An excerpt from

# The Graphviz Cookbook

Rod Waldhoff
rwaldhoff@gmail.com
http://heyrod.com/

**ABOUT THIS BOOK**

*The Graphviz Cookbook*, like a regular cookbook, is meant to be a practical guide that *shows you how to create something tangible* and, hopefully, *teaches you how to improvise your own creations* using similar techniques.

The book is organized into four parts:

***Part 1: Getting Started*** introduces the Graphviz tool suite and provides "quick start" instructions to help you get up-and-running with Graphviz for the first time.

***Part 2: Ingredients*** describes the elements of the Graphviz ecosystem in more detail, including an in-depth review of each application in the Graphviz family.

***Part 3: Techniques*** reviews several idioms or "patterns" that crop up often when working with Graphviz such as how to tweak a graph's layout or add a "legend" to a graph. You might think of these as "micro-recipes" that are used again and again.

***Part 4: Recipes*** contains detailed walk-throughs of how to accomplish specific tasks with Graphviz, such as how to spider a web-site to generate a sitemap or how to generate UML diagrams from source files.

# Chapter 4

# The DOT Language in Depth

DOT is a text-based format for describing graphs.

It is used as an input to and sometimes as an output from (or intermediary between) individual programs within the Graphviz family. Many third-party tools also support the DOT format.

There are three core "objects" within DOT: graphs, nodes and edges. We'll discuss each in turn.

## 4.1 Graphs

A graph is a container for nodes and edges (and, as we'll see later, other graphs).

All nodes and edges must be contained within a graph. The outermost layer of any DOT file will contain one (or more) `graph` declarations.

Hence the simplest valid DOT file might be that found in Figure 4.1, which declares an anonymous, empty graph.

```
graph { }
```

**Figure 4.1:** *The simplest valid DOT file is an empty, anonymous graph.*

More commonly, a graph is assigned an identifier, which goes between the word `graph` and the first curly brace (Figure 4.2).

```
graph MyGraph { }
```

**Figure 4.2:** *This graph has been assigned the identifier (`MyGraph`).*

Both graphs are rendered in the same way (in this case, as aN empty canvas).

DOT recognizes two basic types of graph:

1. ***directed graphs***, in which the edges are directional, such that an edge from *node A* to *node B* is different than an edge from *node B* to *node A*, and
2. ***un-directed graphs***, in which the edges have no inherent direction. In an un-directed graph, an edge from *node A* to *node B* is the same as an edge from *node B* to *node A*.

A **digraph** is identified by the keyword `digraph`. A **graph** is identified by the keyword `graph`.

Within a graph, edges are indicated by two "dash" characters (`--`) (Figure 4.3).

```
graph G {
  A -- B
}
```

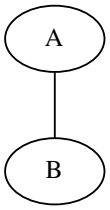


**Figure 4.3:** *Within a graph, edges are indicated by two "dash" characters (`--`).*

```
digraph G {
  A -> B
}
```

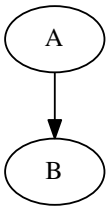


**Figure 4.4:** *Within a digraph, edges are indicated by an ASCII-art arrow (`->`).*

Within a digraph, edges are indicated by little ASCII-art arrows made from a dash and a greater-than sign (`->`) (Figure 4.4).

A ***strict digraph*** is a digraph that contains at most one *directed* edge between any pair of nodes. The keywords `strict digraph` are used to specify a strict digraph.

For example, in Figure 4.5, there are multiple directed edges running from *node B* to *node C*, from *node C* to *node B* and from *node A* to itself. But when we add the keyword `strict` before the keyword `digraph`, `dot` and other Graphviz applications will quietly ignore the duplicate edges, as seen in Figure 4.6.

```
digraph NotStrictG {
  A -> B -> C
  B -> C -> B
  B -> C -> B
  B -> C -> B
  A -> A -> A
}
```

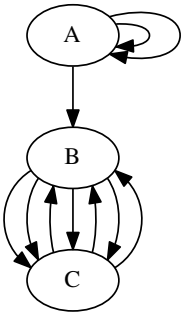


**Figure 4.5:** *In a regular (non-strict) digraph,* `dot` *will render duplicate edges as seen here between node B and node C, node C and node B and node A and itself. (Compare with Figure 4.6.)*

```
strict digraph StrictG {
  A -> B -> C
  B -> C -> B
  B -> C -> B
  B -> C -> B
  A -> A -> A
}
```

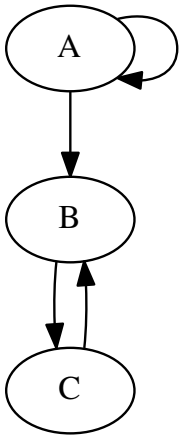


**Figure 4.6:** *In a* strict *digraph,* `dot` *will silently ignore duplicate edges such as those declared but not rendered between node B and node C, node C and node B and node A and itself. (Compare with Figure 4.5.)*

## 4.2 Nodes

You can create a node within a graph by simply declaring it (Figure 4.7).

```
digraph G {
  Foo
}
```

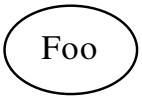


**Figure 4.7:** *A node is created whenever a new identifier is encountered.*

Node declarations can be separated by a single space, but by convention node declarations are often separated by a semi-colon or placed on separate lines (or

both) (Figure 4.8).

```
digraph G { Foo Bar }
```

```
digraph G { Foo; Bar; }
```

```
digraph G {
  Foo
  Bar
}
```

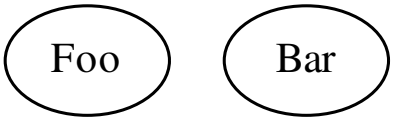


**Figure 4.8:** *Node declarations can be separated by semicolons or white-space characters.*

```
digraph G {
  Foo
  Foo
  Foo
}
```

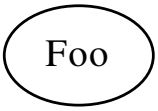


**Figure 4.9:** *Declaring the same node several times doesn't have any effect*

Every node must have a unique identifier. Even when a node declaration appears several times within a DOT file, only one node is created for each unique identifier. (Figure 4.9).

## 4.3 Edges

You can create an edge by putting the "edge operator" (`--` for graphs and `->` for digraphs) between two node identifiers in your DOT file (Figure 4.10).

When creating an edge, you can declare the nodes first and later add an edge, or can declare the nodes at the same time as you declare the edge between them. As seen in Figure 4.10, this doesn't change the way in which the graph is rendered.

In contrast to nodes (where multiple declarations of the same node don't have any effect), each `edge` declaration creates a unique edge in the graph. (See Figure 4.11.)

Edge operators can be chained together (`A -> B -> C`) or declared one at a time (`A -> B; B -> C`). As illustrated in Figure 4.12, this doesn't change the way in which the graph is rendered.

While there are ways to add or supress the arrowheads that appear in the rendered graph, digraphs are not allowed to contain un-directed edges (`--`) and un-directed graphs are not allowed to allowed to contain directed edges (`->`). Some Graphviz tools will refuse to process a DOT file that uses the wrong edge operator.

```
digraph G {
  Foo
  Bar
  Foo -> Bar
}
```

```
digraph G { Foo -> Bar }
```

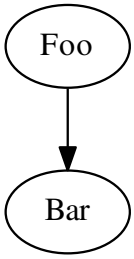


**Figure 4.10:** *Edges are created by placing the "edge operator" (-- or ->) between two nodes. If you want, you can declare the nodes and later add an edge between them. This doesn't change the way in which the graph is rendered.*

```
digraph {
  Foo -> Bar;
  Foo -> Bar;
  Foo -> Bar;
}
```

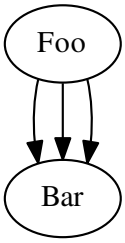


**Figure 4.11:** *Each* `edge` *declaration creates new edge in the graph.*

```
digraph {
  A -> B -> C -> A
}
```

```
digraph {
  A -> B;
  B -> C;
  C -> A;
}
```

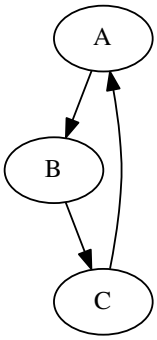


**Figure 4.12:** *Edge operators can be chained together or declared one-at-a-time. This doesn't change the way in which the graph is rendered.*

## 4.4 Some Syntax Details

Let's take a moment to review some specific syntax details of the DOT language.

