Bioinformatics III

First Assignment

Thibault Schowing (2571837)
Wiebke Schmitt (2543675)

April 18, 2018

Exercise 1.1: The random network

(a) Implementation of the missing methods for the Node-class. Listing 1 shows source code of Node.py.

Listing 1: Node.py

```python
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
#Usefull link to begin with classes: http://introtopython.org/classes.html

class Node:
    def __init__(self, identifier):
        """
        Sets node id and initialize empty node list that references its
            connected nodes
        """
        self.identifier = identifier

        # More about lists in Python 3.5:
        # https://docs.python.org/3.5/tutorial/datastructures.html
        self.neighbours_list = []

    def hasLinkTo(self, node):
        """
        Returns True if this node is connected to node asked for,
        False otherwise
        """
        return node in self.neighbours_list

    def addLinkTo(self, node):
        """
        Adds link from this node to parameter node (only if there is no link
            connection already),
        does not automatically care for a link from parameter node to this
            node
        """
        if node not in self.neighbours_list:
            self.neighbours_list.append(node)

    def degree(self):
        """
        Returns degree of this node
        """
        return len(self.neighbours_list)

    def __str__(self):
        """
        Returns id of node as string
        """
        return self.identifier
```

(b) Implementation of the missing methods for the AbstractNetwork-class. Listing 2 shows source code of AbstractNetwork.py.

Listing 2: AbstractNetwork.py

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

class AbstractNetwork:
    """Abstract network definition, can not be instantiated"""

    def __init__(self, amount_nodes, amount_links):
        """
        Creates empty nodelist and call createNetwork of the extending class
        """
        self.nodes = {}
        self.__createNetwork__(amount_nodes, amount_links)

    def __createNetwork__(self, amount_nodes, amount_links):
        """
        Method overwritten by subclasses, nothing to do here
        """
        raise NotImplementedError

    def appendNode(self, node):
        """
        Appends node to network
        """
        self.nodes[node.identifier] = node

    def maxDegree(self):
        """
        Returns the maximum degree in this network
        """
        deg_list = []
        for key, value in self.nodes.items():
            deg_list.append(value.degree())

        #print("Debug: deg_list ", deg_list)
        #print("Debug: max degree: ", max(deg_list))
        return max(deg_list)

    def size(self):
        """
        Returns network size (here: number of nodes)
        """
        return len(self.nodes)

    def __str__(self):
        '''
        Any string-representation of the network (something simply is enough)
        '''
        # will contain: {identifier : neighbours} -> dict are printed pretty
            nicely
        self.networkdict = {}
        for n in self.nodes.values():
            # n is a node -> contains identifier and neighbours
            self.networkdict[n.identifier] = n.neighbours_list
        return str(self.networkdict)



    def getNode(self, identifier):
        """
        Returns node according to key
        """
        return self.nodes[identifier]
```

(c) Implementation of the missing methods for the RandomNetwork-class. Listing 3 shows source code of RandomNetwork.py.

Listing 3: RandomNetwork.py

```
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
from AbstractNetwork import AbstractNetwork
from Node import Node
import random # you will need it :-)

class RandomNetwork(AbstractNetwork):
    """Random network implementation of AbstractNetwork"""

    def __createNetwork__(self, amount_nodes, amount_links): # remaining methods are taken from AbstractNetwork
        """
        Creates a random network
        1. Build a list of n nodes
        2. For i=#links steps, add a connection between for two randomly
           chosen nodes that are not yet connected
        """

        #print("debug: Init RandomNetwork")

        random.seed()

        self.maxLink = ((amount_nodes * (amount_nodes - 1)) / 2)

        if amount_links > self.maxLink:
            # https://docs.python.org/2/library/exceptions.html
            raise ValueError("The requested number of link (amount_links) is too high compared to the number of nodes (amount_nodes).")

        # Probability for an edge between two randomly selected nodes.
        self.probability = amount_links / self.maxLink

        # Average degree
        self.averageDegree = (2 * amount_links)/amount_nodes

        # Creation of the nodes with identifier from 0 to amount_nodes - 1
        # and append them to the network with the AbstractNetwork's function
        for i in range(0, amount_nodes):
            self.appendNode(Node(i))


        # Create "amount_links" random connection between two nodes.
        # Verify if the connection already exists and if it is connecting the node to itself
        # Usefull help: https://stackoverflow.com/questions/22842289/generate-n-unique-random-numbers-within-a-range

        for i in range(0, amount_links):
            self.randNum = random.sample(range(0, amount_nodes), 2)

            self.tmpNode1 = self.nodes[self.randNum[0]]
            self.tmpNode2 = self.nodes[self.randNum[1]]

            # Controls

            if not self.randNum[0] == self.randNum[1] and not self.tmpNode1.hasLinkTo(self.tmpNode2):
                # Not the same and not already connected
                self.symetricConnection(self.tmpNode1, self.tmpNode2)



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

```
        node2.addLinkTo(node1)
```

Exercise 1.2: Degree distribution of random networks

(a) Implementation of the missing methods for the DegreeDistribution-class. Listing 4 shows source code of DegreeDistribution.py.

