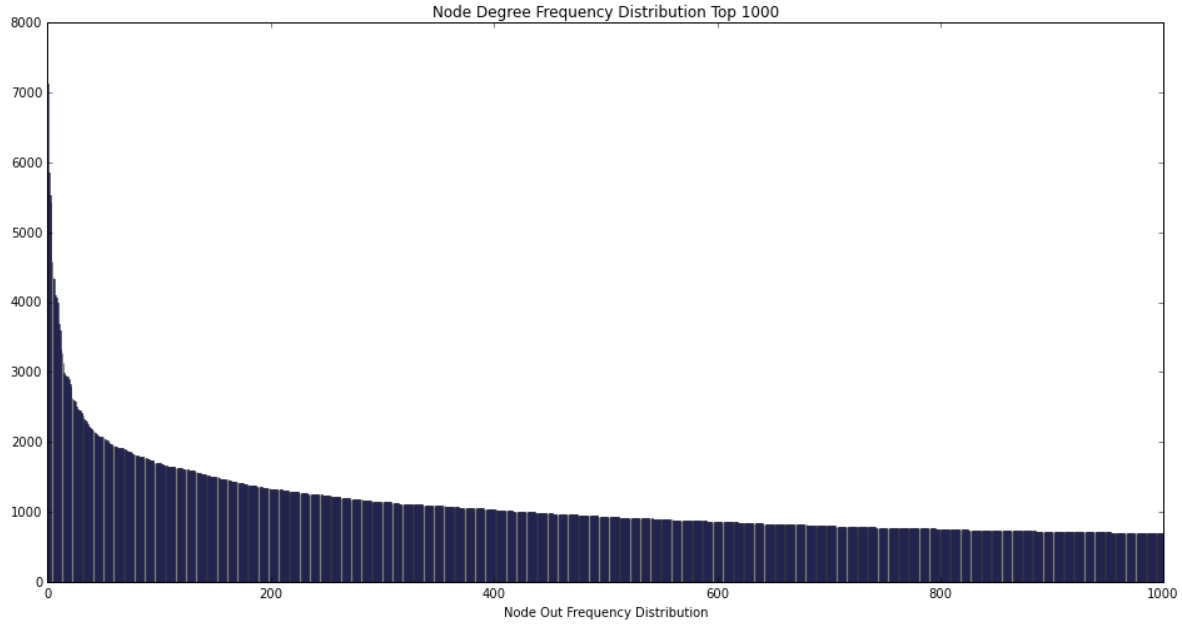
Using MRJob, explore the Wikipedia network data on the AWS cloud. Reuse your code from HW 7.1---does is scale well? Be cautioned that Wikipedia is a directed network, where links are not symmetric. So, even though a node may be linked to, it will not appear as a primary record itself if it has no out-links. This means that you may have to ADJUST your code (depending on its design). To be sure of your code's functionality in this context, run a systems test on the directed_toy.txt network.

In [115]:
```
%%writefile MrJobWikiLinks.py

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

class MrJobWikiLinks(MRJob):
    SORT_VALUES = True

    def configure_options(self):
        super(MrJobWikiLinks, self).configure_options()
        self.add_passthrough_option(
            '--idxFile', type='string', default=None)

    # line is  node \t dictionary where dictionary entries are {nei
ghbor : weight}
    def mapper(self, _, line):
        node_entry = re.split('\t',line.strip())
        node = node_entry[0]
        neighbors = eval(node_entry[1])
        for neighbor in neighbors:
            yield node, 1

    def combiner(self, node, counts):
        yield node, sum(counts)

    # key=node, values=count
    def reducer(self, node, counts):
        yield node, sum(counts)

    def identity_mapper(self, key, value):
        yield value, (value, key)

    def sorting_reducer(self, _, values):
        for value in values:
            yield value

    def steps(self):
        return [MRStep(mapper = self.mapper,
                       combiner = self.combiner,
                       reducer = self.reducer),
                MRStep(mapper=self.identity_mapper,
                       reducer=self.sorting_reducer,
                       jobconf={'mapred.output.key.comparator.clas
s':
                                'org.apache.hadoop.mapred.lib.KeyFi
eldBasedComparator',
                                'mapred.text.key.comparator.option
s': '-k1nr -k2nr',
                                })]

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