### Identifiers

DOT calls the names we assign to things like graphs and nodes ***identifiers***.

DOT recognizes three formats for identifiers: alphanumeric, numeric, and quoted.

An ***alphanumeric identifier*** is any combination of letters, numbers and the underscore character (_), as long as it starts with a letter. See Figure 4.13 for examples.

```
digraph G {
 Foo -> f00
 node_7 -> foo_bar
}
```

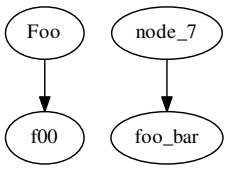


**Figure 4.13:** *An* alphanumeric identifier *is any combination of letters, numbers and the underscore character (_), as long as the first character is a letter.*

A ***numeric identifier*** is any combination of digits (`0` through `9`) and, optionally, a single decimal point (`.`). Leading zeros are OK. (For example, `007` is valid numeric identifier.) A leading or trailing decimal point is also OK. (For example, `.5` and `5.` are both valid numeric identifiers.) See Figure 4.14 for examples.

```
digraph G {
 700 -> 1234 -> 007
 .95 -> 3.14 -> 0123.
}
```

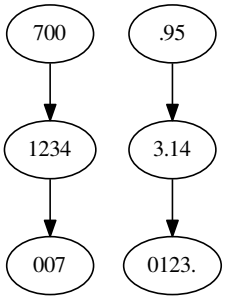


**Figure 4.14:** *A* numeric identifier *is any combination of the digits* `0` *through* `9` *and at most one decimal point (.).*

A ***quoted string identifier*** is any sequence of characters, wrapped in double quotation marks. Quoted string identifiers can contain:

- "White space", even line breaks (`\n`)

- Symbols (E.g., the π and ≈ charcters "π ≈ 3.14".)
- Special characters (like a quotation mark), when escaped with a backslash character. (E.g., write \" to indicate ".)

Note that the blank string ("") and strings composed entirely of white-space characters (" ") are also valid identifiers. See Figure 4.15 for examples of quoted string identifiers.

```
digraph G {
 "spaces &\nnewlines" -> ""
 "  3.14"  -> " "
 "" -> "escaped \"quotes\""
}
```

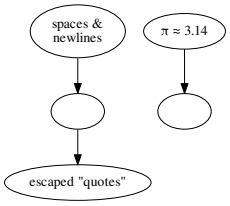


**Figure 4.15:** *A* quoted string identifier *is a string surrounded by double quotation marks (").  The backslash character (\) is used to "escape" certain characters within the string, so that* \n *creates a line break (newline character) and* \" *creates a literal quotation mark (") (rather than closing the string).*

## Comments

A *comment* is text within a DOT file that is intended to be read by humans rather than software. The Graphviz tools will ignore any text that appears in a comment.

DOT supports two common comment formats. Like *C/C++*, within DOT:

- Two forward-slashes (like //) indicate a single line comment. Everything from // to the end of the line will be ignored.
- /* and */ delimit multi-line comments. Everything from the opening /* to the closing */ will be ignored, whether on a single line or split across multiple lines.

DOT also supports a third comment delimiter. Like Ruby, Perl, bash and many other scripting languages:

- A single hash mark (#) indicates a single line comment. Everything from # to the end of the line will be ignored.

See Figure 4.16 for examples.

Note that //, /*, */ and # are *not* treated as comment delimiters when used within quoted strings.

```
digraph G {
  A -> B // A single-line comment.

  // the rest of the line is ignored

  /* This is a multi-line comment.
     Everything up to the star-slash
     delimiter is ignored. */

  # The hash character (#) is
  # an alternative single-line
  # comment format.

  B -> C # alt single-line

  /*
    You can "comment-out" parts
    of your graph

  C -> D

    using multi-line
    comments.
  */

  C /* also */ -> /* inline */ E

  // Comment delimiters can
  // be safely used inside
  // of quoted strings.
  E [label="/* E! */"]
}
```

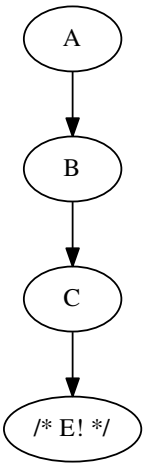


**Figure 4.16:** *Examples of comments in DOT.*

## Double-Quoted Strings

Double-quoted strings appear in the DOT Language as (quoted-string) identifiers and as attribute values.

The text within a double-quoted string is given special treatment relative to "regular" tokens in the DOT language.

Specifically:

1. Within a double-quoted string, the "backslash" character is used to escape certain special reserved characters. For example, `\"` is interpreted as a literal quotation mark (") and `\n` is interpreted as a "newline" character. The sequence `\\` inserts a single literal backslash.

2. Where appropriate[1] DOT supports two special forms of line break within

[1] Specifically, in identifiers and attributes such as `label`, `xlabel`, `headlabel`, etc.

double-quoted strings, `\l` and `\r`. Using `\l` will cause the preceding line to be aligned on the left. Using `\r` will cause the preceding line to be aligned on the right. (Using `\n` will cause the preceding line to be centered, the default alignment.) See Figure .18.

3. Depending upon the context, DOT supports special escape sequences that act as placeholders for other values. These sequences are replaced (at render-time) by the corresponding context-sensitive value.

    a. Within node attribute values, the sequence `\N` is replaced with the name (identifier) of the node. (Note that the default `label` value is `\N`.)

    b. Within edge attribute values:

        - The sequence `\T` is replaced with the name of the tail node.
        - The sequence `\H` is replaced with the name of the head node.
        - The sequence `\E` is replaced with the name (declaration) of the edge.

    c. In all attribute values:

        - The sequence `\G` is replaced with the name (identifier) of the containing graph or subgraph.
        - The sequence `\L` is replaced with the `label` of the corresponding object (graph, node, edge or subgraph).

4. When they appear within double-quoted strings, the character sequences `//`, `/* */` and `#` are interpreted as their literal character values, not as comment delimiters.

5. A single *logical* string can be split into multiple double-quoted substrings using the "string concatenation" operator `+`. DOT will concatenate any pair of strings joined by a plus sign into a single value. For example, the expression `"Foo" + "Bar"` is interpreted identically to the expression `"FooBar"`.

6. Within a double-quoted string, a trailing backslash character (`\`) allows a single *logical* line to be split into multiple *actual* lines. That is, whenever a line within a double-quoted string ends with a backslash character (immediately followed by a newline), the next physical line in the file is treated as if it were a continuation of the preceding line.

### Case Sensitivity

Keywords in DOT such as `graph`, `node` and `edge` are case-insensitive (Figure 4.17). This means that `graph { }` and `GRAPH { }` are equivalent to one another, and to `Graph { }`, `gRAPH { }`, `GrApH { }`, and so on.

Specifically, DOT ignores the case of the following keywords:

- digraph
- edge
- graph
- node
- strict
- subgraph

```
DIGRAPH g {
  Node [shape=box]
  edgE [color=GREEN]
  a -> b [label="comes before"]
  SUBGRAPH h {
    rank=same
    c -> d
  }
}
```

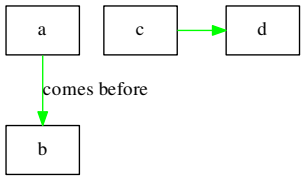


**Figure 4.17:** *Examples of case-insensitivity in DOT.*

Attribute names and values such as `shape` and `box` or `rankdir` and `LR` *are* case sensitive.[2] Hence while `EDGE [color=red]` is equivalent to `edge [color=red]` the `COLOR` attribute in `edge [COLOR=red]` will be ignored.

