CSE3150

Course Project: Building a BGP Simulator

Dr. Justin Furuness

Deadline

December 4th at 11:59PM. NO RESUBMISSIONS AFTER THIS POINT. NO EXTENSIONS.
For Extra Credit only: Dec 2nd at 11:59PM. NO RESUBMISSIONS AFTER THIS POINT. NO EXTENSIONS.

1 Extra Credit

1.1 EC Opportunities

NOTE: for eligibility for extra credit, you must submit your assignment by Dec 2 11:59PM.

- Top 3 fastest programs will get an A in the course and can skip the final (please document your design decisions for the speedups you made)
- The top 10 fastest programs will get +3 on their final grade in the course (please document your design decisions for the speedups you made)
- Reimplement this entire project in C (3 points on your final class grade)
- Reimplement this entire project in Rust (3 points on your final class grade)
- Using WASM, put this simulator on a website with a real domain name so that the simulations can be run purely client-side. I suggest using Cloudflare pages for this. If you configure it correctly, you can just pay for the domain name (10 bucks a year) and Cloudflare will automatically deploy the site every time you push to your GitHub. Ask AI how to do this. The website also should not look like garbage, make it look at least somewhat nice. Many bosses you have will not be programmers, and the presentation matters a lot. You should be able to feed the website with a CSV of announcements, and a target ASN, and get back the announcements with their AS paths at that given AS. NOTE: a bad website that breaks sometimes will result in only partial credit for this. (3 points)

- If you can think of something cool let me know and maybe we can have it as extra credit.

1.2 EC Overview

NOTE: for eligibility for extra credit, you must submit your assignment by Dec 2 11:59PM.

The point of the extra credit is two fold. For those shooting for the fastest program, you'll learn how to make your code significantly faster and this will be a great learning exercise. For those looking to do C/Rust/UI, you'll learn how great AI is once you understand a language from a high level. Whatever you don't understand, AI can explain it to you and help fill in the gaps.

These are hard... but if you are good at using AI, they are easy (since you'll have written the C++ version for the AI to work with) and will take no longer than a day or two. That is the power of agentic AI that you will be able to unlock if you are a great programmer. If you want to work at great companies, this is the level of programming that they will expect from you. Language conversion is a trivial task for AI with the right prompter.

A few things to note:

- If you're paying for the cheap version, they rate limit you, so you'll want to spread your prompts out over a long time period unless you plan on manually coding these.
- The speed needs to be somewhat comparable to the basic C++ version (aside from the WASM code), the AI shouldn't be writing slop
- Rust as a language discourages the use of OO and cyclical references. Feel free in these areas to use the unsafe keyword for the purposes of this project if that's the design pattern you went with for your program. But don't write the entire thing using gigantic unsafe keywords around your base C++ model.
- Don't forget: Dec 2nd 11:59PM is the deadline for these.

Overview

In this assignment you'll be building a BGP (internet) Simulator. This is a unique problem that will involve much of the concepts that we've gone over in class. This will also be semi-open ended, just like problems that you'll encounter at your work. Everything will be manually reviewed, since this is too big of a project to do any other way, and the datasets that we will be using will be too large to have some type of API/automated test suite for them. In the final sections you'll be given a system test to run just to make sure that your output will be compatible with what I'm looking for, but beyond that you'll write your own tests. In fact, you'll write every single part of this assignment yourself without any starter

code given.
My personal recommendation is to start as early as possible, and doing it last minute will definitely be difficult. For those of you that have been using AI on everything, here it will likely fail you unless you prompt it well. AI tends to get worse as the context grows, especially when you get into large projects with many files. Additionally, AI gives general advice and is essentially a syntax completion engine. With open ended projects, it will often choose the wrong design patterns. Additionally, there are several optimizations that can be made around the problem set you are dealing with that AI will not have been trained on. That is part of the point of this project, at this point you should be able to know more than the AI can do! That's what makes you valuable as a C++ developer.

Goal

Note: If this (the goal section) is confusing, just ignore it and move to step 2. I was just trying to give an example of how this problem might be given to a developer, it is not relevant towards the completion of the course project.

You are a C++ Developer that has been hired by Cloudflare, a leading CDN (content delivery network). They want to optimize the announcements (data) that is sent to the rest of the internet so that more traffic from the internet flows through their network. In order to make these decisions, they need to be able to know, at any given AS, what the AS-Paths are for that ASes announcements.

To do this, your manager has given you the task of building a BGP Simulator. You'll use CAIDA's topology information to build the AS graph. You'll build out basic BGP policies. You'll then pull down the 4% of public AS data that will be given to you (up to 100GB of data), and simulate what you expect the other 96% of private ASes data to look like. This data will then be dumped into a CSV, which your boss will then use to make more optimal routing decisions.

2 Building the AS Graph

2.1 How the AS Graph is structured

The internet is a directed acyclic graph that is comprised of about 100,000 nodes called Autonomous Systems (or ASes for short). All the ISPs you know about likely own at least one of these nodes, such as Spectrum, Comcast, Verizon, TMobile, AT&T, etc.

These ASes connect in two types of ways: Provider to customer relationships, and peer to peer relationships. You can think of these relationships as edges in your AS Graph. In a provider to customer relationship, the customer pays the provider to send it's traffic to the

internet. In a peer to peer relationship, traffic flows freely between the two ASes (nodes). You can see an example of this here: `bgpsimulator.com/example`.

Here AS 666 and AS 3 are customers of AS 4. That means AS 4 is a provider of AS 666 and AS 3. AS 4 is a peer of AS 777, and AS 777 is a peer of AS 4. Each of these ASes can be identified by their AS number, or their ASN for short. This acts like an ID for the AS, they are unique and are a maximum of 32 bits, according to some BGP RFCs.

2.2 Downloading the AS Graph information

Our first goal is going to be to build this AS Graph. For this, we will use a dataset provided by CAIDA. While this isn't truly accurate, it's the current best approximation that we have of the AS topology (by topology, I'm referring to the AS graph). First, have your code download the latest data from caida. Use the URL for the month prior, since CAIDA doesn't always release their datasets on the first day of the month. You're welcome to use libcurl for this, or some other solution that you may uncover later. Remember, your design decisions will go beyond the instructions given. Think, if you wanted to run the final program once a day, what are some of the best practices that you can use with this code?

2.3 Reading the AS Graph information and building the AS graph

After you've download the file from CAIDA, now you need to build the graph. There are a few steps involved in this.

- First you'll need to extract the provider to customer relationships from the file, along with the peer to peer relationships. Check out the format in the URL itself (note: we'll ignore the source column).
- Second, you'll need to build the AS graph. There are several ways to do this, but at this stage, you'll need to keep track of at least four things:
  - The Autonomous System Number (ASN), which acts as the ID for the node
  - The providers that the AS (node) has (if any)
  - The customers that the AS (node) has (if any)
  - The peers that the AS (node) has (if any)
- Then you'll need to check, specifically, that there are no provider cycles, or customer cycles in the provided input (peer cycles are okay and are expected)
- Building a graph of 100k nodes with 500k edges takes time. Consider how to make this portion faster as well.

- My personal recommendation is to start this portion with smart pointers, build out the test suite, and then come back later to optimize it once you have your tests and have built out more of the program.
- You're going to want to come back to this portion as the project progresses, since the way that the AS graph is structured is key for speeding up the program.

2.4 Testing the AS Graph

You must test your code! This way you can make sure that you're code isn't breaking whenever you go back and make refactors. Typically for something with a variable input, you'll want some unit tests for certain parts of the graph creation. You'll also want some tests were you create your own mini-graph input and output so that you can see your code is mapping these inputs and outputs correctly (these are often called system tests, but you can use the same testing framework for them).

3 BGP Functionality

3.1 Announcements

Next, we'll focus on propagating data throughout the AS graph. Each AS (node) on this graph owns a block of IP addresses. When you go to a website such as `google.com`, under the hood `google.com` is translated to a 32 bit IPv4 address, or a 128 bit IPv6 address (which is just a binary address on the internet). Every internet service provide needs to know how to get to that IP address, and they need to know which AS (node) on the internet owns that IP address.

To let the internet know which IP addresses it owns, Google will announce a range of IP addresses. For example, Google's AS 15169 owns all of the IP addresses from 8.8.8.0 to 8.8.8.255 (and it's very common for much of the internet to use Google's DNS server at 8.8.8.8). The format for such a block of IP addresses is 8.8.8.0/24, which represents that the first 24 bits of the 32 bit IP address are specified, and the last 8 bits are not specified and fall under the range that is owned by Google. Keep in mind this is an IPv4 prefix, and there are also IPv6 prefixes, and you'll need to support both.

When Google wants to let the internet know "hey I own all IP addresses from 8.8.8.0 to 8.8.8.255", Google will announce 8.8.8.0/24 in a BGP announcement from their AS. Each announcement will have (for this exercise, which is only looking at BGP from a high level), four attributes, listed below:

- A prefix (such as 8.8.8.0/24)

- An AS-Path (at the start, just Google's ASN of 15169)
- The next hop ASN (where the announcement just came from. Initially also Google's ASN of 15169)
- The received from relationship (if the announcement came from a customer, a peer, or a provider).

While our final product will simply need to look at prefixes and AS Paths at each ASN, we will use these other attributes for announcement comparisons.

The prefix is the block of IP addresses that Google owns. The AS-Path is the path that the announcement has taken to get to that point. Every node that the announcement travels to adds itself to the AS-Path. The next hop ASN is where the announcement just came from (the ASN). And the received from relationship is whether it came from a provider, peer, or customer.

You can see a very simple example of this on bgpsimulator.com's home page. I would turn the propagate packets off (a checkbox in the top right corner), since that's not something that we will be covering during this exercise. You can see that AS 777 announces 1.2.0.0/16. At AS 3, the AS-Path becomes 3-777, since each AS prepends itself to the path as the announcement travels.

Go ahead and to start with, create the Announcement struct or class, and feel free to change this out later if need be.

3.2 Storing announcements

ASes store these announcements in what's called the local_rib. From a high level you can think of the local RIB of just a hashmap with prefixes as the keys, and the values being the announcements themselves. Each AS will only every have one entry per prefix.

For our simulator, we not only need to be able to store these announcements, but we'll also need to be able to make decisions off of them. For that reason, a nice place to start is a separate class that will store the announcements and handle these decisions. We can start with an abstract class Policy, and a subclass called BGP. Every AS should have a pointer to a concrete Policy object. Each BGP object can hold a local RIB, which is the data structure we described above. Go ahead an also add a received_queue to this BGP object as well, which will be of the format of a hashmap where the keys are the prefix, and the values are a list of all the announcements that we have received for each prefix.

3.3 Propagating Announcements: Flattening the graph

Now we'll need to start an announcement at one AS (like Google's AS) and propagate it throughout the rest of the internet. To do that, we first need to flatten our graph. This is a

common technique that will turn our DAG into a vector of vectors, where the lowest vector at index 0 represents the edges (who have no customers), and the vectors get higher and higher up the provider chain.

To do this, get all ASes that have no customers and assign them all a rank of 0. Then, for each AS with a rank of 0, assign all of their providers a rank of 1. For each AS with a rank of 1, assign all of their providers a rank of 2. Keep doing this until you have reached ASes that have no providers anymore (remember, there are no cycles).

Then take these assigned ranks, and store pointers or ASNs (anything representing these ASes), in a vector of vectors, where each inner vector represents the rank of those ASes. We can also add this new attribute to each AS if you'd like of propagation_rank.

3.4 Propagating Announcements: Seeding the graph

Now we can start to seed the AS Graph. Go ahead and make a random announcement (perhaps 1.2.0.0/16 with a next hop of 1, an AS-Path of [1], and a received_relationship of origin (which should trump customers, providers, and peers, origins are always preferred the most). Store this announcement in the local_rib of an AS. Congrats, you've now seeded the AS Graph.

3.5 Propagating announcements: The propagation path

From a high level, announcements go up all the way, then across one hop, then down all the way.

Now when we propagate this announcement, we first want to send it up the AS Graph. Starting at the propagation rank 0, for each AS, if they have announcements, they must send those announcements to their providers. Their providers (at this point) can store the announcements in their received_queue. Then we go to rank 1. For anyone that has announcements in their received_queue, process and store them.

When we are processing the announcements, we want to create a new announcement that may potentially be stored. This announcement should have the new received_from_relationship, as well as the current ASes ASN prepended to the AS Path. Ignore conflicts for now, and simply store these announcements in the local RIB (where the key is the prefix and the value is the announcement). After storing the announcements, clear the received_queue. Then for each stored announcement in the local RIB, send to all providers of those ASes. Repeat this process of processing, storing, and sending all the way up the ranks.

Now we want to send across the AS Graph. Unlike when we go up or down, we only go one single hop across to our peers, never multiple hops (which prevents cycles/valleys, also known as valley free routing). Don't use your propagation ranks during this step since those are for customer provider relationships. Instead, simply propagate from all ASes.

After propagating from all ASes, then process from all ASes, storing the announcements and clearing the received_queue. It's critical that this be done in this order (first all ASes send, then all ASes process and store) to avoid announcements traveling multiple hops.

Lastly, we send the announcements back down. This is the reverse of the first step (sending to providers). Now start at the highest propagation rank (NOT rank 0, the opposite of rank 0). For each AS, send all announcements to their customers, which should store them in the received_queue. Then go down one rank. The rank should process and store the announcements (ignoring conflicts for now), then clear the received_queue, and then repeat the process by sending all stored announcements to the next rank down. Repeat this process until you have gone down through all the propagation ranks.

Remember also that every time you send an announcement, you should be setting the next_hop_asn to the ASN that is sending the announcement. And every time you store an announcement, you should be prepending the ASN that is doing the storing to the AS-Path of the announcement.

Any time you are confused as to how this works, each AS displays their local RIB on bgpsimulator.com. You can build any example you like by playing with the Graph Editor and Scenario Editor. Remember to turn the "Propagate Packets" checkbox off, as we are not propagating packets but announcements.

3.6 Propagating announcements: Dealing with conflicts

So what happens then, when an AS receives two announcements for the same prefix? Let's say the AS already has an announcement for 1.2.0.0/16 in it's local RIB and it receives another announcement for 1.2.0.0/16. The AS prioritizes which announcement will make them the most money, which works as follows (from a high level for this project):

- Choose the announcement with the best relationship. Customers > Peers > providers.
- If the relationship is the same, choose the announcement with the shortest AS path.
- If the AS path length is the same, choose the announcement with the lowest next hop ASN

You can see an example of this at AS 4 in the bgpsimulator.com home page (remember to turn off propagate packets in the top right corner, those are packet animations not announcements). AS 4 receives an announcement from AS 3 and from AS 666 for the same prefix. Both are customers, so AS 4 chooses the announcement from AS 666 which has a shorter AS Path length.

3.7 Output and tests

Once we have reached the end of our program, the final output will be for you to dump the AS graph into a CSV, which has the columns of "asn", "prefix", and "as_path". This is what Cloudflare will use to figure out how to optimize their network. We'll get into how to feed in large amounts of data into this program in a minute, but for now, try just seeding in one announcement with a tiny test graph and making sure things are working properly. Then try a larger test graph. Then try two announcements, announced from different ASes, for the same prefix. These are some useful test cases (and remember, you can compare to bgpsimulator.com if need be to determine what ground truth would look like). You can also add more.

3.8 Congrats!

Congrats! You've now built a very basic, working (hopefully), BGP simulator. We will extend this functionality a tiny bit from here, but this is the gist of what the program will look like. Pat yourself on the back!

4 Adding ROV functionality

Now of course, it isn't great that any AS on the internet can simply announce that they own Google's prefix! If you look at bgpsimulator.com, AS 666 has done what is called a prefix hijack. Even though AS 777 is the correct owner of the prefix 1.2.0.0/16, AS 666 has announced the same prefix, and captured much of the traffic on the internet!

We won't get into how defending against this works, but we will simulate it from a high level. Each announcement can have an attribute called rov_invalid. Additionally, some ASes will deploy a defense called ROV. When an ROV AS receives an announcement with rov_invalid attribute set to True, the AS will immediately drop the announcement and never store it.

There are several ways of doing this. One object oriented way is to have your AS class, and your AS object will point a policy object called BGP, which handles the local RIB, announcement storage, etc. You can inherit from BGP and create a new policy class called ROV, which can extend BGP with this new defense. Your program should be able to take in, when it runs, text file where each line represents an ASN deploying ROV.

That's it for this section, and that's it for the BGP functionality of the simulator. Make sure to test this.

5 Usability

Now we need to make this program usable. Like almost all C++ programs out there, we need to wrap this in something that makes it easy to run. To do that, go ahead and wrap this in Python using the C++ bindings that we talked about in class. You can replace certain parts of your program like the downloading of the relationship dataset with a basic Python program. You can likely also replace a lot of the testing code with Pytest as well. The parts that must be fast, leave them fast. The parts that do not require speed, feel free to switch them out to Python to make them more user friendly and easier to maintain. This is very common in these types of programs.

6 Docs

Make sure every design decision that you make is well documented in your README.md, and that your program has decent documentation in general. Every place where you are not fast, your program should be clean and user friendly.

7 Data set

I will be giving you several files for you to use as your test. First is the announcements CSV, which will have the ASN, prefix, and rov_invalid boolean. Then there is the ROV ASNS csv, which will have all of the ASNs that are meant to deploy ROV. And finally, I will give you what your output should be equivalent to, ribs.csv. To compare your output to the files, use the compare script included in the datasets. Your program should be able to ingest these files, seed the announcements, propagate them, and then output the CSV that it already does with the "asn", "prefix", and "as_path" columns. Your program should be able to handle these datasets. Whatever is not optimized should be clean and well designed. And of course, make sure that your program is accurate.

The one test that will not be given in advance is the cycle test. Your program should output a reasonable print statement any time that a cycle is detected in the AS relationships and end the program.

Note: If you're shooting for speed, the only thing I care about is that my input produces the desired output. You can change anything you want in-between.

8 Hardware Requirements for speed tests

Your code should use no more than 2 CPUs, and no more than 8GB of RAM. While in theory we could handle much larger datasets (especially as you optimize the program), we'll

stick with the datasets that are given to everyone for now.

For the speed tests we'll just use the provided files, unless I find that some people have hardcoded their programs to them, in which case I will generate different datasets to test over.

9 Submission

See the prior two sections above and make sure your program is compatible. If your program fails any tests, you'll be given a zero, as your simulator does not work. If you pass the tests, and you are using C++ bindings, you will get a 100%. Extra credit points may be awarded beyond that (see the section on Extra Credit). To submit, just zip your program up and put it on HuskyCT under the course project folder. If you'd like you can submit early and I can grade it early as well since it should be pretty quick.