Listing 4: DegreeDistribution.py

```
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
class DegreeDistribution:
    """Calculates a degree distribution for a network"""

    def __init__(self, network):
        """
        Inits DegreeDistribution with a network and calculate its distribution
        """
        self.histogram = [0] * (network.maxDegree() + 1)

        for key, node in network.nodes.items():
            self.histogram[node.degree()] += 1

        #print("Debug: histogram list ", self.histogram)

        # Other option:
        # Dict containing {id:degree}
        # self.degrees = {}
        # for node in network.nodes.iteritems():
        # self.degrees[node.identifier] = node.degree()
        # for i in range(0, network.maxDegree() + 1:
        #     self.histogram[i] = self.degrees.values().count(i)

    def getNormalizedDistribution(self):
        '''
        Returns the computed normalized distribution
        '''

        maxvalue = max(self.histogram)
        norm = [float(i) / maxvalue for i in self.histogram]
        #print("Debug: normalized histogram ", norm)
        return norm
```

(b) Implementation of the missing methods for the Tools.py methods. Listing 5 shows source code of Tools.py

Listing 5: Tools.py

```
#Bioinformatics 3 : Wiebke Schmitt & Thibault Schowing
import matplotlib.pyplot as plt
import math


def poisson(k, lambda_):

    #If k == 0, the first part is equal to 1 -> less computation (?)
    if k == 0:
        return math.exp(-lambda_)
    else:
        return ((lambda_**k) / (math.factorial(k))) * math.exp(-lambda_)


def plotDistributionComparison(histograms, legend, title):
    '''
    Plots a list of histograms with matching list of descriptions as the
        legend
    '''
```

```
    # adjust size of elements in histogram
    # https://stackoverflow.com/questions/13400876/python-length-of-longest-
        sublist

    maxlength = len(max(histograms,key=len))
    #print("Debug max len", maxlength)
    for h in histograms:
        # Expant the current histogram (table) to the size of the biggest one
        h.extend([0.0] * (maxlength - len(h)))

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

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

    # you don't have to do something here
    plt.legend(legend)
    plt.title(title)
    plt.tight_layout()# might throw a warning, no problem

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

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


def getPoissonDistributionHistogram(num_nodes, num_links, k):
    '''
    Generates a Poisson distribution histogram up to k
    '''

    lambda_ = (2 * num_links) / num_nodes

    poissonDistribution = []

    # From 0 to k included
    for i in range(0, k + 1):
        poissonDistribution.append(poisson(i, lambda_))

    #print("Debug Poisson histogram", poissonDistribution)
    return poissonDistribution
```

To get visually pleasing plots ensure that all distributions that are plotted together have the same length. This can be done by appropriately extending the shorter ones. Why does this happen and how do you need to "fill" the shorter distributions? Are the ranges of the discrete distributions we obtain in (c) deterministic in our case?

Plot 1:	50/100	500/1000	5000/10000	50000/100000
Plot 2:	20000/5000	20000/17000	20000/40000	20000/70000

Figure 1:

The data in the figure 1, corresponds with figure 2 (plot 1) and figure 3 (plot 2). In figure 2 we can observe that the ratio node/edges is equal among all plots. We have seen in the lectures that the larger is the graph, the more the probability that a random node has a link to k other nodes is Poisson distributed. In the second plot, we keep the same number of nodes but increase the number of edges. This has the effect to increase the mean of the

Poisson distribution. We can easily observe the orange, red, brown and grey bell curves sliding to the right as the number of edges increases.

We fill the shorter distributions with zeros because Poisson distributions tend to zero and we assume the missing values (*P(k)*) at the end of our tables, so for large *k's*, will tend to zero.

The ranges of the discrete distributions we obtain cannot be deterministic because we generate our graphs randomly. So the max degree of the distribution is random.

