COSC 417 – Topics In Networking
Lab 3 – Fall 2025

For this lab, we'll be taking our previously automated tracing and ASN lookup scripts, combining them with a web crawler, and using it to generate a mapping file. This mapping file will contain the IP, the IP's that it is linked to (come before and after it in traceroutes), the URLs that route through that IP, and the ASN information of the IP. This will allow us to analyze individual nodes in the network.

Modularizing Our Code

In the previous lab, we wrote our automated ASN/tracing code as a singular program. For this lab, we'll start by making that code more modular. We'll turn it into a set of ***functions*** that we can import into our other script files and use.

First, create a new Python file called ***tracert.py***. Note that we cannot use the name ***trace.py***, as this is already a built-in module in Python (and it has nothing to do with traceroutes). Inside this file, we'll begin by importing our dependent libraries:

```
import os, re, ipwhois
```

Then, we'll define our first function, ***trace()***. This function will take an IP address input, and return a ***list of IP addresses*** from our traceroute. This uses the os.***popen()*** and re.***findall()*** functions that we used in the previous lab:

```
# Perform a traceroute on input IP address and return list of IPs in route
def trace(ip):
    myData = os.popen('tracert ' + ip).read()
    route = re.findall('(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})', myData)[1:]
    return route
```

That handles programmatically fetching all the IP addresses in a trace. Now, we'll define a function to programmatically fetch the ASN information of an IP address. It will return this as a ***dictionary*** of ASN values, just like we did in the previous lab. We'll call this function ***asn()***:

```
# Fetch ASN information of IP address and return as dictionary, or None
def asn(ip):
    try:
        myIp = ipwhois.net.Net(ip)
        myObj = ipwhois.asn.IPASN(myIp)
        results = myObj.lookup()
        return results
    except:
        return None
```

Note that the ***asn()*** function returns ***None*** if we are unable to fetch the ASN information. This may happen occasionally, so it's important we include the necessary try/except to catch it. We can easily test our code by running the small fragment provided here. ***Make sure you comment this out once you have tested it – we don't want this test code running when we import our module!***

```
# An example of how these functions can be used
#ips = tracert('8.8.8.8')
#for ip in ips:
#    print(asn(ip))
```

The Web Crawler

Create a new Python file in the ***same directory*** called ***crawl.py***. This will be our modular ***web crawler*** code. Refer back to the web crawler slides for assistance with this section. To begin, we'll need our imports:

```
from urllib.request import Request, urlopen
from random import choice, randint
import re
from time import sleep
```

First, we'll define a ***create_request()*** function, which takes a URL input and creates a new ***Request*** object from it. We'll use a ***rotating user agent***, based on a random choice from a list of user agents called ***agents***. The function will look like this:

```
# Generate a request to input URL with random User-Agent string
def create_request(url):
    headers = {'User-Agent': choice(agents)}
    req = Request(url, data=None, headers = headers)
    return req
```

Note the usage of the ***choice()*** function from the ***random*** module to randomly pick a user agent from the agents list. We'll then take the returned ***Request*** object, and pass it to another function we'll define called ***fetch()***. The fetch function will actually return the HTML code of the page:

```
# Fetch the HTML of a the given web page at URL, or return None
def fetch(url):
    try:
        response = urlopen(create_request(url), timeout = 10)
        if response.getcode() == 200:
            data = response.read()
            return data
        else:
            return None
    except:
        return None
```

Remember to include the check on the ***response code*** to make sure it's a 200 code (OK), and not an error. Also make sure to set an ***appropriate timeout value***. This is an integer in seconds. Ten seconds should be a long enough timeout in most cases. Like our previous ASN function, this code may throw an exception, so we need to use a try/except to make sure that we return ***None*** if we are unable to fetch the page. If successful, this function returns the ***HTML*** of the web page we've requested.

Next, we'll define the ***extract_links()*** function. This function should take raw HTML data in, and return a ***unique list*** of URLs extracted from the text. Reference the slides for assistance with this function, I will leave it up to you to implement. Don't forget to cast the data to a string using ***str()*** before you try using your regular expression, and don't forget to use the ***list(set())*** trick to return a list without any duplicates in it.

