Assignment 2

Ben Napier

February 17, 2022

Note that the networks in this exercise are all undirected.

We recall some definitions and introduce some new ones.

Let d_{ij} be the distance (the length of the shortest path) between vertices i to j.

Then the closeness centrality of vertex j is $\mathrm{CC}(j) = \frac{1}{\sum_i d_{ij}}$.

The nearness centrality of vertex j is $\mathrm{NC}(j) = \sum_i \frac{1}{d_{ij}}$. In both these definitions, the sums are over all vertices i, $i \neq j$, in the network.

The degree centrality of vertex j is simply its degree (the number of neighbours it has) and is denoted $\mathrm{DC}(j)$.

The adjacency centrality of vertex j is $\mathrm{AC}(j) = \frac{1}{d_j} \sum_i \frac{d_j - d_i}{d_j + d_i}$ where the sum is over all vertices i that are adjacent to j and d_i denotes the degree of a vertex. (So $\mathrm{DC}(j)$ and d_j are different notations for the same measure.)

1 Question 1

1. *[5 marks] Calculate the values of the four centrality measures defined above on each vertex in the network below. (The diagram and the dictionary are two representations of the same network.) Present your answer as four lists — one for each centrality measure — that gives the vertices and the calculated values ordered by those values.*

[1]:
```
network = {1: [4],
           2: [4],
           3: [4],
           4: [1, 2, 3, 5, 6],
           5: [4],
           6: [4, 7, 8, 9, 10, 11],
           7: [6, 8, 11],
           8: [6, 7, 9, 11],
           9: [6, 8, 10],
           10: [6, 9, 11, 12],
           11: [6, 7, 8, 10],
           12: [10]}
```

First, we import the required libraries.

[2]:
```python
from typing import Dict
from IPython.display import clear_output
import matplotlib.pyplot as plt
import random
from pprint import pprint


def print_status_bar(progress: float, block_count: int = 10) -> None:
    clear_output(wait=True)
    dark_string = " " * round(progress * block_count)
    light_string = " " * (block_count - len(dark_string))
    print(f"[{dark_string}{light_string}]")


def round_sig(x: float, sig: int = 3) -> str:
    return "%s" % float(f"%.{sig}g" % x)


def format_number(x: float, sig: int = 3) -> str:
    out = round_sig(x, sig)
    if x >= 0:
        out = " " + out
    return out
```

Below find the code for a breadth-first search function and a function to compute the distance matrix from an adjacency list.

[3]:
```python
def bfs(g: Dict[int, list[int]], node: int) -> Dict[int, int]:
    vstd = {node: 0}
    q = [node]
    while q:
        v = q.pop(0)
        for nghbr in g[v]:
            if nghbr not in vstd:
                vstd[nghbr] = vstd[v] + 1
                q.append(nghbr)
    return vstd


def compute_dst_mat(g: Dict[int, list[int]]) -> Dict[int, Dict[int, int]]:
    dst_mat = {}
    for v in g:
        dst_mat[v] = bfs(g, v)
    return dst_mat
```

We compute the distance matrix for our network.

[4]:
```python
network_dst_mat = compute_dst_mat(network)
```

We now define functions for the four measures.

```
[5]: def cc(distance_matrix: Dict[int, Dict[int, int]], node: int) -> float:
         total = 0
         for other_node in distance_matrix:
             total += distance_matrix[node][other_node]
         return 1 / total


     def nc(distance_matrix: Dict[int, Dict[int, int]], node: int) -> float:
         total = 0
         for other_node in distance_matrix:
             if other_node == node:
                 continue
             total += 1 / distance_matrix[node][other_node]
         return total


     def dc(network: Dict[int, list[int]], node: int) -> int:
         return len(network[node])


     def ac(network: Dict[int, list[int]], node: int) -> float:
         total = 0
         for neighbour in network[node]:
             total += (dc(network, node) - dc(network, neighbour)) / (dc(network,␣
      ↪node) + dc(network, neighbour))
         total /= len(network[node])
         return total
```

