Published on **Linux DevCenter** (http://www.linuxdevcenter.com/)
See this if you're having trouble printing code examples

An Introduction to GraphViz and dot

by Michele Simionato
05/06/2004

You must give a presentation tomorrow and you haven't prepared any figures yet; you must document your last project and you need to plot your most hairy class hierarchies; you are asked to provide ten slightly different variations of the same picture; you are pathologically unable to put your finger on a mouse and draw anything more complex than a square. In all these cases, don't worry! `dot` can save your day!

What is `dot`?

`dot` is a tool to generate nice-looking diagrams with a minimum of effort. It's part of `GraphViz`, an open source project developed at AT&T and released under an MIT license. It is a high-quality and mature product, with very good documentation and support, available on all major platforms, including Unix/Linux, Windows, and Mac. There is an official home page and a supporting mailing list.

What Can I Do with `dot`?

First of all, let me make clear that `dot` is not just another paint program, nor a vector graphics program. `dot` is a scriptable, batch-oriented graphing tool; it is to vector drawing programs as `LaTeX` is to word processors. If you want to control every single pixel in your diagram, or if you are an artistic person who likes to draw free hand, then `dot` is not for you. `dot` is a tool for the lazy developer, the one who wants the job done with the minimum effort and without caring too much about the details.

Since `dot` is not a WYSIWYG tool—even if it comes with a WYSIWYG tool, `dotty`—it is not primarily an interactive tool. Its strength is the ability to generate diagrams *programmatically*. To fulfill this aim, `dot` uses a simple but powerful graph description language. Give `dot` very high level instructions and it will draw the diagrams for you, taking into account all the low level details. Though you have a large choice of customization options and can control the final output in many ways, it is not at all easy to force `dot` to produce *exactly* what you want, down to the pixel.

Expecting that would mean to fight with the tool. You should think of `dot` as a kind of smart boy, who likes to do things his own way and who is very good at it, but becomes nervous if the master tries to put too much

pressure on him. The right attitude with `dot` (just as with LaTeX) is to trust it and let it to do the job. At the end, when `dot` has finished, you can always refine the graph by hand. (`dotty`, the `dot` diagram interactive editor, comes with GraphViz and can read and generate `dot` code.) In most cases, you do not need to do anything manually, since `dot` works pretty well. The best approach is to customize `dot` options, so that you can programmatically generate one or one hundred diagrams with the least effort.

`dot` is especially useful in repetitive and automatic tasks, since it easy to generate `dot` code. For instance, `dot` comes in handy for automatic documentation of code. UML tools can also do this work, but `dot` has an advantage over them in terms of ease of use, a flatter learning curve, and greater flexibility. On top of that, `dot` is very fast and can generate very complicated diagrams in fractions of second.

Hello World from `dot`

`dot` code has a C-ish syntax and is quite readable even to people who have not read the manual. For instance, this `dot` script:

```
graph hello {

// Comment: Hello World from ``dot``
// a graph with a single node Node1

Node1 [label="Hello, World!"]

}
```

generates the image shown in Figure 1.

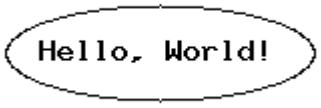

Figure 1. "Hello, World!" from GraphViz

Save this code in a file called `hello.dot`. You can then generate the graph and display it with a simple one-liner:

```
$ dot hello.dot -Tps | gv -
```

The `-Tps` option generates PostScript code, which is then piped to the ghostview utility. I've run my examples on a Linux machine with ghostview installed, but `dot` works equally well under Windows, so you may trivially adapt the examples.

If you're satisfied with the output, save it to a file:

```
$ dot hello.dot -Tps -o hello.ps
```

You'll probably want to tweak the options, for instance adding colors and changing the font size. This is not difficult:

```
graph hello2 {

// Hello World with nice colors and big fonts

Node1 [label="Hello, World!", color=Blue, fontcolor=Red,
    fontsize=24, shape=box]
```

```
}
```

This draws a blue square with a red label, shown in Figure 2.

Figure 2. A stylish greeting

You can use any font or color available to X11.

Editor's note: or presumably to Windows, if you're not running an X server.

`dot` is quite tolerant: the language is case insensitive and quoting the options `color="Blue", shape="box"` will work too. Moreover, in order to please C fans, you can use semicolons to terminate statements; `dot` will ignore them.

Basic Concepts of `dot`

A generic `dot` graph is composed of nodes and edges. Our `hello.dot` example contains a single node and no edges. Edges enter in the game when there are relationships between nodes, for instance hierarchical relationships as in this example, which produced Figure 3:

```
digraph simple_hierarchy {

B [label="The boss"]      // node B
E [label="The employee"]  // node E

B->E [label="commands", fontcolor=darkgreen] // edge B->E

}
```

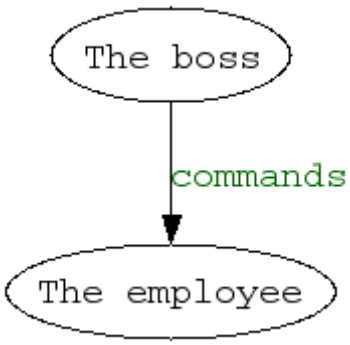

Figure 3. A hierarchical relationship

`dot` is especially good at drawing directed graphs, where there is a natural direction. (GraphViz also includes the similar `neato` tool to produce undirected graphs). In this example the direction is from the boss, who commands, to the employee, who obeys. Of course `dot` gives you the freedom to revert social hierarchies, as seen in Figure 4:

```
digraph revolution {
```

```
B [label="The boss"]      // node B
E [label="The employee"]  // node E

B->E [label="commands", dir=back, fontcolor=red]
// revert arrow direction

}
```

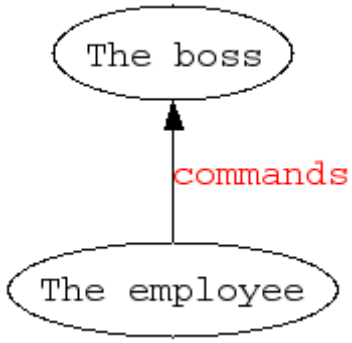

Figure 4. An inverted hierarchy

Sometimes, you want to put things of the same importance on the same level. Use the `rank` option, as in the following example, which describes a hierarchy with a boss, two employees, John and Jack, of the same rank, and a lower ranked employee Al who works for John. See Figure 5 for the results.

```
digraph hierarchy {

nodesep=1.0 // increases the separation between nodes

node [color=Red,fontname=Courier]
edge [color=Blue, style=dashed] //setup options

Boss->{ John Jack } // the boss has two employees

{rank=same; John Jack} //they have the same rank

John -> Al // John has a subordinate

John->Jack [dir=both] // but is still on the same level as Jack
}
```

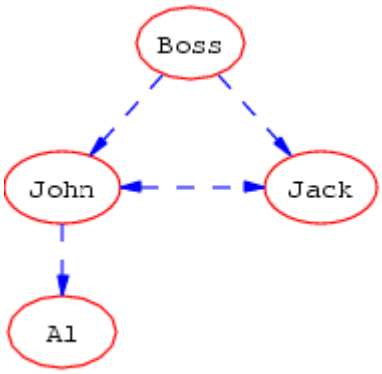

Figure 5. A multi-level organizational chart

This example shows a nifty feature of `dot`: if you forget to give explicit labels, it will use the name of the nodes as default labels. You can also set the default colors and style for nodes and edges respectively. It is even possible to control the separation between (all) nodes by tuning the `nodesep` option. I'll leave it as an

exercise for the reader to see what happens without the `rank` option (hint: you get a very ugly graph).

`dot` is quite sophisticated, with dozen of options which you can find in the excellent documentation. In particular, the man page (`man dot`) is especially useful and well done. The documentation also explains how to draw graphs containing subgraphs. However, those advanced features are outside the scope of this brief article.

We'll discuss another feature instead: the ability to generate output in different formats. Depending on your requirements, different formats can be more or less suitable. For the purpose of generating printed documentation, the PostScript format is quite handy. On the other hand, if you're producing documentation to convert to HTML format and put on a Web page, PNG format can be handy. It is quite trivial to select an output format with the `-T` output format type flag:

```
$ dot hello.dot -Tpng -o hello.png
```

There are *many* others available formats, including all the common ones such as GIF, JPG, WBMP, FIG and more exotic ones.

