MIDS-W261-2016-HWK-Week07-Lane

July 11, 2016

0.1 MIDS UC Berkeley, Machine Learning at Scale Assignment 7

W261-1 Summer 2016
Week 7: SSSP
Jackson Lane

General Description
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!

In [3]:
```
%%writefile MRJob7_0_findshortestpath.py

#This job takes in a graph file, optionally an index file, a start point or name, and an end po
#The job will then find the shortest path between the start and points using breadth first searc
# There are two general sections of this job
#The first is the pre-processing.  Here, the graph data is transformed in the following ways:
#   -Initialize fields for visit status, distance, and shortest path
#   -Join graph data with name from index file if present
#   -Put start point into queue

# In the second phase, the breadth first search algorithm is applied repeatedly until either it
# or it runs out of nodes to process.
# This phase will also terminate after 999 iterations.
# Each iteration of breadth first search will:
#   -Update the statuses of the nodes currently in queue and add their neighbors to the queue
#   -Update the shortest paths and distances of the nodes currently in queue
#   -Terminate if either the end node has been reached or if there are no nodes left to process

# The second phase passes back either the shortest path to the end node or an error message

from mrjob.job import MRJob
```

```
from mrjob.step import MRStep
import sys

class MRJob7_0_findshortestpath(MRJob):
    MRJob.SORT_VALUES = True

    def steps(self):
        #Run 999 iterations of BFS search
        step =  [MRStep(
                    mapper=self.mapper
                       , reducer=self.reducer
                ,jobconf =  {
            "mapred.map.tasks":4,
            "mapred.reduce.tasks":2
        }
                      )
               ]
        return [MRStep(mapper=self.mapper_preprocess,reducer=self.reducer_preprocess
            ,jobconf =  {
            "mapred.map.tasks":4,
            "mapred.reduce.tasks":2
        })] + step * 999

    def configure_options(self):
        super(MRJob7_0_findshortestpath, self).configure_options()
        self.add_passthrough_option('--startNode', default='1')
        self.add_passthrough_option('--endNode', default='1')

    def mapper_preprocess(self, _, line):
        fields = line.strip().split('\t')
        #Check if line comes from a graph data file
        # or an index file.  You usually can tell if its
        # a graph data file if the first field is a number.
        if fields[0].isdigit():
            id = fields[0]
            neighbors = eval(fields[1])
            # if it turns out that this was actually a line from an index file
            # and the node name was just all numeric, then discard
            if(str(type(neighbors)) == "<type 'dict'>"):
                yield id, neighbors
        else:
            # if line is from an index file, yield node id and name with order inversion
            id = fields[1]
            name = fields[0]
            yield id, {"*":name}

    # The reducer here will perform left on graph data with the names from the index file, if pr
    def reducer_preprocess(self,id,values):
        name = id
        id = int(id)
        for neighbors in values:
            if neighbors.get("*"):
                name= neighbors["*"]
            else:
```

```
            if name == self.options.startNode:
                yield id, [name,neighbors, 0, 'queue', []]
            else:
                yield id,[name, neighbors,sys.maxint, 'frontier', []]

    # This mapper takes in a node and computes the results of a BFS search iteration on that no
    # The mapper then yields the resulting nodes from that BFS search.
    def mapper(self, node, values):
        name,neighbors, distance, status,shortest_path = values
        node = str(node)
        neighbors = map(str,neighbors)
        # First part of termination condition.  Each mapper will yield a special key *Q with va
        # If the mapper processes any queued elements, it will emit *Q with a value of 1
        # In the reducer, if none of the mappers emitted *Q with a value of 1, then that means
        # the queue is empty and there is no path between the start and end nodes.
        yield "*Q",0
        self.increment_counter("7_0","NA",1)

        if status == 'queue':
            #Yield *Q,1 to indicate that the mapper has processed a node in queue and that ther
            # nodes left to process in the graph
            yield "*Q",1
            if(name == self.options.endNode):
                self.increment_counter("7_0",map(str,shortest_path),1)

            # Update status from "queue" to "visited"and yield
            yield node, [name,neighbors, distance, 'visited', shortest_path]
            if neighbors:
                #Yield each of node's neighbors.
                #Note that mapper does not have any information on the neighbors
                    # except for node id.  So the mapper just emits what it can,
                    # and it's up to the reducer to fill in the missing information
                for neighbor in neighbors:
                    temp_path = list(shortest_path)
                    temp_path.append(name)
                    yield neighbor, ['',None, distance + 1, 'queue', temp_path]
        else:
            #Even if a node has been visited before or is on the frontier, the mapper still yie
            # so that reducer can use it to fill in missing information if needed
            yield node, [name,neighbors, distance, status, shortest_path]


    #The reducer fills in the missing information for each of the nodes emitted by the mapper
    #The reducer also will terminate the MRJob if the endNode is found or
    # if the mapper did not process any nodes in status "queue"
    def reducer(self, key, values):
        if key == "*Q":
            queue_count = sum(values)
            if (queue_count == 0):
                raise Exception("No Path")
        else:
            name = ""
            neighbors = {}
            distance = sys.maxint
```

```
                    status = None
                    path = []
                    for value in values:
                        # If the reducer runs into a visited node, the reducer assumes all information i
                        # If the visited node is the endNode, the reducer terminates the job
                        if value[3] == 'visited':
                            [name,neighbors,distance,status,path] = value
                            if(name == self.options.endNode):
                                raise Exception("Success")
                            break

                        #Handle frontier nodes.
                        elif value[3] == "frontier":
                            name = value[0]
                            neighbors = value[1]
                            #If we previously processed a queued node, with same ID, then leave status a
                            #Otherwise, update status to frontier
                            if status != 'queue':
                                status = value[3]

                        # Handle queued nodes.  This type of node should always come with a frontier noc
                        # The frontier node updates the name and neighbors while the queued node updates
                        else:
                            status = value[3]
                            path = value[4]
                        distance = min(distance, value[2])

                    # After all values are processed, yield node
                    yield key, [name,neighbors, distance, status, path]

        if __name__ == '__main__':
            MRJob7_0_findshortestpath.run()
```

```
Overwriting MRJob7_0_findshortestpath.py
```