```
Overwriting MrJobWikiLinks.py
```

In [119]:
```
from MrJobWikiLinks import MrJobWikiLinks

infile = 's3://ucb-mids-mls-networks/wikipedia/all-pages-indexed-out.txt'
outfile = '/Users/rcordell/Documents/MIDS/W261/week07/HW7/wiki_out_counts.txt'
#idxFile = '/Users/rcordell/Documents/MIDS/W261/week07/HW7/Data/synNet/indices.txt'

mr_job = MrJobWikiLinks(args = [infile,
                                '-r', 'emr',
                                '--conf-path','mrjob.conf',
                                '--pool-emr-job-flows'
                               ])

with open(outfile, 'w') as countFile:
    with mr_job.make_runner() as runner:
        runner.run()
        for line in runner.stream_output():
            mr_job.parse_output_line(line)
            countFile.write(line)
```

In [3]:
```
import matplotlib.pyplot as plt
import re
import numpy as np

nodes = []
counts = []
with open('wiki_out_counts.txt','r') as cntFile:
    for line in cntFile.readlines():
        count, node = re.split('\t',line.strip())
        counts.append(int(count))
        nodes.append(node.strip('"'))

y_pos = np.arange(1000)
fig = plt.figure(figsize=(16,8))
plt.bar(y_pos, counts[:1000], align='center', alpha=0.5)
plt.xlim(-1,1000)
#plt.xticks(y_pos, words[:100], rotation=90)
plt.xlabel('Node Out Frequency Distribution')
plt.title('Node Degree Frequency Distribution Top 1000')
plt.show()
```



In [4]:
```
print "Number of Nodes: {0}".format(len(nodes))
print "Number of Links: {0}".format(sum(counts))
print "Average Degree:  {0}".format(float(sum(counts))/len(nodes))
```

```
Number of Nodes: 5781290
Number of Links: 142114057
Average Degree:  24.5817208616
```

HW 7.4: Shortest path graph distances (Wikipedia)

Using MRJob, find shortest path graph distances in the Wikipedia network on the AWS cloud. Reuse your code from 7.2, but once again be warned of Wikipedia being a directed network. To be sure of your code's functionality in this context, run a systems test on the directed_toy.txt network.

When running your code on the Wikipedia network, proof its function by running the job:

- shortest path from "Ireland" (index=6176135) to "University of California, Berkeley" (index=13466359),

and show your code's output. Show the shortest path in terms of just page IDS but also in terms of the name of page (show of your MapReduce join skills!!)

Once your code is running, find some other shortest paths and report your results.

NB this is a different setup than the job above. However the outputs are the same. We solved the problems separately and wanted to demonstrate several different methods and drivers

In [16]:
```
from MrJobTransform import MrJobTransform
from sys import maxint
import re

# transform the adjacency list into a priority queue list
def runTransformJob(ifname,ofname, startNode): #include files and s
tartNode
    mr_job = MrJobTransform(args=[ifname,
                                  '-r', 'emr',
                                  '--output-dir', ofname,
                                  '--no-output',
                                  '--conf-path','mrjob.conf',
                                  '--pool-emr-job-flows',
                                  '--startNode', startNode])

    with mr_job.make_runner() as runner:
        runner.run()

if __name__ == '__main__':
    runTransformJob('s3://ucb-mids-mls-networks/wikipedia/all-pages
-indexed-out.txt',
                    's3://w261-rlc-hw7/out/queue', '6176135')
```

In [18]:
```
from MrJobTransform import MrJobTransform
from MrJobGraph70 import MrJobGraph70
from sys import maxint
from shutil import copy
import re

def runSSSPJob(qfile, start_node, end_node): #set up job
    mr_job = MrJobGraph70(args=[qfile,
                                '-r', 'emr',
                                '--conf-path','mrjob.conf',
                                '--emr-job-flow-id', 'W261-HW7-Clus
ter'
                               ])