Finally, we'll define the ***crawl()*** function, which takes a ***starting URL*** and an integer number of ***steps***. The steps is the maximum number of pages to crawl before terminating the loop. We'll use two lists, ***queue*** and ***crawled***, to hold the URLs we might crawl and the URLs we have already crawled, respectively. We'll add all extracted links to the ***queue***, and then the crawled URL to the ***crawled*** list. Here's what the function looks like:

```
# Crawls repeatedly for X steps and returns a list of crawled URLs
def crawl(start_url, steps):
    queue = [start_url]
    crawled = []
    for _ in range(steps):
        try:
            url = choice(queue)
            queue.remove(url)
            queue = list(set(queue + extract_links(fetch(url))))
            crawled.append(url)
            sleep(5 + randint(0, 10))
        except:
            pass
    return crawled
```

DO NOT FORGET TO INCLUDE A SLEEP() IN YOUR LOOP! This is required to ensure that there is a delay between crawl requests. Otherwise, you may be IP banned while attempting to crawl websites. A delay of at least 5 seconds is usually sufficient, although longer delays are typically safer. When the loop is ***finished***, we return the ***crawled*** list of URLs. This should effectively be a loosely-linked, moreorless random list of URLs that we can then analyze using our ***tracert*** module. We can test the code with the following snippet (use a small number of steps):

```
# Example of using the crawler to crawl up to 5 pages
#starting = 'https://en.wikipedia.org/wiki/Special:Random'
#print(crawl(starting, 5))
```

The Map Module

Finally, we'll make one more Python file (in the same directory) called ***map.py***. This will be our actual program to execute. We'll begin by importing the dependent modules, including our own ***tracert*** and ***crawl*** modules:

```
import tracert
import crawl
import socket
import json
from urllib.parse import urlparse
```

We will need the ***socket*** module to convert a ***domain name*** to an equivalent IP address (effectively a DNS lookup). We'll need the ***json*** module to help us format out print out output data. And we'll need the ***urlparse*** function to retrieve the domain from our crawled URL, while removing extraneous data like the ***path*** or ***protocol*** (i.e. https://) from the URL.

Next, we'll go ahead and define some ***major variables***. Typically constants in python use ***ALL_CAPS*** format, although one of these values (the ***OUTPUT***) is not actually a constant – Python doesn't care, as constants do not ***have*** to actually be constant in Python, they are purely for style and readability. Our constants will be: a starting URL to begin the crawl, a maximum number of steps (or fetches) to perform in our crawl, an output dictionary object, and an output ***.json*** filename (the file does not have to exist yet):

```
STARTING_URL = 'https://en.wikipedia.org/wiki/Special:Random'
STEPS = 10
OUTPUT = {}
OUTFILE = 'netdata.json'
```

I recommend Wikipedia's ***Random Article*** page as a starting point, but you're free to start your crawl elsewhere if you desire. I keep the steps small for now so we can easily test. The output is just an empty dictionary, and the outfile will be called ***netdata.json***.

The first thing we'll need to do is perform our crawl. This will take some time (depending on the number of steps you use). We'll store the results in a list called ***urls***:

```
print("Performing crawl...")
urls = crawl.crawl(STARTING_URL, STEPS)
```

Once we've got a list of URLs from the crawler, we can begin our analysis. For ***every IP***, we'll store the following information:

- A ***unique list*** of URLs that the IP appears in the traceroute of
- A list of ***links***, or IP addresses which are directly adjacent to the IP in a traceroute
- A ***dictionary*** of ASN information from the ***asn()*** function for that IP

First, we'll want to loop over our list of URLs returned by the crawler. For each URL, we'll want to use ***urlparse*** to get the ***domain*** or ***host***, and then use ***socket.gethostbyname()*** to return an IPv4 address that points to that domain, which we'll call ***addr***. We'll then call the ***trace()*** function from our tracert module on the IP address, and store the returned list of IP addresses in the route:

```
print("Crawl complete. Performing traces.")
for url in urls:
    try:
        print("Tracing: " + url)
        url_parts = urlparse(url)
        addr = socket.gethostbyname(url_parts.netloc)
        route = tracert.trace(addr)
```

At this point, we have a list of IP addresses in ***route*** that represent the path taken from ***our computer*** to the remote server that the ***URL*** is pointing towards. We'll then ***loop over this list***, and begin adding data to our ***OUTPUT*** dictionary:

```
route = tracert.trace(addr)
previous = None
for ip in route:
    if ip in OUTPUT:
        OUTPUT[ip]['url'].append(url)
        OUTPUT[ip]['url'] = list(set(OUTPUT[ip]['url']))
    else:
        OUTPUT[ip] = {'asn': None, 'links': [], 'url': [url]}
```

First, we check if the IP address already exists in the dictionary. If not, we create a new entry that is ***empty*** except for a list containing the currently crawled URL. This empty object is a ***dictionary*** that includes an ***asn*** object, a ***links*** object, and a ***url*** object. If the IP ***is*** already in the OUTPUT variable, we append the current URL to the ***url*** list, and use the ***link(set())*** trick to remove any duplicates from the list.

