DAVID N. BLANK-EDELMAN

practical Perl tools: let me draw you a picture

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration*. He has spent the past 20+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

A COUPLE OF YEARS AGO I HAD THE unusual experience of being asked to help design my office when we moved to a new building. Interior design is not something I've ever really dabbled in, but I knew one thing for sure: It had to have as many whiteboards as possible. Readers of this magazine know that I asked for this not out of some fetish for white, slick surfaces. For people like us, drawing often equals thinking. We also know the value of drawing pictures to document infrastructure design, network configurations, data structures, and the lot.

Tools that can make drawing these pictures easier are great. Tools that will actually automate the process are even better. We're going to look at both kinds of tools in this column. I want to introduce you to two of my favorite Perl modules: GraphViz and Graph::Easy.

Pedant alert: The name of the AT&T package is Graphviz; the name of the Perl module that acts as a wrapper around Graphviz is called GraphViz (with a capital V). I don't know why the difference in capitalization is there; I can only assume it was an attempt to drive proofreaders batty.

GraphViz and Graphviz

GraphViz is an easy-to-use Perl module that provides a wrapper around the graph visualization package from AT&T. This package contains a number of programs, which they describe like this:

> The Graphviz layout programs take descriptions of graphs in a simple text language, and make diagrams in several useful formats such as images and SVG for Web pages, Postscript for inclusion in PDF or other documents; or display in an interactive graph browser. (Graphviz also supports GXL, an XML dialect.)
>
> Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

To use the Perl module, you will need to make sure that the Graphviz programs are installed and working on your machine. You can download the source code from http://www.graphviz.org if necessary, but it is pretty likely that there is a Graphviz package available for your operating system through your packaging/installation system of choice (.deb, .rpm, fink/macports, .exe, etc.). From that point on you can choose to ignore the native Graphviz text language (DOT) if you'd like and write only Perl code.

Let's look at the basics of this Perl code because there isn't very much beyond the basics you'll ever need to know to use the module effectively.

The first step is to create a GraphViz object. The creation step is rather important because it is the constructor call (i.e., new()) that determines the format of the graph. This format is passed in via parameters such as layout, as in:

```
my $graph = GraphViz->new( layout => 'neato' );
```

This says that the resulting graph will be processed using the neato algorithm. The Graphviz layout program neato creates spring model graphs (i.e., the ones that consist of balls attached together by lines, mimicking the old molecule-building kits you used in chemistry class). Other layout options include dot (for directed graphs, i.e., trees), twopi (for radial graphs), circo (for circular graphs), and fdp (for spring model graphs like neato but using a different algorithm). The figures in this column are created using the default dot algorithm (i.e., no layout parameter supplied).

By default the GraphViz module will create diagrams with arrows on the lines connecting the shapes on the graph. This can be changed by specifying a "directed" parameter whose value is 0 in the new() call. There are a number of other GraphViz options available in the new() call, so be sure to see the module's (and Graphviz's) documentation.

Once we have a GraphViz object we can start populating the graph. This is quite simple:

```
$graph->add_node('router'); # "router" is the name of that new node
```

If we were to ask GraphViz to create the graph at this point we'd get something quite Zen (see Figure 1).

FIGURE 1

If after years of making this diagram the center of your meditation practice you decide the shape should be a box instead of an oval, you would use this instead:

```
$graph->add_node( 'router', shape => 'box' );
```

Furthermore, if you'd prefer the picture in Figure 2 instead, a third attribute would be specified:

```
$graph->add_node('router', shape => 'box',
                           label => 'Big Blinky Important Thing');
```

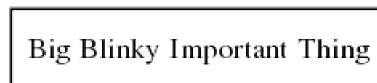

FIGURE 2

See the module documentation for other attributes that can be used to change how a node is displayed.

Now that you know how to make all of the shapes you want for your diagram, it is time to connect the dots, err, and nodes. That is done by calling add_edge() for each connection:

```
# add another node first so we have something to connect to
$graph->add_node('web server');
```

```
# connect the node with the name 'router' to the node named 'web server'
$graph->add_edge('router' => 'web server');
```

You probably can guess that there is a panoply of possible optional parameters we can use. For example, if we wanted to label the link between the router and the Web server with its connection type, that would be:

```
$graph->add_edge('router' => 'web server', label => '1000GB-FX');
```

which produces the output shown in Figure 3.

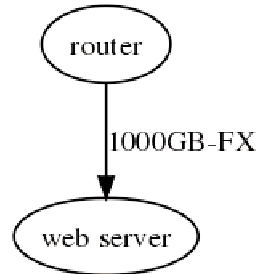

FIGURE 3

Other parameters let us set presentation attributes such as font, arrow size, and color and give hints to Graphviz about how to lay out the resulting graph.

We now know how to make nodes and how to connect them, but we haven't yet seen how to generate a graph that contains those nodes and connections. There are a number of methods that start with as_ for creating the actual graph. For example, as_gif() will create a GIF version, as_png() creates a PNG, as_ps creates Postscript, and so on. GraphViz supports a healthy number of output formats. It can also do neat tricks such as spitting out HTML image map tags.

With the as_* methods it is up to you to decide where the requested output goes. The as_* methods can take filenames, filehandles, references to scalar variables, and even code references if you want to feed the data to a subroutine. If you don't specify an argument it just returns the data, so you can say something like this:

```
print $graph->as_ps;
```

to print the generated Postscript file to stdout.

Congratulations! You have now learned everything you need to know to go off and start making interesting graphs of your own. To help jumpstart your creative process I'll show you one of my examples, and then we'll mention some GraphViz-related modules that can further spark your imagination.

Here's some code that attempts to sniff packets off the Net to show you the connections from hosts on your network to Web servers:

```
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP qw(:strip);
use NetPacket::TCP;
use Net::PcapUtils;
use GraphViz;

my $filt = "port 80 and tcp[13] = 2";
my $dev  = "en1";
my %traffic;    # for recording the src/dst pairs

die "Unable to perform capture:"
    . Net::Pcap::geterr($dev)
    . "\n"
```

```
    if (
    Net::PcapUtils::loop(
        \&grabipandlog,
        DEV        => $dev,
        FILTER     => $filt,
        NUMPACKETS => 50
    )
    );

my $g = new GraphViz;

for ( keys %traffic ) {
    my ( $src, $dest ) = split(/:/);
    $g->add_node($src);
    $g->add_node($dest);
    $g->add_edge( $src => $dest );
}
$g->as_jpeg("fig4.png");


sub grabipandlog {
    my ( $arg, $hdr, $pkt ) = @_;

    my $src = NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )
        ->{'src_ip'};

    my $dst = NetPacket::IP->decode( NetPacket::Ethernet::strip($pkt) )
        ->{'dest_ip'};

    $traffic{"$src:$dst"}++;
}
```

First we load up the modules we'll use for network sniffing and dissection plus GraphViz. We set a Berkeley Packet Filter (BPF) filter string to capture SYN packets (i.e., the start of a TCP/IP conversation) to the HTTP port. We set the device for capture and start a capture that will continue until it has received 50 packets. Each time a packet is captured by the filter it will call a subroutine called grabipandlog(). That subroutine takes each packet apart to find the source and destination IP addresses. It then stores a record for each unique source and destination IP address pair encountered.

After the capture has concluded it is a simple to pull all of the connection records out of traffic, adding a node for each source and destination IP address and connecting the two nodes. A graph is generated and written out as a JPEG file. Figure 4 is a simple example of what this program will draw.

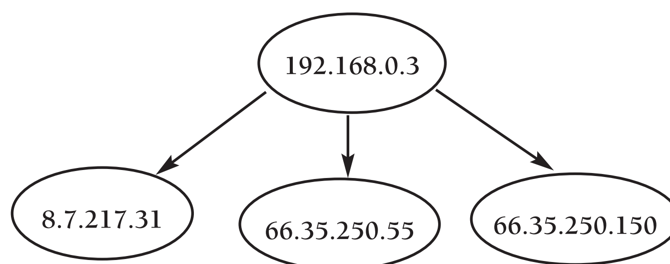

FIGURE 4

If we wanted to, we could make this code a little more complex by:

- thickening or labeling the links between the nodes to indicate amount of traffic or
- showing an internal vs. external Web server distinction by varying the node shapes.

Network traffic diagrams are just one application. The GraphViz module itself comes with several other examples. GraphViz::Data::Grapher and GraphViz::Data::Structure can help you understand complex Perl data structures using two different kinds of graphs. Here's a sample from the GraphViz::Data::Grapher examples directory:

Given the data structure defined this way:

```
@d = ("red",
     { a => [3, 1, 4, 1], b => { q => 'a', w => 'b'}},
     "blue", undef);
```

GraphViz::Data::Grapher will output the graph shown in Figure 5.

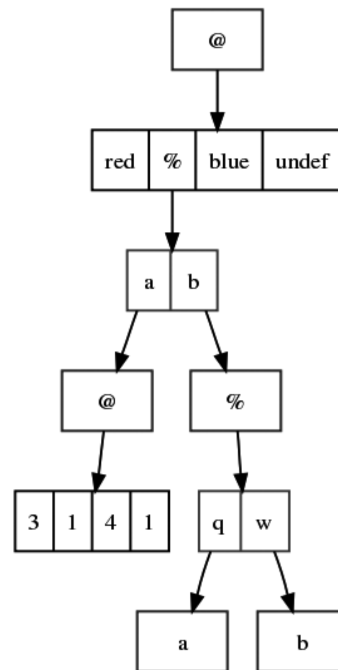

FIGURE 5

Outside of the GraphViz package, there are cool modules to visualize regular expressions, database schema, class diagrams, Makefile structures, parser grammars, XML code, and so on.

Graph::Easy, Baby

I'd like to show you one more package that is similar to GraphViz but is spiffier in a number of ways. Graph::Easy (which is well documented at http://bloodgate.com/perl/graph/manual/index.html) works with a similar idea to that of GraphViz but takes it even further. For example, in addition to writing Perl code like that we've seen for GraphViz, Graph::Easy can input and output data in Graphviz's native format. Being able to output DOT files means Graph::Easy can use Graphviz to create graphs in any graphics format Graphviz supports. Graph::Easy also has a really legible text format it will happily parse to create a graph. Let's look at the Perl and the plain-text method for graph creation.

Here's some sample Perl:

```
use Graph::Easy

my $graph = Graph::Easy->new();
$graph->add_edge ('router', 'web server', '1000GB-FX');
print $graph->as_ascii();
```

This code shows that the general approach for graph specification in Perl is very similar to our previous examples but is a bit more compact. Note that we didn't have to add_node() before creating a connection. We just specified that there was a link between two nodes called "router" and "web server" and that this link should be labeled with "1000GB-FX." Following that specification is a method call not found in GraphViz: as_ascii(). This produces an ASCII drawing like the following:

```
+--------+  1000GB-FX  +------------+
| router | ----------> | web server |
+--------+             +------------+
```

If you've ever wanted to make an ASCII flowchart for documentation purposes, now you know an easy way to do it.

I could go on and on about the additional graph features Graph::Easy provides (e.g., multiple links between two nodes, links that loop from a node back to itself, links that can point to other links, the ability to create links that fork in two different directions, node groups, more colors and styles, etc.) but I'd like to get to an even more interesting feature I mentioned earlier. Graph::Easy lets you specify graphs using a very easy-to-read text format.

If we wanted to reproduce the simple "two nodes with a link" example that has dogged our every step in this column, we could write:

```
[ router ] -- 1000GB-FX --> [ web server ]
```

If we decided the picture made more sense with a bidirectional link, it then becomes:

```
[ router ] <-- 1000GB-FX --> [ web server ]
```

You can specify more complicated pictures equally easily. For example, the doc shows this example:

```
[ car ] { shape: edge; }

[ Bonn ] -- train --> [ Berlin ] -- [ car ] --> [ Ulm ]

[ rented ] --> [ car ]
```

which becomes this picture when as_ascii() is printed:

```
+------+  train  +--------+        car          +-----+
| Bonn | ------> | Berlin | ------------------> | Ulm |
+------+         +--------+                     +-----+
                                   ^
                                   |
                                   |
                              +--------+
                              | rented |
                              +--------+
```

Turning this textual description into a graph for Graph::Easy to generate and output can be done in one of two ways:

- Use the provided graph-easy utility script.
- Ask Graph::Easy to parse the description using Graph::Easy::Parser:

```
use Graph::Easy::Parser;

my $descript = '[ router ] — 1000GB-FX —> [ web server ]';

my $parser = Graph::Easy::Parser->new();
my $graph  = $parser->from_text($descript);

print $graph->as_ascii();
```

Graph::Easy::Parser has a from_file() method if you'd prefer to read the graph description from a file. See the Graph::Easy::Parser doc for more details.

In parting, I think it is important to mention that the ease and power of Graph::Easy hasn't gone unnoticed by other module writers. They've created add-on modules like those mentioned for GraphViz. Here's my favorite example (coincidentally, by the author of Graph::Easy) taken from the module's documentation:

```
use Devel::Graph;

my $grapher = Devel::Graph->new();

my $graph = $grapher->decompose( \'if ($b == 1) { $a = 9; }' );

print $graph->as_ascii();
```

This takes in a piece of Perl code and attempts to generate a flowchart that Graph::Easy can display. Here's the output:

```
################
#   start      #
################
  |
  |
  v
+--------------+
| if ($b == 1) |--+
+--------------+  |
  |               |
  | true          |
  v               |
+--------------+  |
|  $a = 9;     |  | false
+--------------+  |
  |               |
  |               |
  v               |
################  |
#    end       # <+
################
```

Let's end with that pretty picture, drawn just for you. Take care, and I'll see you next time.