    iterations = 0
    more_nodes_to_visit = True #logical flag for breakout
    while(more_nodes_to_visit):
        iterations += 1
        print 'Iteration: {0}'.format(iterations)
        with open('_queue.txt','w+') as qFile:
            with mr_job.make_runner() as runner:
                runner.CLEANUP_CHOICES='JOB'
                runner.run()
                more_nodes_to_visit = False
                output_line = None
                for line in runner.stream_output():
                    q = mr_job.parse_output_line(line)
                    qFile.write(line)
                    # peek at the q entry status - if one is not vi
sited, keep going
                    status = q[1][2]
                    if status != 'V':
                        more_nodes_to_visit = True
                    else:
                        # if it is Visited, check to see if this is
our end node
                        if q[0].strip('"') == str(end_node):
                            output_line = 'Path from node {0} to no
de {1} is: {2}' \
                                .format(start_node, end_node, q[1]
[3])
                if output_line:
                    more_nodes_to_visit = False
                    print output_line

        copy('_queue.txt',qfile)

if __name__ == '__main__':
    runSSSPJob('Data/wikipedia/s3/queue.txt', 6176135, 13466359)
```

```
Iteration: 1
Path from node 6176135 to node 13466359 is: ['6176135', '1160779
1', '13466359']
```

```
Iteration: 1

Path from node 6176135 to node 13466359 is: ['6176135', '11607791', '13
466359']
```

In [117]:
```
%%writefile mrjob.conf
include: /Users/rcordell/.mrjob.conf
runners:
    hadoop:
        hadoop_home: '/usr/local/Cellar/hadoop/2.7.2/libexec'

    emr:
        ssh_tunnel_to_job_tracker : true
        ec2_instance_type : m1.medium
        num_ec2_instances : 4
        enable_emr_debugging: true
        bootstrap:
        - sudo apt-get install -y python-pip || sudo yum install -y
python-pip
        - sudo pip install boto mrjob
```

```
Overwriting mrjob.conf
```

In [ ]:
```
#Using find shortest path function from above
#Ireland to Guinness (one of my favorite beers)
findShortestPath2('s3://ucb-mids-mls-networks/wikipedia/all-pages-i
ndexed-out.txt', '6176135', '5341467', 'j-3NFXT598M070G')
```

The path is: ['6176135', '5341467'] In a distance of: 1

In [ ]:
```
%%writefile path.txt
6176135,1
11607791,2
13466359,3
```

In [ ]:
```
%%writefile join.py

#Quick job based on Homework 5 to do a join with the results of a t
ext file, matching indices
from mrjob.job import MRJob

class JoinJob(MRJob):

    def mapper_init(self):
        self.webIDs = {} #web ID dictionary to store IDs in the pat
h
        with open('path.txt', 'r') as myfile: #provided file
            lines = myfile.readlines()
            for line in lines:
                line = line.split(',') #stored as comma separated
                self.webIDs[line[0]] = line[1].strip()

    def mapper(self, _, line):
        line = line.strip().split('\t') #split input
        if line[1] in self.webIDs.keys(): #check if found a word in
our dictionary
            yield self.webIDs[line[1]], line[0] #yield

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

In [ ]:
```
from join import JoinJob

mr_job = JoinJob(args = ['indices.txt', '--file', 'path.txt'])

#This runs locally, would probably want to extend and make run on A
WS

with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        print mr_job.parse_output_line(line)
```

('1', 'Ireland')

('2', 'Seamus Heaney')

('3', 'University of California, Berkeley')

HW 7.5: Conceptual exercise: Largest single-source network distances

Suppose you wanted to find the largest network distance from a single source, i.e., a node that is the furthest (but still reachable) from a single source.

How would you implement this task? How is this different from finding the shortest path graph distances?

Is this task more difficult to implement than the shortest path distance?

As you respond, please comment on program structure, runtimes, iterations, general system requirements, etc...

One algorithm to find the longest path in a directed acyclic graph is to perform a BFS from any node to find the farthest node, T. Then run a BFS from T to find the longest path and that will be the longest path.

There is no way to "short circuit" this algorithm like when you search for the shortest path - you must visit all the nodes in the graph twice to know that you have checked for the longest path. The run time is $O(2 * (|V| + |E|))$ in this case.

The implementation could use most of what we already have but we need to keep track of the longest paths at every iteration instead of the shortest and not stop until all nodes have been visited.

HW 7.6: Computational exercise: Largest single-source network distances (optional)

Using MRJob, write a code to find the largest graph distance and distance-maximizing nodes from a single-source. Test your code first on the toy networks and synonyms network to proof its function.

==========END HW 7==========

In [ ]: