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 7

The Rest of the Graphviz Suite

7.1 `acyclic`

`acyclic` is a relatively simple tool for detecting and "correcting" digraphs that contain cycles (directed loops).

Example of Use

By default, `acyclic` will write (to *STDOUT*) a version of the input digraph with a sufficient number of edges reversed to ensure the digraph does not contain any cycles.

```
> echo "digraph { A -> B -> A }" | acyclic
digraph {
  A -> B;
  A -> B;
}
```

Figure 7.1: `acyclic` *will ensure that a digraph contains no cycles by reversing edges if necessary.*

Command Line Options

Verbose Output (`-v`)

When the `-v` flag is present, `acyclic` will emit a line (to *STDERR*) that states whether the input graph is acyclic, contains a cycle or is an undirected graph. Also see the `-n` option.

Supress graph Output (`-n`)

When the `-n` flag is present, `acyclic` will NOT emit the acyclic version of input digraph. This is most useful when coupled with the `-v` parameter.

```
> echo "digraph G { A -> B -> A }" | acyclic -nv
Graph G has cycles
```

Figure 7.2: *The* `-v` *parameter tells* `acyclic` *to write a brief summary to STDERR. The* `-n` *parameter tells* `acyclic` not *to output the modified graph.*

Output File (`-o`)

`acyclic`'s output can be written to file in the typical Unix way (by redirecting the *STDOUT* stream):

```
> acyclic in.gv > out.gv
```

or by using the `-o` flag to specify an output file:

```
> acyclic in.gv -o out.gv
```

Exit Status

Note that (in addition to the text generated by the `-v` flag) `acyclic` reports whether or not the input graph was acyclic via the application's *exit status* (also known as a *return code*). Specifically:

- `acyclic` returns `0` (the success status) if the input digraph is acyclic,
- `acyclic` returns `1` (an error status) if the digraph contains a cycle, and
- `acyclic` returns `2` (an error status) if the input isn't digraph at all, but an undirected graph instead.

Most users can ignore the exit status most of the time, but there are at least two situations in which the value returned by `acyclic` might be of interest:

1. Even if you invoke `acyclic` for the express purpose of converting a cyclic graph into an acyclic graph, `acyclic` returns what is technically an error status. Some applications and scripts[1] will halt when a program or command they invoke returns an error status.

   If needed, you can ensure a success status by appending `|| true` to the end of the `acyclic` command. E.g:

```
> (acyclic in.gv -o out.gv || true)
```

   which will exit with a success status (`0`) no matter how `acyclic` responds.

[1] `make`, for example, and the `&&` operator in `bash`

2. You can use `acyclic`'s response code to your advantage when writing your own shell scripts, as in demonstrated Figure 7.3.

```
#!/bin/bash
# This script will run either 'dot' or 'neato' on a graph
# depending upon whether or not the input file contains an
# acyclic (di)graph.

acyclic -n $1     # Run acyclic on the input file ($1)

RESULT=$?         # Grab the return code ($?)

case $RESULT in   # Take the appropriate action

  0) dot $1 ;;

  1) echo "That digraph contains a cycle.";
     echo "Please fix it and try again.";
     exit 1 ;;

  2) neato $1 ;;

esac

exit 0
```

Figure 7.3: *Example of a* `bash` *script that uses* `acyclic`*'s exit status.*

7.2 gc

Just as the Unix program `wc` counts the characters, words or lines in text file, the Graphviz program `gc` counts the nodes, edges and other components of a DOT file.

Example of Use

The command:

```
> gc -a MyGraph.gv
       4       3       1       0 G (MyGraph.gv)
```

counts the various components of the DOT format graph defined in `MyGraph.gv` and reports the number of nodes, edges, *connected* components and clusters found, as well as the graph name and source. Here `gc` reports that the file `MyGraph.gv` contains a graph named `G` with `4` nodes, `3` edges, `1` connected component and no clusters.

Inputs

If multiple files are enumerated on the command line, `gc` will report counts for each of them independently as well as a total count across all files.

If no files are specified, `gc` will read input data from *STDIN* instead.

Command Line Paramaters

Default

When no count flags are specified on the command line, `gc` will report the number of nodes and edges found (equivalent to `-ne`).

Count nodes (`-n`)

When `-n` is specified, `gc` will report the number of *distinct* nodes found in the input file, files or stream.

```
> echo ”graph G { A; A; B; A; }” | gc -n
       2 G (<stdin>)
```

Count edges (`-e`)

When `-e` is specified, `gc` will report the number of edges found in the input file, files or stream.

```
> echo ”digraph G { A -> { B, C, D } }” | gc -e
       3 G (<stdin>)
```

Count connected components (`-c`)

When `-c` is specified, `gc` will report the number of “connected components” found in the input file, files or stream (where each ***connected component*** is a group of nodes connected directly or indirectly by edges that aren’t connected to any nodes not in the group).

```
> echo ”graph G { A -- B; C -- D }” | gc -c
       2 G (<stdin>)
```

Count clusters (`-C`)

When `-C` is specified, `gc` will report the number of cluster found in the input file, files or stream (where each cluster is a graph or subgraph whose name starts with “cluster”).

```
> echo "graph G { A -- B; C -- D }" | gc -c
      2 G (<stdin>)
```

Count all (`-a`)

When `-a` is specified, `gc` will report the number of nodes, edges, connected components and clusters, as if `-necC` was specified.

Undirected Only (`-U`)

When `-U` is specified, `gc` will ignore any digraphs found in the input file, files or stream, and only report upon the undirected graphs that are found.

Directed Only (`-D`)

When `-D` is specified, `gc` will ignore any undirected graphs found in the input file, files or stream, and only report upon the digraphs that are found.

Recurse Over Subgraphs (`-r`)

When `-r` is specified, `gc` will print a similiar report for each subgraph found in the input file, files or stream.

```
> echo "graph G { subgraph S1 { a -- b }; subgraph S2 { c } }" | gc -r
      3       1 G (<stdin>)
        2       1 S1
        1       0 S2
```

Verbose Output (`-v`)

When `-v` is specified, `gc` will print additional messages about the files and streams it is processing.

Supress Output (`-s`)

When `-s` is specified, `gc` will not print any information to *STDOUT*. This is primarily useful when you are only interested in the exit status of the `gc` process.

Help (`-?`)

When `-?` is specified, `gc` prints a brief summary of its command line options to *STDOUT*.

```
> gc -?
Usage: gc [-necCaDUrsv?] <files>
  -n - print number of nodes
  -e - print number of edges
  -c - print number of connected components
  -C - print number of clusters
  -a - print all counts
  -D - only directed graphs
  -U - only undirected graphs
  -r - recursively analyze subgraphs
  -s - silent
  -v - verbose
  -? - print usage
By default, gc prints nodes and edges
If no files are specified, stdin is used
```

Exit Status

When `-U` (`-D`) is specified but *no* graphs (digraphs) are found in the input, `gc` will exit with an error status (value `1`). Otherwise `gc` will exit with an success status (value `0`).

Most users can safely ignore the exit status virtually all of the time. See Section 9.3 for more information.

7.3 `gvcolor`

`gvcolor` colors the nodes of a (directed) graph based on the color of the nodes that are connected to it.

Given a graph that has been "seeded" by setting the `color` attribute of at least one node, `gvcolor` will cause the colors to "flow" through the graph along each (directed) edge. Each node (with an otherwise unspecified color) is assigned the average color of all the nodes that "point" to it.

For instance, consider a simple digraph like the one in Figure 7.4.

```
digraph {
 node [style=filled]
   A -> B
   B -> D
   C -> D
   D -> E
   D -> F
}
```

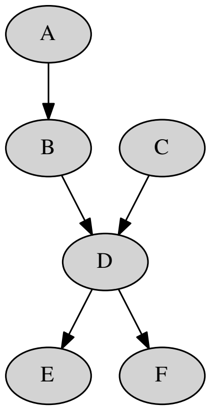

Figure 7.4: *A simple digraph without any specified colors.*

Normally, if we were to assign a color to one of the nodes, the color would only apply to the specific node it was assigned to (as in Figure 7.5).

```
digraph {
 node [style=filled]
   A -> B
   B -> D
   C -> D
   D -> E
   D -> F
   B [color=gold1]
}
```

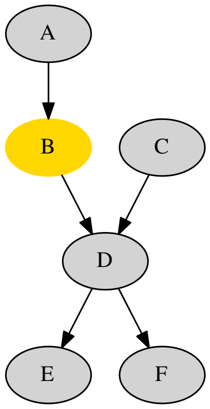

Figure 7.5: *A simple digraph with one colored node.*

When run through `gvcolor`, however, the color that was explicitly assigned is copied to all of the nodes "downstream" from it. Hence in Figure 7.6 the color assigned to *node B* propagates to *nodes D*, *E* and *F* (but not to *nodes A* or *C*, since there is no directed path from from *B* to *A* or *C*).

```
digraph {
 node [style=filled]
   A -> B
   B -> D
   C -> D
   D -> E
   D -> F
   B [color=gold1]
}
```

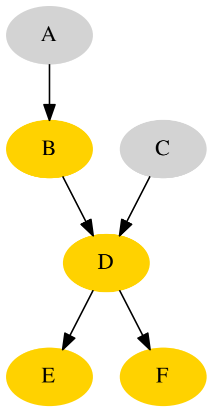

Figure 7.6: *A simple digraph with one colored node, processed through* `gvcolor`.

When more than one of a node's "upstream" neighbors has color, the node is assigned the *average* of upstream nodes' colors. Hence in Figure 7.7, *node D* is assigned a color mid-way between that of *node B* and *node C* (and that same color is propagated to *nodes E* and *F*).

```
digraph {
 node [style=filled]
   A -> B
   B -> D
   C -> D
   D -> E
   D -> F
   B [color=gold1]
   C [color=lightblue]
}
```

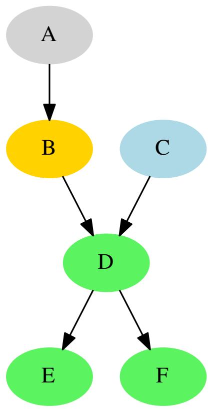

Figure 7.7: *A digraph with more than one colored node, processed through* `gvcolor`.

Note that it is really the *edges* leading into a node that determine its color under `gvcolor`. The more edges between a pair of nodes, the more strongly the node at the tail (beginning) influences the color of the node at the head (end). (See Figure 7.8.)

```
digraph {
 node [style=filled]
   A -> B
   B -> D
   B -> D
   B -> D
   B -> D
   B -> D
   C -> D
   D -> E
   D -> F
   B [color=gold1]
   C [color=lightblue]
}
```

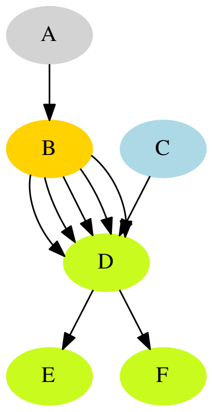

Figure 7.8: *When determining a node's color,* `gvcolor` *uses the average color of the node at the other side of* each *inbound edge. When multiple edges stem from the same tail, the color of the node is used multiple times in the computation of the average.*

Example of Use

Before `gvcolor` can process a graph each node must be assigned a explicit position via the `pos` attribute. An easy way to achieve this is to run the DOT file through the `dot` layout engine.

Hence the typical invocation of `gvcolor` will be something like this:

```
> dot MY-GRAPH.gv | gvcolor | dot -Tpng -o MY-IMAGE.png
```

`gvcolor`'s internal logic relies on the relative physical position of nodes in the graph. In order for `gvcolor` to work properly, not only does each node need an explicit `pos` attribute, but the relative position of connected nodes must match the directed-edge relationship between them. That is, if there is an edge `A -> B` in the graph then the *node B* should not be positioned above *node A*.[2] `gvcolor` may not behave as expected when presented with graphs that contain cycles[3] or layouts that have been manipulated with `rank` or similar attributes.

Controlling `gvcolor` through graph attributes

Certain graph attributes will influence the way in which `gvcolor` assigns colors to nodes.

[2] Or, if `rankdir=LR` is specified, the *node B* should not be positioned to the left of *node A*.
[3] A cycle is a directed path that forms a loop, such as `A -> B -> C -> A`.

`Defcolor` `gvcolor` uses `Defcolor` for the color of nodes that are not otherwise specified (whether through the `color` attribute directly or by virtue of being downstream from a node that has an explicitly set `color` attribute).

```
digraph {
  Defcolor=gray
  A -> C
  B -> C
  B [color=blue]
}
```

Absent an explict `Defcolor` attribute, `gvcolor` defaults to white.

Note that the name is case-sensitive, it must be `Defcolor` rather than `defcolor`.

`flow` Setting the graph attribute `flow` to `back` will cause `gvcolor` to reverse the direction in which colors propagate through the graph. When `flow=back`, a node's color will be determined by the nodes at the *head* of each edge for which it is the tail.

```
digraph {
 flow=back
 node [style=filled]
   A -> B
   B -> D
   C -> D
   D -> E
   D -> F
   B [color=gold1]
   C [color=lightblue]
}
```

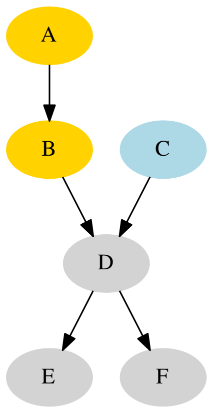

Figure 7.9: *Caption TK*

Hence in Figure 7.9 the color that has been assigned to *node B* flows *up* to *node A*, rather than *down* to *nodes D*, *E* and *F*.

Note that `flow=back` doesn't reverse the direction of the arrows themselves, it just changes the direction in which the colors propagate.

`saturation` When the `saturation` attribute is set, `gvcolor` will–in addition to its regular color-propagation logic–vary the degree of saturation for each node based on the node's `rank` in the overall graph. The `saturation` attribute defines the range for the degree of saturation as a pair of numbers between `0` and `1`.

```
digraph {
  saturation="0.9,0.1"
  a -> b
  a -> c
  b -> d
  c -> e
  d -> g
  d -> h
  e -> h
  e -> i
  e -> j
  x -> j
  a [color=dodgerblue]
  b [color=firebrick]
  x [color=dodgerblue]
}
```

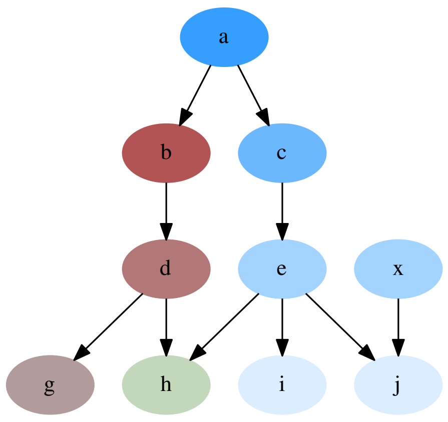

Figure 7.10: *Caption TK*

Note that the `saturation` value is not determined by how close or far a node is to an assigned color value. It is strictly a function of the rank of each node within the overall graph. Hence in Figure 7.10, while we've explicitly assigned *node x* the same color as *node a*, the color of *x* matches that of *node c* and not that of *node a*. This is because *node c* and *node x* share the same `rank` in the graph, and hence the same degree of saturation.

7.4 `gvgen`

`gvgen` generates DOT Language descriptions of several graphs that are found in discrete mathematics and computer science.

Example of Use

The size, shape and other aspects of the graphs that are generated by `gvgen` are determined by command-line options. For example, the command:

```
> gvgen -c3
```

generates a *cycle* (polygon) with three vertices, as seen in Figure 7.11.

```
> gvgen -c3
graph {
  1 -- 2
  2 -- 3
  1 -- 3
}
```

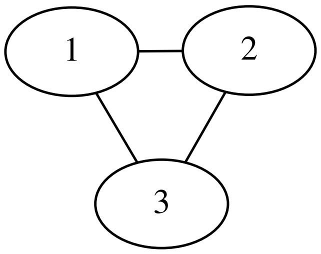

Figure 7.11: *The graph generated by* `gvgen -c 3` *and a* `neato` *rendering of it.*

The DOT Language graph descriptor generated by `gvgen` can be saved to file in the typical Unix fashion:

```
> gvgen -c3 > c3.gv
```

or the `-o` parameter can be used to specify an output file:

```
> gvgen -c3 -o c3.gv
```

But it isn't necessary to save the output of `gvgen` to file. It can be piped directly to a rendering engine. Hence:

```
> gvgen -c3 | neato -Tpng -o c3.png
```

generates a PNG image like that found in Figure 7.11.

See Section 7.4 for a description of the various kinds of graph that `gvgen` can generate. See Section 7.4 for a discussion of other command line parameters.

graph Types

`gvgen` can generate many different "classes" of graph. Command line parameters control the shape and size of the graphs that are generated. The following sections describe the `gvgen` graph types in detail.

Path (`-p`)

A *path* is a simple line of nodes.

```
> gvgen -p4
graph {
  1 -- 2
  2 -- 3
  3 -- 4
}
```

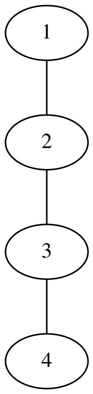

Figure 7.12: *A path on 4 nodes.*

The command `gvgen -p N` is used to generate a *path* with *N* nodes.

Cycle (`-c`)

A *cycle* is a "closed" *path*–one in which an edge connects the last node back to the first.

```
> gvgen -c5
graph {
  1 -- 2
  2 -- 3
  3 -- 4
  4 -- 5
  1 -- 5
}
```

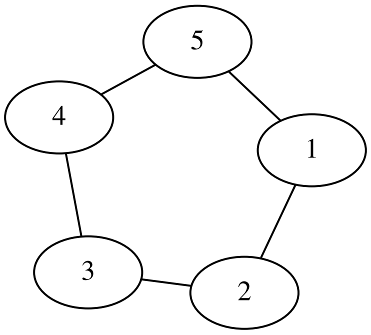

Figure 7.13: *A cycle on 5 nodes.*

The command `gvgen -c N` is used to generate a *cycle* with *N* nodes.

Star (`-s`)

A *star* graph is shaped like an octopus or starfish, with multiple "legs" radiate out from a central "hub". In a *star* graph, The center node is connected to every other node, but none of the outer nodes are connected to one another.

```
> gvgen -s5
graph {
  1 -- 2
  1 -- 3
  1 -- 4
  1 -- 5
}
```

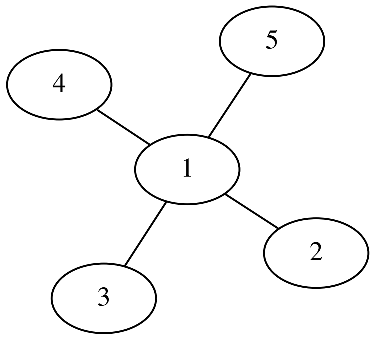

Figure 7.14: *A star on 5 nodes.*

The command `gvgen -s N` generates a *star* with *N* nodes.

Wheel (`-w`)

A *wheel* graph is a star graph in which all of the “outer” nodes are connected to form a cycle. A wheel with *N* nodes is a cycle of *N-1* nodes, each connected to a central “hub”.

```
> gvgen -w5
graph {
  1 -- 2
  1 -- 3
  1 -- 4
  1 -- 5
  2 -- 3
  3 -- 4
  4 -- 5
  2 -- 5
}
```

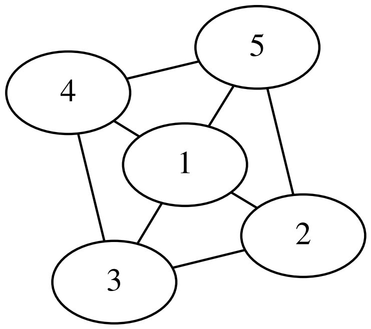

Figure 7.15: *A wheel on 5 nodes.*

The command `gvgen -w N` is used to generate a *wheel* with *N* nodes.

Complete graph (`-k`)

A *complete graph* is a graph in which every node is connected to every other node.

```
> gvgen -k5
graph {
  1 -- 2
  1 -- 3
  1 -- 4
  1 -- 5
  2 -- 3
  2 -- 4
  2 -- 5
  3 -- 4
  3 -- 5
  4 -- 5
}
```

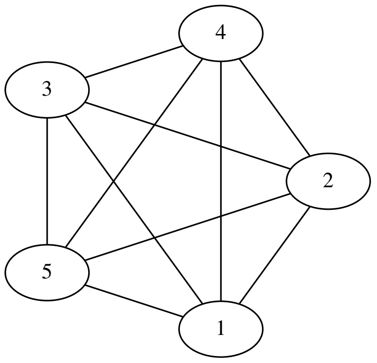

Figure 7.16: *The* complete graph *on 5 nodes.*

The command `gvgen -k N` is used to generate a *complete graph* with N nodes.

Complete bipartite graph (`-b`)

The nodes of *complete bipartite graph* are divided into two groups. Each node is connected to every node in the *other* group.

```
> gvgen -b2,3
graph {
  1 -- 3
  1 -- 4
  1 -- 5
  2 -- 3
  2 -- 4
  2 -- 5
}
```

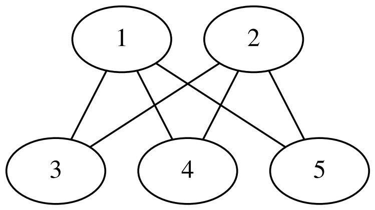

Figure 7.17: *A 2-by-3 complete bipartite graph.*

The command `gvgen -b N,M` is used to generate a *complete bipartite graph* with one group of N nodes and one of M nodes.

Grid (`-g`)

The command `gvgen -g N,M` generates an N by M rectangular *grid* of nodes.

```
> gvgen -g3,4
graph {
  1 -- 2
  1 -- 5
  2 -- 3
  2 -- 6
  3 -- 4
  3 -- 7
  4 -- 8
  5 -- 6
  5 -- 9
  6 -- 7
  6 -- 10
  7 -- 8
  7 -- 11
  8 -- 12
  9 -- 10
  10 -- 11
  11 -- 12
}
```

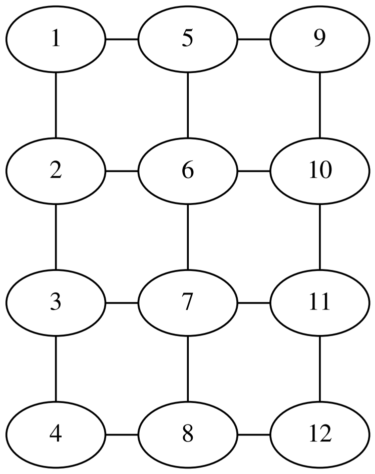

Figure 7.18: *A 3-by-4 rectangular grid.*

Folded grid (`-gf`) Appending the letter `f` to the *Grid* parameter (e.g., `gvgen -gf N,M`) will cause `gvgen` to generate a *folded grid*—one in which diagonally-opposite corners are joined by an edge.

```
> gvgen -gf3,4
graph {
  1 -- 2
  1 -- 5
  1 -- 12
  2 -- 3
  2 -- 6
  3 -- 4
  3 -- 7
  4 -- 8
  4 -- 9
  5 -- 6
  5 -- 9
  6 -- 7
  6 -- 10
  7 -- 8
  7 -- 11
  8 -- 12
  9 -- 10
  10 -- 11
  11 -- 12
}
```

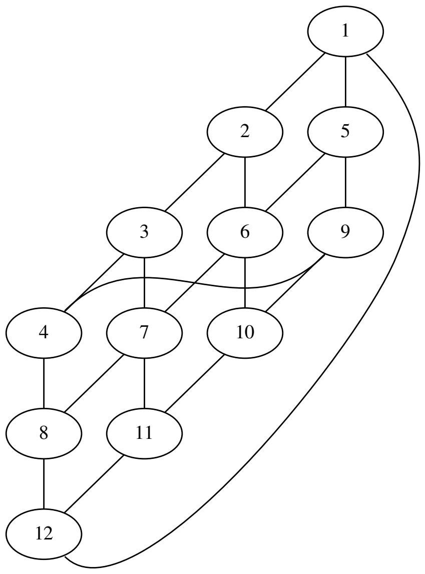

Figure 7.19: *A 3-by-4* folded *rectangular grid.*

Cylinder (`-C`)

A *cylinder* is a stack of identically-sized *cycles* stacked on top of each other, with the corresponding nodes in each *cycle* have been connected to form a *path*.[4]

```
> gvgen -C3,4
graph {
  1 -- 2
  2 -- 3
  3 -- 4
  1 -- 4
  5 -- 6
  6 -- 7
  7 -- 8
  5 -- 8
  9 -- 10
  10 -- 11
  11 -- 12
  9 -- 12
  1 -- 5
  5 -- 9
  2 -- 6
  6 -- 10
  3 -- 7
  7 -- 11
  4 -- 8
  8 -- 12
}
```

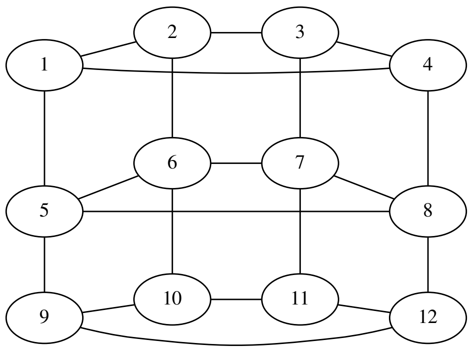

Figure 7.20: *A 3-by-4* cylinder.

The command `gvgen -C N,M` is used to generate a *cylinder* with *N* cycles of *M* nodes.

Ball (`-B`)

A *ball* is a *cylinder* that has been “pinched off” on each end by inserting a node at the top and bottom of the stack.

[4] Alternatively, you could think of a collection of identically sized *paths* arranged in a circle such that the corresponding nodes in each *path* are connected to form a *cycle*.

```
> gvgen -B2,3
graph {
  1 -- 2
  2 -- 3
  1 -- 3
  4 -- 5
  5 -- 6
  4 -- 6
  1 -- 4
  2 -- 5
  3 -- 6
  0 -- 1
  0 -- 2
  0 -- 3
  4 -- 7
  5 -- 7
  6 -- 7
}
```

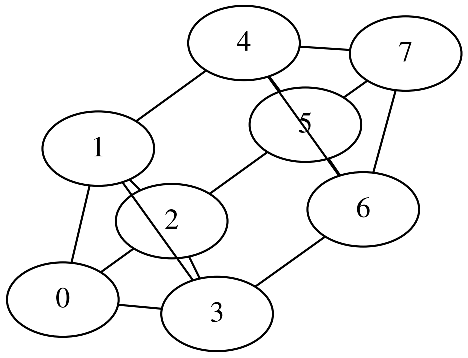

Figure 7.21: *A 2-by-3 ball.*

The command `gvgen -B N,M` is used to generate the *ball* created by adding nodes at the top and bottom of a *cylinder* with N cycles of M nodes.

Torus (`-T`)

In the field of mathematics known as *topology*, the term ***torus*** is used to describe objects shaped like a donut or wedding ring—any three-dimensional object with precisely one hole running through it[5]–as illustrated in Figure 7.22.

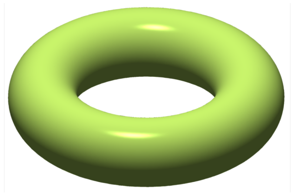

Figure 7.22: *A torus is a donut-shaped object, a three-dimensional surface with exactly one hole running through it.*

`gvgen` can create the graph-based equivalent of a *torus*. A ***torus***, as a graph is a *cylinder* that has been sort of folded back onto itself, connecting the corresponding nodes of the *cycles* at the top and bottom of the stack. Figure 7.23 depitcs a *torus* composed of three *cycles* of four nodes[6]

The command `gvgen -T N,M` is used to generate an N-by-M *torus*.

[5] Curiously, by this topological definition, most coffee cups are also *tori*, with one hole formed by the handle. To a topologist, donuts and coffee cups have the same shape.

[6] Although, if you look closely, you'll note that a three-by-four *torus* could be seen as either three *cycles* of four nodes or four *cycles* of three nodes.

```
> gvgen -T3,4
graph {
  1 -- 2
  2 -- 3
  3 -- 4
  1 -- 4
  5 -- 6
  6 -- 7
  7 -- 8
  5 -- 8
  9 -- 10
  10 -- 11
  11 -- 12
  9 -- 12
  1 -- 5
  5 -- 9
  1 -- 9
  2 -- 6
  6 -- 10
  2 -- 10
  3 -- 7
  7 -- 11
  3 -- 11
  4 -- 8
  8 -- 12
  4 -- 12
}
```

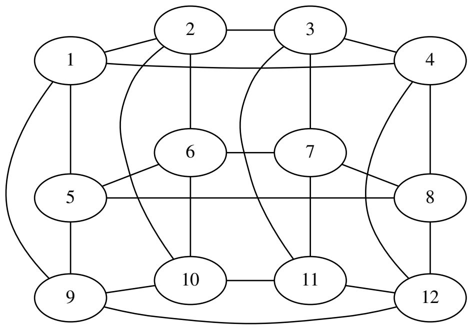

Figure 7.23: *A 3-by-4 torus graph. A torus is a cylinder in which the corresponding nodes at the top and bottom of have been connected.*

Hypercube (`-h`)

A *hypercube* of degree N is an N-dimensional cube.

This is a curious construct, and difficult to visualize for degree four and above, but we can develop a more intuitive sense of the structure by building it up iteratively.

The *hypercube* with degree zero is a single point (node).[7]

```
graph {
  1
}
```

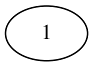

Figure 7.24: *The zero-dimensional* hypercube *is just a single point.*

The *hypercube*with degree *one* can generated by taking that point and "stretch-

[7] While the zero-dimensional hypercube is a good place to start constructing the higher-dimensional cubes, note that `gvgen` won't actually generate this degenerate case. `gvgen -h 0` yields an error.

ing” it, such that you’re left with two points (nodes) connected by a single line (edge).

```
> gvgen -h1
graph {
  1 -- 2
}
```

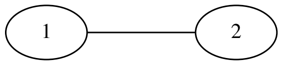

Figure 7.25: *The one-dimensional* hypercube *is two copies of the zero-dimensional version, with edges connecting corresponding nodes. Our point becomes a line.*

The *hypercube*with degree *two* is generated by taking that line and “stretching” it in a direction orthogonal to the first, such that you’re left with two lines with connected end-points, in other words, a square.

```
> gvgen -h2
graph {
  1 -- 2
  1 -- 3
  2 -- 4
  3 -- 4
}
```

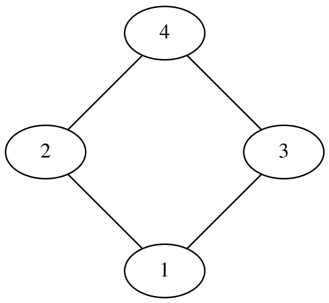

Figure 7.26: *The two-dimensional* hypercube *is two copies of the one-dimensional version, with edges connecting corresponding nodes. Our line becomes a square.*

The *hypercube* with degree *three* is generated by taking that square and “stretching” it in yet another direction orthogonal to the first two, such that you’re left with two squares with connected end-points, in other words, a cube.

More generally, the *hypercube* with degree *N* is generated by taking two copies of the *hypercube* with degree *N-1* and connecting the corresponding vertices.

The command `gvgen -h N` is used to generate an *N*-dimensional hypercube.

Sierpinski graph (`-S`)

The *Sierpiński graph* is a fractal structure that is defined by iteratively replacing portions of the graph with copies of itself. The number of recursive iterations is known as the *order* of the graph.

The *Sierpiński graph* of order one is simply a triangle.

To generate the *Sierpiński graph* of order *two*, we insert a new node at the midpoint of every edge. (Here, *node 4* between *nodes 1* and *3*, *node 5* between *nodes 1* and *2* and *node 6* between *nodes 2* and *3*), connecting the new nodes to form a triangle.

```
> gvgen -h3
graph {
  1 -- 2
  1 -- 3
  1 -- 5
  2 -- 4
  2 -- 6
  3 -- 4
  3 -- 7
  4 -- 8
  5 -- 6
  5 -- 7
  6 -- 8
  7 -- 8
}
```

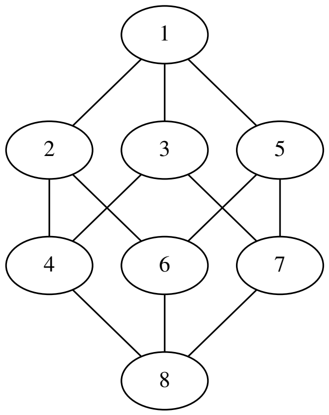

Figure 7.27: *The three-dimensional* hypercube *is two copies of the two-dimensional version, with edges connecting corresponding nodes. Our square becomes a cube.*

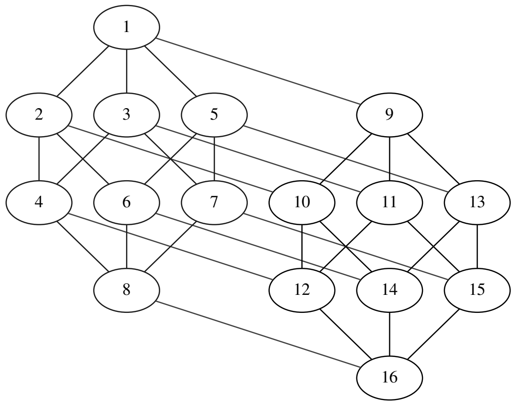

Figure 7.28: *The four-dimensional hypercube is two copies of the three-dimensional version, with edges connecting corresponding nodes. Our cube becomes an object that can't exist in a three-dimensional world.*

```
> gvgen -S1
graph {
  1 -- 2
  1 -- 3
  2 -- 3
}
```

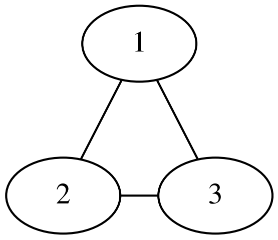

Figure 7.29: *The Sierpiński graph of order one.*

```
> gvgen -S2
graph {
  1 -- 4
  1 -- 5
  2 -- 5
  2 -- 6
  3 -- 4
  3 -- 6
  4 -- 5
  4 -- 6
  5 -- 6
}
```

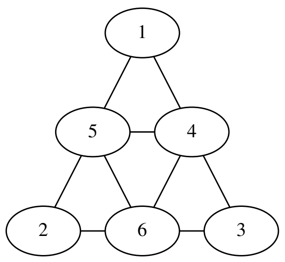

Figure 7.30: *The Sierpiński graph of order two.*

To generate the *Sierpiński graph* of order *three*, we repeat the process—introducing a new node at the midpoint of every edge–but only for the "upward" pointing triangles (when oriented as in Figure 7.31). We ignore the triangle formed by *nodes 4*, *5* and *6*.

To generate the *Sierpiński graph* of order *four*, we repeat the process. once again ignoring the downward-pointing triangles (*7-8-9*, *10-11-12*, *13-14-15*).

The command `gvgen -S N` is used to generate the *Sierpiński graph* of order *N*.

```
> gvgen -S3
graph {
  1 -- 7
  1 -- 8
  2 -- 10
  2 -- 11
  3 -- 13
  3 -- 14
  4 -- 8
  4 -- 9
  4 -- 14
  4 -- 15
  5 -- 7
  5 -- 9
  5 -- 11
  5 -- 12
  6 -- 10
  6 -- 12
  6 -- 13
  6 -- 15
  7 -- 8
  7 -- 9
  8 -- 9
  10 -- 11
  10 -- 12
  11 -- 12
  13 -- 14
  13 -- 15
  14 -- 15
}
```

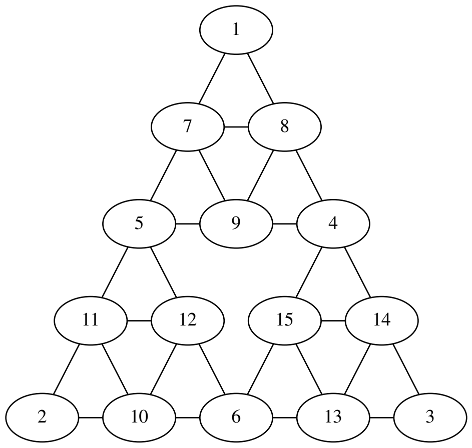

Figure 7.31: *The Sierpiński graph of order three.*

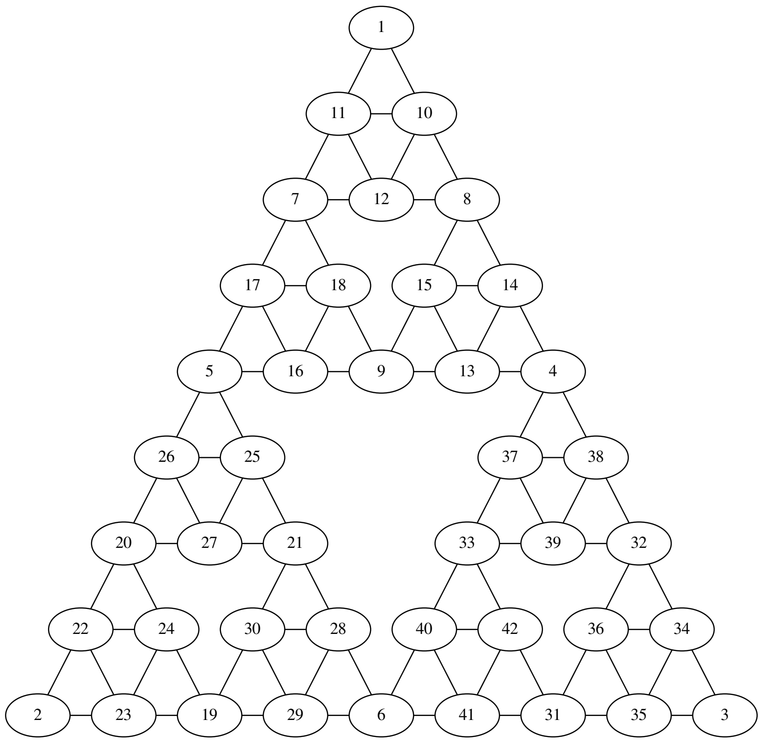

Figure 7.32: *The* Sierpiński graph *of order four.*

Other Command Line Options

Help (-?)

A brief summary of the `gvgen` command line parameters can be obtained by passing the `-?` flag (Figure 7.33).

```
> gvgen -?
Usage: gvgen [-dv?] [options]
 -c<n>          : cycle
 -C<x,y>        : cylinder
 -g[f]<h,w>     : grid (folded if f is used)
 -G[f]<h,w>     : partial grid (folded if f is used)
 -h<x>          : hypercube
 -k<x>          : complete
 -b<x,y>        : complete bipartite
 -B<x,y>        : ball
 -i<n>          : generate <n> random
 -m<x>          : triangular mesh
 -M<x,y>        : x by y Moebius strip
 -n<prefix>     : use <prefix> in node names ("")
 -N<name>       : use <name> for the graph ("")
 -o<outfile>    : put output in <outfile> (stdout)
 -p<x>          : path
 -r<x>,<n>      : random graph
 -R<n>          : random rooted tree on <n> vertices
 -s<x>          : star
 -S<x>          : sierpinski
 -t<x>          : binary tree
 -t<x>,<n>      : n-ary tree
 -T<x,y>        : torus
 -T<x,y,t1,t2>  : twisted torus
 -w<x>          : wheel
 -d             : directed graph
 -v             : verbose mode
 -?             : print usage
```

Figure 7.33: `gvgen`*'s help text. Also see* `man gvgen`.

Directed graphs (-d)

Although it generates un-directed graphs by default, `gvgen` can be used to create both directed and un-directed graphs. Add the `-d` flag to the command line to generate a directed graph (Figure 7.34).)

Output Files (-o)

`gvgen`'s output can be written to file in the typical Unix way (by redirecting the *STDOUT* stream):

```
> gvgen -h 3 > cube.gv
```

or by using the `-o` flag to specify an output file:

```
> gvgen -dp3
digraph {
  1 -> 2
  2 -> 3
}
```

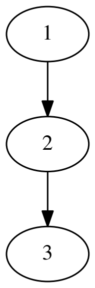

Figure 7.34: *A* directed *path on three nodes.*

```
> gvgen -h 3 -o cube.gv
```

Node Prefix (`-n`)

By default, `gvgen` assigns each node in the generated graph a number. The first node is generally named *1*, the second *2*, and so on.

