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 26

A treemap of folder contents

A ***treemap*** is a diagram that uses rectangular tiles to represent hierarchical collections of quantitative data. The *area* of each tile is proportional to the *magnitude* of the corresponding data point. The tiles are grouped into rectangular clusters according to the hierarchical relationship of the data points they represent, and tiles and clusters are further grouped into larger and larger rectangles until the top of the hierarchy is reached.[1]

Treemaps are particularly suited for visualizing the distribution of some resource across nested categories. For instance, a treemap might represent the distribution of a company's staff across divisions, departments and teams, or the way in which a family's budget is allocated to various categories and subcategories of expenditures.

For the same reason, several applications use treemaps to visualize how much disk space is taken up by each file and directory on a hard drive. In fact, treemaps were literally invented to solve this very problem.[2]

In this recipe, we'll use `patchwork`, a relatively new addition to the Graphviz family, to create treemaps representing the allocation of disk space across the files and sub-directories of a given folder on a hard drive,

26.1 What this recipe shows

- How to gather raw data with Unix programs like `du` and `find`.
- How to transform raw data into valid DOT files using `awk`.
- How to use `patchwork` to create treemaps.
- How to automate a multi-step process with a shell script.

[1] See Section 6.7 for more background on treemaps.

[2] Treemaps and the first treemap rendering algorithms were developed by Ben Shneiderman at the University of Maryland in 1990. Shneiderman's 1991 paper "Tree visualization with treemaps: a 2-d space-filling approach" was published in the January 1992 edition of the journal *ACM Transactions on Graphics*. See `http://www.cs.umd.edu/hcil/treemap-history/` for a history and much more information.

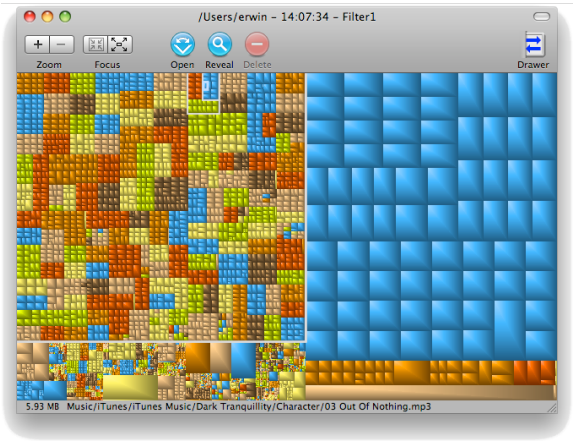

Figure 26.1: *A screenshot of* `GrandPerspectiv`*, an Apple's OS X application for visualizing disk usage as a tree map.*

- How to modify DOT files using `gvpr`.
- How to use the `colorscheme` attribute to define a color palette.

26.2 Ingredients

- `du`, a Unix application for calculating the size of files and directories (Section 9.1).
- `awk`, a Unix application for scanning and processing text files (Section 9.2).
- `patchwork`, Graphviz's treemap layout engine (Section 6.7).
- `bash`, a shell-scripting environment for Unix, Apple's OS X and Cygwin for Microsoft Windows (Section 9.3).
- `find`, a Unix application for creating and filtering lists of files (Section 9.6).
- `gvpr`, Graphviz's `awk`-like tool for modifying graphs (**??**).

26.3 The recipe

The high-level plan

Our objective is to create a process that can generate a treemap diagram representing the contents of an arbitrary directory on the file system.

Each tile in the treemap will represent an individual file. Each cluster will represent a directory. The area of each tile (and each cluster) will be proportional to the disk space used by the corresponding file (or sub-directory).

Our process will need to take the following steps:

1. Create a list (by name and size) of each file and sub-directory within the specified directory.
2. Convert that list into a DOT format graph that can be processed by `patchwork`.
3. Use `patchwork` to convert that DOT file into our final image.

We'll use `du` to implement the first step and `awk` to implement the second (and of course, `patchwork` to implement the third).

A modest start

So that we can more easily wrap our heads around the problem, let us first tackle a simpler objective: Let's create a "flat" treemap that represents only that content *directly* contained within the root directory.

We can later build on this solution to handle the full directory tree.

A `patchwork` friendly DOT file

Working backwards, let's begin by considering the output of Step 2. What should the DOT file look like?

Recall that when `patchwork` converts a DOT file into a treemap, every node in the graph becomes a tile in the treemap. The area of each tile is determined by the `area` attribute of the corresponding node.

Hence a basic `patchwork`-friendly graph representation of a directory will contain one node for every file within the directory, and each node will have an `area` attribute proportional to the size of the corresponding file. For example, Figure 26.2 lists a small directory (left) and the corresponding DOT format graph description (right).

```
> tree "My Documents/"
My Documents/
|-- accounts.xls
|-- a-video.avi
|-- cal.pdf
|-- memo.doc
|-- photo-1.jpg
|-- photo-2.jpg
|-- photo-3.jpg
|-- song-1.mp3
`-- song-2.mp3

0 directories, 9 files
```

```
graph "My Documents" {

  "accounts.xls" [area=12.9]
  "a-video.avi"  [area=46.7]
  "cal.pdf"      [area= 4.5]
  "memo.doc"     [area= 6.5]
  "photo-1.jpg"  [area=14.5]
  "photo-2.jpg"  [area=15.6]
  "photo-3.jpg"  [area=12.1]
  "song-1.mp3"   [area=27.9]
  "song-2.mp3"   [area=18.8]

}
```

Figure 26.2: *A* `patchwork`*-friendly DOT representation of a single directory's contents. The graph on the right represents the directory listing on the left. Each node represents a file, identified by name. Here, the* `area` *attribute is the size of the corresponding file (in hundreds of kilobytes). A rendering of this graph can be found in Figure 26.3.*

In Figure 26.2, the identifier for each node is the name of the corresponding file. The `area` associated with each node is the size of the file, in hundreds of kilobytes.[3]

Rendering this DOT file with `patchwork` (via `patchwork -Tpng mydocuments.gv`, for example) yields an image like that found in Figure 26.3.

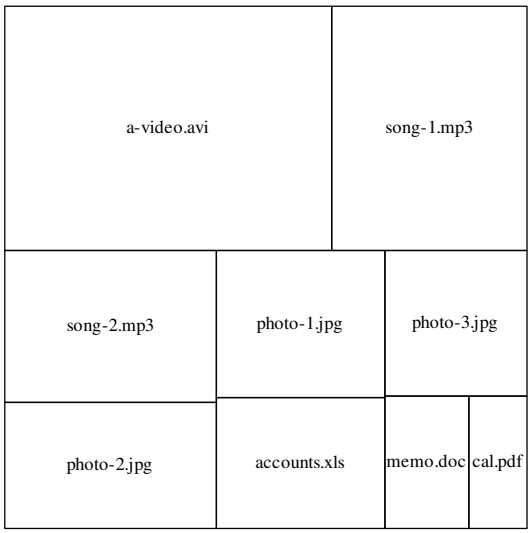

Figure 26.3: *Our basic directory listing graph, as rendered by* `patchwork`*. Also see Figure 26.2.*

[3] The reason for this unconventional unit is discussed below.

The raw input data

Now that we have a better sense of the DOT file we're trying to create, let's consider the raw data we'll have to work with.

`du` is tailor-made for Step 1 of our process. It can easily enumerate the name and size of each item within a directory, and the default output is already pretty close to the format we're aiming for. For example, when we run `du *` from within the `My Documents` directory listed in Figure 26.2, `du` generates something like Figure 26.4.[4]

```
> du -d 0 *
4672  a-video.avi
1288  accounts.xls
452 cal.pdf
652 memo.doc
1448  photo-1.jpg
1556  photo-2.jpg
1212  photo-3.jpg
2792  song-1.mp3
1884  song-2.mp3
```

Figure 26.4: *The contents of* `My Documents` *(Figure 26.2), as reported by* `du` *(when run from within the* `My Documents` *directory). The* `-d 0` *flag on the command line ensures that we'll only see the direct contents of* `My Documents` *in the report, even if it happens to contain sub-directories.*

Note that for each file in `My Documents`, `du` outputs a line of the form:

```
[SIZE-IN-BYTES] <TAB> [FILENAME]
```

We "just" need to flip the columns around and insert some boilerplate text to create valid DOT node and graph declarations.

The conversion

We'll use `awk` to convert the raw data generated by `du` into the `patchwork`-friendly DOT file.[5]

We are seeking a DOT file of the form:

```
graph {
  "FILENAME" [area=SIZE]
}
```

[4] See Section 9.1 for more information about `du`.
[5] See Section 9.2 for more information about `awk`.

where SIZE and FILENAME are the first and second fields of each line of du's output. The AWK program in Figure 26.5 does this.

```
1 BEGIN { print "graph {" }
2 { print "  \"" $2 "\" [area=" $1 "]" }
3 END { print "}" }
```

Figure 26.5: *An AWK script that generates a valid DOT format graph descriptor based on the content of a directory as reported by* du. *See Section 9.2 for more information about* awk *and Section 9.1 for more information about* du.

Lines 1 and 3 — of Figure 26.5 are each invoked once, at the beginning and end of the processing, respectively. We use them to generate the graph { and } text that wraps the rest of our data.

Line 2 — is the action that is fired for each line of input. We use $1 and $2 to print the first and second tab-delimited field in each line, adding the node and area declarations around the input data.

Using the -f parameter to load the AWK script from a file, we can invoke our end-to-end conversion as seen in Figure 26.6.

```
> du -d 0 * | \
 awk -F "\t" -f script.awk
graph {
  "a-video.avi" [area=4672]
  "accounts.xls" [area=1288]
  "cal.pdf" [area=452]
  "memo.doc" [area=652]
  "photo-1.jpg" [area=1448]
  "photo-2.jpg" [area=1556]
  "photo-3.jpg" [area=1212]
  "song-1.mp3" [area=2792]
  "song-2.mp3" [area=1884]
}
```

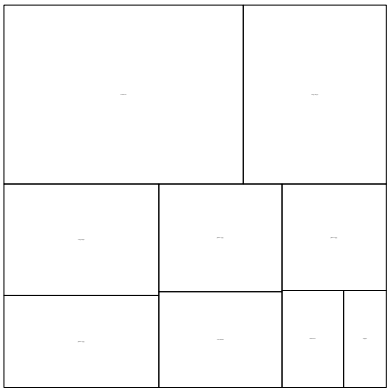

Figure 26.6: *A DOT format graph and* patchwork *rendering based on the contents of a directory as reported by* du *and transformed by* awk.

Figure 26.6 is *almost* right, but the node labels are so small that you have to look closely in order to see them at all. The trouble is that patchwork doesn't (currently) make any attempt to scale the size of the node labels to match the overall size of the diagram. Hence when large area attributes are used, the size of the tiles dwarf the size of the labels. To work around this, we must scale the area attributes down to a more reasonable size. Since patchwork's default area value is 1, we should keep the area values between 1 and, say, 100.

Luckily, du provides an option that makes this easy. The -B flag can be used to specify the base unit used when reporting the size of files and directories. Our files vary in size roughly from 500 KB to 30 MB. That makes bytes and kilobytes a little bit too small for us to use, and megabytes a little bit too large, but reporting the size of each file in hundreds of kilobytes (one tenth of a megabyte) works out about right. To achieve this, we can pass the argument -B 100K to du. Our revised DOT file and the corresponding treemap image can be found in Figure 26.7.

```
> du -d 0 -B 100K * | \
 awk -F "\t" -f script.awk
graph {
  "a-video.avi" [area=47]
  "accounts.xls" [area=13]
  "cal.pdf" [area=5]
  "memo.doc" [area=7]
  "photo-1.jpg" [area=15]
  "photo-2.jpg" [area=16]
  "photo-3.jpg" [area=13]
  "song-1.mp3" [area=28]
  "song-2.mp3" [area=19]
}
```

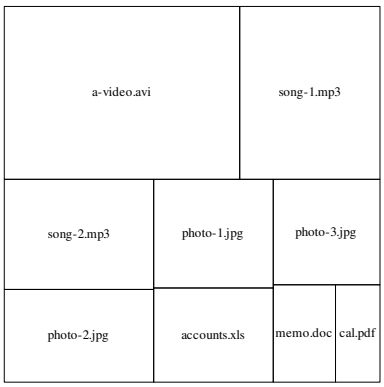

Figure 26.7: *A scaled-down DOT format graph and* patchwork *rendering based on the contents of a directory as reported by* du *(with the* -B 100k *parameter) and transformed by* awk.

Not bad. We can now programatically create a basic treemap representing the contents of a single directory.

Scripted

To avoid typing the entire command found in Figure 26.7 every time we want to create a treemap, we can create a small shell script to automate the process.

```
1 #!/bin/bash
2 AWK_SCRIPT="
3  BEGIN { print \"graph {\" }
4  { print \"  \\\"\" \$2 \"\\\" [area=\" \$1 \"]\" }
5  END { print \"}\" }
6 "
7 cd "$1"
8 du -d 0 -B 100K * | awk -F "\t" "${AWK_SCRIPT}"
```

Figure 26.8: *A* bash *script that automates our process for creating a* patchwork*-friendly DOT format graph descriptor from the files and sub-directories directory contained within a given directory.*

Looking at the script line by line:

Line 1 — the "she-bang" declaration that tells the operating system how to execute this script. (In this case, using the `bash` interpreter.)

Lines 2 through 6 — create a variable (named `AWK_SCRIPT`) containing our AWK script.

Line 7 — moves into the specified root directory (`$1`, representing the first parameter passed to the script on the command line). This allows the output from `du` to be relative to the specified root directory.

Line 8 — executes `du` and processes it with our `awk` script, generating the `patchwork`-friendly DOT format graph description.

Saving the contents of Figure 26.8 into a file named `treemapper` (and ensuring the file is executable using the command `chmod a+x treemapper` or something similar), we can then run:

```
treemapper "My Documents"
```

to generate the graph description, or:

```
treemapper "My Documents" | patchwork -Tpng -o my-documents.png
```

to pipe that description directly to `patchwork` in order to generate the `my-documents.png` image immediately.

A more robust solution

Our script would be more useful, of course, if it could handle more than one level of the directory tree.

Let's turn our attention to the larger challenge of mapping the full tree beneath the specified directory.

The multi-level DOT file

Recall that `patchwork` keeps the nodes within a cluster together so that the DOT file's clusters form the clustered groups of tiles within the treemap.

The clusters aren't given an `area` directly. The area of the cluster is the total area of all the files it contains.

Hence our enhanced `patchwork`-friendly graph will once again include a node for every file found in the specified directory tree, but this time we will include a cluster for every sub-directory. An example can be found in Figure 26.9.

```
> tree "My Documents/"
My Documents/
|-- cal.pdf
|-- Documents/
|   |-- accounts.xls
|   `-- memo.doc
`-- Media/
    |-- Music/
    |   |-- song-1.mp3
    |   `-- song-2.mp3
    |-- Pictures/
    |   |-- photo-1.jpg
    |   |-- photo-2.jpg
    |   `-- photo-3.jpg
    `-- Video/
        `-- a-video.avi

5 directories, 9 files
```

```
graph "My Documents" {
  "cal.pdf" [area=4.5]
  subgraph "cluster Documents" {
    "accounts.xls" [area=12.9]
    "memo.doc" [area=6.5]
  }
  subgraph "cluster Media" {
    subgraph "cluster Music" {
      "song-1.mp3" [area=27.9]
      "song-2.mp3" [area=18.8]
    }
    subgraph "cluster Pictures" {
      "photo-1.jpg" [area=14.5]
      "photo-2.jpg" [area=15.6]
      "photo-3.jpg" [area=12.1]
    }
    subgraph "cluster Video" {
      "a-video.avi" [area=46.7]
    }
  }
}
```

Figure 26.9: *A* `patchwork`*-friendly DOT language description of a graph representing a complete directory tree. Each file is represented by a node. Each directory is represented by a cluster.*

Rendering this DOT file with `patchwork` yields an image like that found in Figure 26.10.

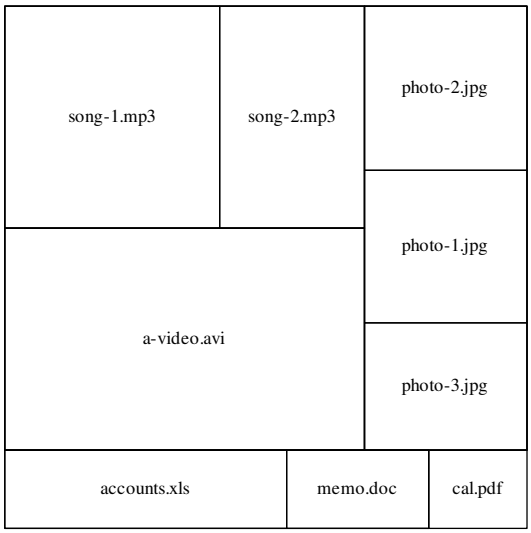

Figure 26.10: *A* `patchwork` *rendering of the DOT file listed in Figure 26.9. Note that each cluster forms a contiguous rectangle within the treemap.*

Note that unlike the treemap in Figure 26.3, in Figure 26.10 the files (and sub-directories) that are contained within a given directory are grouped into rectangular clusters. (This is particularly noticeable in the placement of the tiles representing `song-2.mp3` and `photo-2.jpg`.)

The raw input data

Recall that by default, when given a directory `du` will enumerate all of the sub-directories (and none of the files) from the specified directory down, and that when the command line parameter `-a` is provided `du` will enumerate both files and directories.

So using `du` we can get a listing of the sub-directories contained within our root directory or a listing of the files *and* sub-directories contained within our root directory. But `du` doesn't seem to offer a direct way to get at the information we really want, a list of *files* contained within the directory tree, without any of the sub-directories.

This makes our task is a little trickier than before.

As may have already occurred to experienced Unix users, the solution is to take advantage of the Unix program `find`.

Like `du`, `find` can recursively list the contents of a directory tree. `find` won't tell us the size of the files, but it is a veritable Swiss army knife for scanning and filtering files and directories.

We can leverage two of `find`'s capabilities in particular:

1. The `-type f` predicate, which will restrict `find`'s output to files (ignoring directories).
2. The `-exec` action, which allows us to specify an action to perform for each file that is found.

Specifically, with these options we can use `find` to list each *file* in the directory tree and execute `du` on each to determine its size, as seen in the command listed in Figure 26.11.

```
> find "My Documents" -type f -exec du -B 100k {} \;
47  My Documents/Media/Video/a-video.avi
28  My Documents/Media/Music/song-1.mp3
19  My Documents/Media/Music/song-2.mp3
16  My Documents/Media/Pictures/photo-2.jpg
15  My Documents/Media/Pictures/photo-1.jpg
13  My Documents/Media/Pictures/photo-3.jpg
7 My Documents/Documents/memo.doc
13  My Documents/Documents/accounts.xls
5 My Documents/cal.pdf
```

Figure 26.11: *The size of each regular file within* `My Documents`*, as identitified by* `find` *and reported by* `du`*.*

Loosely translated, the command in Figure 26.11 says:

1. `find ¨My Documents¨`
   Find all files and sub-directories below `My Documents`.
2. `-type -f`
   Filter out everything that isn't a "normal" file.
3. `-exec du -B 100k {} \;`
   Execute `du -B 100k` for every file (`{}`) that was found.

This leaves us with the size and full path and name of every file in the specified directory.

The conversion

The output in Figure 26.11 tells us precisely which directories and sub-directories each file is contained in, but it does this on a file-by-file basis. In order to create a DOT file that groups each file into the appropriate clusters (as in Figure 26.10) we'd need to do a fair bit of processing.[6] Luckily, we don't have to.

Taking advantage of the split-declaration technique described in Chapter 21, we won't bother trying to bundle together all of a cluster's node declarations into a contiguous group. We'll just repeat the cluster declaration for each node it contains. I.e., rather than declaring each cluster exactly once:

[6] This processing would be relatively straightforward with a procedural programming language, but it is substantially more difficult to implement using a single-pass, line-by-line view of the input such as that provided by `awk`.

```
graph {
  subgraph clusterOne {
    A
    subgraph clusterTwo {
      B
      C
    }
  }
}
```

we can just "reopen" the cluster on every line that needs it:

```
graph {
  subgraph clusterOne { A }
  subgraph clusterOne { subgraph clusterTwo { B } }
  subgraph clusterOne { subgraph clusterTwo { C } }
}
```

This format fits much better into the `awk` programming model. Our logic is essentially the same as it was before (Figure 26.5). The only change is that rather than simply outputting the file name as the node identifier, we much split the file and path into individual segments, and provide a cluster declaration for each segment other than the very last one. The `awk` script in Figure 26.5 is one such implementation.

```
1  BEGIN { print "graph {" }
2  {
3    printf "  "
4    size = split($2,segment,"/")
5    for (i=1; i <size; i++) {
6      printf "subgraph \"cluster " segment[i] "\" { "
7    }
8    printf "\"" segment[size] "\" [area=" $1 "]"
9    for (i=0; i <size-1; i++) {
10     printf " }"
11   }
12   print ""
13 }
14 END { print "}" }
```

Figure 26.12: *An AWK script for transforming a recursive* `du` *listing into a* `patchwork`*-friendly graph containing clusters corresponding to each sub-directory of the specified* root *directory.*

Let's look at the `awk` script in detail:

Line 1 and 14 — as before, the first and last line generate the `graph {` and `}` text that wraps the rest of our data.

Lines 2 through 13 — define the action that is invoked on every line of input.

Line 3 and 12 — for readability, each line of output begins with a small indent and ends with a newline character.

Line 4 — splits each input file path (`$2`) into a `/`-delimited array of path segments, and stores it into a variable named `segment`.

Lines 5 through 7 — loop through the elements of the array, printing the beginning part of the cluster declaration.

Line 8 — prints the node definition, using the name of the file as the identifier and the size of the file as the `area` attribute.

Lines 9 through 11 — loop through the elements of the array once again, printing the `}` character that closes the previously opened cluster declarations.

Putting it all together, Figure 26.13 lists the graph generated by the revised script in Figure 26.12 when run on our example directory.

```
> find "My Documents" -type f -exec du -B 100k {} \; | \
   awk -F "\t" -f script.awk
graph {
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Video" { "a-…
    video.avi" [area=47] } } }
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Music" { "song…
    -1.mp3" [area=28] } } }
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Music" { "song…
    -2.mp3" [area=19] } } }
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Pictures" { "…
    photo-2.jpg" [area=16] } } }
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Pictures" { "…
    photo-1.jpg" [area=15] } } }
  subgraph "cluster My Documents" { subgraph "cluster Media" { subgraph "cluster Pictures" { "…
    photo-3.jpg" [area=13] } } }
  subgraph "cluster My Documents" { subgraph "cluster Documents" { "memo.doc" [area=7] } }
  subgraph "cluster My Documents" { subgraph "cluster Documents" { "accounts.xls" [area=13] } }
  subgraph "cluster My Documents" { "cal.pdf" [area=5] }
}
```

Figure 26.13: *The DOT format graph created by applying the AWK script defined in Figure 26.12 to the directory listing in Figure 26.9.*

When rendered with `patchwork`, this yields:

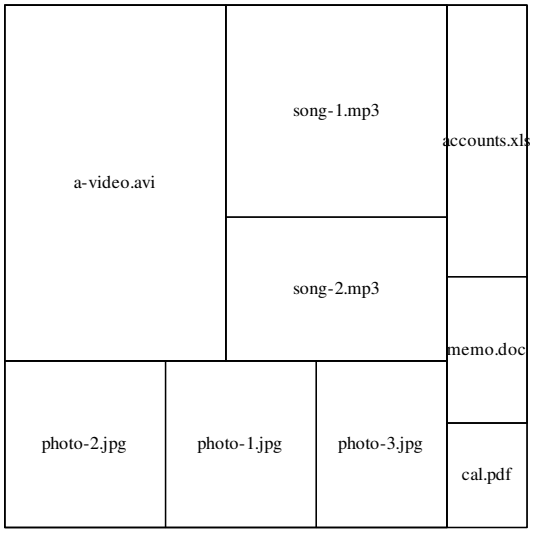

Figure 26.14: *The graph in Figure 26.13, as rendered by* `patchwork`.