We now calculate the requested measures.

```
[6]: measures = dict()

     for measure_name in ["cc", "nc", "dc", "ac"]:
         measures[measure_name] = []
     for node in network:
         measures["cc"].append((node, cc(network_dst_mat, node)))
         measures["nc"].append((node, nc(network_dst_mat, node)))
         measures["dc"].append((node, dc(network, node)))
         measures["ac"].append((node, ac(network, node)))
     for measure_name in measures:
         measures[measure_name] = sorted(measures[measure_name], key=lambda k: k[1],␣
      ↪reverse=True)
```

And finally print them in a nice way.

[7]:
```
column_width = 12
print("_" * ((5 * 4) + (column_width * 4)))
print(
    f"{'CC'.ljust(5 + column_width)}{'NC'.ljust(5 + column_width)}{'DC'.ljust(5␣
 ↪+ column_width)}{'AC'.ljust(5 + column_width)}")
print("_" * ((5 * 4) + (column_width * 4)))
for cc_tuple, nc_tuple, dc_tuple, ac_tuple in zip(measures["cc"],␣
 ↪measures["nc"], measures["dc"], measures["ac"]):
    cc_v = format_number(cc_tuple[1])
    nc_v = format_number(nc_tuple[1])
    dc_v = format_number(dc_tuple[1])
    ac_v = format_number(ac_tuple[1])
    print(
        f"{str(cc_tuple[0]).ljust(3)}: {str(cc_v).
 ↪ljust(column_width)}{str(nc_tuple[0]).ljust(3)}: {str(nc_v).
 ↪ljust(column_width)}{str(dc_tuple[0]).ljust(3)}: {str(dc_v).
 ↪ljust(column_width)}{str(ac_tuple[0]).ljust(3)}: {str(ac_v).
 ↪ljust(column_width)}")
print("_" * ((5 * 4) + (column_width * 4)))
```

```
____________________________________________________________________
CC                NC                DC                AC
____________________________________________________________________
6  : 0.0625       6  : 8.5          6  : 6.0          4  : 0.515
4  : 0.0556       4  : 7.83         4  : 5.0          6  : 0.226
10 : 0.0455       10 : 6.83         8  : 4.0          10 : 0.136
11 : 0.0455       11 : 6.83         10 : 4.0          8  : 0.0214
8  : 0.0435       8  : 6.67         11 : 4.0          11 : -0.0143
9  : 0.0435       9  : 6.33         7  : 3.0          7  : -0.206
7  : 0.0417       7  : 6.17         9  : 3.0          9  : -0.206
1  : 0.0357       1  : 4.92         1  : 1.0          12 : -0.6
2  : 0.0357       2  : 4.92         2  : 1.0          1  : -0.667
3  : 0.0357       3  : 4.92         3  : 1.0          2  : -0.667
5  : 0.0357       5  : 4.92         5  : 1.0          3  : -0.667
12 : 0.0312       12 : 4.5          12 : 1.0          5  : -0.667
____________________________________________________________________
```

2 Question 2

2. *[20 marks] Obtain the three datasets in topic3networks.zip (under Topic 3 on Learn Ultra, see the descriptions below). Load these networks. Again, they are all undirected. We wish also to work with connected graphs so find the largest connected component of each and discard other vertices. For each dataset, for each of the four centrality measures, list, in order, the 20 vertices with the highest values of that measure (include more if the values are tied). Comment on whether you think, based on what you have found, that nearness centrality is a good alternative to closeness centrality and that adjacency centrality is a good alternative to degree centrality. The datasets:*

- *london_transport_raw_edges.txt: The network is of London rail and underground stations that are linked if they are adjacent on some line. The second and third items on each line in the file are a pair of nodes that are joined by an edge (the first item describes how they are linked and can be ignored for this exercise).*
- *Roget.txt: This is a network of words that are linked if they appear together in a thesaurus. At the start of the file is a list of words (the nodes) and their numeric identifiers. Then there are lists (one per line) of words that appear together in the thesaurus. There should be an edge between any pair of nodes that appear in the same list. For example, the list 3 4 323 325 implies the existence of six edges: (3,4), (3, 323), (3, 325), (4, 323), (4, 325), (323, 325)*
- *CCSB-Y2H.txt: The network is of interactions amongst proteins in yeast (living cells can be considered as complex webs of macromolecular interactions known as interactome networks). The first two items on each line are a pair of nodes joined by an edge (the rest of the line can be ignored).*

First we load the three graphs as adjacency lists.