In [4]:
```
# Driver function to call the findshortestpath MRJob.
# It passes the graph data file and index file if present to the MRJob.
# StartNode, endNode, and runnerType are all parameterized, making this driving function backen
# Since the MRJob should terminate early, the driver function actually handles the results of t
# a try-catch statement.

%load_ext autoreload
%autoreload 2
from MRJob7_0_findshortestpath import MRJob7_0_findshortestpath
import os
def findShortestPath(filenames, startNode, endNode,runnerType):
    # Separate graph data file from index file (if present)
    filenames = filenames.split()
    fileData = filenames[0]
    if (len(filenames) > 1):
        fileIndex = filenames[1]
    else:
        fileIndex = ''
    mr_job = MRJob7_0_findshortestpath(args=[
        fileData,fileIndex, '-r', runnerType,
```

```
            '--startNode', startNode,
            '--endNode',endNode
        ])
        with mr_job.make_runner() as runner:
            try:
                runner.run()
            except:
                #The MRJob should terminate early when it finds the endNode or runs out
                # of nodes to process.  Since MRJob does not let you stream_output on
                # jobs that have terminated early, the MRJob uses counters to pass
                # the shortest path to the driver function.
                counters = sorted(runner.counters()[-1]["7_0"].keys())
                path =counters[-1].replace(";",",")
                if path == "NA":
                    raise Exception("No path")
                path = map(str,eval(path))
                # Append endNode to the path.  The MRJob path alone only shows nodes leading up to
                return path + [endNode]
        raise Exception("No Path after 999 iterations")
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Test against toy data

```
In [37]: %%writefile toy_indicies.txt
         A        1
         B        2
         D        3
         C        4
```

```
Overwriting toy_indicies.txt
```

```
In [38]: # Testing undirected toy with index file
         filename = 'Data/undirected_toy.txt toy_indicies.txt'
         print 'Shortest path in', filename
         # Because we are using an index file, we refer to nodes by name rather than ID
         path = findShortestPath(filename, 'A', 'D','local')
         print path
```

```
Shortest path in Data/undirected_toy.txt toy_indicies.txt
['A', 'B', 'D']
```

```
In [18]: # Testing directed toy without index file
         filename = 'Data/directed_toy.txt'
         print 'Shortest path in', filename
         path = findShortestPath(filename, '1', '5','local')
         print path
```

```
Shortest path in Data/undirected_toy.txt toy_indicies.txt
['A', 'B', 'D']
Shortest path in Data/directed_toy.txt
['1', '2', '4', '5']
```

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) (dictionary of links)

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

(word) (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)

In [223]: %%writefile MRJob7_1_findnodes.py

```
#MR Job code to find all the nodes in a graph.  It does this by emitting a key for each node i

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

class MRJob7_1_findnodes(MRJob):

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

    def combiner(self, node, values):
        # Even if there are duplicates, emit a value of 1.
        # We only want to know whether a node exists, not its degree
        yield str(node), 1

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

if __name__ == '__main__':
    MRJob7_1_findnodes.run()
```

```
Overwriting MRJob7_1_findnodes.py
```

In [4]: %%writefile MRJob7_1_findlinks.py

```
#MR Job code to find all the edges in a graph.  It does this by creating a frequency distributio

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

```
class MRJob7_1_findlinks(MRJob):

    def mapper(self, _, line):
        node, neighbors = line.strip().split('\t')
        neighbors = eval(neighbors)
        # compute the degree of the node and yield
        degree = len(neighbors)
        yield degree, 1

    def combiner(self, degree, counts):
        # Sum up counts
        yield degree, sum(counts)

    def reducer(self, degree, counts):
        yield degree, sum(counts)

if __name__ == '__main__':
    MRJob7_1_findlinks.run()
```

```
Overwriting MRJob7_1_findlinks.py
```

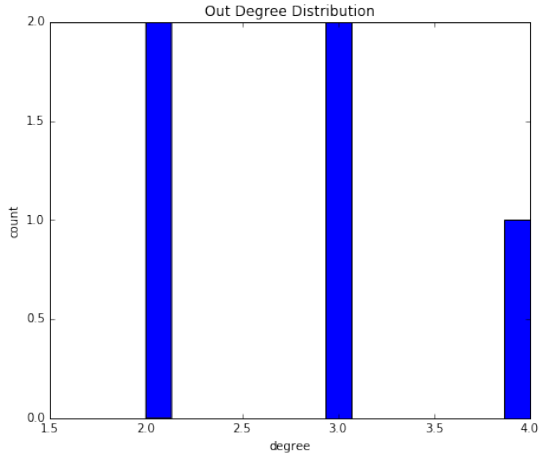
Driver class

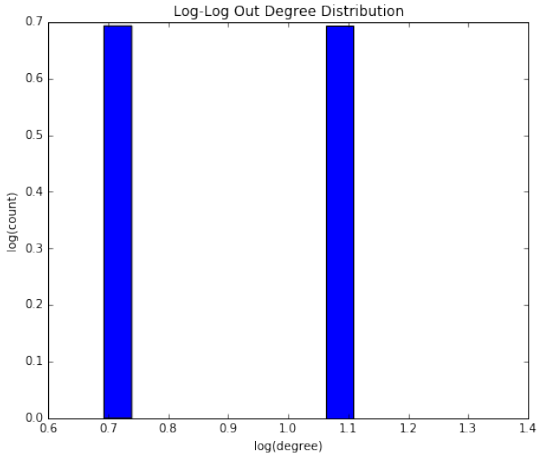
In [5]:
```
# This is a driver function to call the findNodes and findEdges MRJobs.
# It will generate a file with the results that can be used in the next function
# to show EDA

from __future__ import division
from MRJob7_1_findnodes import MRJob7_1_findnodes
from MRJob7_1_findlinks import MRJob7_1_findlinks
import numpy

def exploreData(filename,runnerType):
    mr_job = MRJob7_1_findnodes(args=[
        filename, '-r', runnerType ])
    nodes = 0

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

    # Compute the distribution of the links
    mr_job = MRJob7_1_findlinks(args=[
        filename, '-r', runnerType
    ])
    # Output number of nodes + degree distribution to a file
    with open(filename+ "_degreedist", 'w') as myfile:
        myfile.write(str(nodes)+"\n")
        with mr_job.make_runner() as runner:
            runner.run()
            for line in runner.stream_output():
                myfile.write(line)
```

In [6]:
```
%matplotlib inline

#This function takes the data generate by the findNodes and findEdges MRJobs and displays
# number of nodes, links, and average outgoing links per node
#This function also displays a histogram of the degree distribution and a log-log histogram
# of the degree distribution

import matplotlib
import numpy as np

def generateHistogram(filename):
    with open(filename+ "_degreedist", 'r') as myfile:
        nodes = int(myfile.readline())
        degree_occurences = [line.split("\t") for line in myfile.readlines()]

    # Get needed histogram statistics from graph data
    bins,weights = np.transpose(degree_occurences)
    bins = map(int,bins)
    weights = map(float,weights)
    links = numpy.sum([int(degree) * float(weight) for degree, weight in degree_occurences])
    print 'Number of nodes:',nodes
    print 'Number of links:', links
    print 'Average links:',links / nodes

    # Create a histogram
    fig = matplotlib.pyplot.figure(figsize=(16, 6))

    plot = fig.add_subplot(1, 2, 1)
    plot.set_xlabel("degree")
    plot.set_ylabel("count")
    plot.set_title("Out Degree Distribution")
    plot.hist(bins, 15, weights = weights)

    # Create a log-log histogram using matplotlib.
    log_bins = numpy.log(bins)
    log_weights = numpy.log(weights)

    log_plot = fig.add_subplot(1, 2, 2)
    log_plot.set_xlabel('log(degree)')
    log_plot.set_ylabel('log(count)')
    log_plot.set_title('Log-Log Out Degree Distribution')
    log_plot.hist(log_bins, 15, weights = log_weights)
```

In [7]:
```
print 'Undirected Toy Data Local'
exploreData('Data/undirected_toy.txt','inline')
generateHistogram('Data/undirected_toy.txt')
```

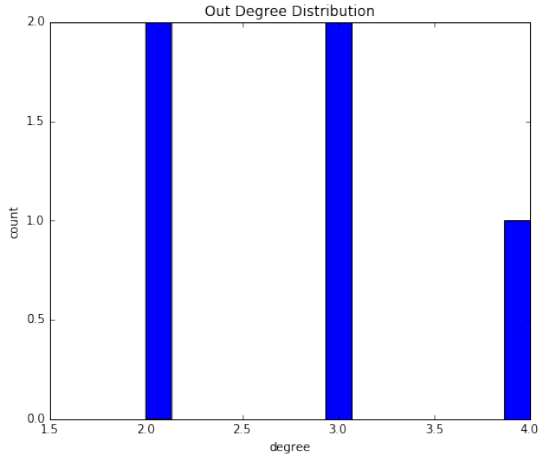
```
Undirected Toy Data Local
Number of nodes: 5
Number of links: 14.0
Average links: 2.8
```

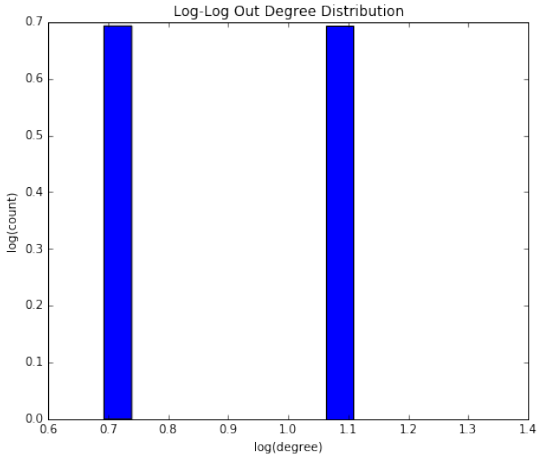





```
In [218]: print 'Undirected Toy Data  Hadoop'
          exploreData('Data/undirected_toy.txt','hadoop')
          generateHistogram('Data/undirected_toy.txt')
```

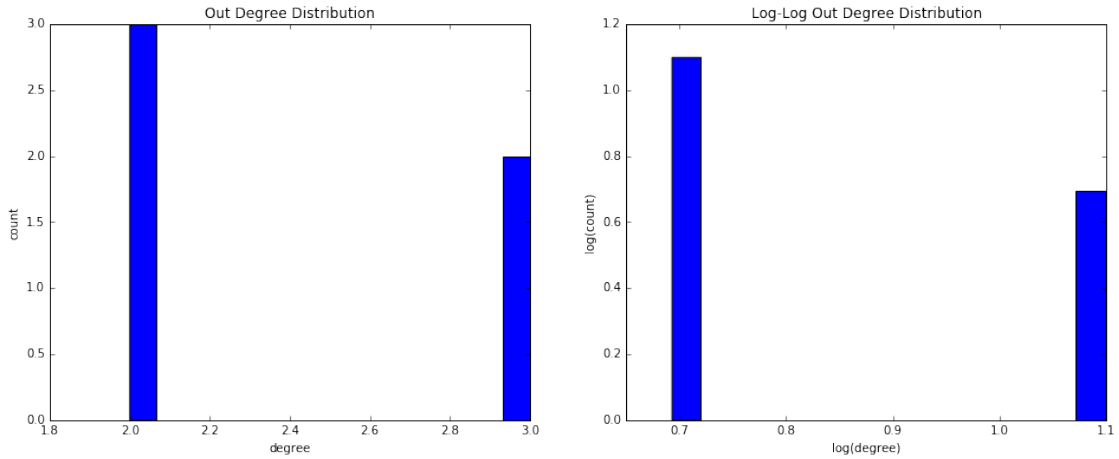
```
Undirected Toy Data  Hadoop
Number of nodes: 5
Number of links: 14.0
Average links: 2.8
```





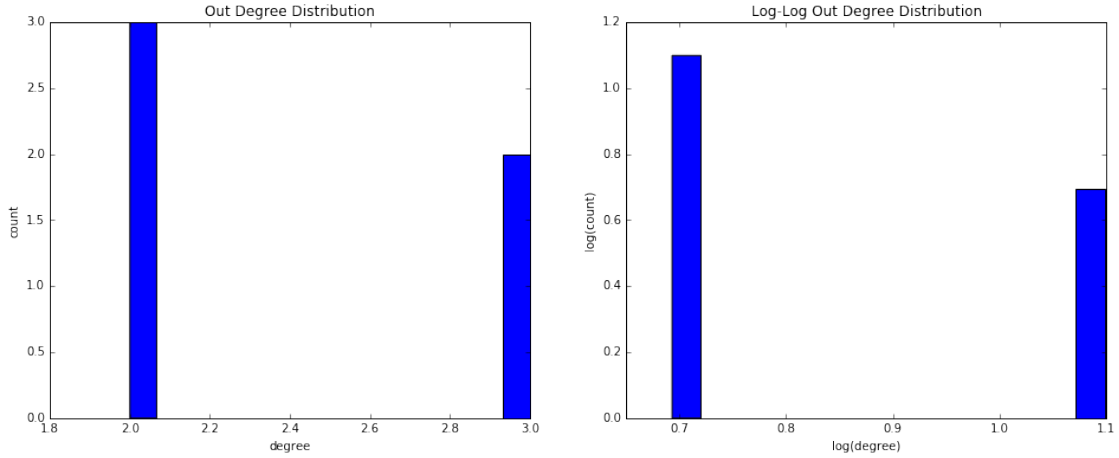

```
In [22]: print 'Directed Toy Data Local'
         exploreData('Data/directed_toy.txt','inline')
         generateHistogram('Data/directed_toy.txt')
```

```
Directed toy data on local
Number of nodes: 6
Number of links: 12.0
Average links: 2.0
```




```
In [46]: print 'Directed Toy Data Hadoop'
         exploreData('Data/directed_toy.txt','hadoop')
         generateHistogram('Data/directed_toy.txt')
```

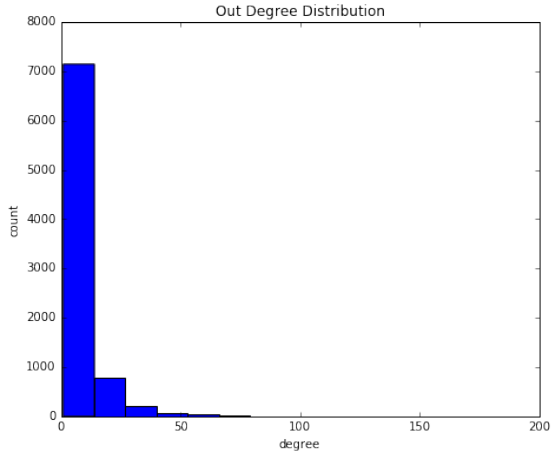
```
Directed Toy Data Hadoop
Number of nodes: 6
Number of links: 12.0
Average links: 2.0
```

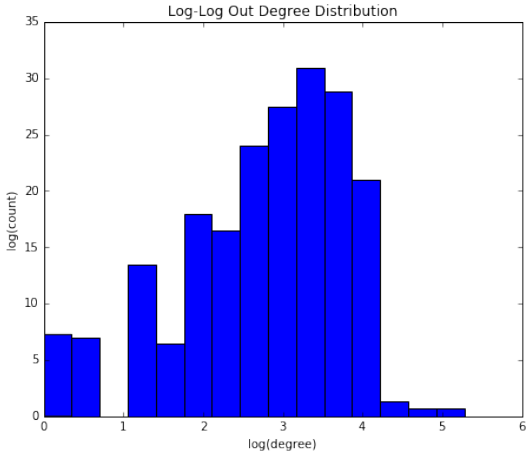


Now run on the full data set for real

```
In [47]: print 'NLTK Synonyms Local'
         exploreData('Data/synNet/synNet.txt','inline')
         generateHistogram('Data/synNet/synNet.txt')
```

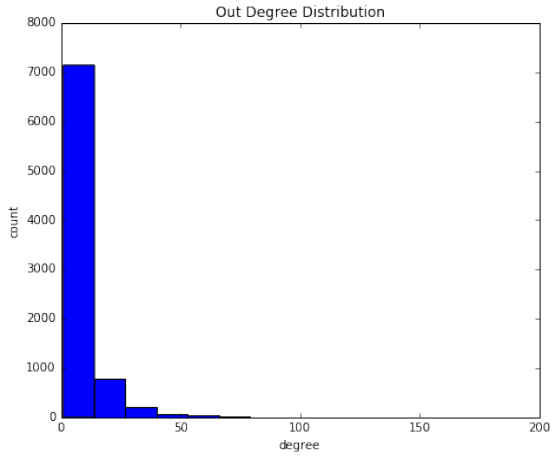
```
NLTK Synonyms Local
Number of nodes: 8271
Number of links: 61134.0
Average links: 7.39136742836
```

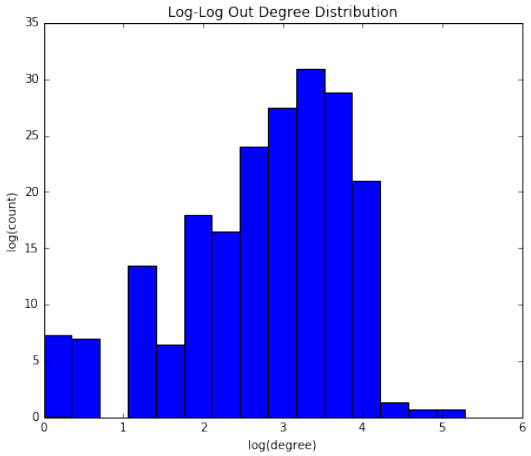





```
In [48]: print 'NLTK Synonyms Hadoop'
         exploreData('Data/synNet/synNet.txt','hadoop')
         generateHistogram('Data/synNet/synNet.txt')
```

```
NLTK Synonyms Hadoop
Number of nodes: 8271
Number of links: 61134.0
Average links: 7.39136742836
```





HW 7.2: Shortest path graph distances (NLTK synonyms)

```
In [20]: print "NLTK Synonyms Shortest Path Local"
         findShortestPath('Data/synNet/synNet.txt Data/synNet/indices.txt', 'walk', 'make','local')
```

```
NLTK Synonyms Shortest Path Local
```

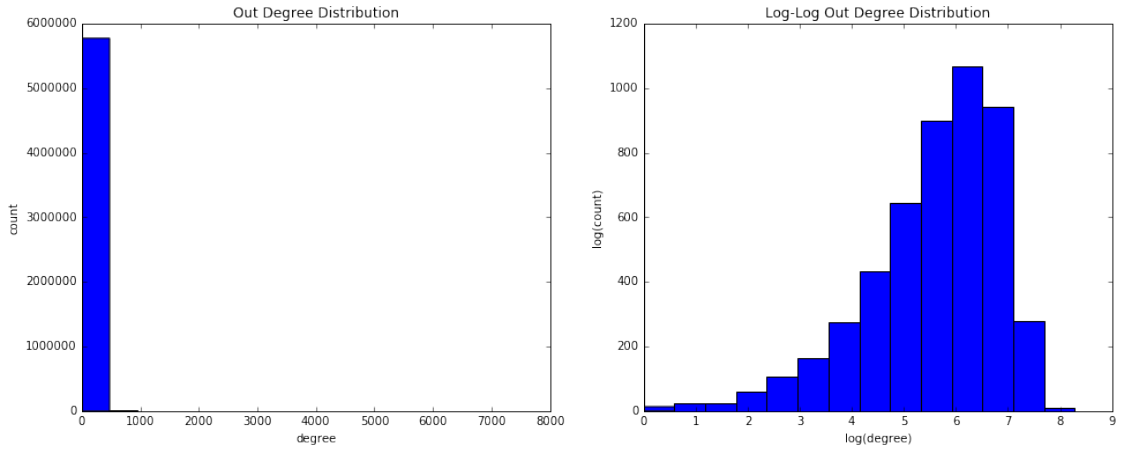
```
Out[20]: ['walk', 'passes', 'reach', 'make']
```

```
In [21]: print "NLTK Synonyms Shortest Path Hadoop"
         findShortestPath('Data/synNet/synNet.txt Data/synNet/indices.txt', 'walk', 'make','hadoop')
```

```
NLTK Synonyms Shortest Path Hadoop

Out[21]: ['walk', 'passes', 'reach', 'make']
```

HW 7.3: Exploratory data analysis (Wikipedia)

```
In [29]: print 'Wikipedia Out Data'
         exploreData('Data/wikipedia/all-pages-indexed-out.txt','hadoop')
         generateHistogram('Data/wikipedia/all-pages-indexed-out.txt')

Wikipedia Out Data
Number of nodes: 15192277
Number of links: 142114057.0
Average links: 9.35436189058
```