## 4.5 Attributes

DOT supports a large number of *attributes* that influence how nodes, edges and graphs are rendered.

These attributes are specified as name-value pairs using the syntax `name=value` or `name="value"`.

Attribute declarations are typically enclosed in square brackets (`[ ]`). When more than one name-value pair is used, they must be separated by whitespace or comma characters. E.g.,

```
[penwidth=10 fontname="Helvetica"]
```

or

```
[penwidth=10,fontname="Helvetica"]
```

or

[2] In the general case. Color names, for example, can be written in lower or upper case.

```
[penwidth=10, fontname="Helvetica"]
```

Graph and subgraph attributes are listed (without enclosing brackets) directly within the container (Figure 4.18) or enclosed in square brackets following the keyword `graph` (Figure 4.19).

```
digraph G {
  rankdir=RL
  bgcolor=gray
  A -> B
}
```

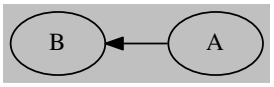


**Figure 4.18:** *Graph and subgraph attributes can be listed directly within the container. (Also see Figure 4.19.)*

```
digraph G {
  graph [rankdir=RL bgcolor=gray]
  A -> B
}
```

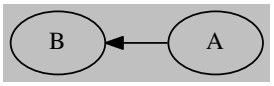


**Figure 4.19:** *Graph and subgraph can be enclosed in square brackets (`[ ]`) following the keyword `graph`. (Also see Figure 4.18.)*

Node and edge attribute lists are grouped within square brackets and listed immediately following the node or edge declaration (Figure 4.20).

```
digraph G {
  A [shape=box color=blue]
  B [style=dashed
     label="Node B"]
  A -> B [arrowhead=vee]
}
```

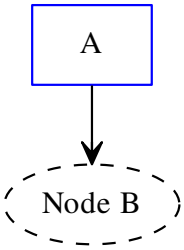


**Figure 4.20:** *Node and edge attributes are grouped with square brackets (`[ ]`) and can be listed immediately following the node or edge declaration.*

An attribute list can also follow the keywords `node` and `edge`, rather than a node or edge declaration (respectively), in order to set (or reset) the attributes' default values. Subsequent nodes or edges will inherit this new default value unless the default is changed or overwritten at an individual node or edge level. See Figure 4.21 for examples.

Setting an attribute's value to a empty string (`""`) restores the original default value.

```
digraph G {

  node [shape=box
        style=filled
        color=dodgerblue]
    A
    B [shape=triangle]
    C

    node [color=pink]
      D
      E [shape="" color=""]

  edge [style=bold]
    A -> B
    A -> C [style=dashed]

    edge [color=firebrick]
      B -> D [color=""]
      C -> E
      D -> E [constraint=false]
}
```

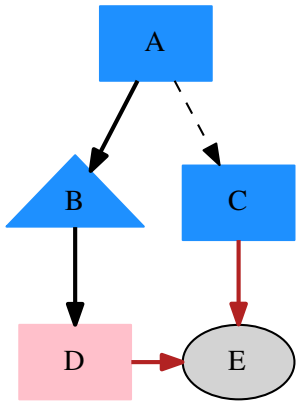


**Figure 4.21:** *The* `node` *and* `edge` *keywords can be used to change an attribute's value for all subsequent nodes and edges.*

In the general case, attribute values *may* be enclosed within double-quotes, but this isn't always strictly necessary. The assignments:

```
node [color=red]
```

and

```
node [color="red"]
```

are equivalent.

However, attribute values that contain white-space characters, commas or other characters with special meaning within DOT files *must* be enclosed in quotation marks.

Similiarly, DOT does not make a distinction between numeric values and string values in attribute assignment. The assignments:

```
node [width=3]
```

and

```
node [width="3"]
```

are equivalent.

See Part V to learn more about individual graph, node and edge attributes.

