Network Data Analysis - Assignment 1

Felipe M. Megale, 100806980

February 2022

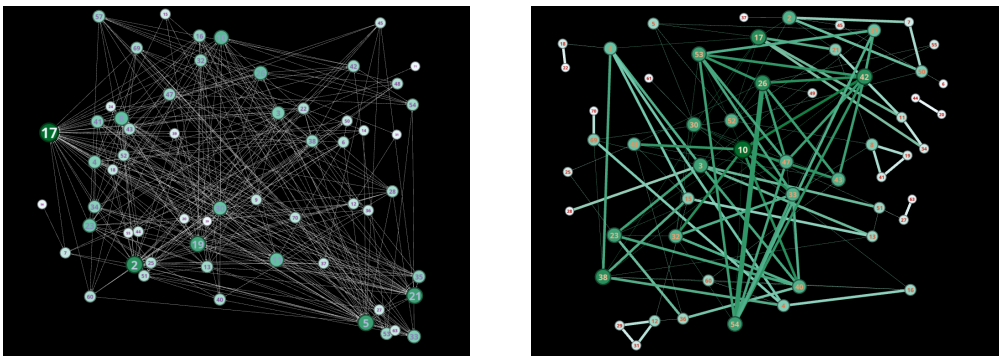
Contents

1 Pre-processing data

To complete the pre-processing of the original dataset, I used Pandas and Numpy. The first step for pre-processing the original file, after downloading it, was to remove all rows that had issues. Specifically speaking, this issue is not having either person name or ID. If either of these two information are missing, the row is removed from the original set. After removing those rows, I also removed those columns that did not have a label i.e., a question, and the two last questions that were further apart from the first 7. Furthermore, in order to have a more intelligible data set, I renamed the first column from *"Unnamed: 0"* to *"Name"*. Then, I isolated all questions and for each one of them, I created a new data set comprised of three attributes: *"Name"*, *"ID"*, *"Question"*. This resulted in 7 new data sets, one for each question, which were saved as CSV files. The questions are:

1. "Which person you have hear of their voice or seen their faces?"
2. "Which person you have met (in person+online) and exchange conversation?"
3. "Which person you have collaborated with?"
4. "Which person you have eye contact?"
5. "Which peson you have eaten lunch with?"
6. "Which person you have shared a ride?"
7. "Which person you have taken at least two courses with?"

After having separated the questions, I processed the inputs of each question in each individual file. The reason for this was to simply adequate each person's input to comma-separated IDs by removing spaces and trailing characters such as commas and hyphens. Finally, for each of the 7 questions, I created 7 other files in the CSV format Gephi expects. For example, a line with the following values *"1,2,3,4,5"* means that node 1 is connected to nodes 2, 3, 4 and 5.



(a) Network 1 **(b)** Network 2

Figure 1: Networks 1 & 2

2 Analyses of 6 networks

2.1 Available networks

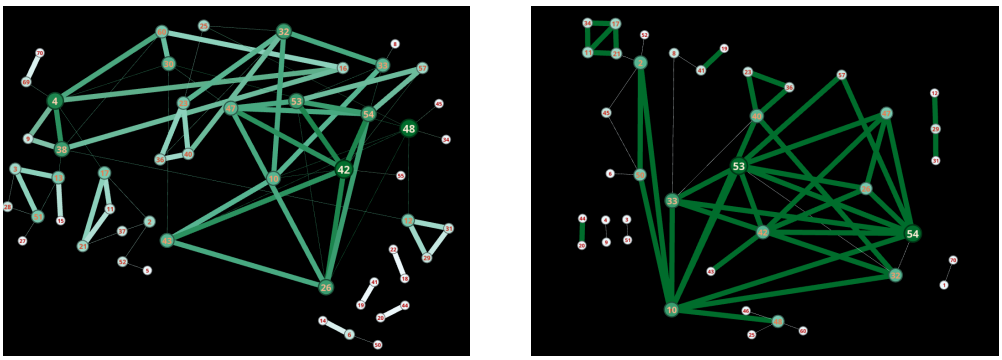
For this assignment, out of the 7 available networks, the ones I chose to work with are networks 1 through 5, and 7.

Before describing each network individually, one characteristic all of them share is that the edges in all 7 graphs are unweighted for the same reason. The questions asked *"which people?"*, meaning that for each person in the list, you only place an edge if the answer is yes. Because it is binary (edge or no edge), all graphs have unweighted edges. However, if we had a question asking *"how many times?"*, then there is a possibility of having weighted edges. Also, Gephi does not render a node if its degree equals zero.

2.1.1 Network 1

The first network asks the following question: *"Which person you have hear of their voice or seen their faces?"*.

Directed or undirected? This network is an example of a directed graph. The reason for this is that in online conferences, it is not all participants who speak and/or have their computer cameras on. Therefore, I may see other people's faces and/or hear their voices, but they may not see or hear me.



(a) Network 3 **(b)** Network 4

Figure 2: Networks 3 & 4

2.1.2 Network 2

The second network asks the following question: *"Which person you have met (in person+online) and exchange conversation?"*.

Directed or undirected? This network is an example of an undirected graph. In order to have a conversation, both parties must engage. If only one of them speak, there is no dialog. Therefore, is is not possible to exchange conversation.

2.1.3 Network 3

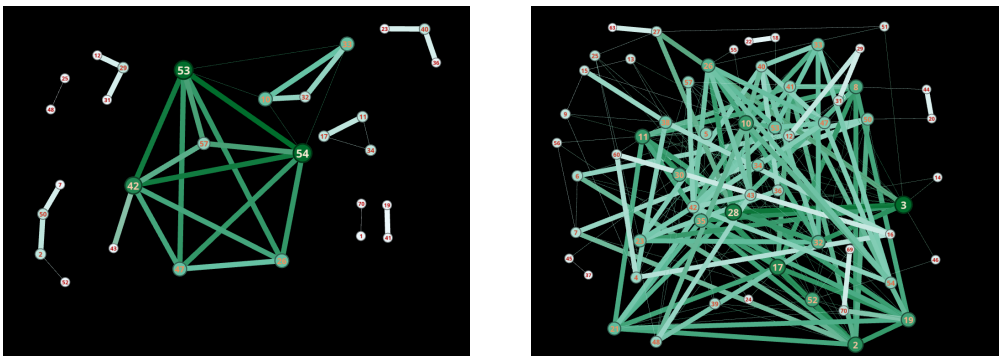
The third network asks the following question: *"Which person you have collaborated with?"*.

Directed or undirected? This is an undirected graph because collaboration must be enforced by both parties. It has to be a mutual agreement.

2.1.4 Network 4

The fourth network asks the following question: *"Which person you have eye contact?"*.

Directed or undirected? This network is undirected because in order to establish eye contact, two people must engage. There is no way for one person to look into someone else's eyes and not be looked back.



(a) Network 5 **(b)** Network 7

Figure 3: Networks 5 & 7

2.1.5 Network 5

The fifth network asks the following question: *"Which peson you have eaten lunch with?"*.

Directed or undirected? This is an undirected graph because eating lunch with another person implies that both people had to simultaneously engage in the activity.

2.1.6 Network 7

The seventh network asks the following question: *"Which person you have taken at least two courses with?"*.

Directed or undirected? This network is an undirected graph because you cannot be simultaneously enrolled in a course another person has not and consider that as taking a course together. Also, the course must be taken in the same year.

Metric	Net. 1	Net. 2	Net. 3	Net. 4	Net. 5	Net. 7
Num. Nodes	60	60	59	60	59	60
Num. Edges	426	120	73	48	30	214
Density	0.120	0.067	0.042	0.027	0.017	0.120
Avg. Clust. Coef.	0.394	0.304	0.296	0.241	0.168	0.488
Num. Nodes SCC	49	-	-	-	-	-
Num. Nodes WCC	58	-	-	-	-	-
Num. Nodes CC	-	52	42	20	10	59
Avg. Path Len. SCC	2.599	-	-	-	-	-
Avg. Path Len. CC	-	3.143	4.234	2.857	1.711	2.654
Diameter of SCC	7	-	-	-	-	-
Diameter of CC	-	7	11	6	4	6

Table 1: Network statistics

3 Metrics of 6 networks

In this section I will present some statistics about the 6 chosen networks. The results will be presented in tables. All metrics and plots were calculated and generated using the Python package NetworkX.

3.1 Number of nodes

The number of nodes of each graph is the amount of people who have participated in the survey. Whether they are connected to someone else, or not, they will be represented as nodes.

3.2 Number of edges

The number of edges of each graph will be the amount of connections that exist between each person for each question.

3.3 Edge density

Graph density tells us how connected nodes are between each other. If the density value is high, we say the graph is connected, if the density value is low, we say it is sparse. For undirected graphs, this metric can be calculated as

$$D_{undirected} = \frac{2|E|}{|N|(|N|-1)} \tag{1}$$

and the density for directed graphs is defined as

$$D_{directed} = \frac{|E|}{|N|(|N|-1)} \tag{2}$$

where E is the number of edges and V is the number of nodes in the graph.

3.4 Degree distribution

The degree distribution can superficially tell us what are the preferences for people when connecting to each other. Figure 4 plots the degree distributions for the 6 chosen networks.

3.5 Average clustering coefficient

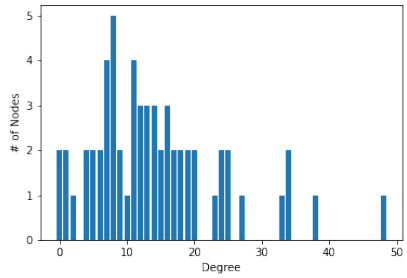
The average clustering coefficient for a graph helps determine how transitive a relationship is. For example, if persons A and B are friends, and persons A and C are friends, there is a high chance that persons B and C are also friends. The clustering coefficient is defined as

$$C_i = \frac{2e_i}{k_i(k_i-1)} \tag{3}$$

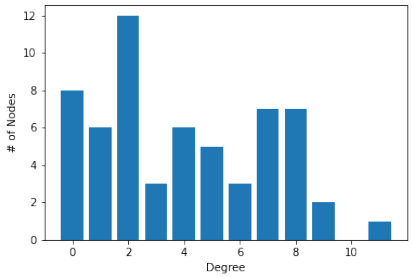
where e_i is the number of edges between the neighbors of node i.

The average clustering coefficient of the graph is calculated as

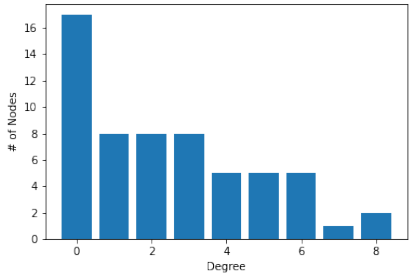
$$\langle C \rangle = \frac{1}{N}\sum_{i}^{N} C_i \tag{4}$$



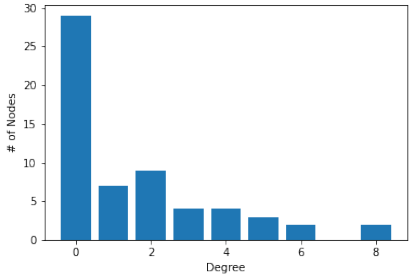
(a) Network 1



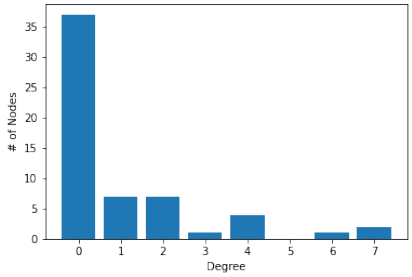
(b) Network 2



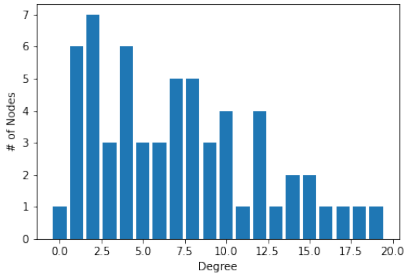
(c) Network 3



(d) Network 4



(e) Network 5



(f) Network 7

Figure 4: Degree distributions

where N is the number of nodes in the graph, and C_i is the clustering coefficient of node i.

3.6 Number of nodes in strongly connected component (SCC)

The strongly connected component (SCC) metric can only be obtained from directed graphs. Since only the first network is directed, it is the only one that can provide this value. For networks 2 through 5, and 7, the values are from the connected components. Refer to table 1 for the values.

3.7 Number of nodes in weakly connected component (WCC)

The weakly connected component (WCC) metric can only be obtained from directed graphs. Since only the first network is directed, it is the only one that can provide this value. For networks 2 through 5, and 7, the values are from the connected components. Refer to table 1 for the values.

3.8 Average path length in SCC

The average path length metric indicates how far apart two nodes are from each other in the connected graph. In other words, how many jumps, in average, it takes to reach other nodes. For a directed graph, it is calculated as

$$\langle d \rangle \equiv \frac{1}{2L_{max}} \sum_{i,j \neq i} d_{ij} \tag{5}$$

whereas for undirected graphs, it is calculated as

$$\langle d \rangle \equiv \frac{1}{L_{max}} \sum_{i,j > i} d_{ij} \tag{6}$$

For networks 2 through 5, and 7, I calculated the average path length in the connected component because these networks are undirected graphs, thus not being possible to determine strongly connected components.

3.9 Diameter of SCC

This metric represents the maximum shortest distance between two nodes in a connected graph. It is be represented as

$$diameter \equiv \max_{ij} d_{ij} \tag{7}$$

For the first network, I collected the diameter of the strongly connected component. However, since all other networks are undirected graphs, I collected the diameter of the largest connected component.

3.10 Community detection

To detect communities in each of the chosen networks, I ran the Girvan–Newman algorithm, implemented in NetworkX. Figure 5 illustrates the communities in each network. The communities the algorithm found make sense, given that it ran by removing the edges with highest betweenness, separating the communities the edge held together. Also, by comparing with figures 1, 2, and 3, we can see that the nodes with most connections between each other form a community the algorithm was able to find. Another interesting feature that the community detection algorithm allows us to perceive is that the more sparse the graph, the more communities we have. Figures 5c, 5d, and 5e depict this behavior.

3.11 Centrality Measures

Centrality tries to determine which node is the most central in a graph. The four centrality measures will be used to determine which node is the most important in each network. The measures are in-degree, out-degree, betweenness, and closeness. The choice for each centrality measure for each graph was arbitrary.

3.11.1 Network 1

For this network, in-degree and out-degree will be used to determine the two most central nodes. Because it is a directed graph, we can use these measures. The in-degree centrality of a node is calculated based on how many edges arrive in it. Similarly, the out-degree centrality of a node is calculated based on how many edges leave it. Table 2 summarizes the two most people.

Rank	Person	Score
1	17	0.576
2	5	0.508

(a) In-degree

Rank	Person	Score
1	10	0.271
2	19	0.254

(b) Out-degree

Table 2: Network 1 centrality

Rank	Person	Score
1	10	0.186
2	42	0.152

(a) Degree

Rank	Person	Score
1	17	0.169
2	30	0.131

(b) Betweenness

Table 3: Network 2 centrality

3.11.2 Network 2

The centrality measures analyzed for this network were degree centrality and betweenness centrality. Degree centrality is similar to in-degree and out-degree centrality measures, except for the direction of the edges, which do not exist. The degree centrality measure for a node is defined by the amount of edges connected to it. Betweenness, on the other hand, is defined by the amount of shortest paths that go through a given node, and is calculated as follows

$$C_B(i) = \sum_{j<k} \frac{g_{jk}(i)}{g_{jk}} \tag{8}$$

where g_{jk} is the amount of shortest paths connecting nodes j and k, and $g_{jk}(i)$ is the node currently being analyzed. Table 3 summarizes the values found.

3.11.3 Network 3

The chosen centrality measures for this network were degree and closeness. Degree centrality has been previously defined. However, the closeness centrality measure is defined by the average length of shortest paths between a

Rank	Person	Score
1	48	0.137
2	42	0.137

(a) Degree

Rank	Person	Score
1	48	0.245
2	30	0.243

(b) Closeness

Table 4: Network 3 centrality

Rank	Person	Score
1	10	0.039
2	33	0.032

(a) Betweenness

Rank	Person	Score
1	10	0.169
2	33	0.165

(b) Closeness

Table 5: Network 4 centrality

node and all other nodes in a graph. It can be calculated as

$$C_C(i) = \left[\frac{1}{N-1} \sum_{j=1}^{N} d(i,j) \right]^{-1} \tag{9}$$

where N is the number of nodes in a graph, and $d(i,j)$ is the distance between nodes i and j. Table 4 summarizes the centrality results for this network.

3.11.4 Network 4

The centrality measures chosen to analyze from this network were betweenness and closeness. These two measures have been previously defined. Table 5 summarizes the values found.

3.11.5 Network 5

The centrality measures analyzed for this network were degree and betweenness centralities. These measures already have been introduced. Table 6 summarizes the two most influential nodes of this network and their centrality score.

Rank	Person	Score
1	53	0.120
2	54	0.120

(a) Degree

Rank	Person	Score
1	42	0.005
2	53	0.004

(b) Betweenness

Table 6: Network 5 centrality

Rank	Person	Score
1	3	0.322
2	28	0.305

(a) Degree

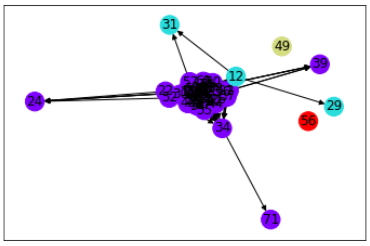
Rank	Person	Score
1	28	0.543
2	17	0.509

(b) Closeness

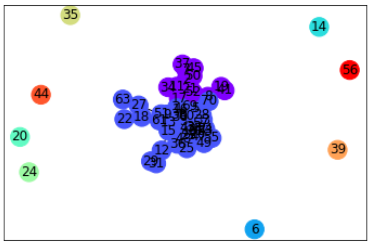
Table 7: Network 7 centrality

3.11.6 Network 7

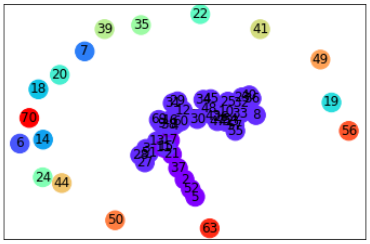
The chosen centrality measures for network 7 were degree and closeness. I will refrain from going deeper into these concepts since they have already been introduced. Table 7 summarizes the centrality findings for this network.



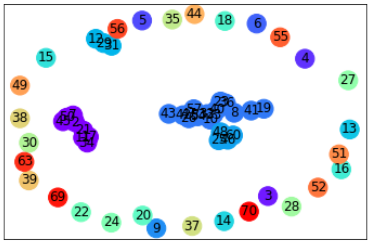
(a) Network 1



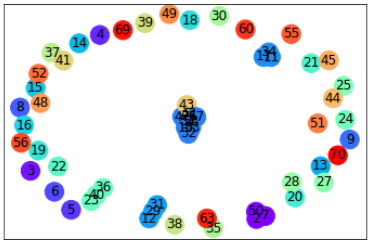
(b) Network 2



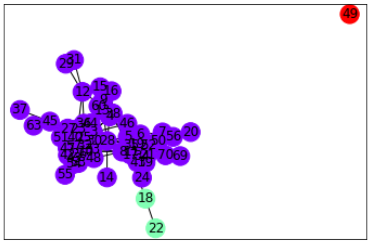
(c) Network 3



(d) Network 4



(e) Network 5



(f) Network 7

Figure 5: Detected Communities

4 Insights

All the metrics previously calculated and visualized allow us to extract insightful knowledge about the networks that exist in our graduate program. Tackling the aspects of low degree distribution, low density and high number of communities, we can see that the questions *"Which person you have collaborated with?"*, *"Which person you have eye contact?"*, and *"Which peson you have eaten lunch with?"* create the most sparse graphs. That may be due to the fact that most classes are still online and people haven't gotten the chance to interact with each other in a more meaningful way.

It is also interesting to notice unexpected behaviors on these sparse networks. Let us look at network 3 for example. It was one the three networks that were low in density and very sparse. However, the existing connected component was well connected. This means that despite existing many people who did not collaborate with their peers, the ones who did were thorough in doing it. However, because there were few edges in the whole graph, the average path length in the connected component was high. This is an indication that people collaborate together but they have their usual partners.

For network 5, which asks the question *Which peson you have eaten lunch with?"*, we see that not many people have participated in this activity. Those who did, usually do frequently and there seems to be a closed group. Visualizing the network we are able to see that the connected component is almost a complete graph. This means that even though there are few connections in the graph, the ones that exist are meaningful.

Finally, discovering which nodes in each graph are more central is important. The reason for this is that knowing which people connect two or more components may contribute to how the people in these networks interact and collaborate with each other. Removing some of the key people from can result in disrupting an intricate collaboration network.

5 Information Networks

5.1 Crawler Implementation

```
1  ''' IMPORTS '''
2  # regular expression module
3  import re
4
5  # import HTML parser
6  from bs4 import BeautifulSoup
7
8  # import HTTP library
9  import requests
10
11 # import URL utility library
12 from urllib.parse import urlsplit
13
14 # import another URL utility library
15 import tldextract
16
17 # import sleep to prevent IP lockout
18 #     or intentional delays
19 from time import sleep
20
21 # import datetime to improve logging
22 from datetime import datetime
23
24 # python native queue data structure
25 from collections import deque
26
27 # native python json module, allows to serialize/
       deserialize json
28 import json
29 '''END IMPORTS'''
30
31 '''GLOBAL VARIABLES'''
32 # email regex
33 email_regex = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-
```

```
    Z|a-z]{2,}\b'
34
35 # list of accepted domains
36 university_domains = ['uoit', 'ontariotechu']
37
38 # list of excluded extensions
39 bad_file_extensions = ["mp4","mkv","pdf","docx","doc","
    mp3","wav","webp", "jpg", "png"]
40
41 # BFS will use this URL as starting node
42 INITIAL_URL = "https://ontariotechu.ca"
43
44 # this is a dictionary. each key is a URL and each
    value is a set of URLs that are referenced by the
    key
45 graph = {}
46
47 # this will be the queue of unvisited URLs
48 queue = deque()
49 '''END GLOBALS VARIABLES'''
50
51 ''' BEGIN FUNCTIONS '''
52 # helper function to check if string is an email
53 def is_email(string):
54     is_string_an_email = re.fullmatch(email_regex,
        string)
55     return is_string_an_email
56
57 # helper function to get formatted current datetime
58 def get_now():
59     return datetime.now().strftime("%Y-%m-%d_%H:%M:%S")
60
61 # lambda function to disregard all those hrefs that are
    not http or https pages, e.g. mailto
62 def filter_non_http(url):
63     splitted_url = urlsplit(url)
64     scheme = splitted_url.scheme
65     if scheme not in ['http', 'https']:
```

```
66          return False
67      if is_email(netloc):
68          return False
69      return True
70
71  # lambda function to filter off URLs with bad file
        extensions
72  def detect_bad_file_extensions(url):
73      splitted_url = urlsplit(url)
74      path = splitted_url.path
75
76      for ext in bad_file_extensions:
77          if ext in path:
78              return False
79      return True
80
81  # lambda function to remove www. from URLs, trailing
        slashes, index pages and double slashes.
82  def rstrip_url(url):
83      splitted_url = urlsplit(url)
84      netloc = splitted_url.netloc.replace("www.", "")
85      path = splitted_url.path\
86          .rstrip('/')\
87          .replace('index.php', '')\
88          .replace('index.html', '')\
89          .replace('//','/')\
90          .rstrip('/')
91      new_url = f'https://{netloc}{path}'.rstrip('/')
92      return new_url
93
94  # lambda function to convert uoit domain to
        ontariotechu
95  def uoit_to_ontariotechu(url):
96      splitted_url = urlsplit(url)
97      netloc = splitted_url.netloc.replace('uoit','
            ontariotechu')
98      path = splitted_url.path
99      new_url = f'https://{netloc}{path}'.rstrip('/')
```

```
100     return new_url
101
102 # key function for BFS. This expands the current node (
    URL)
103 def generate_children(url):
104     try:
105         # 1) sleep for 1 second to avoid ip blocking or
              throttling
106         sleep(1)
107
108         # 2) perform HTTP request to given URL
109         print(get_now(), "GET HTTP request", url)
110         r = requests.get(url)
111         html_content = r.content
112
113         # 3) parse HTML response to Python object
114         soup = BeautifulSoup(html_content, "html.parser
              ")
115
116         # 4) get all anchor tags
117         all_anchors = soup.find_all("a")
118
119         # 5) get all hrefs from all anchors
120         all_hrefs = []
121         for anchor in all_anchors:
122             try:
123                 all_hrefs.append(anchor['href'])
124             except:
125                 pass
126
127         # 6) filter off URLs that are not in the UOIT
              or OTU domains
128         university_hrefs = list(filter(lambda d:
              tldextract.extract(d).domain in
              university_domains, all_hrefs))
129
130         # 7) filter off all URLs that are not http or
              https (e.g. mailto)
```

```
131             university_hrefs = list(filter(filter_non_http,
                    university_hrefs))
132
133             # 8) filter off URLs that point to files
134             university_hrefs = list(filter(
                    detect_bad_file_extensions, university_hrefs
                    ))
135
136             # 9) remove www., index.php or index.html,
                    trailing slashes and double slashes
137             university_hrefs = list(map(rstrip_url,
                    university_hrefs))
138
139             # 10) convert uoit domains to ontariotechu
                    since uoit redirects to ontariotechu
140             university_hrefs = list(map(
                    uoit_to_ontariotechu, university_hrefs))
141
142             # 11) return all found children
143             print(get_now(), 'Returning children...')
144             return list(set(university_hrefs))
145         except:
146             # return empty children list if HTTP request
                    fails for any reason
147             return []
148
149 # bfs driver function
150 def bfs(url):
151     print(get_now(), "Started BFS...")
152
153     # 1) start by populating the queue with the initial
            node
154     queue.append(url)
155     skipped_count = 0
156
157     # 2) while the queue isnt empty
158     while len(queue) != 0:
159
```

```
160         # 3) dequeue the first item in line
161         u = queue.popleft()
162
163         # 4) get current nodes of the graph
164         graph_keys = list(graph.keys())
165
166         print(get_now(), "Current number of nodes", len
                (graph_keys))
167
168         # 5) if the dequeued item is not a node in the
                graph
169         if u not in graph_keys:
170             print(get_now(), "Skipped", skipped_count,
                    "before expanding next node!")
171             skipped_count = 0
172             print(get_now(), f'Generating children of {
                    u}...')
173
174             # 6) expand all nodes from current node
175             children = generate_children(u)
176             print(get_now(), 'Generated', len(children)
                    , 'children!')
177
178             # 7) queue new URLs if they have not been
                    queued yet
179             for child in children:
180                 if child not in queue    :
181                     queue.append(child)
182
183             # 8) add current node's adjacency list to
                    graph
184             graph[u]=children
185
186         # 9) if web page has been visited before, skip
187         else:
188             skipped_count += 1
189             print(get_now(), f'Skipped {u} !')
190
```

```
191             print(get_now(), "Queue length", len(queue))
192
193         # 10) after traversing all web pages, dump graph to
                a json file for posterior processing and
                analysis
194         print(get_now(), 'Writing adjacency list to file...
                ')
195         with open('bfs_adj_list.json', 'w') as f:
196             f.write(json.dumps(graph))
197         print(get_now(), 'Wrote adjacency list to file!')
198
199 ''' END FUNCTIONS '''
200
201
202 # Python's main function
203 ''' BEGIN MAIN '''
204 if __name__ == '__main__':
205     program_begin = datetime.now()
206     print(program_begin, "Starting BFS...")
207     # start BFS from initial URL
208     bfs(INITIAL_URL)
209     program_end = datetime.now()
210     print(program_end, "Program finished!")
211     delta = program_end - program_begin
212     print("Program started", program_begin)
213     print("Program finished", program_end)
214     print("Program took", delta, "to complete!")
215 ''' END MAIN '''
```