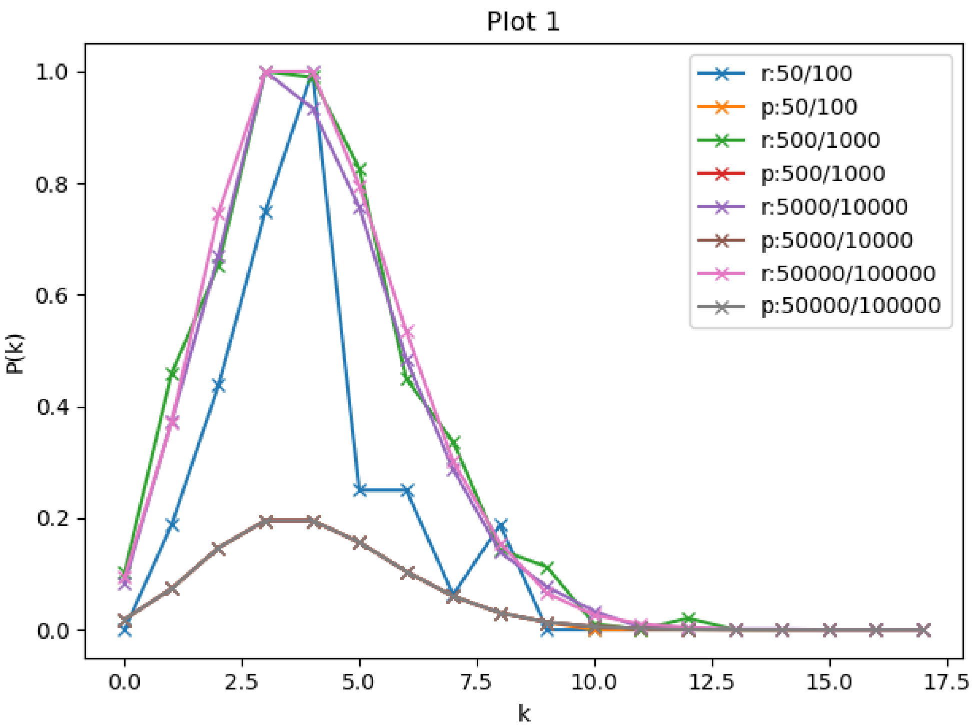

Figure 2:

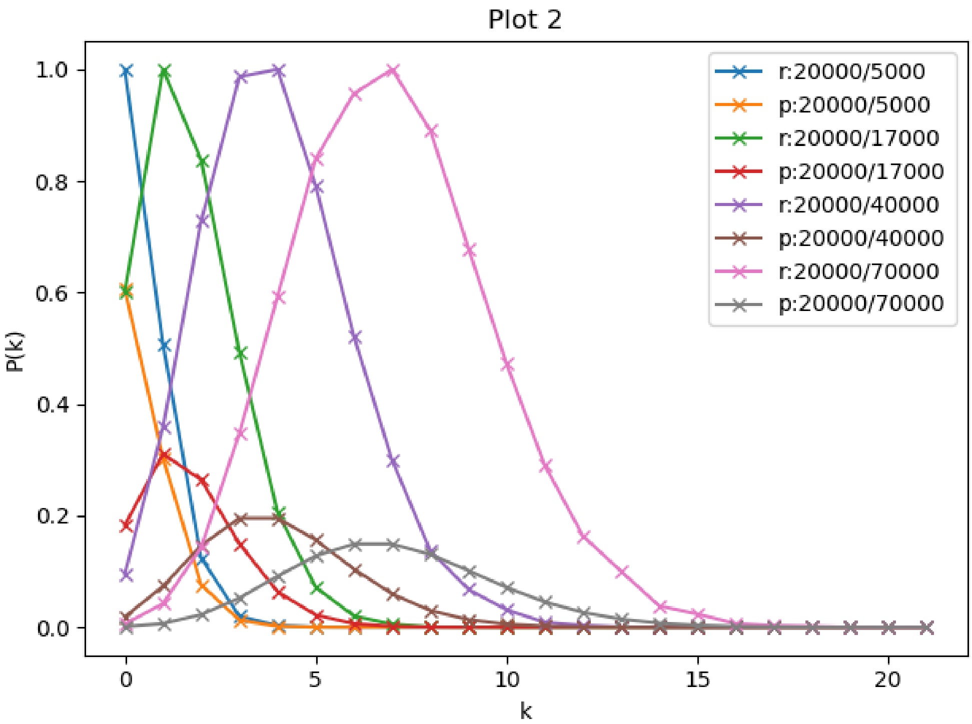
Plot 2
r:20000/5000
p:20000/5000
r:20000/17000
p:20000/17000
r:20000/40000
p:20000/40000
r:20000/70000
p:20000/70000
P(k)
k
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

Figure 3: