Bioinformatics III

Fifth Assignment

Thibault Schowing (2571837)
Wiebke Schmitt (2543675)

May 23, 2018

Exercise 5.1: Cliques and Network Evolution

(a) *Reading network files*
The class GenericNetwork contains the functions to read the network from a file and also the function to count cliques.

Listing 1: generic_network.py

```python
from node import Node
import itertools
import copy


class GenericNetwork:
    def __init__(self):
        # key: node identifier, value: Node-object
        self.nodes = {}
        self.nb_edges = 0

    def read_from_tsv(self, file_path):
        """
        Reads white-space-separated files that contain two or more columns.
            The first two columns contain the
        identifiers of two nodes that have an undirected edge. The two nodes
            are added to the network.
        :param file_path: path to the file
        """
        # clear the prior content of the network
        self.nodes = {}

        # open the file for reading
        with open(file_path, 'r') as file:
            # iterate over the lines in the file
            for line in file:
                #
                columns = line.split()

                # skip lines that do not have two node identifiers
                if len(columns) < 2:
                    continue

                # We ignore if there is more than one connection
                # create the two nodes and remove potential whitespace such as
                    new-line from their identifiers
                node_1 = Node(columns[0].strip())
                node_2 = Node(columns[1].strip())
                # add the nodes and the edge between them to the network
                self.add_node(node_1)
                self.add_node(node_2)
                self.add_edge(node_1, node_2)


    def get_nodes(self):
        """

        :return: the dict of nodes
        """
        return copy.deepcopy(self.nodes)


    def add_node(self, node):
        """
        Adds the specified node to the network.
        :param node: Node-object
        """
        if node.identifier not in self.nodes.keys():
            self.nodes[node.identifier] = node

    def add_edge(self, node_1, node_2):
        """
        Adds an (undirected) edge between the two specified nodes.
        :param node_1: Node-object
        :param node_2: Node-object
        :raises: KeyError if either node is not in the network
        """
        # raise an error if the nodes are not in the network
        if node_1.identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 node_1)
        if node_2.identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 node_2)

        # add the (undirected) edge
        self.nodes[node_1.identifier].add_edge(node_2)
        self.nodes[node_2.identifier].add_edge(node_1)

        # increment the number of edge of 1
        self.nb_edges += 1

    def get_node(self, identifier):
        """
        :param identifier: node identifier
        :return: Node-object corresponding to the given node identifier, if
            the node is in the network
        :raises: KeyError if there is no node with that identifier in the
            network
        """
        if identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 identifier)
        return self.nodes[identifier]

    def has_edge(self, node_1, node_2):
        """
        :param node_1: Node-object
        :param node_2: Node-object
        :return: True if the two nodes have an (undirected) edge, False
            otherwise
        :raises: KeyError if either node is not in the network
        """
        # raise an error if the nodes are not in the network
        if node_1.identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 node_1)
        if node_2.identifier not in self.nodes.keys():
                raise KeyError('There is no node in the network with identifier:',
                    node_2)

        return node_1.has_edge_to(node_2) and node_2.has_edge_to(node_1)

    def size(self):
        """
        :return: number of nodes in the network
        """
        return len(self.nodes.keys())

    def nb_edges(self):
        """

        :return: number of edges
        """
        return self.nb_edges()

    def max_degree(self):
        """
        :return: highest node degree in the network, 0 if there are no nodes
            in the network
        """
        return max([node.degree() for node in self.nodes.values()], default=0)

    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
            nblist = []
            for elem in n.neighbour_nodes:
                nblist.append(elem)
            self.networkdict[n.identifier] = nblist

        niceprint = str(("\n".join("{}\t\t{}".format(k, v) for k, v in self.
            networkdict.items())) + "\n\n")
        return niceprint

    # remove the link between two nodes and return true or false if link don't
        exist.
    def remove_link(self, node1, node2):
        """

        :param node1:
        :param node2:
        :return:
        """

        if isinstance(node1, str):
            node1 = self.nodes[node1]
        if isinstance(node2, str):
            node2 = self.nodes[node2]

        if node1.has_edge_to(node2) and node2.has_edge_to(node1):
            node1.remove_edge(node2)
            node2.remove_edge(node1)
            self.nb_edges -= 1
            return True
        else:
            return False

    # Find all the cliques of k nodes in the network
    # We tried an other recursive manner (Bron-Kerbosch with pivot) but missed
        time to succeed
    def find_cliques(self, k):

        # main loop (recursive)
        def clique_loop(k, list):

            # Recursivity stop condition
            if k == 1:
                return list
            else:
                tmp_list = []
                for tuple in list:
                    for node1 in self.nodes.keys():
                        if node1 in tuple:
                            break
                        else:
                            for node2 in tuple:
                                hasLink = True

                                if not self.nodes[node1].has_edge_to(self.
                                    nodes[node2]):
                                    hasLink = False

                        if hasLink:
                            tmp_list.append(tuple + (node1,))

                return clique_loop(k - 1, tmp_list)

        # Here we call the main loop, the list argument contains a map object
        # nodelist == iterable containing all the nodes keys formatted: (x, )


        # http://www.secnetix.de/olli/Python/lambda_functions.hawk
        nodelist = map(lambda x: (x,), self.nodes.keys())
        lst = clique_loop(k, nodelist)
        ret = sorted(lst)
        ret = [ret for ret, _ in itertools.groupby(ret)]
        return ret
```

(b) *Finding Cliques*

The function to find cliques of n nodes in a network is in the class generic_network.py in listing 7. This function returns the list of cliques of size n. In the main program (listing 3), the function *remove_contained_cliques* remove the smaller cliques contained in the bigger one as requested. **The code seems to work, but the execution time is too long.** We are aware that this is not the optimal solution.

(c) *Evolving Network*

In the listing 3, the main program is executed and different functions are implemented. The function evolve takes a network and a number of time steps and randomly remove or add edges in the network.

Listing 2: main5.py

```python
from generic_network import GenericNetwork
import random
from random import randint
import matplotlib.pyplot as plt
from randomized_network import RandomizedNetwork
from motif_enrichment import MotifEnrichment


def remove_contained_cliques(res1, res2, res3):
    """

    :param clik3: cliques of 3 nodes
    :param clik4: cliques of 4 nodes
    :param clik5: cliques of 5 nodes
    :return: remove the smaller cliques contained in the big ones as requested
    """
    # If the clique of 4 is already in a clique of 5 -> remove
    for clique5 in res3:
        for clique4 in res2:
            if contains(clique5, clique4):
                res2.remove(clique4)
        # Same with size 3
        for clique3 in res1:
            if contains(clique5, clique3):
                res1.remove(clique3)

    for clique4 in res2:
        for clique3 in res1:
            if contains(clique4, clique3):
                res1.remove(clique3)

def contains(list1, list2):
    """
    http://thispointer.com/python-check-if-a-list-contains-all-the-elements-of
        -another-list/
    check if list1 contains all elements in list2

    :param list1:
    :param list2:
    :return: boolean value
    """
    result = all(elem in list1 for elem in list2)
    return bool(result)


def evolve(t, network, plot = None):
    """
    Randomly select two nodes and delete the edge if existing or add it
        otherwise
    :param t: number of time steps
    :param network: network class object
    :return:
    """

    def get_two_random_nodes(add):
        """
        :add: if "add" is true, we want to add an edge so the two nodes must
            not be connected
        :return: two different random nodes from the network
        """

        # Pick a node with a degree > 1
        node1 = network.get_node(random.sample(list(network.get_nodes()), 1)
            [0])
        node2 = network.get_node(random.sample(list(network.get_nodes()), 1)
            [0])

        while not node1.degree() > 1:
            node1 = network.get_node(random.sample(list(network.get_nodes()),
                1)[0])

        while not node2.degree() > 1:
            node2 = network.get_node(random.sample(list(network.get_nodes()),
                1)[0])

        # If we want to add an edge, the two nodes mustn't be connected. To
            avoid blockage
        # it is necessary to rechoose both nodes.
        if add:
            while node1.has_edge_to(node2) or node1 == node2:
                node1 = network.get_node(random.sample(list(network.get_nodes
                    ()), 1)[0])
                node2 = network.get_node(random.sample(list(network.get_nodes
                    ()), 1)[0])
        else:
            # if the node are note connected, take a random neighbour of node1
            while not node1.has_edge_to(node2) or node1 == node2:
                node1_list = node1.get_neighbours()
                node2 = network.get_node(node1_list[randint(0,len(node1_list)
                    -1)])

        return (node1, node2)

    # return cliques values for t = 100
    ret1 = []
    ret2 = []
    ret3 = []

    for _ in range(0, t):
        print("Evolution step: ", _)

        # 1 = Add or 0 = delete edge
        add = bool(random.getrandbits(1))

        # Get to nodes according to the decision to add or remove an edge
        nodes = get_two_random_nodes(add)

        if not add:
            network.remove_link(nodes[0], nodes[1])
        else:
            network.add_edge(nodes[0], nodes[1])

        # For t = 100 - plot each step.
        if t == 100:
            print("Calculating intermediate cliques...")
            res1 = network.find_cliques(3)
            res2 = network.find_cliques(4)
            res3 = network.find_cliques(5)
            remove_contained_cliques(res1, res2, res3)

            # Save the number of cliques of size 3, 4 and 5 after each step
            ret1.append(len(res1))
            ret2.append(len(res2))
            ret3.append(len(res3))

    # return the different clique values for all the 100 steps (empty if t !=
        100)
    return (ret1, ret2, ret3)


#
    ##################################################################################
# MAIN
#
    ##################################################################################



if __name__ == "__main__":

    print("Assignment 5 - Schmitt Schowing\n\n")

    # (b) - Read Network
    PATH = "../Data/sup53/rat_network.tsv"
    net = GenericNetwork()
    net.read_from_tsv(PATH)

    # # (c) - Count cliques
    # res1 = net.find_cliques(3)
    # res2 = net.find_cliques(4)
    # res3 = net.find_cliques(5)
    #
    # # Total number of cliques
    # print("\n\nNumber of cliques of 3 nodes: ", len(res1))
    # print("Number of cliques of 4 nodes: ", len(res2))
    # print("Number of cliques of 5 nodes: ", len(res3))
    #
    #
    # # # Do not count the cliques of smaller size that are contained in a
        larger
    # # # clique.
    # # remove_contained_cliques(res1, res2, res3)
    # #
    # #
    # # print("\n\nNumber of cliques of 3 nodes after cleaning: ", len(res1))
    # # print("Number of cliques of 4 nodes after cleaning: ", len(res2))
    # # print("Number of cliques of 5 nodes after cleaning: ", len(res3))
    #
    #
    #
    #
    #
    #
    # # 100 EVOLUTION - reset the network
    #
    # print("\n\n
        ----------------------------------------------------------------------------------"
    #       "\n                              Network Evolution"
    #       "\n
        ----------------------------------------------------------------------------------\n")
    #
```

```
#
# print("Start evolution 100 time steps.")
#
# evo100_net = GenericNetwork()
# evo100_net.read_from_tsv(PATH)
# evolution_data_100 = evolve(100, evo100_net)
#
# print("Evolution done. Counting cliques.")
#
# evo100_res1 = evo100_net.find_cliques(3)
# evo100_res2 = evo100_net.find_cliques(4)
# evo100_res3 = evo100_net.find_cliques(5)
#
# # remove_contained_cliques(evo100_res1, evo100_res2, evo100_res3)
#
# print("\n\nNumber of cliques of 3 nodes after 100 evolutions: ", len(
    evo100_res1))
# print("Number of cliques of 4 nodes after 100 evolutions: ", len(
    evo100_res2))
# print("Number of cliques of 5 nodes after 100 evolutions: ", len(
    evo100_res3))
#
# print("Plot Evolution Data")
#
# plt.plot(evolution_data_100[0], label='Cliques of size 3')
# plt.plot(evolution_data_100[1], label='Cliques of size 4')
# plt.plot(evolution_data_100[2], label='Cliques of size 5')
# plt.xlabel("Evolution")
# plt.ylabel("Number of cliques")
# plt.legend()
# plt.show()
#
#
#
# # Too damn long !
# # 1000 EVOLUTION - reset the network
#
#
# print("Reset Network")
# evo1000_net = GenericNetwork()
# evo1000_net.read_from_tsv(PATH)
#
# print("Start evolution 1000 time steps.")
# evolution_data_1000 = evolve(1000, evo1000_net)
#
# print("Counting cliques for the 1000 time evolved network")
# evo1000_res1 = evo1000_net.find_cliques(3)
# evo1000_res2 = evo1000_net.find_cliques(4)
# evo1000_res3 = evo1000_net.find_cliques(5)
#
# # remove_contained_cliques(evo1000_res1, evo1000_res2, evo1000_res3)
#
#
# print("\n\nNumber of cliques of 3 nodes after 1000 evolutions: ", len(
    evo1000_res1))
# print("Number of cliques of 4 nodes after 1000 evolutions: ", len(
    evo1000_res2))
# print("Number of cliques of 5 nodes after 1000 evolutions: ", len(
    evo1000_res3))



# print("\n\n
    ------------------------------------------------------------------------"
#       "\n                              Randomized network"
#       "\n
    ------------------------------------------------------------------------\n")
```

```
# print("Original Network ")
# rat_net = GenericNetwork()
# rat_net.read_from_tsv("../Data/sup53/rat_network.tsv")
#
# print("nb cliques 3: ", len(rat_net.find_cliques(3)))
#
#
# print("Randomized Network ")
# randomized_net = RandomizedNetwork(rat_net).get_randomized_network()
# print("nb cliques rand: ", len(randomized_net.find_cliques(3)))

#---------------------------------------------------------------------------
#    Motif Enrichment
#---------------------------------------------------------------------------

rat_net = GenericNetwork()
rat_net.read_from_tsv("../Data/sup53/rat_network.tsv")
print("Start Motif Enrichment")
enrich = MotifEnrichment(100, rat_net)
print("P-Values: ", enrich.pis)
```

