Bioinformatics III

Second Assignment

Thibault Schowing (2571837)
Wiebke Schmitt (2543675)

April 25, 2018

Exercise 2.1: The Scale-Free network

(a) *Implement the algorithm given in the lecture to set up a scale-free network according to the Barabási-Albert model (see Lecture 2, slide 8). Start from the first three connected nodes and add each new node with a given number of links. Connect the new links with increasing preference to nodes that have higher degrees. This ScaleFreeNetwork-class should again use the abstract network class that you wrote in the first assignment. To obtain a much faster implementation and full points, think of a method to map the probabilities to connect to nodes somehow instead of computing them from scratch in each iteration.*
Implementation of the missing methods for the ScaleFreeNetwork-class. Listing 1 shows source code of ScaleFreeNetwork.py.

Listing 1: ScaleFreeNetwork.py

```
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
import random
from Node import Node
from AbstractNetwork import AbstractNetwork

class ScaleFreeNetwork(AbstractNetwork):
    """Scale-free network implementation of AbstractNetwork"""



    def __createNetwork__(self, amount_nodes, amount_links):
        """
        Create a network with an amount of n nodes, add m links per iteration
            step
        for n nodes:
            for m links:
                link node to other nodes
        """

        def symetricConnection(node1, node2):
            node1.addLinkTo(node2)
            node2.addLinkTo(node1)

        def genProbList():

            # Generate probability range for each nodes

            prob_list = []
            sumkj = self.degreeSum()

            for key, node in self.nodes.items():
                # calculate pi
                # ki = degree of node i
                #print("Debug: node degree: ", node.degree())
                ki = node.degree()
```

```
                # Sum kj = sum of all degrees

                prob_list.append(ki / sumkj)
                #print("Debug select neighbours prob list: ", prob_list)

            # now we have a probability list -> select the number_neighbours
                future neighbours

            #print("DEBUG probability list generated: ", prob_list)
            return prob_list



    random.seed()
    print("Debug: ", amount_nodes, " nodes and ", amount_links, "links ....
        creating ScaleFree network")

    # Initial m0 nodes connected to each other
    #
    # #QUESTION: is 3 fixed ??? Should it be dynamic ?

    m0 = 3
    # Number of links per node
    number_neighbours = amount_links

    # Contains the degrees of the initial complete network
    degree_list = [2,2,2]

    for i in range(0, m0):
        self.appendNode(Node(i))

    symetricConnection(self.getNode(0), self.getNode(1))
    symetricConnection(self.getNode(1), self.getNode(2))
    symetricConnection(self.getNode(0), self.getNode(2))

    # useless and slow
    # http://didar-physics.blogspot.de/2015/02/barabasialbert-model-
        generated-code.html
    # https://stackoverflow.com/questions/38008748/python-implementing-a-
        step-by-step-modified-barabasi-albert-model-for-scale-fr

    # Random failure measure
    random_failure = 0

    # new nodes id (without the 3 initial nodes)
    for new_node_id in range(3, amount_nodes - 3):

        #print("Debug: population: ", population)
        new_node = Node(new_node_id)
        self.appendNode(new_node)

        # Just a sequence of all node ids
        population = list(range(0, self.size()))

        # Generate probability list of existing nodes
        prob_list = genProbList()

        for i in range(amount_links):


            while(True):
                chosen_neighbour = random.choices(population, weights=
                    prob_list, k = 1)[0]
```

```
                    # if it's a new link and it's not a self-connection
                    if not new_node.hasLinkTo(chosen_neighbour) and not
                        chosen_neighbour == new_node.id:

                        symetricConnection(new_node, self.getNode(
                            chosen_neighbour))
                        break

                    # Random failure increment
                    random_failure += 1

            # debug info: print degrees
            self.degrees = []
            for id, node in self.nodes.items():
                self.degrees.append(node.degree())
            #print(self.degrees)


        print("Debug Random failure count: ", random_failure)
```

(b) *Determine the degree distributions for scale-free networks of 10 000 and 100 000 nodes (each with two new links per iteration), respectively, and plot them with double logarithmic axes. A new pre-implemented method in Tools.py will help you with that. What are the differences? Next, compare one of the distributions to the degree distribution of an equally sized random network (play around with the plot-scaling). What are the major differences?*