[8]:
```
transport = dict()
with open("london_transport_raw_edges.txt") as g:
    for l in g:
        data = l.split(" ")
        if data[1] not in transport:
            transport[data[1]] = set()
        if data[2][:-1] not in transport:
            transport[data[2][:-1]] = set()

        transport[data[1]].add(data[2][:-1])
        transport[data[2][:-1]].add(data[1])

print(f"Imported transport with {len(transport)} nodes.")
```

```
Imported transport with 369 nodes.
```

[9]:
```
thesaurus = dict()
thesaurus_names = dict()
with open("Roget.txt") as g:
    for ln, l in enumerate(g):
        if ln == 0 or ln == 1023:
            continue
        data = l.split(" ")
        if ln < 1023:
            thesaurus[int(data[0])] = set()
            thesaurus_names[int(data[0])] = data[1][:-1].replace("\"", "")
        if ln > 1023:
            for node in data:
                for u in data:
                    if node == u:
                        continue
                    thesaurus[int(node)].add(int(u))
```

```
                        thesaurus[int(u)].add(int(node))

print(f"Imported thesaurus with {len(thesaurus)} nodes.")
```

```
Imported thesaurus with 1022 nodes.
```

[10]:
```
proteins = dict()
with open("CCSB-Y2H.txt") as g:
    for ln, l in enumerate(g):
        if ln == 0:
            continue
        data = l.split("\t")
        if data[0] not in proteins:
            proteins[data[0]] = set()
        if data[1] not in proteins:
            proteins[data[1]] = set()
        proteins[data[0]].add(data[1])
        proteins[data[1]].add(data[0])

print(f"Imported proteins with {len(proteins)} nodes.")
```

```
Imported proteins with 1278 nodes.
```

We reuse the connected components code from the first assignment.

[11]:
```
def get_ccs(network: Dict[int, list[int]]) -> list[list[int]]:
    ccs = []
    nodes = set(list(network.keys()))
    while nodes:
        clear_output(wait=True)
        print(f"Current number of components: {len(ccs)}")
        print(f"Unvisited nodes remaining: {len(nodes)}")
        node = nodes.pop()
        node_reach = bfs(network, node)
        ccs.append(node_reach.copy())
        nodes = nodes.difference(node_reach)
    return ccs


def get_largest_vertex_set(vertex_sets: list[list[int]]) -> list[int]:
    return max(vertex_sets, key=lambda k: len(k))


def get_induced_subgraph(network: Dict[int, list[int]], vertex_set: list[int])␣
 ↪-> Dict[int, list[int]]:
    induced_network = {}
    for vertex in vertex_set:
        induced_network[vertex] = network[vertex]
    return induced_network
```

[12]:
```
transport_max = get_induced_subgraph(transport,␣
 ↪get_largest_vertex_set(get_ccs(transport)))
clear_output(wait=True)
print(f"Largest conected component in transport has {len(transport_max)} nodes.
 ↪")
```

```
Largest conected component in transport has 369 nodes.
```

[13]:
```
thesaurus_max = get_induced_subgraph(thesaurus,␣
 ↪get_largest_vertex_set(get_ccs(thesaurus)))
clear_output(wait=True)
print(f"Largest conected component in thesaurus has {len(thesaurus_max)} nodes.
 ↪")
```

```
Largest conected component in thesaurus has 994 nodes.
```

[14]:
```
proteins_max = get_induced_subgraph(proteins,␣
 ↪get_largest_vertex_set(get_ccs(proteins)))
clear_output(wait=True)
print(f"Largest conected component in proteins has {len(proteins_max)} nodes.")
```

```
Largest conected component in proteins has 964 nodes.
```

We calculate the centrality measures of each vertex for all three datasets.