(d) *Cliques in evolving networks.* Due to the execution time, this has been run on a minimized version of the rat network.

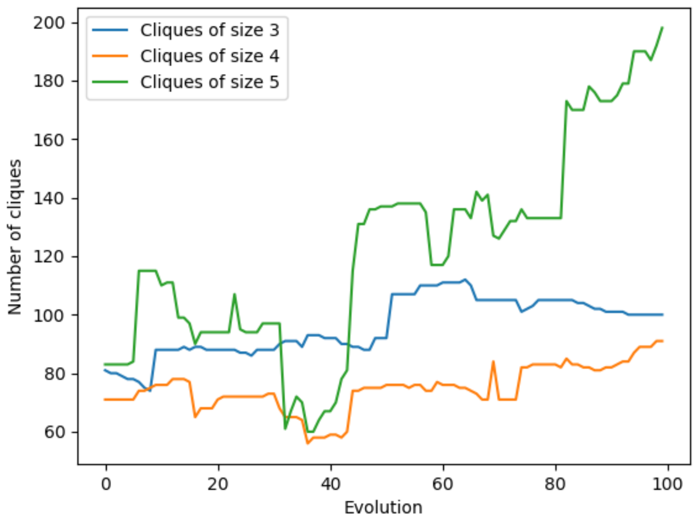

Figure 1: Evolution of the number of cliques during 100 randomization steps on a randomly minimized version of the rat network.

(e) *Randomizing Network* The class *randomized_network* builds a randomized network. Randomizing a network this way, keep its degree (number of edges) the same but change the topological structure of the graph. According to the Wikipedia definition of Degree-preserving randomization: "*Degree Preserving Randomization is a technique used in Network Science that aims to assess whether or not variations observed in a given graph could simply be an artifact of the graph's inherent structural properties rather than properties unique to the nodes, in an observed network.* "(https://en.wikipedia.org/wiki/Degree-preserving_randomization, Mai 2018). In other words, we use the randomization to verify whether the topology of the original graph is due to randomness or has a specific structure.

Listing 3: randomized_network.py

```
from node import Node
from generic_network import GenericNetwork
import random
from random import randint
from copy import deepcopy

def intersect(a, b):
    """ return the intersection of two lists """
    return list(set(a) & set(b))

class RandomizedNetwork:
    '''
    Randomize a given network
    '''
    def __init__(self, network):
        '''
        Initialization: deep copy the given network and randomize the copy
        '''

        self.rand_network = deepcopy(network)
        m = self.rand_network.nb_edges

        for _ in range(0, 2*m):
            #print("debug loop: ", _)

            # Randomly select 2 nodes with degree > 0
            def choose_random_node():
                return self.rand_network.get_node(random.sample(list(self.
                    rand_network.get_nodes()), 1)[0])

            def choose_two_nodes_condition():
                """
                Chose two nodes
                 - not the same nodes
                 - with more than 0 neighbour
                 -
                :return:
                """

                node01 = choose_random_node()
                node11 = choose_random_node()

                while not node01.degree() > 0 or node01.has_edge_to(node11):
                    node01 = choose_random_node()

                while (not node11.degree() > 0 and node11 != node01) or node11
                    .has_edge_to(node01):
                    node11 = choose_random_node()

                return node01, node11


            # CHOSE TWO RANDOM NODES - not identical, with degree > 1
            node01, node11 = choose_two_nodes_condition()

            # Randomly select a neighbour in the neighbours lists !!!!! not
                already connected to node 11 !!!!
            # #TODO proof to self link and duplicate link
            # #UPDATE self link ok, as the two nodes are different

            # Below: choose two node in the neighbour list of node01 and node
                11
            # the resulting edges should be switched without producing
                duplicate nodes

            def chose_rand_neighbour(list):
                return self.rand_network.get_node(list[randint(0, len(list) -
                    1)])

            # chose 02 and 12 a random neighbour of 01 and 11
            # In the next while loop, all random operation a executed again to
                 avoid blockage.

            node02 = chose_rand_neighbour(node01.get_neighbours())
            node12 = chose_rand_neighbour(node11.get_neighbours())

            # chose a random other neighbour !! Might cause blockage if
                neighbour are all connected to node 01
            while node01.has_edge_to(node12) or node11.has_edge_to(node02):
                node01, node11 = choose_two_nodes_condition()

                node02 = chose_rand_neighbour(node01.get_neighbours())
                node12 = chose_rand_neighbour(node11.get_neighbours())


            # e1 = (node01, node02) -> (node01, node12)
```

```
                # e2 = (node11, node12) -> (node 11, node02)

                node01.remove_edge(node02)
                node01.add_edge(node12)

                node11.remove_edge(node12)
                node11.add_edge(node02)



    def get_randomized_network(self):
        return self.rand_network
```