Generating `dot` Code

`dot` is not a real programming language, but it is pretty easy to interface `dot` with a real programming language. Bindings exist for many programming languages—including Java, Perl, and Python. A more lightweight alternative is just to generate the `dot` code from your preferred language. Doing so will allow you to automate the entire graph generation.

Here is a simple Python example using this technique. This example script shows how to draw Python class hierarchies with the least effort; it may help you in documenting your code.

```
# dot.py

"Require Python 2.3 (or 2.2. with from __future__ import generators)"

def dotcode(cls):
    setup='node [color=Green,fontcolor=Blue,fontname=Courier]\n'
    name='hierarchy_of_%s' % cls.__name__
    code='\n'.join(codegenerator(cls))
    return "digraph %s{\n\n%s\n%s\n}" % (name, setup, code)

def codegenerator(cls):
    "Returns a line of dot code at each iteration."
    # works for new style classes; see my Cookbook
    # recipe for a more general solution
    for c in cls.__mro__:
        bases=c.__bases__
        if bases: # generate edges parent -> child
            yield ''.join([' %s -> %s\n' % ( b.__name__,c.__name__)
                           for b in bases])
        if len(bases) > 1: # put all parents on the same level
            yield " {rank=same; %s}\n" % ''.join(
                ['%s ' % b.__name__ for b in bases])

if __name__=="__main__":
    # returns the dot code generating a simple diamond hierarchy
    class A(object): pass
    class B(A): pass
    class C(A): pass
    class D(B,C): pass
    print dotcode(D)
```

The function `dotcode` takes a class and returns the `dot` source code needed to plot the genealogical tree of that class. `codegenerator` generates the code, traversing the list of the ancestors of the class (in the Method Resolution Order of the class) and determining the edges and the nodes of the hierarchy. `codegenerator` is a generator which returns an iterator yielding a line of `dot` code at each iteration. Generators are a cool recent addition to Python; they come particularly handy for the purpose of generating text or source code.

The output of the script is the following self-explanatory `dot` code:

```
digraph hierarchy_of_D {

node [color=Green,fontcolor=Blue,font=Courier]

 B -> D
 C -> D

 {rank=same; B C }

 A -> B

 A -> C

 object -> A

}
```

Now the simple one-liner:

```
$ python dot.py | dot -Tpng -o x.png
```

generates Figure 6.

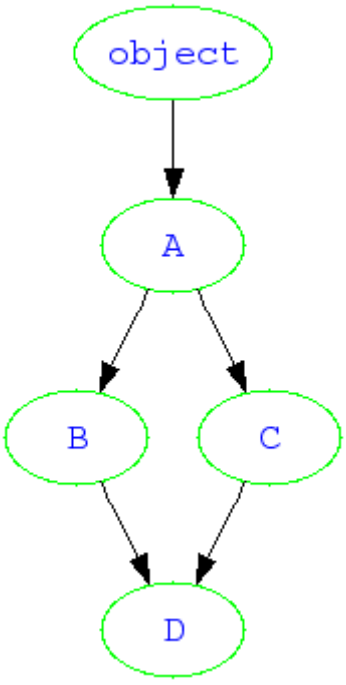

Figure 6. A Python class diagram

References

You may download `dot` and the others tool coming with GraphViz at the official GraphViz homepage. You will also find plenty of documentation and links to the mailing list.

Perl bindings (thanks to Leon Brocard) and Python bindings (thanks to Manos Renieris) are available. Also, Ero Carrera has written a professional-looking Python interface to `dot`.

The script `dot.py` I presented in this article is rather minimalistic. This is on purpose. My Python Cookbook recipe, Drawing inheritance diagrams with `Dot`, presents a much more sophisticated version with additional examples.

Michele Simionato is employed by Partecs, an open source company headquartered in Rome. He is actively developing web applications in the Zope/Plone framework.

Return to the LinuxDevCenter.com.