[15]:
```
centrality_measures = dict()

for dataset in [
    {"name": "transport", "data": transport_max,
     "dst_mat": compute_dst_mat(transport_max)},
    {"name": "thesaurus", "data": thesaurus_max,
     "dst_mat": compute_dst_mat(thesaurus_max)},
    {"name": "proteins", "data": proteins_max,
     "dst_mat": compute_dst_mat(proteins_max)}
]:
    centrality_measures[dataset["name"]] = dict()
    for node in dataset["data"]:
        centrality_measures[dataset["name"]][node] = dict()
        centrality_measures[dataset["name"]][node]["cc"] = cc(
            dataset["dst_mat"], node)
        centrality_measures[dataset["name"]][node]["nc"] = nc(
            dataset["dst_mat"], node)
        centrality_measures[dataset["name"]][node]["dc"] = dc(dataset["data"],␣
 ↪node)
        centrality_measures[dataset["name"]][node]["ac"] = ac(
            dataset["data"], node)
```

We then list the top 20 vertices (with ties) sorted each centrality measure for each dataset.

[16]:
```
sorted_centrality_measures = dict()
for dataset_name in ["transport", "thesaurus", "proteins"]:
    sorted_centrality_measures[dataset_name] = dict()
    for measure in ["cc", "nc", "dc", "ac"]:
        sorted_centrality_measures[dataset_name][measure] = [
            (v[0], v[1].get(measure)) for v in␣
 ↪centrality_measures[dataset_name].items()]
        sorted_centrality_measures[dataset_name][measure] = sorted(
            sorted_centrality_measures[dataset_name][measure], key=lambda tup:␣
 ↪tup[1], reverse=True)

top_20_vertices = dict()
for dataset_name in ["transport", "thesaurus", "proteins"]:
    top_20_vertices[dataset_name] = {}
    for measure in ["cc", "nc", "dc", "ac"]:
        top_20_vertices[dataset_name][measure] = [v
                                                   for v in␣
 ↪sorted_centrality_measures[dataset_name][measure][:20]]
        tie_measure_value =␣
 ↪sorted_centrality_measures[dataset_name][measure][19][1]
        ties = [v for i, v in enumerate(sorted_centrality_measures[dataset_name]
                                        [measure]) if v[1] == tie_measure_value␣
 ↪and i >= 20]
        top_20_vertices[dataset_name][measure] += ties
```