(f) *Examining motif enrichment*
Please be aware that the function to extract the cliques is still not optimal here.

Exercise 5.2: Annotations in Protein–Protein–Interaction Networks

(a) *Adding annotations to PPI-networks*

The listings for this exercise are at the end of the document.

(b) *Generating an overview*

For Chicken:

Table 1: Chicken network overview

Interactions in the network	300				
Proteins in the network	281	Protein without annotation	44	Percentage	15.6
Annotation per protein					
Smallest number	0	Average number	7.7	Biggest number	88
Protein per annotation					
Smallest number	1	Average number	1.55	Biggest number	27

For pig:

Table 2: Pig network overview

Interactions in the network	50				
Proteins in the network	51	Protein without annotation	13	Percentage	25.5
Annotation per protein					
Smallest number	0	Average number	5.5	Biggest number	40
Protein per annotation					
Smallest number	1	Average number	1.13	Biggest number	5

for Human:

Table 3: Human network overview

Interactions in the network	275472				
Proteins in the network	17087	Protein without annotation	2262	Percentage	13.2
Annotation per protein					
Smallest number	0	Average number	7.22	Biggest number	184
Protein per annotation					
Smallest number	1	Average number	10.6	Biggest number	1554

(c) *Examining the most/least common annotations*

Table 4: Function of the 5 most common GO identifiers of the human network.

GO id	**Quantity**	**Biological Process**
GO:0006351	1562	The cellular synthesis of RNA on a template of DNA.
GO:0045944	1029	Any process that activates or increases the frequency, rate or extent of transcription from an RNA polymerase II promoter.
GO:0007165	1010	Signal transduction
GO:0006357	960	Any process that modulates the frequency, rate or extent of transcription mediated by RNA polymerase II.
GO:0006355	765	Any process that modulates the frequency, rate or extent of cellular DNA-templated transcription

We can observe that these annotations concerns general process happening almost in every cell. This explains why they are the most common in opposition as the annotations in the table below, which concerns specific reaction or process concerning particular location on molecules.

Table 5: Function of the 5 least common GO identifiers of the human network

GO id	**Quantity**	**Biological Process**
GO:0000003	1	Reproduction
GO:0000011	1	Vacuole inheritance
GO:0000032	1	Cell wall mannoprotein biosynthetic process
GO:0000053	1	Argininosuccinate metabolic process
GO:0000097	1	Sulfur amino acid biosynthetic process

(d) **Investigating annotation enrichment** *The hypergeometric distribution can be used to find out if a given annotation is significantly overrepresented in interacting compared to non–interacting protein pairs. Implement a function that computes pA for every annotation A in a given annotated network.*

Table 6: Number and percentage of annotation with certain p-value

p-value	Number	Percentage
p <0.05	35	2.721%
p >0.5	43	3.343%
p >0.95	1243	96.656%

Table 7: Annotations with the **five lowest** *pA* and **five highest** *pA*

GO:ID	**pA**	**Nb Protein**	**Nb Interact. protein**	**Annotation**
GO:0009409	4.3907e-07	3	3	Response to cold
GO:0030154	1.7908e-05	7	4	Cell differentiation
GO:0007169	0.0002	3	2	Transmembrane receptor protein tyrosine kinase signaling pathway
GO:0000712	0.0002	3	2	Resolution of meiotic recombination intermediates
GO:0032570	0.0002	3	2	Response to progesterone
GO:0007049	1	10	0	Cell cycle
GO:0006096	1	9	0	Glycotic process
GO:0055114	1	9	0	Oxydation-reduction process
GO:0006457	1	9	0	Protein folding
GO:0006094	1	8	0	Gluconeogenesis

TODO

Are interacting proteins functionally more similar than non–interacting protein ?
No, the annotations of the interacting proteins...

Was this to be expected? Why (not)?

(e) **e) Investigating annotation combinations***s: Implement a function that computes if certain annotation combinations occur more frequently than expected. The function should take the combination size k and the number of random distributions r. Additionally, let n be the number of proteins in the network and nA the number of proteins with annotation*

Table 8: Number and percentage of combination with certain p-value

p-value	Number	Percentage
p <0.05	9794	49.25%
p >0.5	0	0.0%
p >0.95	1252	6.295%

Table 9: The m combinations with the smallest pc and the m combinations with the highest pc

Three smallest Pc:				
GO:IDs	Occurence	p-Value	Annotation 1	Annotation 2
'GO:0006897', 'GO:0006898'	1	0.0	endocytosis	Receptor-mediated endocytosis
'GO:0006898', 'GO:0021517'	2	0.0	Receptor-mediated endocytosis	Ventral spinal cord development
'GO:0008203', 'GO:0048813'	1	0.0	Cholesterole metabolism process	Dendrites morphogenesis
Three biggest Pc:				
'GO:0006355', 'GO:0006355'	1	0.71	Regulation of transcription DNA-templated	Regulation of transcription DNA-templated
'GO:0006351', 'GO:0050821'	1	0.71	Transcription DNA-templated	Protein stabilization
'GO:0006351', 'GO:0043066'	1	0.69	Transcription DNA-templated	Negative regulation of apoptotic process