HW 7.4: Shortest path graph distances (Wikipedia)

```
In [30]: print "Wikiedpia Shortest Path Between Ireland and University of California, Berkeley"
         path = findShortestPath('Data/wikipedia/all-pages-indexed-out.txt Data/wikipedia/indices.txt'
                                 ,'Ireland', 'University of California, Berkeley','hadoop')
         print path

Wikiedpia Shortest Path Between Ireland and University of California, Berkeley
['Ireland', 'Seamus Heaney', 'University of California, Berkeley']

In [5]: print "Wikiedpia Shortest Path Between Nelson_le_Follet and Mairy"
        path = findShortestPath('Data/wikipedia/all-pages-indexed-out.txt Data/wikipedia/indices.txt',
                                'Nelson_le_Follet', 'Mairy','hadoop')
        print path

Wikiedpia Shortest Path Between Nelson_le_Follet and Mairy


        ---------------------------------------------------------------------------

        Exception                                 Traceback (most recent call last)

        <ipython-input-5-521a63196523> in <module>()
          1 print "Wikiedpia Shortest Path Between Nelson_le_Follet and Mairy"
```

```
      2 path = findShortestPath('Data/wikipedia/all-pages-indexed-out.txt Data/wikipedia/indices.tx
----> 3                          'Nelson_le_Follet', 'Mairy','hadoop')
      4 print path


    <ipython-input-4-8addc3509881> in findShortestPath(filenames, startNode, endNode, runnerType)
     33             path =counters[-1].replace(";",",")
     34             if path == "NA":
---> 35                 raise Exception("No path")
     36             path = map(str,eval(path))
     37             # Append endNode to the path.  The MRJob path alone only shows nodes leading up


    Exception: No path
```

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

At the moment, I think I would implement this task by checking each permutation of the N -2 non-start or end nodes in the network. For each permutation, I'd check if that path is possibe from the start node to the end node. If while transversing a permutation, I find that the next node is not reachable from the current node, just drop the next node from the permutation and try to move to the node after that. Once all permutations have been checked, return the one with the longest length that reaches the end node. If all N-2 nodes on a permutation are reachable in order, then terminate early and return that permutation (you can't have a longer path than N).

The main difference between the task of finding the longest path and finding the shortest path is that the task of finding the shortest path needs to only visit each node once, giving it a worst case run time of O(N+E). In contrast, the task of finding the longest path must revisit nodes.

The task of finding the longest path would be significantly more difficult than the task of finding the shortest path. It has O(N(N-2)!) complexity, far greater than the O(N+E) complexity for the task of finding the shortest path. While the meat of this task can be accomplished in parallel with (N-2)! machines (each machine would transverse one of the (N-2)! permutations), you still would inenvitably need to find the maximum length of each of the found paths, and that comparison alone would take O((N-2)!) time.

HW 7.5.1:

Combiners can help in the shortest path implementation. In theory, you could benefit from a combiner in instances where a queued node is on the same mapper as one of its neighbors. The combiner would combine the two yields from the mapper and send just one in queued node with complete information to the reducer. However, the reducer would need to be modified to handle queued nodes from the mapper that may have complete or incomplete information.

While order inversion may speed up some parts of the task, it is not technically neccesary for my shortest path implementation, at least not in my implementation. Each of my reducer functions will yield only one record per input key, so one could you collect all the values first and then process and yield in a reducer_final. However, order inversion in my implementation saves some time by making the reducers run the termination checks before processing the node information from the mappers. If a termination condition is met, then all the work by the reducer is going to be discarded anyway, so it's better to perform the termination check upfront rather than let the reducer process node information first.

HW 7.5.2: OPTIONAL

HW 7.6: Computational exercise: Largest single-source network distances: OPTIONAL

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

In [ ]: