#!/usr/bin/env python3

import pybgpstream

"""

CS 6250 BGP Measurements Project

Notes:

- Edit this file according to the project description and the docstrings provided for each function

- Do not change the existing function names or arguments

- You may add additional functions but they must be contained entirely in this file

"""

# Task 1A: Unique Advertised Prefixes Over Time

def unique\_prefixes\_by\_snapshot(cache\_files):

    """

    Retrieve the number of unique IP prefixes from each of the input BGP data files.

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A list containing the number of unique IP prefixes for each input file.

        For example: [2, 5]

    """

    # the required return type is 'list' - you are welcome to define additional data structures, if needed

    unique\_prefixes\_by\_snapshot = []

    for fpath in sorted(cache\_files):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "rib-file", fpath)

        unique\_prefixes = set()

        for el in stream:

            prefix = el.fields['prefix']

            unique\_prefixes.add(prefix)

        unique\_prefixes\_by\_snapshot.append(len(unique\_prefixes))

    return unique\_prefixes\_by\_snapshot

# Task 1B: Unique Autonomous Systems Over Time

def unique\_ases\_by\_snapshot(cache\_files):

    """

    Retrieve the number of unique ASes from each of the input BGP data files.

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A list containing the number of unique ASes for each input file.

        For example: [2, 5]

    """

    # the required return type is 'list' - you are welcome to define additional data structures, if needed

    unique\_ases\_by\_snapshot = []

    for fpath in sorted(cache\_files):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "rib-file", fpath)

        unique\_ases = set()

        for el in stream:

            if el.fields['as-path'] == "": continue

            ases = el.fields["as-path"].split(" ")

            unique\_ases.update(ases)

        unique\_ases\_by\_snapshot.append(len(unique\_ases))

    return unique\_ases\_by\_snapshot

# Task 1C: Top-10 Origin AS by Prefix Growth

def top\_10\_ases\_by\_prefix\_growth(cache\_files):

    """

    Compute the top 10 origin ASes ordered by percentage increase (smallest to largest) of advertised prefixes.

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A list of the top 10 origin ASes ordered by percentage increase (smallest to largest) of advertised prefixes

        AS numbers are represented as strings. In the event of a tie, the AS with the lower number should come first.

        For example: ["777", "1", "6"]

          corresponds to AS "777" as having the smallest percentage increase (of the top ten) and AS "6" having the

          highest percentage increase (of the top ten).

      """

    # the required return type is 'list' - you are welcome to define additional data structures, if needed

    top\_10\_ases\_by\_prefix\_growth = []

    data = {}

    for ndx, fpath in sorted(cache\_files):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "rib-file", fpath)

        counts = {}

        for el in stream:

            prefix = el.fields['prefix']

            if el.fields['as-path'] == "": continue

            origin = el.fields["as-path"].split(" ").pop()

            tmp = counts.get(origin, set())

            tmp.add(prefix)

            counts[origin] = tmp

        for key in counts:

            if key in data:

                data[key][1] = len(counts[key])

            else:

                data[key] = [len(counts[key]), 0]

    for key in data:

        if data[key][1] == 0: continue

        perc\_diff = (float(data[key][1]) / float(data[key][0])) - 1

        top\_10\_ases\_by\_prefix\_growth.append([key, perc\_diff])

    top\_10\_ases\_by\_prefix\_growth.sort(key=lambda r: -r[1])

    return [r[0] for r in top\_10\_ases\_by\_prefix\_growth[:10]]

# Task 2: Routing Table Growth: AS-Path Length Evolution Over Time

def shortest\_path\_by\_origin\_by\_snapshot(cache\_files):

    """

    Compute the shortest AS path length for every origin AS from input BGP data files.

    Retrieves the shortest AS path length for every origin AS for every input file.

    Your code should return a dictionary where every key is a string representing an AS name and every value is a list

    of the shortest path lengths for that AS.

    Note: If a given AS is not present in an input file, the corresponding entry for that AS and file should be zero (0)

    Every list value in the dictionary should have the same length.

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A dictionary where every key is a string representing an AS name and every value is a list, containing one entry

        per file, of the shortest path lengths for that AS

        AS numbers are represented as strings.

        For example: {"455": [4, 2, 3], "533": [4, 1, 2]}

        corresponds to the AS "455" with the shortest path lengths 4, 2 and 3 and the AS "533" with the shortest path

        lengths 4, 1 and 2.

    """

    # the required return type is 'dict' - you are welcome to define additional data structures, if needed

    shortest\_path\_by\_origin\_by\_snapshot = {}

    for ndx, fpath in enumerate(sorted(cache\_files)):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "rib-file", fpath)

        tmp = {}

        for el in stream:

            prefix = el.fields['prefix']

            if el.fields['as-path'] == "": continue

            ases = el.fields["as-path"].split(" ")

            origin = ases[-1]

            unique\_ases = set(ases)

            if len(unique\_ases) == 1:

                continue

            if len(unique\_ases) < tmp.get(origin, float('inf')):

                tmp[origin] = len(unique\_ases)

        for key in tmp:

            if key not in shortest\_path\_by\_origin\_by\_snapshot:

                shortest\_path\_by\_origin\_by\_snapshot[key] = [0 for \_ in range(len(cache\_files))]

            shortest\_path\_by\_origin\_by\_snapshot[key][ndx] = tmp[key]

    return shortest\_path\_by\_origin\_by\_snapshot

# Task 3: Announcement-Withdrawal Event Durations

from collections import defaultdict

def aw\_event\_durations(cache\_files):

    """

    Identify Announcement and Withdrawal events and compute the duration of all explicit AW events in the input BGP data

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A dictionary where each key is a string representing the address of a peer (peerIP) and each value is a

        dictionary with keys that are strings representing a prefix and values that are the list of explicit AW event

        durations (in seconds) for that peerIP and prefix pair.

        For example: {"127.0.0.1": {"12.13.14.0/24": [4.0, 1.0, 3.0]}}

        corresponds to the peerIP "127.0.0.1", the prefix "12.13.14.0/24" and event durations of 4.0, 1.0 and 3.0.

    """

    # the required return type is 'dict' - you are welcome to define additional data structures, if needed

    aw\_event\_durations = {}

    nested\_dict = lambda: defaultdict(nested\_dict)

    data = nested\_dict()

    for ndx, fpath in enumerate(sorted(cache\_files)):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "upd-file", fpath)

        stream.add\_filter("ipversion", "4")

        for record in stream.records():

            for el in record:

                addr = el.peer\_address

                prefix = el.fields["prefix"]

                if el.type == "A":

                    data[addr][prefix] = el.time

                if el.type == "W" and prefix in data[addr]:

                    time\_diff = el.time - data[addr][prefix]

                    if time\_diff > 0:

                        tmp = aw\_event\_durations.get(addr, {})

                        tmp[prefix] = tmp.get(prefix, []) + [time\_diff]

                        aw\_event\_durations[addr] = tmp

                    del data[addr][prefix]

    return aw\_event\_durations

# Task 4: RTBH Event Durations

def rtbh\_event\_durations(cache\_files):

    """

    Identify blackholing events and compute the duration of all RTBH events from the input BGP data

    Identify events where the prefixes are tagged with at least one Remote Triggered Blackholing (RTBH) community.

    Args:

        cache\_files: A chronologically sorted list of absolute (also called "fully qualified") path names

    Returns:

        A dictionary where each key is a string representing the address of a peer (peerIP) and each value is a

        dictionary with keys that are strings representing a prefix and values that are the list of explicit RTBH event

        durations (in seconds) for that peerIP and prefix pair.

        For example: {"127.0.0.1": {"12.13.14.0/24": [4.0, 1.0, 3.0]}}

        corresponds to the peerIP "127.0.0.1", the prefix "12.13.14.0/24" and event durations of 4.0, 1.0 and 3.0.

    """

    # the required return type is 'dict' - you are welcome to define additional data structures, if needed

    rtbh\_event\_durations = {}

    nested\_dict = lambda: defaultdict(nested\_dict)

    data = nested\_dict()

    for fpath in sorted(cache\_files):

        stream = pybgpstream.BGPStream(data\_interface="singlefile")

        stream.set\_data\_interface\_option("singlefile", "upd-file", fpath)

        stream.add\_filter("ipversion", "4")

        for record in stream.records():

            for el in record:

                addr = el.peer\_address

                prefix = el.fields["prefix"]

                if el.type == "A":

                    comms = el.fields["communities"]

                    if "666" in " ".join(comms):

                        data[addr][prefix]["start"] = el.time

                    elif prefix in data[addr]:

                        del data[addr][prefix]

                if el.type == "W" and prefix in data[addr]:

                    time\_diff = el.time - data[addr][prefix]["start"]

                    if time\_diff > 0:

                        tmp = rtbh\_event\_durations.get(addr, {})

                        tmp[prefix] = tmp.get(prefix, []) + [time\_diff]

                        rtbh\_event\_durations[addr] = tmp

    return rtbh\_event\_durations

# The main function will not be run during grading.

# You may use it however you like during testing.

#

# NB: make sure that check\_solution.py runs your

#     solution without errors prior to submission

if \_\_name\_\_ == '\_\_main\_\_':

    # do nothing

    pass