## 4.6 Subgraphs and Clusters

In DOT, graphs and digraphs can contain other graphs. These are known as ***subgraphs***. Like the top-level graph, subgraphs can contain nodes, edges and other subgraphs.

A subgraph can be created by nesting a collection of node, edge and subgraph declarations in a pair of curly braces, optionally prefixed by the keyword `subgraph`. An identifier for the subgraph, if any, can be specified between the keyword `subgraph` and the opening curly brace. See Figure 4.22 for examples.

```
digraph G {
  A -> B
  { C -> D }
  E -> F
}
```

```
digraph G {
  A -> B
  subgraph { C -> D }
  E -> F
}
```

```
digraph G {
  A -> B
  subgraph S { C -> D }
  E -> F
}
```

**Figure 4.22:** *In DOT, graphs and digraphs can contain other graphs, known as subgraphs. A subgraph is created by wrapping the contained node and edge declrations within a pair of curly braces. Optionally, the keyword* `subgraph` *can be used and an identifier for the subgraph can be assigned.*

Note that both graphs and digraphs use the keyword `subgraph`. There is no `subdigraph`. A subgraph shares directionality with its parent graph—a subgraph within a digraph must only contain directed edges, a subgraph within a graph must only contain un-directed edges.

### Using subgraphs to limit scope

A subgraph declaration creates a "local scope" for attribute assignment. Any default attribute value changed within a subgraph (using the `node` or `edge` keyword, for example) reverts to its previous value at the end of the subgraph declaration. See Figure 4.23 for an example.

```
digraph G {
  node [color=blue]
  A -> B;
  {
    rank=same;
    node [color=orange shape=box]
    C -> D;
  }
  E -> F
}
```

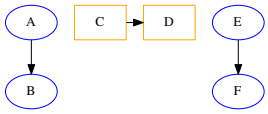


**Figure 4.23:** *A subgraph defines a local scope for attribute assignment. Any default attribute values assigned within the subgraph revert to their original value at the end of the subgraph.*

### Clusters

A cluster is a special class of subgraph with an identifier that begins with the (case-sensitive) prefix `cluster`.

A cluster is treated as an inseparable unit by `dot`, `patchwork` and other layout engines. All of the nodes within a cluster, and *only* the nodes within that cluster, are laid out within their own rectangular region. See Figure 4.24 and Figure 4.25 for examples.

```
digraph G {
  node [color=blue]
  subgraph S {
    node [color=orange shape=box]
    a -> c;
  }
  a -> b -> c;
}
```

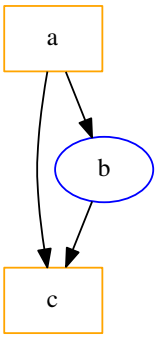


**Figure 4.24:** *By default, a subgraph doesn't change the way in which the graph is laid out. The nodes within the subgraph don't need to be contiguous, and nodes from outside the subgraph might be mixed in. (Compare with Figure 4.25.)*

```
digraph G {
  node [color=blue]
  subgraph clusterS {
    node [color=orange shape=box]
    a -> c;
  }
  a -> b -> c;
}
```

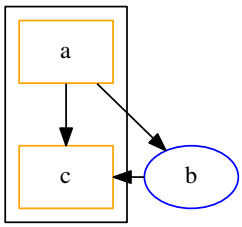


**Figure 4.25:** *When a subgraph's identifier begins with the prefix* `cluster`*, the nodes within the subgraph are grouped together within a rectangular region. Nodes from outside the subgraph are not allowed within that rectangle. (Compare with Figure 4.24.)*

Certain attributes make use of clusters in other special ways as well. See "Appendices" (Part V) for details.

## 4.7 What Next?

- Visit "Appendices" (Part V) for more information on graph, node and edge attributes supported by Graphviz and the DOT Language.
- Jump to "Techniques" (Part III) to explore tips and tricks for working with the DOT Language.
- Visit "Output Formats " (Chapter 5) to learn about alternative ways to describe graphs as text.