[17]:
```
def print_measures(measures, thesaurus_lookup=False):
    value_width = 10
    vertex_width = 20
    col_sep = 5
    column_width = value_width + vertex_width + 2
    print("_" * ((column_width * 2) + col_sep))
    print(
        f"{'CC'.center(column_width)}{' ' * col_sep}{'NC'.
 ↪center(column_width)}")
    print("_" * ((column_width * 2) + col_sep))
    for i in range(max(len(measures["cc"]), len(measures["nc"]))):
        if i >= len(measures["cc"]):
            cc_vertex = ""
            cc_value = ""
        else:
            cc_tuple = measures["cc"][i]
            if thesaurus_lookup:
                cc_vertex = str(thesaurus_names[cc_tuple[0]])[:vertex_width].
 ↪ljust(vertex_width)
            else:
                cc_vertex = str(cc_tuple[0])[:vertex_width].ljust(vertex_width)
            cc_value = format_number(cc_tuple[1]).ljust(value_width)
        if i >= len(measures["nc"]):
            nc_vertex = ""
            nc_value = ""
        else:
            nc_tuple = measures["nc"][i]
            if thesaurus_lookup:
                nc_vertex = str(thesaurus_names[nc_tuple[0]])[:vertex_width].
↪ljust(vertex_width)
            else:
                nc_vertex = str(nc_tuple[0])[:vertex_width].ljust(vertex_width)
            nc_value = format_number(nc_tuple[1]).ljust(value_width)

        print(f"{cc_vertex}: {cc_value}{' ' * col_sep}{nc_vertex}: {nc_value}")
    print("_" * ((column_width * 2) + col_sep))
    print()
    print("_" * ((column_width * 2) + col_sep))
    print(
        f"{'DC'.center(column_width)}{' ' * col_sep}{'AC'.
↪center(column_width)}")
    print("_" * ((column_width * 2) + col_sep))
    for i in range(max(len(measures["dc"]), len(measures["ac"]))):
        if i >= len(measures["dc"]):
            dc_vertex = ""
            dc_value = ""
        else:
            dc_tuple = measures["dc"][i]
            if thesaurus_lookup:
                dc_vertex = str(thesaurus_names[dc_tuple[0]])[:vertex_width].
↪ljust(vertex_width)
            else:
                dc_vertex = str(dc_tuple[0])[:vertex_width].ljust(vertex_width)
            dc_value = format_number(dc_tuple[1]).ljust(value_width)
        if i >= len(measures["ac"]):
            ac_vertex = ""
            ac_value = ""
        else:
            ac_tuple = measures["ac"][i]
            if thesaurus_lookup:
                ac_vertex = str(thesaurus_names[ac_tuple[0]])[:vertex_width].
↪ljust(vertex_width)
            else:
                ac_vertex = str(ac_tuple[0])[:vertex_width].ljust(vertex_width)
            ac_value = format_number(ac_tuple[1]).ljust(value_width)

        print(f"{dc_vertex}: {dc_value}{' ' * col_sep}{ac_vertex}: {ac_value}")

    print("_" * ((column_width * 2) + col_sep))
    print("\n" * 3)
```

2.1 Initial measure investigation

We first look at the measures on some simple graphs..

[18]:
```
n = 1000

print_status_bar(0, block_count=80)

k = dict()
for i in range(n):
    k[i] = set(range(n)).difference({i})

print_status_bar(0.25, block_count=80)

k_plus_e = {-1: {0}}
for i in range(n):
    k_plus_e[i] = set(range(n)).difference({i})

print_status_bar(0.5, block_count=80)

star_network = {-1: set(range(n))}
for i in range(n):
    star_network[i] = {-1}

print_status_bar(0.75, block_count=80)
c = dict()
for i in range(n):
    c[i] = {(i + 1) % n, (i - 1) % n}

print_status_bar(0, block_count=80)
clear_output(wait=True)

column_width_large = 30
column_width_small = 10
column_sep = 5

print("_" * ((column_width_small * 2) + column_width_large + (2 * column_sep)))
print(f"{'Graph'.ljust(column_width_large)}{' ' * column_sep}", end='')
print(f"{'DC'.ljust(column_width_small)}{' ' * column_sep}", end='')
print(f"{'AC'.ljust(column_width_small)}{' ' * column_sep}", end='')
print()
print("_" * ((column_width_small * 2) + column_width_large + (2 * column_sep)))
print(f"{'Complete graph'.ljust(column_width_large)}{' ' * column_sep}", end='')
print(f"{format_number(dc(k, 0)).ljust(column_width_small)}{' ' * column_sep}",␣
↪end='')
```

```
print(f"{format_number(ac(k, 0)).ljust(column_width_small)}{' ' * column_sep}",␣
↪end='')
print()
print(f"{'Complete graph + e (endpoint)'.ljust(column_width_large)}{' ' *␣
↪column_sep}", end='')
print(f"{format_number(dc(k_plus_e, -1)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print(f"{format_number(ac(k_plus_e, -1)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print()
print(f"{'Star graph (center)'.ljust(column_width_large)}{' ' * column_sep}",␣
↪end='')
print(f"{format_number(dc(star_network, -1)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print(f"{format_number(ac(star_network, -1)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print()
print(f"{'Star graph (end-point)'.ljust(column_width_large)}{' ' *␣
↪column_sep}", end='')
print(f"{format_number(dc(star_network, 0)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print(f"{format_number(ac(star_network, 0)).ljust(column_width_small)}{' ' *␣
↪column_sep}", end='')
print()
print(f"{'Cycle graph'.ljust(column_width_large)}{' ' * column_sep}", end='')
print(f"{format_number(dc(c, 0)).ljust(column_width_small)}{' ' * column_sep}",␣
↪end='')
print(f"{format_number(ac(c, 0)).ljust(column_width_small)}{' ' * column_sep}",␣
↪end='')
print()
print("_" * ((column_width_small * 2) + column_width_large + (2 * column_sep)))
```

```
____________________________________________________________
Graph                              DC              AC
____________________________________________________________
Complete graph                      999.0           0.0
Complete graph + e (endpoint)       1.0            -0.998
Star graph (center)                 1000.0          0.998
Star graph (end-point)              1.0            -0.998
Cycle graph                         2.0             0.0
____________________________________________________________
```