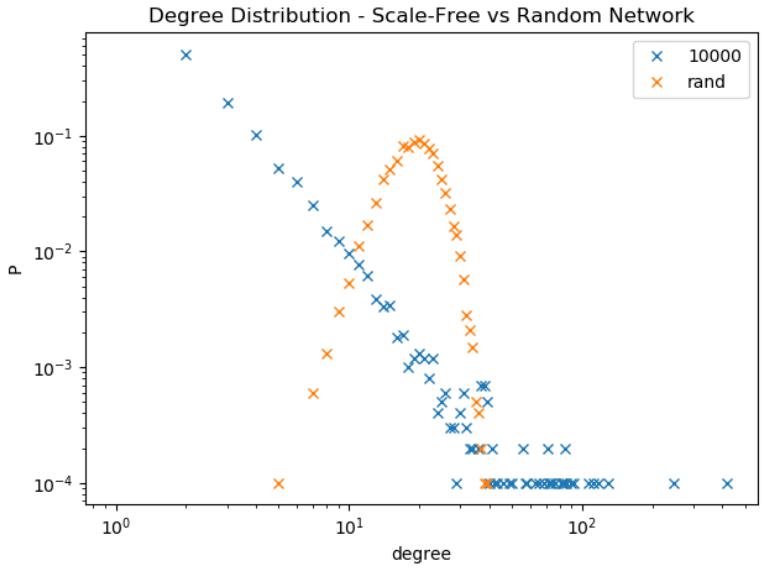

Figure 1: We can observe that the Scale-Free distribution follow a power-law style, as it is almost a straight line when ploted with log-log axes. On the other hand, the random network is more Poisson distributed as seen in Assignment 1.

Listing 2: ScaleFreeTest.py

```
#!/usr/bin/python
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
from ScaleFreeNetwork import ScaleFreeNetwork
from DegreeDistribution import DegreeDistribution
from RandomNetwork import RandomNetwork
```

```
import matplotlib.pyplot as plt
import Tools as Tools


if __name__=="__main__":

    # TASK 2.1 a AND b
    # Creating two networks and taking the degree distributions
    small = 100
    big = 1000
    sf_net = ScaleFreeNetwork(small,2)
    sf_net2 = ScaleFreeNetwork(big,2)

    rand_net = RandomNetwork(10000, 100000)

    sf_degree = DegreeDistribution(sf_net).getNormalizedDistribution()
    sf_degree2 = DegreeDistribution(sf_net2).getNormalizedDistribution()
    rand_degree = DegreeDistribution(rand_net).getNormalizedDistribution()

    # Plot the degree distribution
    Tools.plotDistributionComparisonLogLog([sf_degree, sf_degree2, rand_degree
        ],[small,big, "rand"], "Degree Distribution Scale-Free network")
    Tools.plotDistributionComparisonLogLog([sf_degree2, rand_degree], [big, "
        rand"],"Degree Distribution - Scale-Free vs Random Network")

    # TASK 2.1 c

    sf_net_c = ScaleFreeNetwork(10000, 2)
    sf_net_c_degree = DegreeDistribution(sf_net_c).getNormalizedDistribution()

    k = len(sf_net_c_degree)

    gamma_distance = []

    # Foreach gamma, calculate the KS distance
    steps = [x * 0.1 for x in range(10, 30)]
    for gamma in steps:
        theoretical_dist = Tools.getScaleFreeDistributionHistogram(gamma, k)
        #Normalize
        #TODO normalize according to what ?

        distance = Tools.simpleKSdist(sf_net_c_degree, theoretical_dist)
        gamma_distance.append((gamma, distance))

    # Sort the distances-gamma tuples
    gamma_distance.sort(key=lambda x: x[1], reverse=True)

    print("All gamma-distance: ", gamma_distance)
    print("Best gamma: ", gamma_distance[0][0])

    # Optimal theoretical distribution (powerlaw)
    optimal_theoretical = Tools.getScaleFreeDistributionHistogram(
        gamma_distance[0][0], k)
    Tools.plotDistributionComparisonLogLog([sf_net_c_degree,
        optimal_theoretical], ['Scale Free Network', 'PowerLaw'],'Compare
        theory to practice')
```

(c) *The degree distribution of a scale-free network follows a power law, which has the form* $P(k)\ k^{-\lambda}$ *To simplify the exercise, we assume* $P(k)\ Ck^{-\lambda}$ *with* C *being a fixed normalization constant to obtain a proper distribution. Try to fit this theoretical distribution to the degree distribution of a random network using the Kolmogorov-Smirnov distance.*

Listing 3: Tools.py

```
import matplotlib.pyplot as plt
import math
```

```
def plotDistributionComparison(histograms, legend, title):
    '''
    Plots a list of histograms with matching list of descriptions as the
        legend
    '''
    # determine max. length
    max_length = max(len(x) for x in histograms)

    # extend "shorter" distributions
    for x in histograms:
        x.extend([0.0]* (max_length-len(x)) )

    # plots histograms
    for h in histograms:
        plt.plot(range(len(h)), h, marker = 'x')

    # remember: never forget labels!
    plt.xlabel('degree')
    plt.ylabel('P')

    # you don't have to do something stuff here
    plt.legend(legend)
    plt.title(title)
    plt.tight_layout()
    plt.show()


def plotDistributionComparisonLogLog(histograms, legend, title):
    '''
    Plots a list of histograms with matching list of descriptions as the
        legend
    '''
    fig = plt.figure()
    ax = plt.subplot()
    # determine max. length
    max_length = max(len(x) for x in histograms)

    # extend "shorter" distributions
    for x in histograms:
        x.extend([0.0]* (max_length-len(x)) )

    ax.set_xscale("log")
    ax.set_yscale("log")

    # plots histograms
    for h in histograms:
        ax.plot(range(len(h)), h, marker = 'x', linestyle='')

    # remember: never forget labels!
    plt.xlabel('degree')
    plt.ylabel('P')

    # you don't have to do something stuff here
    plt.legend(legend)
    plt.title(title)
    plt.tight_layout()

    # Uncomment the line below to display normally
    # plt.show()

    # Comment the 2 lines below to display normally
    filename = title + ".png"
    fig.savefig(filename)

def getScaleFreeDistributionHistogram(gamma, k):
```

```
        '''
        Generates a Power law distribution histogram with slope gamma up to degree
            k
        '''
        histogram = []
        # NORMALISATION_CONSTANT \
        # Todo here or in ScaleFreeTest.py

        for i in range(1, k):
            histogram.append(i**-gamma)


        return histogram


    def simpleKSdist(histogram_a, histogram_b):
        '''
        Simple Kolmogorov-Smirnov distance implementation
        '''
        #print("KSDIST: size hist 1: ", len(histogram_a), "  Size hist2: ", len(
            histogram_b))
        D = 0
        # Assume that the two hists have the same length
        for i in range(0, len(histogram_a)-1):
            D = max(D, (histogram_a[i] - histogram_b[i]))

        return D
```

Use the KS distance to determine a γ (between 1 and 3, 0.1 steps sufficient) that fits best to the degree distribution of a scale-free network with 10 000 nodes and two new links per iteration. Compare the empirical distribution of the network to the theoretical distribution with optimal γ in a double-log. plot. Comment on the quality of your fit, reason why it may fail and how it could be vastly improved.

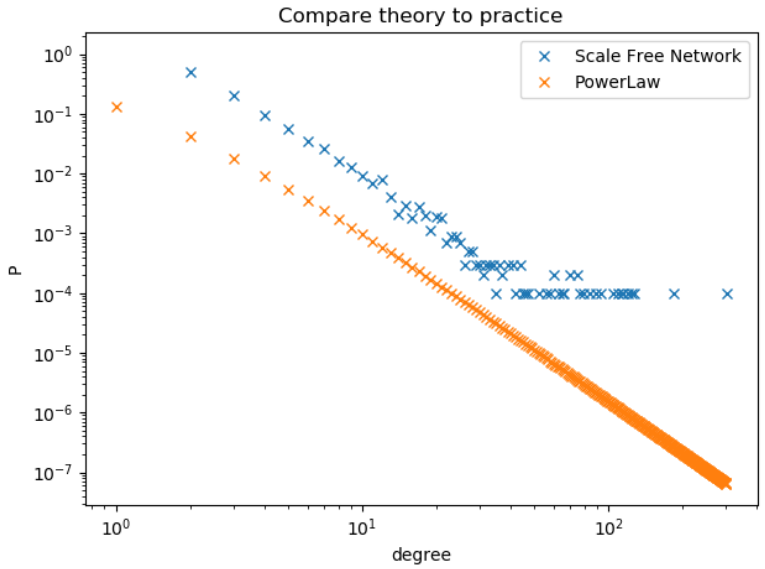

Figure 2:

Figure 3:

Exercise 2.2: Classify real-world network examples

(a) bla

(b) bla

(c) bla

Exercise 2.3: Real interaction networks

(a) Here is the implementation of the BioGRIDReader-class

Listing 4: BioGRIDReader.py

```
import operator
from GenericNetwork import GenericNetwork
from DegreeDistribution import DegreeDistribution
import Tools as Tools

class BioGRIDReader:
    '''Reads BioGRID tab files'''
    def __init__(self, filename):
        '''
        Initialization, read in file and build any data structure that makes
            you happy
        '''

        content_start = False

        # Temporary tab -> contains one line
        line_tab = []

        self.INTERACTOR_A = []
        self.INTERACTOR_B = []
        self.OFFICIAL_SYMBOL_FOR_A = []
        self.OFFICIAL_SYMBOL_FOR_B = []
        self.ALIASES_FOR_A = []
        self.ALIASES_FOR_B = []
        self.EXPERIMENTAL_SYSTEM = []
        self.SOURCE = []
        self.PUBMED_ID = []
        self.ORGANISM_A_ID = []
        self.ORGANISM_B_ID = []

        with open(filename, "r") as f:
            for line in f:
                if line.startswith("INTERACTOR_A"):
                    content_start = True
                    continue
                if content_start:
                    # Process data
                    line = line.rstrip()
                    line_tab = line.split('\t')

                    self.INTERACTOR_A.append(line_tab[0])
                    self.INTERACTOR_B.append(line_tab[1])
                    self.OFFICIAL_SYMBOL_FOR_A.append(line_tab[2])
                    self.OFFICIAL_SYMBOL_FOR_B.append(line_tab[3])
                    self.ALIASES_FOR_A.append(line_tab[4])
                    self.ALIASES_FOR_B.append(line_tab[5])
                    self.EXPERIMENTAL_SYSTEM.append(line_tab[6])
                    self.SOURCE.append(line_tab[7])
                    self.PUBMED_ID.append(line_tab[8])
                    self.ORGANISM_A_ID.append(line_tab[9])
                    self.ORGANISM_B_ID.append(line_tab[10])

        # The file has now been read and all infos are in lists
```

```
        # Tuples can store pairwise interactions


55
    def getMostAbundantTaxonIDs(self, n):

        interact = {}
        organism_pairs_list = zip(self.ORGANISM_A_ID, self.ORGANISM_B_ID)
60
        for A, B in organism_pairs_list:
            if not A in interact:
                interact[A] = 1
            else:
65              interact[A] += 1

            # If both are the same, the interaction must be counted only once
            if A != B:
                if not B in interact:
70                  interact[B] = 1
                else:
                    interact[B] += 1
        # Sort the dict to retrieve the n first
        # https://stackoverflow.com/questions/613183/how-do-i-sort-a-
            dictionary-by-value
75
        sorted_interact = sorted(interact.items(), key=operator.itemgetter(1))


        nFirst = []
80      for i in range(1, n+1):
            nFirst.append(sorted_interact[-i])

        return nFirst

85  def getHumanInteraction(self):
        # Search for human-human interactions
        nb_human_human_interact = 0

        organism_pairs_list = zip(self.ORGANISM_A_ID, self.ORGANISM_B_ID)
90      for A, B in organism_pairs_list:
            if A == B == "9606":
                nb_human_human_interact += 1

        print("HUMAN_INTERACTIONS\n")
95      print("\nNumber of Human-Human interactions (human id = 9606): ",
            nb_human_human_interact)

        # Now we need the indices of the human - human interactions
        # The code below extract the indices where ORGANISM A / ORGANISM B are
            human and take the intersection
        # Order dict: https://stackoverflow.com/questions/16772071/sort-dict-
            by-value-python
100
        # Get Indexes
        indexesA = [i for i, x in enumerate(self.ORGANISM_A_ID) if x == '9606'
            ]
        indexesB = [i for i, x in enumerate(self.ORGANISM_B_ID) if x == '9606'
            ]

105     # Get intersection
        indexes = list(set(indexesA).intersection(indexesB))

        proteins = [self.INTERACTOR_A[i] for i in indexes]
        proteins.extend([self.INTERACTOR_B[i] for i in indexes])
110
        proteins_count = {}
        for prot in proteins:
```

```
                if prot not in proteins_count:
                    proteins_count[prot] = 1
                else:
                    proteins_count[prot] += 1
            proteins_count = sorted(proteins_count.items(), key=lambda x:x[1])

            # Obtain the n most used proteins
            n = 10
            nFirst = []
            for i in range(1, n + 1):
                nFirst.append(proteins_count[-i])

            print("\nThe ", n, " proteins with the highest degree are: ")
            print(nFirst)

        def writeInteractionFile(self, taxon_id, filename):

            organism_pairs_list = zip(self.ORGANISM_A_ID, self.ORGANISM_B_ID)
            file = open(filename, "w+")

            # Get Indexes
            indexesA = [i for i, x in enumerate(self.ORGANISM_A_ID) if x ==
                taxon_id]
            indexesB = [i for i, x in enumerate(self.ORGANISM_B_ID) if x ==
                taxon_id]

            # Get intersection
            indexes = list(set(indexesA).intersection(indexesB))

            for i in indexes:
                file.write(self.INTERACTOR_A[i])
                file.write("\t")
                file.write(self.INTERACTOR_B[i])
                file.write("\n")

            file.close()


    if __name__== "__main__":

        bio = BioGRIDReader("BIOGRID-ALL-3.4.159.tab.txt")
        abundantTaxon = bio.getMostAbundantTaxonIDs(5)
        print("The most abundent TaxonIDs are (id, qty): ", abundantTaxon)
        bio.getHumanInteraction()

        # Export human interactions to a file
        EXPORT_FILE_NAME = "humanFile.txt"
        EXPORT_ORGANISM = "9606"
        bio.writeInteractionFile(EXPORT_ORGANISM, EXPORT_FILE_NAME)

        # Create GenericNetwork with previously exported file
        gen = GenericNetwork(EXPORT_FILE_NAME)
        print(str(gen))

        # Get distribution
        gen_degree = DegreeDistribution(gen).getNormalizedDistribution()
        # Plot the degree distribution
        Tools.plotDistributionComparisonLogLog([gen_degree], ["Human"],"Degree
            Distribution Generic network ")
```