TODO - COMMENT

(f) Listings:

Listing 4: task52_main.py

```
from UniprotReader import UniprotReader
from generic_network import GenericNetwork
from GOreader import GOReader

from annotated_network import AnnotatedNetwork

if __name__== "__main__":

    # ========================================================================
        #
    print("------------------------------------------------------------------"
          "\n                   Chicken Annotated Network"
          "\n-------------------------------------------------------------------\
             n")
    # ========================================================================
        #

    #TODO check version (mini)
    path_chicken_network = "../Data/sup51/chicken_network.tsv"
    chicken_network = GenericNetwork()
    chicken_network.read_from_tsv(path_chicken_network)

    path_chicken_uniprot = "../Data/sup51/chicken_uniprot.tsv"
    chicken_uniprot = UniprotReader(path_chicken_uniprot)

    path_chicken_ontology = "../Data/sup51/chicken_GO.gaf"
    chicken_GO = GOReader(path_chicken_ontology)

    Anet_chicken = AnnotatedNetwork(path_chicken_network,
        path_chicken_ontology, path_chicken_uniprot)

    # GENERATE OVERVIEW
    Anet_chicken.generate_overview()

    # # COMMON GO:IDs - Only requested for human
    # common_chicken_GOids = Anet_chicken.get_common_GOid(5)

    # ANNOTATION ENRICHMENT
    print("\n\nInvestigating annotation enrichment for the chicken network\n")
    Anet_chicken.annotation_enrichment(5)

    # ANNOTATION COMBINATION
    print("\n\nInvestigating annotation combinations for the chicken network\n
        ")
    Anet_chicken.annotation_combination(2, 100, 3)



    # ========================================================================
        #
    print("\n\n
        ------------------------------------------------------------------"
          "\n                     Pig Annotated Network"
          "\n-------------------------------------------------------------------\
             n")
    # ========================================================================
        #

    path_pig_network = "../Data/sup53/pig_network.tsv"
    pig_network = GenericNetwork()
```

```
pig_network.read_from_tsv(path_pig_network)

path_pig_uniprot = "../Data/sup53/pig_uniprot.tsv"
pig_uniprot = UniprotReader(path_pig_uniprot)

path_pig_ontology = "../Data/sup53/pig_GO.gaf"
pig_GO = GOReader(path_pig_ontology)

Anet_pig = AnnotatedNetwork(path_pig_network, path_pig_ontology,
    path_pig_uniprot)
Anet_pig.generate_overview()

## Only requested for human
## common_pig_GOids = Anet_pig.get_common_GOid(5)


# ==============================================================================
    #
print("\n\n
    ------------------------------------------------------------------"
     "\n                    Human Annotated Network"
     "\n-------------------------------------------------------------------\
        n")
# ==============================================================================
    #

path_human_network = "../Data/sup53/human_network.tsv"
human_network = GenericNetwork()
human_network.read_from_tsv(path_human_network)

path_human_uniprot = "../Data/sup53/human_uniprot.tsv"
human_uniprot = UniprotReader(path_human_uniprot)

path_human_ontology = "../Data/sup53/human_GO.gaf"
human_GO = GOReader(path_human_ontology)

Anet_human = AnnotatedNetwork(path_human_network, path_human_ontology,
    path_human_uniprot)
Anet_human.generate_overview()

common_human_GOids = Anet_human.get_common_GOid(5)
```

Listing 5: UniprotReader.py

```
from collections import defaultdict

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

        # structure containing ENTRY : [list of other names]
        self.mapping = defaultdict(set)

        # structure containing other names : ENTRY
        self.reverse_mapping = defaultdict(set)

        self.ENTRY = []
        self.ENTRY_NAME = []
        self.STATUS = []
        self.PROTEIN_NAMES = []
        self.GENE_NAMES = []
        self.ORGANISM = []

        # Read file
        content_start = False
        with open(filename, "r") as f:
            for line in f:
                if content_start:
                    # Process data
                    line = line.rstrip()
                    line_tab = line.split('\t')

                    self.ENTRY.append(line_tab[0])
                    self.ENTRY_NAME.append(line_tab[1])
                    self.STATUS.append(line_tab[2])
                    # Split the different names
                    self.PROTEIN_NAMES.append(line_tab[3].split('␣'))
                    self.GENE_NAMES.append(line_tab[4].split('␣'))
                    self.ORGANISM.append(line_tab[5])

                if line.startswith("Entry"):
                    content_start = True
                    continue

        # Construct mapping and reverse mapping
        for i in range(0, len(self.ENTRY)):
            for gene in self.GENE_NAMES[i]:
                self.mapping[self.ENTRY[i]].add(gene)
                self.reverse_mapping[gene].add(self.ENTRY[i])

    def get_uniprot_names_mapping(self):
        return self.mapping

    def get_names_uniprot_mapping(self):
        return self.reverse_mapping

    # Print mapping to file or to console
    # OPTIONAL
    def print_mapping(self):
        print("TODO")

    def print_reverse_mapping(self):
        print("TODO")
```

Listing 6: GOreader.py

```
from collections import defaultdict

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

        self.DB_NAME = []
        self.ACCESS_NUMBER = []
        self.ALTERNATIVE_NAME = []
        self.GO_IDENTIFIER = []
        self.ONTOLOGY_INDICATOR = []

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

                    # Skip all entries not belonging to biological process
                        ontology
                    if line_tab[8] != 'P':
                        continue

                    self.DB_NAME.append(line_tab[0])
                    self.ACCESS_NUMBER.append(line_tab[1]) # Protein name to
                        map
                    self.ALTERNATIVE_NAME.append(line_tab[2])
                    self.GO_IDENTIFIER.append(line_tab[4])
                    self.ONTOLOGY_INDICATOR.append(line_tab[8])

        # for i in range(0, len(self.DB_NAME)):
        #     print(self.DB_NAME[i], "\t", self.ACCESS_NUMBER[i], "\t", self.
            ALTERNATIVE_NAME[i], "\t", self.GO_IDENTIFIER[i], "\t", self.
            ONTOLOGY_INDICATOR[i])

        # Create a data structure with all information


        self.DATA = []
        for i in range(0, len(self.DB_NAME)):
            #TODO delete DATA if not used

            entry_line = [self.DB_NAME[i],
                          self.ACCESS_NUMBER[i], # real name in uniprot
                          self.ALTERNATIVE_NAME[i],
                          self.GO_IDENTIFIER[i],
                          self.ONTOLOGY_INDICATOR[i]]

            self.DATA.append(entry_line)

        # Create 4 dictionaries to map all GO ids of the GO file with the
            other data (prot names)
        # dict {GOID : access_number}
        # dict {GOID : alternative_name}
        # dict {alternative_name : GOID}
        # dict {access_number : GOID}

        self.goid_accessnb = defaultdict(set)
        self.accessnb_goid = defaultdict(set)
        self.alternativename_goid = defaultdict(set)
        self.goid_alternativename = defaultdict(set)
        # For readability

        idx_db_name = 0
        idx_access_nb = 1
        idx_alter_name = 2
        idx_go_id = 3
        idx_onto_id = 4

        # For every entry, fill the mappers.
        # The commented mappers are not used but could be useful
        for entry_line in self.DATA:
            #self.goid_accessnb[entry_line[idx_go_id]].add(entry_line[
                idx_access_nb])
            self.accessnb_goid[entry_line[idx_access_nb]].add(entry_line[
                idx_go_id])
            #self.alternativename_goid[entry_line[idx_alter_name]].add(
                entry_line[idx_go_id])
            #self.goid_alternativename[entry_line[idx_go_id]].add(entry_line[
                idx_alter_name])




        # print("Verify mapers")
        #
        # for key in self.goid_accessnb:
        #     print("\nKey: ", key)
        #     for elem in self.goid_accessnb[key]:
        #         print(elem)

    def get_GO_IDs(self, proteinID):
        """
        Get a protein name, returns all GO ids related to it
        :param proteinID:
        :return:
        """

        lst1 = []
        for prot in proteinID:
            tmp = self.accessnb_goid[prot]
            lst1.extend(list(tmp))

        return lst1


    def get_data(self):
        return self.DATA
```

