Practical Perl Tools

Come Fly With Me

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to 30 years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'. dnb@usenix.org

I occasionally read Quora for fun. I recently stumbled upon the following question:

> What are the two closest airports to each other in the world?

The very first answer to the question I saw was from Kevin Lin who said:

> For fun, I wrote a Python script to do the following:
>
> (1) Take the list of airports from http://www.airportcodes.org/ and remove all the airports listed as "Bus service" or "Rail service" or "Van service" or "All airports".
>
> (2) Plug the remaining airports into http://www.gpsvisualizer.com/geo...to get their GPS coordinates.
>
> (3) Finally, compute the distances between pairs of airports by plugging their GPS coordinates into the haversine formula http://stackoverflow.com/questio...

You can find this question and answer here: https://www.quora.com/What-are-the-two-closest-airports-to-each-other-in-the-world/answer/Kevin-Lin.

Lin didn't include his Python code, but I was intrigued by the problem and thought I would take a swipe at doing this in Perl using roughly his method to see how hard it would be. Turns out it isn't that difficult, though there are a few tricky bits and some limitations we'll discuss later on. Let's take a walk through my implementation of Lin's solution and see what we can learn.

Oh, the Modules You Will Go

I don't know how hard Lin's implementation leans on existing extensions to Python, but since the availability of modules to do almost anything is one of Perl's strengths, I decided it would be fine to use them whenever they would make things easier for me. Here's the collection in play:

```
use HTTP::Tiny;
use HTML::Strip;
use Geo::Coder::Google;
use Algorithm::Combinatorics qw(combinations);
use GIS::Distance;   #::Fast
```

The first two will be used to grab the airport list Web page and remove all of the HTML from it. The second will be used to geolocate all of the airports. Algorithm::Combinatorics will make it easy to come up with all of the distances we will need to compute, and GIS::Distance will perform that calculation for us. The comment on GIS::Distance is meant to be a reminder that it would be advantageous to us to also install GIS::Distance::Fast in addition to GIS::Distance. The Fast module implements the distance calculations in C (versus the pure Perl implementations that ship with the main module). These much faster implementations will get used by GIS::Distance automatically if the Fast module has been previously installed.

Let's Get the Airports

Here's some code to fetch the contents of the page, strip off the HTML in the page, and then extract a list of all of the airports from the remaining text:

```
my $code_source = "http://www.airportcodes.org";
my $reply = HTTP::Tiny->new->get($code_source);
my $hs = HTML::Strip->new();
my @airports =
  grep ( /\w, [\w\s-]+\s?\(/,
    ( split( /\s?\n/, $hs->parse( $reply->{content} ) ) ) );
```

That last line is kinda gnarly (sorry), so let's take it apart piece by piece, working from the inside out.

First off, we need the contents of the page as returned by the HTTP GET operation:

```
$reply->{content}
```

Then we will want to strip out any of the HTML tags in the page:

```
$hs->parse( $reply->{content} )
```

Now that we have just the text, which largely consists of a string containing a bunch of lines (most of which contain an airport), we'll want to split the text into a list of lines:

```
split( /\s?\n/, $hs->parse( $reply->{content} ) )
```

With me so far?

As an aside, the use of `\s` in the `split()` takes care of an annoying property of the data where some of the airport listings have a trailing space. Mostly a cosmetic problem, but it was bugging me while I was writing the code. A few seconds ago I said "most of which contain an airport." The use of `grep()` here makes sure we only collect the lines that appear to contain an airport listing:

```
  grep ( /\w, [\w\s-]+\s?\(/,
    ( split( /\s?\n/, $hs->parse( $reply->{content} ) ) ) );
```

I suspect there are more direct ways to extract only the airport data from this page using one of the HTML-parsing/extraction modules, but this method of tossing the HTML and grabbing only the lines we wanted seemed relatively straightforward.

Let's Geocode

We've dived into Geocoding in previous columns a couple of times, so I won't dwell too much on the process. One thing I do need to note is that in this example code, I'm using the Google Maps API Geocoding service, which is (after a certain number of calls) a paid service. More info on it here: https://developers.google.com/maps/documentation/geocoding/start (pricing can be found here: https://developers.google.com/maps/pricing-and-plans/).

Let's look at the code:

```
my $geo =
  Geo::Coder::Google->new( "key" => "{YOUR API CODE HERE}",
  );

my %airports;

foreach my $airport (@airports) {
    next if $airport =~ /[vV]an service/;
    next if $airport =~ /[bB]us service/;
    next if $airport =~ /[bB]us station/;
    next if $airport =~ /Park&Ride Bus/;
    next if $airport =~ /Van Galder Bus/;
    next if $airport =~ /[rR]ail service/;
    next if $airport =~ /[aA]ll airports/;
    next if $airport =~ /Heliport/;
    print STDERR "Locating $airport...";
    my $location = $geo->geocode( 'location' => $airport );

    if ( !defined $location ) {
        print STDERR "not found.\n";
        next;
    }

    $airports{$airport} = [
        $location->{geometry}{location}{lat},
        $location->{geometry}{location}{lng}
    ];
    print STDERR "done.\n";
}
```

I think the process is pretty straightforward. Once we initialize the geocoded object with our API key (see the doc I mentioned earlier for how to get one), we walk through the list of airports we scraped and attempt to geocode each one. As per Lin's solution, there are a number of bus and van service listings that aren't real airports, so we attempt to skip them.

As an aside, there's another thing I would probably do in the next version of this program to clean the data that Lin doesn't mention. There are (by my count) 59 duplicates in the data where the same airport code is listed in two places with slightly different descriptions—for instance:

```
Biloxi/Gulfport, MS (GPT) & Gulfport, MS (GPT)
Endicott, NY (BGM) & Binghamton, NY (BGM)
Leon, Mexico (BJX) & Guanajuato, Mexico (BJX)
Canton/Akron, OH (CAK) & Akron/Canton, OH (CAK)
```

It would be very simple to extract the airport code from each airport and store it in a hash after you attempt to geocode an airport. Then, before geocoding the rest, just skip any airports you previously have a hash entry for already. I leave this (and any other data cleanup you want to do) as an exercise for the reader.

Back to the action. For each airport, we store its latitude and longitude if the geocoder can find them. In my experience, it finds a very large percentage of the airports. If this were a setting where I really cared deeply about the results, I might choose to call a second geocoding service to attempt to find any that Google doesn't have listed.

The lookup (at least from my laptop and decent Internet connection) on average takes about a second or so to complete for each airport. If we wanted to speed this whole thing up, we could use one of the techniques we've discussed in past columns to make a number of queries in parallel. I have no doubt that Google can handle the multiple queries at once, so this would provide a dramatic speedup.

And while we are discussing optimizations, an even better addition would be code that could avoid doing the geolocation at all. It would be best to cache previous results we get back and drop them into some sort of persistent store (even just to a flat file). When we ran the program again, we could skip a query if we've already made it. This would save time, save you money from API calls, and speed things up tremendously on future runs. Given how seldom airports move locations, this is probably a safe thing to do in almost all cases.

Go the Distance

Okay, time to calculate the distance between every possible pair of airports. This process consists of determining all of those pairs and then computing the distance for each.

Figuring out the pairs is something we could do with some loops, but instead let's use this opportunity to learn about two of the easier modules for this process: Algorithm::Combinatorics and Math::Combinatorics. Both have an easy way to ask for all of the combinations of list elements. I choose the former because it uses some C extensions for speed, but if you need a pure Perl solution, Math::Combinatorics will work as well.

Algorithm::Combinatorics' combinations() subroutine will hand us back an iterator. We just call next() on that iterator each time we want a new pair of airports (when it runs out of pairs, it returns undef):

```
my $pairs = combinations( [ keys %airports ], 2 );

my %distances;

my $gis = GIS::Distance->new();

while ( my $pair = $pairs->next ) {

    my $trip = $pair->[0] . '-' . $pair->[1];
```

Above we snuck in the initialization of the GIS::Distance object, so let's talk about that next. There are a number of different ways to compute distance between two points, the most common is the haversine formula. So sayeth Wikipedia:

> The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes.

(Be sure to check out the Wiki page on this for some other interesting trivia.)

By default, GIS::Distance uses this formula by default. Calculating the distance between the two airports becomes this easy:

```
print STDERR "computing distance between $trip...\n";
$distances{$trip} = $gis->distance(
    $airports{ $pair->[0] }->[0],
    $airports{ $pair->[0] }->[1] =>
    $airports{ $pair->[1] }->[0],
    $airports{ $pair->[1] }->[1]
)->{values}->{kilometre};
```

We just ask the module to compute the distance between the pair of airports by feeding in the latitude and longitude of the first airport followed by the same for the second airport. GIS::Distance wants to hand us back a Class::Measure object (which could be handy later if we wanted to do conversions), but we immediately look up the actual value in kilometers and store it in the %distances hash instead.

Show Me the Distances

The last piece of code prints out the results (all 5,016,528 of them) sorted from shortest distance to longest distance. This was, by the way, the moment I realized that there were duplicate entries in the data as mentioned above. Finding two airports with 0 distance between them seemed mighty suspicious. Here's the code:

```
foreach my $trip (
  sort { $distances{$a} <=> $distances{$b} } keys %distances )
{
    print "$trip: $distances{$trip} kilometres\n";
}
```

And the Answer Is...

If you run the code, you get an answer. I find interesting that I got a slightly different answer from the one mentioned in Quora (though Lin's top answer is in the top five list). Here are the airports with the shortest distance between them:

```
Comox, BC (YQQ)-Vancouver, BC (YVR): 1.30501111815652
  kilometres
Vancouver, BC - Coal Harbour (CXH)-Comox, BC (YQQ):
  1.37633222675128 kilometres
Vancouver, BC - Coal Harbour (CXH)-Vancouver, BC (YVR):
  2.11243449299644 kilometres
Omsk, Russia (OMS)-Orsk, Russia (OSW): 2.28071127591897
  kilometres
```

```
Port Protection, AK (PPV)-Point Baker, AK (KPB):
   2.68865415751707 kilometres
Lebanon, NH (LEB)-White River, VT (LEB): 2.80395259148673
   kilometres
```

And just for the sake of completeness, here are the top five longest distances:

```
Rio Cuarto, CD, Argentina (RCU)-Fuyang, China (FUG):
   19993.286433724 kilometres
Padang, Indonesia (PDG)-Esmeraldas, Ecuador (ESM):
   19994.1381628879 kilometres
Ile Des Pins, New Caledonia (ILP)-Zouerate, Mauritania (OUZ):
   20000.9443793096 kilometres
Long Lellang, Malaysia (LGL)-Tefe, AM, Brazil (TFF):
   20002.7713227265 kilometres
Palembang, Indonesia (PLM)-Neiva, Colombia (NVA):
   20011.325933595 kilometres
```

If you do happen to fly any of these distances, do write me, I'd love to hear about it. And with that, take care, and I'll see you next time.