(b) The class getMostAbundantTaxonIDs(n) is listed in the listing 4 above.

(c) *How big is the human interaction network and which are the 10 proteins with the highest degree? Take one of them as an example and briefly explain the biology behind the connectivity.*

The most abundent TaxonIDs are (id, qty): [('559292', 704012), ('9606', 414501), ('316407',

184023), ('284812', 72149), ('7227', 67935)]

- 559292: Saccharomyces cerevisiae (Baker's Yeast)
- 9606: Human (Homo sapiens)
- 316407: Escherichia coli
- 284812: Schizosaccharomyces pombe (Fission yeast)
- 7227: Drosophila melanogaster

Number of Human-Human interactions (human id = 9606): 386192

The 10 proteins with the highest degree are: [('ETG7157', 3024), ('ETG7706', 2559), ('ETG351', 2454), ('ETG1956', 2134), ('ETG7316', 2042), ('ETG4914', 2002), ('ETG4193', 1939), ('ETG672', 1876), ('ETG1994', 1840), ('RP4-811H24.2', 1646)]

This network has 17087 nodes.

(d) Generic network implementation

Listing 5: GenericNetwork.py

```
from AbstractNetwork import AbstractNetwork
from Node import Node

# from standard library module
from itertools import islice
import sys

class GenericNetwork(AbstractNetwork):


    def __init__(self, filename):
        """
        Create a network from a file
        """

        self.nodes = {}
        # We first need to create all Nodes (unique)
        allEntries = []
        pairs = []
        with open(filename) as f:

            # Run through the entire file to make a set of entries
            for line in f:
                line = line.rstrip()
                line_tab = line.split('\t')
                pairs.append(line_tab)
                allEntries.extend(line_tab)

            allUniqueEntries = set(allEntries)
            for n in allUniqueEntries:
                self.appendNode(Node(n))

            for pair in pairs:
                self.getNode(pair[0]).addLinkTo(self.getNode(pair[1]))
                self.getNode(pair[1]).addLinkTo(self.getNode(pair[0]))
```