The `-n PREFIX` option specifies a string to be prefixed to the name of each node in the generated graph, as demonstrated in Figure 7.35.

```
> gvgen -n Foo -p3
graph {
  Foo1 -- Foo2
  Foo2 -- Foo3
}
```

Figure 7.35: *The* `-n` *parameter specifies a string to be prefixed to the name of each node in the generated graph.*

Graph Name (`-N`)

By default, `gvgen` creates an unnamed graph such as code{graph { 1 -> 2 }}. The `-N NAME` option tells `gvgen` to use the specified name instead {Figure 7.36.}

```
> gvgen -N MyGraph -k4
graph MyGraph{
  1 -- 2
  1 -- 3
  1 -- 4
  2 -- 3
  2 -- 4
  3 -- 4
}
```

Figure 7.36: *The* `-N` *parameter specifies the name to assign to the generated graph.*

7.5 nop

`nop` is a pretty-printer and syntax-checker for DOT language graph descriptions. `nop` converts DOT language graph descriptions into a conventional format without changing the graph itself or the way it is rendered.

Example of Use

The command:

```
> nop MyGraph.gv
```

will convert the file `MyGraph.gv` to `nop`'s canonical format:

`nop` doesn't change the original file, but writes the re-formatted graph description to *STDOUT*.

```
> cat MyGraph.gv
digraph MyGraph{

  splines=curved // Use curved edges

  node [style=filled, shape=box]

  A -> B -> D -> { E; F }

  C -> D [style=dashed]

  B [color="Green", label="Beta"]

  A [label="Alpha",color="Red"]

}
```

```
> nop MyGraph.gv
digraph MyGraph {
  graph [splines=curved];
  node [shape=box,
    style=filled
  ];
  A [color=Red,
    label=Alpha];
  B [color=Green,
    label=Beta];
  A -> B;
  B -> D;
  D -> E;
  D -> F;
  C -> D [style=dashed];
}
```

Figure 7.37: `nop` *is a pretty-printer for DOT. It creates "normalized" versions of input graphs. Here we see the before and after view of a graph processed by* `nop`.

You can re-direct `nop`'s output stream to write it to a file:

```
> nop in.gv > out.gv
```

or pipe it to another program:

```
> nop in.gv | dot -Tpng -o out.png
```

to generate an image immediately.

7.6 `prune`

`prune` removes nodes from digraphs, writing the resulting digraph to *STDOUT*.

```
> echo ”digraph { A -> B -> C }” | prune -n B -N color=blue
digraph G {
  B [color=blue];
  A -> B;
}
```

Example of Use

The command:

```
> prune my-graph.gv -nA -nB -Nshape=box
```

transforms the digraph defined in `my-graph.gv`, pruning any and all nodes downstream from *Node A* or *Node B* and setting the `shape` attribute of *Node A* and *Node B* to the value `box`.

Command Line Options

Cut-point node (`-n`)

The parameter `-n` specifies a node that will act a as “cut-point” in the pruning process. All of the specified node's outbound edges are removed from the digraph, as are any nodes and edges that are separated from the rest of the graph by the removal of those outbound edges.

The `-n` parameter can be repeated multiple times to specify multiple cut-points.

```
> cat my-graph.gv
digraph {
  A -> B -> C -> E
  E -> { F, G }
  A -> G
  D -> C
}
```

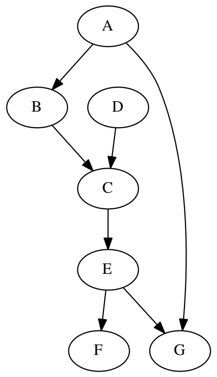

Figure 7.38: *An un-pruned digraph. See Figure 7.39.*

```
> prune my-graph.gv -nC
digraph {
  A -> B;
  A -> G;
  B -> C;
  D -> C;
}
```

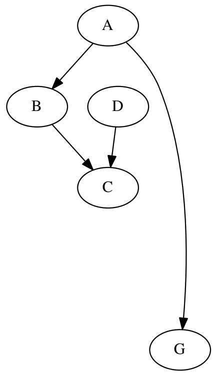

Figure 7.39: *The pruned digraph that results from running* `prune -nC` *on the digraph illustrated in Figure 7.38. Note that nodes E and F have been removed since the outbound edge from node C was the only directed path connecting them to the rest of the graph, but Node G was retained because it is still reachable via the outbound edge from node A.*

Cut-point node Attribute (`-N`)

The parameter `-N` specifies an attribute to be assigned to each node identified as a cut-point (via the `-n` parameter.

The `-N` parameter can be repeated multiple times to specify multiple attributes. Note that `prune` assigns the specified attributes to every node listed in a `-n` parameter, whether or not any outbound edges or downstream nodes were actually removed from the graph.

Help (`-?`)

When `-?` (or `-h`) is specified, `prune` prints a brief summary of its command line options.

```
> cat my-graph.gv
digraph {
  A -> B;
  B -> C;
  B -> D;
  C -> F;
  D -> E;
  E -> F;
}
```

```
> prune my-graph.gv -nC -nD
 -Ncolor=orange -Nshape=box
digraph {
  A -> B;
  C [color=orange,
    shape=box];
  B -> C;
  D [color=orange,
    shape=box];
  B -> D;
}
```

Figure 7.40: *A digraph before (left) and after (right) pruning. The* `-N` *parameter specifies attributes to be assigned to each node that was used as a cut-point.*

```
> prune -?
Usage: prune [options] [<files>]

Options:
  -h :           Print this message
  -? :           Print this message
  -v :           Verbose
  -n<node> :     Name node to prune
  -N<attrspec> : Attribute specification to apply to pruned nodes

Both options '-n' and '-N' can be used multiple times on the command line
```

Figure 7.41: `prune`*'s usage summary.*

See Figure 7.41 for an example.

7.7 `tred`

When there is a directed *path* from *Node A* to *Node Z* in a given digraph, we might say that *Node Z* is ***reachable*** from *Node A*.

Consider, for example, the digraph in Figure 7.42. In this digraph, *Nodes B*, *C*, *D* and *E* are all *reachable* from *Node A*, *Nodes D* and *E* are *reachable* from *Node C*, and so on.

Note that one could remove several of the edges from Figure 7.42 without changing the list of nodes that are reachable from any given starting point. E.g., if one were to delete *Edge A->D*, *Node A* is would still be *indirectly* connected to *Node D* via the path from *A* to *B* to *C* to *D*.

`tred` is a digraph filter that will eliminate these "redundant" edges, yielding the graph with the minimum number of edges needed to ensure that each node can *reach* all of the nodes that it could reach at the start. This process is known as *transitive reduction* (and the name "tred" is derived from *t*ransitive *red*uction).

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

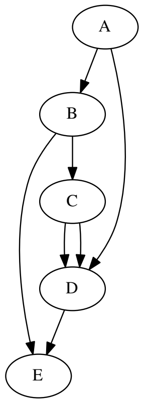

Figure 7.42: *TK*

For example, eliminating *Edge A->D*, *B->E* and one of the edges from *Node C* to *Node D* (as shown in Figure 7.43) yields a subgraph in which every previously *reachable* node remains *reachable*.