5.2 Code Explanation

5.2.1 Imports

The *re* module provides functions to test and validate regular expressions. In this case it was used to check if a given string was a valid email.

BeautifulSoup is responsible for converting the HTML string retrieved from the website into a serialized Python object. This allows for easy querying of all HTML tags and their attributes.

The *requests* library is used to perform HTTP GET requests. The actual web page information is stored in the response's *content* property.

From *urllib* I imported the urlsplit function to easily split the URL into scheme, domain, and path. Query string parameters were ignored. In the same line of though, *tldextract* was also used to easily extract the domain of the URL and change it in a lambda function from *uoit* to *ontariotechu* if it was the case.

The *sleep* function from module *time* was imported to force a 1-second wait before issuing the next HTTP request. This was done to avoid throttling, which would slow down the crawler's activity, and to prevent the domain from locking out my IP.

The *datetime* function from module *datetime* was imported to simply print out current date and time. The reason behind this was to improve logging quality.

The *deque* class was imported from module *collections* to ease the development of the BFS algorithm. It is a queue data structure and allows for queueing and dequeueing elements in it in a first in, first out scheme. It works as the queue of web pages that have not been visited yet.

Finally, the *json* module was imported to dump the assembled graph into a JSON file. Because the crawling was a time-consuming task, having the graph readily available is important.

5.2.2 Global Variables

The *email_regex* variable holds the pattern used to match strings that contain valid email addresses. It is used in the *is_email* helper function.

A *university_domains* list containing *uoit* and *ontariotechu* was used to hold the two University's known domains. It is employed on the filtering of URLs that point to outside of the University's network. This way I can keep crawl only the pages that belong and refer to the Ontario Tech University.

The *bad_file_extensions* list holds a few of the common file extensions that may be present on a URL. It, too, is used to filter off web pages that would host files. Since they are not actual web pages, it makes no sense to go through them.

Simply, the *INITIAL_URL* is just a variable that is used to trigger the Breadth First Search Algorithm. It is passed in the main method as the starting node.

The *graph* variable is a dictionary that will be the assembled graph. The

keys will be the URLs and the adjacency list will be a set of URLs the node has hyperlinks to.

Finally, the *queue* variable starts as an empty queue. This queue is used to hold the web pages that have yet to be visited.

5.2.3 Functions

The *is_email* tests a string against a regular expression and verifies whether it is a valid email or not. It is used as a helper function in the *filter_non_http* function.

The *get_now* function is a wrapper for the *datetime* function. It's only purpose is to return a formatted string containing the current date and time, for logging purposes.

Next, the *filter_non_http* function is a lambda function. It is used in a *filter* list function to remove all URLs that are not HTTP or HTTPS schemes. An example of a URL that was removed was a *mailto*.

The *detect_bad_file_extensions* function is another lambda filter function. It works by splitting the URL and grabbing its path - the path is everything that comes after the country (e.g. *.ca*) - and searching for any of the aforementioned bad file extensions.

Following the list of helper functions, the *rstrip_url* is another lambda function. This one, on the other hand, is used in a list mapping. It works by splitting a URL and altering the path by removing the **www** subdomain, trailing slashes and any files that are index files (HTML or PHP). This function then returns the adjusted URL.

The next lambda function is called *uoit_to_ontariotechu*. The sole purpose of this function is to map all URLs that are in the UOIT domain to Ontario Tech University's domain. This avoids adding nodes to the graph that are different in name, but that point to the same content.

Next, the core of the BFS algorithm, is the *generate_children* function. It works by trying to perform an HTTP GET request. When the request fails by any reason, I say that no children were found, since no web page was collected and no hyper links are present. If the request does not fail, I collect the content (namely, the HTML page as a string), parse it with *BeautifulSoup*, query all anchor HTML tags, then try to collect every href attribute from the anchors. Next, I process all hrefs using a series of mappings and filters. I first filter off those URLs that are not in the UOIT/Ontario Tech University domain. Then the remaining URLs are filtered to remove every scheme that

is not HTTP not HTTPS. The remaining URLs are again filtered to remove those that host files. Next, I map the URLs to remove trailing slashes and index pages. Finally, the remaining set of URLs is mapped again, but this time to change all UOIT domains into Ontario Tech University domains. After this series of mappings and filters, the remaining URLs are put into a set, which in Python removes duplicates from any list, and returned. This finishes the execution of the *generate_children* function.

Lastly, the *bfs* function is the actual implementation of the Breadth First Search algorithm. It starts by pushing to the queue the URL that was passed as a parameter. Then, until the queue is not empty, it dequeues a node that has not been visited yet and gets the current list of nodes in the graph. If the dequeued node is not a node in the graph, expand the graph by generating the current node's children. Then, for each child of the current node, add it to the queue if it has not been queued yet, then add all children as the adjacency list of the current node in the graph. If the dequeued node is in the list of nodes in the graph, skip it to avoid performing unnecessary HTTP requests. When the queue is empty, the function dumps the assembled graph to a JSON file for persistent storage and avoid running the crawler every time the graph is needed.

5.2.4 Main

The main function in Python is used to call the *bfs* function. It passes the *INITIAL_URL* as parameter. This function is the starting point of a Python program. It also calculates how long it took for the crawler to finish traversing all web pages in the Ontario Tech University's domain.

5.3 Analyses

After dumping the graph to a JSON file, the contents were read as a string and converted back to a dictionary. Using NetworkX, this dictionary was then used to instantiate a directed graph. All metrics were extracted using NetworkX.

5.3.1 Directed or undirected?

The network formed by the web pages within the University's domain is a directed graph. This choice is due to the fact that even though web page A

has a hyperlink to web page B, there is no guarantee that web page B will have a hyperlink to web page A. Also, when traversing a graph in a search, it is not interesting to be able to go back to the previous node. This risks the search never finding a solution.

5.3.2 Weighted or unweighted?

This graph was modelled to have unweighted edges. The reason for this is that the objective was to only traverse the web pages within the University's domain. Therefore, the main interest was to know whether or not one web page has a hyperlink to another. However, even if it is desired to find the most referenced node, one can accomplish this by querying the graph for the node with the greatest in-degree.

5.3.3 Metrics

Number of nodes. The total number of nodes, that is, unique web pages within the University's domain is **2592**.

Number of edges. The total number of edges, that is, how many hyperlinks exist within the University's domain is **112137**.

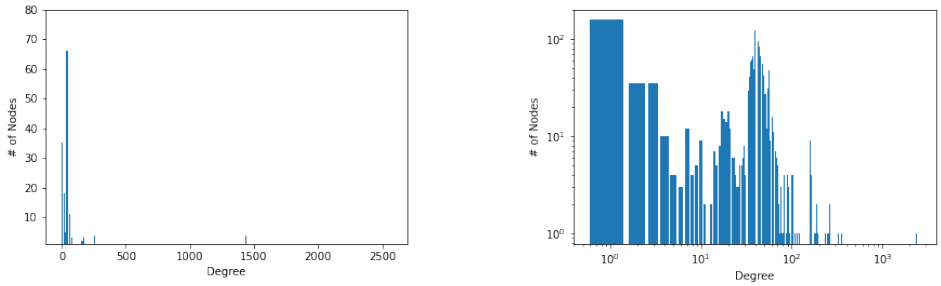
Edge density. Defined in equation 2, the density for the graph formed by the University's web pages is **0.0166**.

Degree distribution. Figure 6 shows the degree distribution for this network's graph. Figure 6a shows the degree distribution in normal scale. Figure 6b shows the degree distribution in log scale. Doing log-log scaling allows us to better see what the degree distribution looks like. The first figure may indicate that the degree distribution for this graph follows a power law. This would mean that a great number of nodes have low degree, meaning that web page does not reference may pages, and few pages refer it back.

Average clustering coefficient. Previously defined in equation 4, the average clustering coefficient for the graph assembled by traversing all web pages in the University's domain is **0.493**.

Number of nodes in strongly connected component. The number of nodes in the subgraph detected by querying the strongly connected component is **2343**.

Number of nodes in weakly connected component. The number of nodes in the subgraph detected by querying the weakly connected component is **2592**.



(a) Degree distribution **(b)** Degree distribution (log-log)

Figure 6: Degree distributions

Average path length in SCC. For the strongly connected component of the graph, the average path length is **4.737**.

Average path length in WCC. The average path length of the weakly connected component is **4.261**.

Diameter in SCC. The diameter of the strongly connected component is **63**.

Diameter in WCC. For the weakly connected component, however, NetworkX's implementation calculates is to be infinite because the directed graph is not strongly connected. Therefore, I calculated the diameter as though the graph is undirected. The diameter in this case becomes **7**.

Community detection. Running the Gilvan-Newman algorithm to detect communities renders us two communities from the graph formed by the University's web pages. The largest one has **2589 nodes**, and the second one has **3 nodes**. Due to the size of the graph, creating a good visualization is not possible, as seen in figure 7. Therefore, I had to resource to extracting information that could capture an overall picture of the detected communities. From the largest community, depicted in blue in figure 7, the node with both greatest in-degree and greatest out-degree is *https://ontariotechu.ca*. The in-degree for this node is **2310** and the out-degree is **257**. As for the smaller community, depicted in red in figure 7, the node with greatest in-degree is *https://fr.mindsight.ontariotechu.ca* and the value is **2**. The node with greatest out-degree in the smaller community is *https://mindsight.ontariotechu.ca* and the value is **1**. The communities found by the Gilvan-Newman algorithm were interesting. One would expect that all web pages within a domain would be closely intertwined, but that was not the case for Ontario Tech Univer-

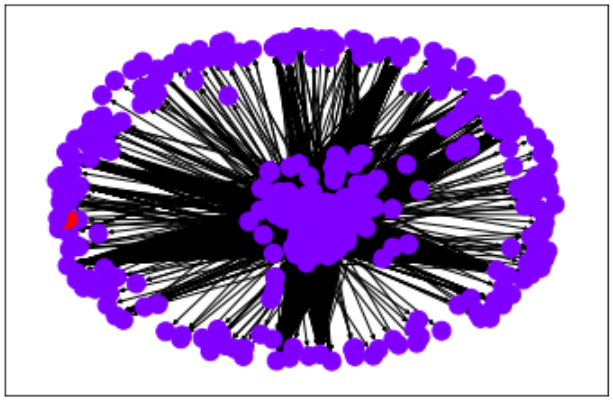


Figure 7: Detected communities

sity. The web pages in the smallest community only have links to themselves, making it impossible to go back to the main community.