Listing 7: annotated_network.py

```python
from UniprotReader import UniprotReader
from generic_network import GenericNetwork
from GOreader import GOReader
import numpy as np
from collections import defaultdict
import itertools
from itertools import combinations
import math


def nCr(n,r):
    """
    # https://stackoverflow.com/questions/4941753/is-there-a-math-ncr-function
        -in-python
    :param n: Total number of object in the set
    :param r: Number of object in the subset
    :return: Number of possible subset
    """
    return math.factorial(n) // math.factorial(r) // math.factorial(n-r)


class AnnotatedNetwork:

    def __init__(self, network_path, GO_path, uniprot_path):

        self.network = GenericNetwork()
        self.network.read_from_tsv(network_path)

        self.uniprot = UniprotReader(uniprot_path)
        self.GO = GOReader(GO_path)

        self.to_uniprot_mapper = self.uniprot.get_names_uniprot_mapping()

        #self.to_othername_mapper = self.uniprot.get_uniprot_names_mapping()

        # dict containing network node {network node id : go ids}
        self.net_go = defaultdict(list)

        # Mapping protein to GOs
        # {nodeid : [GO, GO, ...]}
        for id, node in self.network.nodes.items():

            # Convert the protein id
            uniprot_id = self.to_uniprot_mapper[id]

            # uniprot_id can contains 0, 1 or more names
            # map the protein names with the GO ids
            goids = self.GO.get_GO_IDs(uniprot_id)
            self.net_go[id] = goids

        # Reverse mapping GO to proteins(net)
        # {GO annot : [node, node, ...]}
        self.go_net = defaultdict(set)

        for node in self.net_go:
            list_annot = self.net_go[node]

            for annot in list_annot:
                self.go_net[annot].add(node)

        # Completing GO in the network and quantity
        # {GO : qty}
        self.go_qty = defaultdict(int)

        for key in self.go_net:
            self.go_qty[key] = len(self.go_net[key])

        # COMPUTE ANNOTATION QUANTITY OCCURRENCE
        # number of protein
        self.nb_prot = 0
        # number of protein without annotation
        self.nb_prot_wo_annotation = 0
        # {number of annotation : occurence}
        self.nb_annotqty_occurence = dict()
        # total annotation (Not unique, see total_annot_unique
        self. total_annot = 0

        # for every node
        for key in self.net_go:
            self.nb_prot += 1
            nb_annotation = len(self.net_go[key])

            self.total_annot += nb_annotation

            if nb_annotation == 0:
                self.nb_prot_wo_annotation += 1

            # increment quantity of annotation
            if nb_annotation in self.nb_annotqty_occurence:
                self.nb_annotqty_occurence[nb_annotation] += 1
            else:
                self.nb_annotqty_occurence[nb_annotation] = 1


        # PROTEIN PER ANNOTATION

        self.total_prot_per_annot = 0

        # dict {number of prot/annot : occurence}
        self.nb_prot_occurence = dict()

        # for every annotation in {GO : nodes}
        for annot in self.go_net:
            nb_prot = len(self.go_net[annot])
            self.total_prot_per_annot += nb_prot

            # increment quantity of annotation
            if nb_prot in self.nb_prot_occurence:
                self.nb_prot_occurence[nb_prot] += 1
            else:
                self.nb_prot_occurence[nb_prot] = 1
            #print annotation: proteins
            #print(annot, "\t\t", self.go_net[annot])




    def generate_overview(self):
        """
        Generate the overview of the network
        :return: nada
        """
        # Task 52
        print("\n---------Annotated Network Overview---------\n")
        print("Total protein in the network: ", len(self.network.nodes))
        print("Total interactions in the network: ", self.network.nb_edges)
        print("Total unique annotation: ", len(self.go_net))

        print("Nb_prot: ", self.nb_prot, "\t\tNb without annotation: ", self.
            nb_prot_wo_annotation, "\t\tPercentage: ",
              (self.nb_prot_wo_annotation / self.nb_prot) * 100)

        print("Smallest number of annotation: ", sorted(self.
            nb_annotqty_occurence)[0], "\t\tAverage␣number␣of␣annotation:␣",
              self.total_annot / self.nb_prot, "\t\tBiggest␣number␣of␣
                annotation:␣", sorted(self.nb_annotqty_occurence)[-1])

        print("Smallest␣number␣of␣protein␣per␣annotation:␣", sorted(self.
            nb_prot_occurence)[0], "\t\tAverage␣number␣of␣protein:␣",
              self.total_prot_per_annot / len(self.go_net), "\t\tBiggest␣
                number␣of␣protein:␣", sorted(self.nb_prot_occurence)[-1])

        print("\n\n")

    def get_common_GOid(self, n):
        """
        Return the n most common GO identifiers of the annotated network
        :param n: number of GO wanted
        :return: tuple of lists (n most common, n least common)
        """
        #sorted_go_qty = sorted(self.go_qty.items(), key=lambda x: x[1])

        # Table of sorted GO quantity (DESC) and sorted GO id (ASC)
        sorted_go_qty1 = [v[0] for v in sorted(self.go_qty.items(), key=lambda
             kv: (-kv[1], kv[0]))]

        # Table of sorted GO quantity (ASC) and sorted GO id (ASC)
        sorted_go_qty2 = [v[0] for v in sorted(self.go_qty.items(), key=lambda
             kv: (kv[1], kv[0]))]


        print("Most␣common␣GO␣ids")
        n_most_common = list(itertools.islice(sorted_go_qty1, n))

        for goid in n_most_common:
            print(goid, "\t", self.go_qty[goid])

        print("Least␣common␣GO␣ids")
        n_least_common = list(itertools.islice(sorted_go_qty2, n))

        for goid in n_least_common:
            print(goid, "\t", self.go_qty[goid])

        return (n_most_common, n_least_common)

    def annotation_enrichment(self, top):
        """

        :param top: number of top annotation probability
        :return: the n highest and lowest p(a)
        """

        # List of all possible protein pairs in the network
        protein_pairs = list(itertools.combinations(self.network.nodes, 2))
        # Number of possible pair
        N = len(protein_pairs)
        # Number of interacting protein pairs
        n = self.network.nb_edges

        # Annotation and interacting pairs {GO : [(prot1,prot2),(prot2,prot3)
             ,...]}
        self.annot_all_pairs = defaultdict(list)
        self.annot_interaction_pairs = defaultdict(list)
        self.annot_probability = defaultdict(float)


        ncr_Nn = nCr(N, n)

        # For each annotation in the network
        for A in self.go_net:
```

```python
            # For every possible pair in the network, check if both have annotation A
            # If they have both annotation A, check if the two proteins are interacting (connected in the network)
            for pair in protein_pairs:
                if A in self.net_go[pair[0]] and A in self.net_go[pair[1]]:
                    self.annot_all_pairs[A].append(pair)

                    # if pair 0 and pair 1 are interacting
                    if self.network.get_node(pair[0]).has_edge_to(self.network.get_node(pair[1])):
                        self.annot_interaction_pairs[A].append(pair)

            # Ka = number of protein pairs where both proteins have annotation A
            Ka = len(self.annot_all_pairs[A])

            # ka = number of interacting protein pairs where both proteins have annotation A
            ka = len(self.annot_interaction_pairs[A])

            N_minus_Ka = N - Ka

            # Trying to optimize here ! (not bad, can do better !)
            if ka == 0:
                self.annot_probability[A] = 1
                # print(A, "\t pA: ", 1)
                continue

            pA = 0
            for i in range(ka, min(Ka, n) + 1):
                nCr_Ka_i = nCr(Ka, i)
                nCr_N_minus_Ka_n_i = nCr(N_minus_Ka, n - i)

                # print("\nn = ", n,
                #       "\nN = ", N,
                #       "\nKa = ", Ka,
                #       "\nka = ", ka,
                #       "\ni = ", i,
                #       "\nnCr(Ka,i) = ", nCr_Ka_i,
                #       "\nmin(Ka, n) = ", min(Ka, n),
                #       "\nnCr(N-Ka, n-i) = ", nCr_N_minus_Ka_n_i, "\n")

                pA += (nCr_Ka_i * nCr_N_minus_Ka_n_i) / ncr_Nn

            self.annot_probability[A] = pA

        # The number and percentage of annotations A with pA < 0.05, pA > 0.5, pA > 0
        pa_005 = pa_05 = pa_095 = 0
        for A in self.annot_probability:
            if self.annot_probability[A] <= 0.05:
                pa_005 += 1
            if self.annot_probability[A] < 0.95:
                pa_05 += 1
            if self.annot_probability[A] >= 0.95:
                pa_095 += 1

        # Percentages
        tot_annot = len(self.go_net)
        pct_005 = pa_005 / tot_annot
        pct_05 = pa_05 / tot_annot
        pct_095 = pa_095 / tot_annot

        print("Number of annotation with pA < 0.05         : ", pa_005, "-> ", pct_005*100, "%")
        print("Number␣of␣annotation␣with␣pA␣>␣0.5␣&␣<␣0.95␣:␣", pa_05, "->␣",
            pct_05*100, "%")
        print("Number␣of␣annotation␣with␣pA␣>␣0.95␣␣␣␣␣␣␣␣␣:␣", pa_095, "->",
            pct_095*100, "%")
        print("\n")

        # The n annotations with the smallest pA and the n annotations with
            the highest pA.
        # If there are several annotations with the same pA, choose the ones
            that are associated
        # with more proteins first

        # Create a (GO, pA, Nb-prot) list for the later sort
        annot_prob_prot = []
        for A in self.annot_probability:
            annot_prob_prot.append((A, self.annot_probability[A], len(self.
                go_net[A]), len(self.annot_interaction_pairs[A])))

        # # All the Annotation A with their probabilities and number of
            protein
        # for e in annot_prob_prot:
        #     print(e)

        # gives [('GO-id', p(A), nb_protein), (..., ..., ...)] with P(a)
            ordered ASC
        sorted_probabilities_ASC = [(v[0], v[1], v[2], v[3]) for v in sorted(
            annot_prob_prot, key=lambda kv: (kv[1], kv[2]))]

        # gives [('GO-id', p(A), nb_protein), (..., ..., ...)] with P(a)
            ordered DSC
        sorted_probabilities_DSC = [(v[0], v[1], v[2], v[3]) for v in sorted(
            annot_prob_prot, key=lambda kv: (-kv[1], -kv[2]))]

        # Take the "top" firsts
        smallest_prob = list(itertools.islice(sorted_probabilities_ASC, top))
        biggest_prob = list(itertools.islice(sorted_probabilities_DSC, top))

        print("\n\n(GO:id␣␣|␣␣pA␣␣|␣␣Nb␣Protein␣|␣Nb␣Interact.␣Protein)\n")
        print("Five␣smallest␣Pa:␣\n")
        for e in smallest_prob:
            print(e)

        print("\nFive␣biggest␣Pa:␣\n")
        for e in biggest_prob:
            print(e)

    def annotation_combination(self, k, r, m):
        """

        :param k: combination size
        :param r: number of random distribution
        :param m: m combinations with the smallest pc and the m annotations
            with the highest pc
        :return:
        """

        annotation_probability = defaultdict(float)

        # number of protein in the network
        n = self.network.size()

        # number of protein with annotation A
        # len(self.go_net[A]

        # For each annotation, compute its probability
        # go_net -> {GO_id : [prot1, prot2, ...]}
        for A in self.go_net:
```

```
        annotation_probability[A] = len(self.go_net[A]) / n

    # Generate a list of all annotation combinations of size k that occur
        in the annotated network
    # https://stackoverflow.com/questions/22799053/combinations-of-
        elements-of-different-tuples-in-the-list
    #all_combinations = list(combinations(self.go_net, k))

    # Combination set contains all combination of k annotation contained
        in the network
    combination_dict = defaultdict(list)
    for node in self.net_go:
        if len(self.net_go[node]) < k:
            continue

        tmp_combinations = combinations(self.net_go[node], k)

        # For each k-combination for this node
        for combination in tmp_combinations:
            # The combination are sorted in order to avoid adding (a,b)
                and (b,a)
            s_combination = tuple(sorted(combination))
            if s_combination in combination_dict:
                combination_dict[s_combination][0] += 1
            else:
                combination_dict[s_combination].append(1)


    # for A in annotation_probability:
    #     print(A, "\t", len(self.go_net[A]), "\t\t",
        annotation_probability[A])

    # For each combination (C1, C2, ...) in the network...
    for C in combination_dict:
        # Cn = how often this combination occurs in the network
         #nc = combination_dict[C]

        Pe_c = annotation_probability[C[0]] * annotation_probability[C[1]]
        combination_dict[C].append(Pe_c)
    # DEBUG - infos
    # for key in combination_dict:
    #     print(key, ":\t", combination_dict[key])

    for key in combination_dict:
        #probability_list = [combination_dict[key][1]] * n
        prob = combination_dict[key][1]

        # nr = number of random sample in which C occurs at least as much
            as in the original network
        nr = 0
        for _ in range(0, r):
            random_list = np.random.choice([0, 1], size=n, p=[1 - prob,
                prob])

            # C in the actual network appears combination_dict[key][0]
                times
            # number of occurence in random network
            nb_occ = np.count_nonzero(random_list)


            if nb_occ >= combination_dict[key][0]:
                nr += 1
        # Calculating and adding the probability pc to the dict "
            combination_dict"
        pc = nr / r
        combination_dict[key].append(pc)
    # IMPORTANT - structure of combination dict.
    # combination_dict = (c1, c2) : [nb_occ, expect_prob, rand_prob]


    pc_0001 = pc_005 = pc_05 = 0
    nb_C = len(combination_dict)
    for c in combination_dict:
        pc = combination_dict[c][2]
        if pc < 0.001:
            pc_0001 += 1
        elif pc < 0.005:
            pc_005 += 1
        elif pc > 0.05:
            pc_05 += 1

    # percentages
    pct_0001 = pc_0001/nb_C
    pct_005 = pc_005/nb_C
    pct_05 = pc_05/nb_C

    print("pc < 0.001 : ", pc_0001, "-> ", pct_0001 * 100, "%")
    print("pc < 0.005 : ", pc_005, "-> ", pct_005 * 100, "%")
    print("pc < 0.05  : ", pc_05, "-> ", pct_05 * 100, "%")

    combination_dict_sorted_ASC = sorted(combination_dict.items(), key=
        lambda e: e[1][2])
    combination_dict_sorted_DSC = sorted(combination_dict.items(), key=
        lambda e: -e[1][2])

    # Take the "m" firsts
    smallest_prob = list(itertools.islice(combination_dict_sorted_ASC, m))
    biggest_prob = list(itertools.islice(combination_dict_sorted_DSC, m))

    print("\n\n(GO: ids  |  Occurence in the data  |  Pe(C) | Pc)\n")
    print("Three smallest Pc: \n")
    for e in smallest_prob:
        print(e)

    print("\nThree biggest Pc: \n")
    for e in biggest_prob:
        print(e)
```