We now look at the top 20 nodes (including ties) for each of the datasets and measures.

[19]:
```
print_measures(top_20_vertices["transport"])
```

```
_____________________________________________________________________
               CC                                    NC
```

```
----------------------------------------------------------------------
greenpark            :  0.000307      greenpark            :  59.5
westminster          :  0.000299      bank                 :  59.3
bondstreet           :  0.000299      kingscrossstpancras  :  58.7
kingscrossstpancras  :  0.000299      bakerstreet          :  58.3
oxfordcircus         :  0.000298      oxfordcircus         :  57.8
bank                 :  0.000295      waterloo             :  57.8
waterloo             :  0.000295      bondstreet           :  57.3
bakerstreet          :  0.000294      westminster          :  56.3
euston               :  0.000291      euston               :  55.7
victoria             :  0.00029       liverpoolstreet      :  55.2
farringdon           :  0.00029       shadwell             :  54.3
angel                :  0.00029       moorgate             :  54.2
hydeparkcorner       :  0.000288      highbury&islington   :  53.7
moorgate             :  0.000286      warrenstreet         :  53.3
barbican             :  0.000285      finchleyroad         :  53.3
oldstreet            :  0.000285      victoria             :  53.0
warrenstreet         :  0.000284      embankment           :  53.0
liverpoolstreet      :  0.000284      piccadillycircus     :  52.7
highbury&islington   :  0.000284      tottenhamcourtroad   :  52.4
eustonsquare         :  0.000284      regentspark          :  52.3
piccadillycircus     :  0.000284      :
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
              DC                                    AC
----------------------------------------------------------------------
kingscrossstpancras  :  7.0           paddington           :  0.48
bakerstreet          :  7.0           stratford            :  0.475
stratford            :  7.0           kingscrossstpancras  :  0.46
oxfordcircus         :  6.0           bakerstreet          :  0.453
greenpark            :  6.0           canningtown          :  0.417
paddington           :  6.0           blackhorseroad       :  0.4
waterloo             :  6.0           stockwell            :  0.4
bank                 :  6.0           chalfont&latimer     :  0.4
earlscourt           :  6.0           willesdenjunction    :  0.365
westham              :  6.0           earlscourt           :  0.339
canningtown          :  6.0           westham              :  0.321
euston               :  5.0           surreyquays          :  0.317
willesdenjunction    :  5.0           shadwell             :  0.309
liverpoolstreet      :  5.0           finchleycentral      :  0.3
shadwell             :  5.0           sydenham             :  0.3
turnhamgreen         :  5.0           turnhamgreen         :  0.294
camdentown           :  4.0           waterloo             :  0.289
highbury&islington   :  4.0           nottinghillgate      :  0.286
tottenhamcourtroad   :  4.0           holborn              :  0.25
piccadillycircus     :  4.0           finsburypark         :  0.25
bondstreet           :  4.0           wembleypark          :  0.25
```

```
holborn             :  4.0          westhampstead       :  0.25
finsburypark        :  4.0          :
shepherdsbush       :  4.0          :
leicestersquare     :  4.0          :
westminster         :  4.0          :
victoria            :  4.0          :
moorgate            :  4.0          :
embankment          :  4.0          :
finchleyroad        :  4.0          :
nottinghillgate     :  4.0          :
westbrompton        :  4.0          :
wembleypark         :  4.0          :
westhampstead       :  4.0          :
londonbridge        :  4.0          :
blackhorseroad      :  4.0          :
stockwell           :  4.0          :
whitechapel         :  4.0          :
mileend             :  4.0          :
actontown           :  4.0          :
canadawater         :  4.0          :
surreyquays         :  4.0          :
canarywharf         :  4.0          :
poplar              :  4.0          :
----------------------------------------------------------------------
```

[20]:
```python
print_measures(top_20_vertices["thesaurus"], thesaurus_lookup=True)
```

```
----------------------------------------------------------------------
               CC                                    NC

----------------------------------------------------------------------
inutility           :  0.000496     inutility           :  540.0
store               :  0.000491     neglect             :  534.0
neglect             :  0.00049      deterioration       :  533.0
deterioration       :  0.00049      truth               :  530.0
truth               :  0.000489     store               :  527.0
unimportance        :  0.000489     unimportance        :  526.0
unconformity        :  0.000488     indication          :  525.0
support             :  0.000487     support             :  524.0
indication          :  0.000484     inactivity          :  523.0
inactivity          :  0.000484     activity            :  523.0
deception           :  0.000482     deception           :  523.0
activity            :  0.000481     aid                 :  521.0
aid                 :  0.000481     unconformity        :  521.0
restraint           :  0.000481     restraint           :  516.0
```

```
care                 :  0.00048       care                 :  516.0
skill                :  0.000478      information          :  515.0
information          :  0.000477      obstinacy            :  515.0
preparation          :  0.000477      skill                :  514.0
plan                 :  0.000475      pleasurableness      :  514.0
hindrance            :  0.000475      uncertainty          :  513.0
---------------------------------------------------------------------

---------------------------------------------------------------------
               DC                                    AC

---------------------------------------------------------------------
inutility            :  145.0         indication           :  0.475
deterioration        :  136.0         deterioration        :  0.449
neglect              :  135.0         inutility            :  0.446
truth                :  128.0         store                :  0.444
activity             :  123.0         neglect              :  0.436
indication           :  123.0         truth                :  0.414
deception            :  121.0         support              :  0.411
inactivity           :  119.0         pleasurableness      :  0.399
obstinacy            :  118.0         obstinacy            :  0.397
unimportance         :  117.0         unconformity         :  0.394
store                :  117.0         inactivity           :  0.39
aid                  :  116.0         aid                  :  0.383
support              :  116.0         activity             :  0.375
destruction          :  110.0         deception            :  0.371
uncertainty          :  109.0         information          :  0.366
pleasurableness      :  109.0         destruction          :  0.362
information          :  108.0         restraint            :  0.362
error                :  105.0         unimportance         :  0.36
skill                :  105.0         painfulness          :  0.357
care                 :  105.0         junction             :  0.347
restraint            :  105.0         :
---------------------------------------------------------------------
```

[21]:
```
print_measures(top_20_vertices["proteins"])
```

```
---------------------------------------------------------------------
               CC                                    NC

---------------------------------------------------------------------
YLR291C              :  0.000336      YLR291C              :  385.0
YBR261C              :  0.000297      YLR423C              :  327.0
YPL070W              :  0.000295      YBR261C              :  325.0
YCL028W              :  0.000292      YPL070W              :  316.0
YPL049C              :  0.000289      YPL049C              :  310.0
```

```
YLR423C             :  0.000289      YCL028W             :  305.0
YNL044W             :  0.000283      YHR113W             :  298.0
YHR113W             :  0.000282      YDR510W             :  297.0
YOR284W             :  0.000282      YNL044W             :  296.0
YLR245C             :  0.00028       YOR284W             :  294.0
YBR080C             :  0.000279      YKR034W             :  294.0
YPL088W             :  0.000278      YLR245C             :  291.0
YGR267C             :  0.000277      YDL239C             :  291.0
YHL018W             :  0.000277      YGL153W             :  290.0
YBR233W             :  0.000276      YPL088W             :  289.0
YMR095C             :  0.000276      YBR080C             :  289.0
YDR256C             :  0.000276      YNL229C             :  288.0
YOR095C             :  0.000275      YHL018W             :  288.0
YHR112C             :  0.000275      YMR095C             :  287.0
YKR034W             :  0.000275      YOL034W             :  287.0
---------------------------------------------------------------------


---------------------------------------------------------------------
               DC                                   AC

---------------------------------------------------------------------
YLR291C             :  86.0          YCR106W             :  0.892
YLR423C             :  58.0          YIR038C             :  0.876
YIR038C             :  51.0          YML051W             :  0.865
YBR261C             :  42.0          YLR423C             :  0.856
YDR510W             :  40.0          YLR291C             :  0.848
YDR479C             :  36.0          YDR510W             :  0.832
YDR100W             :  30.0          YDL100C             :  0.832
YML051W             :  29.0          YDR479C             :  0.79
YPL094C             :  28.0          YPL094C             :  0.747
YPL049C             :  27.0          YMR070W             :  0.722
YPL070W             :  27.0          YBR261C             :  0.72
YAR027W             :  25.0          YPL004C             :  0.719
YNL189W             :  22.0          YDR100W             :  0.665
YDL100C             :  22.0          YAR027W             :  0.644
YDR448W             :  21.0          YER125W             :  0.629
YCR106W             :  20.0          YIR033W             :  0.625
YKR034W             :  19.0          YDR448W             :  0.617
YML029W             :  17.0          YKL117W             :  0.615
YIR033W             :  17.0          YML029W             :  0.59
YHR113W             :  16.0          YJL019W             :  0.585
---------------------------------------------------------------------
```

2.2 Adjacency centrality vs degree centrality

We see that adjacency centrality gives us a notion of how *popular* (that is, the degree centrality) a node in context to its neighbours. A node with high adjacency centrality has a higher popularity compared to its neighbours (on average), while a node with a low adjacency centrality has a lower popularity compared to its neighbours (on average). In contrast, degree centrality has a more localised view, giving only the absolute *popularity* of a node. The best of the two measures depends on how the *influence* of a node is to be decided. For example, in the transport dataset it could be said that adjacency centrality is a more useful measure as it gives you a feel for which nodes may be treated as *hubs*; that is, a node in which people commute to and from on the way to another node (although a measure such as betweenness centrality may do this more aptly). In general, adjacency centrality seems to be a better measure in a context such as flow networks, while degree centrality is more suited in situations where you want to select nodes with a high popularity, such as in a graph SIR model (see next assignment).

2.3 Nearness centrality vs closeness centrality

Closeness centrality gives us the average shortest path from a node to all other nodes. Nearness centrality (otherwise known as harmonic centrality) is a variant of closeness centrality that was devised to help when dealing with disconnected graphs (that is, a graph in which some nodes are unreachable from a given start node). To see this, we define $1/d_{ij}$ to be 0 if there is no path from i to j. In the datasets given, we are working with the largest connected component, so this benefit does not help. We can draw more distinctions between these measures. We can think of calculating nearness centrality as a *scoring* process, for every node that is close to the node we are calculating nearness centrality for, we add the reciprocal of its distance to the total score, doing this for every other node. Here we see that a node is not punished for having a node that it has a large shortest path to, it simply gets a small score added on. In comparison, closeness centrality *does* punish such long shortest paths as it is computing the average shortest path length. Thus one may pick nearness centrality if we are favouring nodes that have a lot of close nodes, while one may pick closeness centrality if we want the node to be (on average) close to all nodes.