Degree centrality. For degree centrality, I opted to report all three degree centrality measures: degree, in-degree and out-degree. Tables 8, 9, and 10 holds the top 10 most central nodes in the graph according to the degree centrality measure.

Closeness centrality. As for the closeness centrality measure, previously defined in equation 9, table 11 summarizes the closeness centrality of the top 10 most central nodes.

Betweenness centrality. Previously defined in equation 8, table 12 summarizes the 10 most central URLs in the University's domain, according to the betweenness centrality.

PageRank centrality. Finally, the PageRank centrality measure will determine the nodes' centrality based on their in-degrees. Table 13 summarizes the 10 most central nodes according to this centrality measure.

5.3.4 Interpretation

This is my interpretation of the results. This is my interpretation of the results. This is my interpretation of the results. This is my interpretation of the results. This is my interpretation of the results. This is my interpretation

Rank	URL	Score
1	https://ontariotechu.ca	0.991
2	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.914
3	https://ontariotechu.ca/faculty_staff	0.912
4	https://ontariotechu.ca/virtualtour	0.911
5	https://ontariotechu.ca/contact-us.php	0.911
6	https://ontariotechu.ca/terms-of-use.php	0.910
7	https://ontariotechu.ca/about	0.910
8	https://ontariotechu.ca/about/campus-buildings	0.910
9	https://usgc.ontariotechu.ca/governance	0.832
10	https://hr.ontariotechu.ca/working_at_uoit/careers	0.832

Table 8: Degree Centrality for OTU

Rank	URL	Score
1	https://ontariotechu.ca	0.892
2	https://ontariotechu.ca/forms/online/view.php	0.819
3	https://news.ontariotechu.ca/media-inquiries	0.816
4	https://ontariotechu.ca/maps	0.816
5	https://ontariotechu.ca/about	0.816
6	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.816
7	https://ontariotechu.ca/virtualtour	0.816
8	https://usgc.ontariotechu.ca/governance	0.816
9	https://hr.ontariotechu.ca/working_at_uoit/careers	0.816
10	https://ontariotechu.ca/terms-of-use.php	0.816

Table 9: In-Degree Centrality for OTU

Rank	URL	Score
1	https://ontariotechu.ca	0.099
2	https://ontariotechu.ca/faculty_staff	0.098
3	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.097
4	https://ontariotechu.ca/connectwithus	0.097
5	https://blog.ontariotechu.ca/all	0.096
6	https://ontariotechu.ca/mycampus	0.096
7	https://ontariotechu.ca/about/campus-buildings/downtown-oshawa	0.096
8	https://ontariotechu.ca/apply	0.095
9	https://ontariotechu.ca/future-students/viewbooks.php	0.095
10	https://ontariotechu.ca/current-students/academic-calendars/academic-calendar-archive.php	0.095

Table 10: Out-Degree Centrality for OTU

Rank	URL	Score
1	https://ontariotechu.ca	0.891
2	https://ontariotechu.ca/forms/online/view.php	0.827
3	https://ontariotechu.ca/maps	0.824
4	https://news.ontariotechu.ca/media-inquiries	0.824
5	https://ontariotechu.ca/about	0.824
6	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.824
7	https://ontariotechu.ca/virtualtour	0.824
8	https://usgc.ontariotechu.ca/governance	0.824
9	https://hr.ontariotechu.ca/working_at_uoit/careers	0.824
10	https://ontariotechu.ca/terms-of-use.php	0.824

Table 11: Closeness Centrality for OTU

Rank	URL	Score
1	https://blog.ontariotechu.ca	0.284
2	https://blog.ontariotechu.ca/all	0.214
3	https://ontariotechu.ca	0.107
4	https://ontariotechu.ca/sites/library	0.066
5	https://news.ontariotechu.ca/archives/2021/12/in-retrospect-ontario-tech-university-in-2021.php	0.057
6	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.043
7	https://ontariotechu.ca/faculty_staff	0.041
8	https://ontariotechu.ca/contact-us.php	0.035
9	https://socialscienceandhumanities.ontariotechu.ca	0.035
10	https://ontariotechu.ca/virtualtour	0.035

Table 12: Betweenness Centrality for OTU

Rank	URL	Score
1	https://ontariotechu.ca	0.022
2	https://ontariotechu.ca/sites/library	0.017
3	https://ontariotechu.ca/forms/online/view.php	0.014
4	https://news.ontariotechu.ca/media-inquiries	0.014
5	https://ontariotechu.ca/maps	0.014
6	https://ontariotechu.ca/about	0.014
7	https://ontariotechu.ca/future-students/undergraduate/campus-tours-and-events	0.014
8	https://ontariotechu.ca/virtualtour	0.014
9	https://usgc.ontariotechu.ca/governance	0.014
10	https://hr.ontariotechu.ca/working_at_uoit/careers	0.014

Table 13: Page Rank Centrality for OTU

of the results.