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

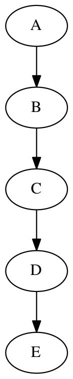

Figure 7.43: *TK*

With a bit of examination you can convince yourself that Figure 7.43 contains smallest number of edges possible without breaking the graph into multiple *components*. In fact, Figure 7.43 contains the *unique* subgraph with this minimal number of edges.

`tred` peforms precisely this kind of graph reduction—eliminaing “redundant” edges to create the smallest possible still-connected subgraph.

Example of Use

The command:

```
> tred MyGraph.gv
```

will perform a transitive reduction on the digraph stored in the file `MyGraph.gv`.

Command Line Parameters

`tred` accepts a of list of one or more DOT files containing graphs to be reduced.

When no files are specified on the command line, `tred` reads digraph data from *STDIN*.

Additional Notes

- `tred` only works with digraphs. Any undirected graphs `tred` encounters will be silently ignored.
- When a digraph contains a *cycle*, there is more than one subgraph that meets the transitive reduction criteria. `tred` still works (by more-or-less arbitrarily choosing which subgraph to present), but it will also report a warning stating that the given solution is not unique.

7.8 `unflatten`

When two nodes have the same *rank*, `dot` will place them on the same horizontal line.[8] Hence when a digraph has a large number of nodes that share the same *rank*—such as when the graph is composed of many *disconnected components* or contains a node with a large number of *neighbors*—`dot`'s default layout logic can result in very wide images.

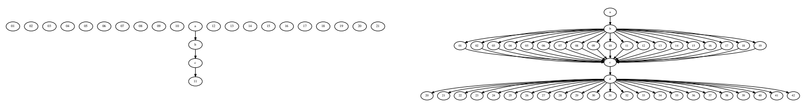

Figure 7.44: *Examples of digraphs that* `dot` *renders as very wide images by default. On the left, a digraph with a large number of disconnected components. On the right, a digraph with a large number of leaf nodes.*

`unflatten` can help in this situation.

`unflatten` preprocesses a graph description and make changes that lead `dot` to generate an image with an *aspect ratio* that is closer to 1:1. In other words, `unflatten` can literally "unflatten" some graphs so that the `dot`-rendered image of the graph is proportioned more like a square.

Specifically, `unflatten` can make changes to a graph in one of two ways:

[8] Unless `rankdir` is set to `LR`, in which case `dot` will place them on the same vertical line.

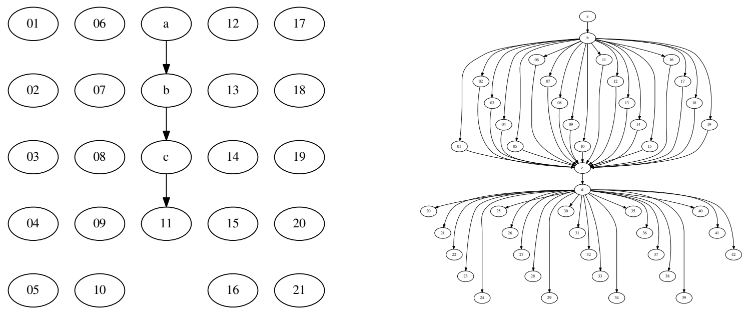

Figure 7.45: *The graphs from Figure 7.44, processed with* `unflatten -c4` *(left) and* `unflatten -f15` *(right) to bring their aspect ratios closer to 1:1.*

1. `unflatten` can add hidden edges to a collection of disconnected nodes in order to split them up into multiple rows.
2. `unflatten` can set the `minlen` attribute on the inbound edge of *leaf* nodes (with exactly one inbound edge) in order to distribute the "children" of a given node across multiple rows.

These behaviors are independently controlled by command line parameters.

Example of Use

The command:

```
> unflatten MyGraph.gv -c 3
```

will process the file `MyGraph.gv` with `unflatten`, connecting otherwised "orphaned" nodes into chains up to `3` edges (`4` nodes) long. The resulting DOT format graph description is written to *STDOUT*.

Command Line Parameters

Chains of Disconnected Nodes (`-c`)

The command line parameter `-c` specifies the number of disconnected nodes to chain together with invisible edges. Specifically, in the command `unflatten -c N`, the `N` parameter is the number of invisible edges to use when forming chains of otherwise isolated nodes. These chains will include `N+1` nodes.

For example, the graph on the left side of Figure 7.45 was generated with the command:

```
> unflatten MyGraph.gv -c 4
```

distributing the otherwise isolated nodes across 5 (N+1) rows.

Length of Leaf-Edges (`-l`)

The command line parameter `-l` specifies the number of rows across which `unflatten` will distribute leaf nodes. The parameter `-l N` will lead `unflatten` to assign a `minlen` value between `1` and `N` (inclusive) to each leaf node on the graph (with exactly one inbound edge).

For example the command:

```
> unflatten MyGraph.gv -l 5
```

distributes each *leaf node* (with only one inbound edge) at up to five different lengths from its “parent”, generating an image like that found on the right side of Figure 7.46.

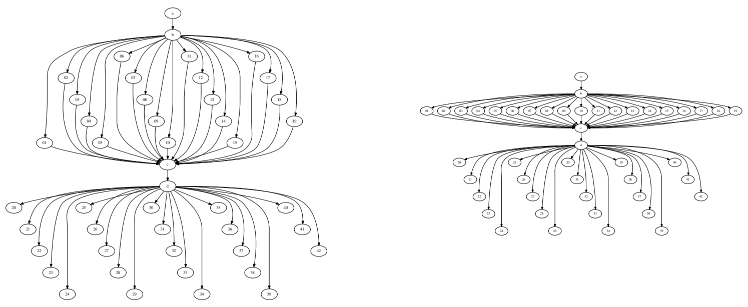

Figure 7.46: *A graph process by* `unflatten` *with (left) and without (right) the* `-f` *parameter. Absent the* `-f` *parameter* `unflatten` *only adjusts the* `minlength` *attribute of leaf nodes. With the the* `-f` *parameter* `unflatten` *only adjusts the* `minlength` *attribute of any node with exactly one inbound edge and at most one outbound edge.*

Length of Non-Leaf Edges (`-f`)

Adding the parameter `-f` *in addition to* the `minlength` parameter `-l` allows `unflatten` to apply the same `minlength`-staggering logic to non-leaf nodes with exactly one inbound and one outbound edge.

For example, the command:

```
> unflatten MyGraph.gv -f -l 5
```

distributes both leaf and non-leaf nodes (with only one inbound edge and at most one outbound edge) at up to five different lengths from its "parent", generating an image like that found on the left of Figure 7.46.

Output File (`-o`)

`unflatten`'s output can be written to file in the typical Unix way (by redirecting the *STDOUT* stream):

```
> unflatten in.gv -l 3 > out.gv
```

or, using the `-o` flag:

```
> unflatten in.gv -l 3 -o out.gv
```

one can specify a file to write to.