Next, we'll want to handle the ***links***. This works based on the ***previous*** variable, which we'll set to the last IP address we looked at in the route. We'll add the previous IP in the route to the current IP's links, and vice-versa, showing a ***non-directional*** relationship between the two:

```
if previous:
    OUTPUT[previous]['links'].append(ip)
    OUTPUT[ip]['links'].append(previous)
    OUTPUT[ip]['links'] = list(set(OUTPUT[ip]['links']))
    OUTPUT[previous]['links'] = list(set(OUTPUT[previous]['links']))
previous = ip
```

This allows us to see the actual physical ***links*** between IP addresses – i.e. where traffic flows from IP to IP while following the path. Note we use the list(set()) trick again to remove duplicates from both the current IP's links list, and the previous IP's links list.

Finally, we'll want to check if the IP address has any ASN info yet. If it doesn't, we'll use our ***asn()*** function to retrieve it. Otherwise, we'll just ignore it. This avoids repeatedly requesting ASN info for the same IP address:

```
previous = ip
if not OUTPUT[ip]['asn']:
    asn_data = tracert.asn(ip)
    OUTPUT[ip]['asn'] = asn_data
```

Finally, we'll want to close our ***try/except*** with a simple printed error and a ***pass***, so that our code continues execution even if part of this process throws an exception:

```
except:
    print("Failed to trace: " + url)
    pass
```

Note that this entire ***try/except*** clause and all the code in it goes ***within*** the ***for url in urls*** loop. When the loop is finished, we'll have performed traces on every URL we crawled, and then performed ASN lookups on every IP address we have found in those traces and linked them together. All of this is now stored in a large nested dictionary in the ***OUTPUT*** variable. Your console output should look something like this:

```
Performing crawl...
Crawl complete. Performing traces.
Tracing: https://en.wikipedia.org/wiki/Special:Random
Tracing: https://www.discogs.com/master/592613
Tracing: https://creativecommons.org/licenses/by-sa/3.0/
Tracing: https://www.mediawiki.org/
Tracing: https://www.mediawiki.org/wiki/Template:Main_page/jv
Tracing: https://www.mediawiki.org/wiki/Template:Main_page/fr
Tracing: https://lists.wikimedia.org/hyperkitty/list/mediawiki-l@lists.wikimedia.org/thread/
5VVH5LTZUIN63WNAJZFREMD66LFQLTFT/
Tracing: https://creativecommons.org/website-icons
Tracing: https://thenounproject.com/search/?q=masks&amp;amp;i=127990
Tracing: https://creativecommons.org/category/weblog/press/
```

Finally, we'll take our ***OUTPUT*** and print it to our output JSON file. We'll use ***open()*** with the ***a+*** flag to set the output of the file as ***append or create if not-existent***, and then use the ***json.dumps()*** function to dump our data in JSON format. We can use the ***indent*** and ***sort_keys*** functions to help pretty-print this in a nice, readable format:

```
with open(OUTFILE, 'a+') as f:
    f.write(json.dumps(OUTPUT, indent = 4, sort_keys = True))
```


```
{
    "129.250.2.206": {
        "asn": {
            "asn": "2914",
            "asn_cidr": "129.250.0.0/16",
            "asn_country_code": "US",
            "asn_date": "1988-04-05",
            "asn_description": "NTT-COMMUNICATIONS-2914, US",
            "asn_registry": "arin"
        },
        "links": [
            "168.143.191.134",
            "129.250.3.124"
        ],
        "url": [
            "https://en.wikipedia.org/wiki/Special:Random",
            "https://www.mediawiki.org/wiki/Template:Main_page/jv",
            "https://www.mediawiki.org/",
            "https://www.mediawiki.org/wiki/Template:Main_page/fr"
        ]
    },
    "129.250.204.6": {
        "asn": {
            "asn": "2914",
            "asn_cidr": "129.250.0.0/16",
            "asn_country_code": "US",
            "asn_date": "1988-04-05",
            "asn_description": "NTT-COMMUNICATIONS-2914, US",
            "asn_registry": "arin"
        },
        "links": [
            "198.35.26.96",
            "129.250.4.43"
        ],
        "url": [
            "https://en.wikipedia.org/wiki/Special:Random",
            "https://www.mediawiki.org/wiki/Template:Main_page/jv",
```


Try running your code with a small number of steps and wait for it to finish. Then check your ***netdata.json*** file. It should look something like this:

Now we have lots of network data in an easy to parse, portable format. We'll be able to use this data in the future to generate ***graphs*** of the network, linking IP address to IP address and determining where the boundaries of various autonomous systems are, and how they interlink. In the next lab, we'll use this to ***graphically*** display this data.

Grading

Run your code with ***at least 100 steps*** once you have it working. Submit your ***three Python files***, plus the resultant ***netdata.json*** file on Moodle for grading.