If you're troubled by the format of Figure 26.13, you could always run the output through `nop` (by adding `| nop` to the end of the command) to clean things up a bit. `nop` will consolidate the cluster definitions to create a listing similar to that found in Figure 26.9.

Re-scripted

Once again, we can create a small shell script to automate the process.

```
1  #!/bin/bash
2  AWK_SCRIPT="
3  BEGIN { print \"graph {\" }
4  {
5    printf \"  \"
6    size = split(\$2,segment,\"/\")
7    for (i=1; i <size; i++) {
8      printf \"subgraph \\\"cluster \" segment[i] \"\\\" { \"
9    }
10   printf \"\\\"\" segment[size] \"\\\" [area=\" \$1 \"]\"
11   for (i=0; i <size-1; i++) {
12     printf \" }\"
13   }
14   print \"\"
15 }
16 END { print \"}\" }
17 "
18 PARENT_DIR="`dirname "$1"`"
19 cd "${PARENT_DIR}"
20 ROOT_DIR="`basename "$1"`"
21 find "${ROOT_DIR}" -type f -exec du -B 100k {} \; | \
22   awk -F "\t" "${AWK_SCRIPT}"
```

Figure 26.15: *A* `bash` *script automating our process for creating a* `patchwork`*-friendly, DOT format graph description based on the contents of an arbitrary directory.*

Reviewing the script in detail:

Line 1 — the "she-bang" declaration that tells the operating system how to execute this script. (In this case, using the `bash` interpreter.)

Lines 2 through 17 — write our AWK script into a variable named `AWK_SCRIPT`.

Line 18-19 — computes the location of the *parent* of the specified root directory (`$1`), saving it to a variable named `PARENT_DIR`, and then moves into that directory. This ensures that the output of `find` will describe the files relative to `ROOT_DIR`, excluding the path up to and including `PARENT_DIR`.

Line 20 — computes the location of the specified root directory (`$1`) relative to its parent directory, saving it to a variable named `ROOT_DIR`

Lines 21 through 22 — execute our command, generating a graph representing the files within `ROOT_DIR`.

As before, we can save that script into a file named `treemapper` (and ensure the file is executable using `chmod` or similar) and then run

```
> treemapper "My Documents"
```

to generate the graph description, or:

```
> treemapper "My Documents" | patchwork -Tpng -o my-documents.png
```

to pipe that description directly to `patchwork` in order to generate the `my-documents.png` image immediately.

26.4 Extensions

Color coding by file type

A "basic" treemap illustrates both the *magnitude* of each data point and the hierarchical *structure* of the overall data set. We can use color to add yet another dimension to the diagram.

For example, hard-drive analysis programs will often color-code the tiles within the treemap to indicate the *type* of file that it represents—one color for images, another for text, another for audio, etc.

Let's extend our treemapping process to color-code each tile according to the file-extension of the file it represents.

To color-code our treemap manually, we'd make sure that the `style` of each node is `filled`, and assign a `fillcolor` value to each node according to the extension of the corresponding file. E.g., something like:

```
song-1.mp3 [style=filled, fillcolor=green]
```

We *could* modify our AWK script to look at each file's extension and assign the appropriate color, but Graphviz's `gvpr` application offers an easier way.

Recall that `gvpr` will process a DOT file invoking user-defined *actions* when a node, edge or graph is encountered. Hence we can define an node *action* to parse the file-extension from the node's identifier and assign the appropriate `fillcolor`. The `gvpr` script in Figure 26.16 does just that.

Let's look at the script in Figure 26.16 more closely.

Lines 1 - 33 — define a `BEGIN` action that is invoked exactly once, at the beginning of `gvpr`'s processing. We use this block to initialize some variables we'll use later on in the processing.

- **Lines 2 - 8** — define an associative array (named `colormap`) that maps generic file types like `image` or `audio` to the appropriate color.
- **Lines 10 - 32** — define an associative array (named `typemap`) that maps a file extension like `jpg` or `mp3` to the appropriate generic file type.

Lines 35 - 53 — define a node action that is invoked every time `gvpr` encounters a node in our graph. This is where we determine the file extension and assign the `fillcolor`.

```
BEGIN {
  int colormap[string];   // maps generic type to color
  colormap[""]         = 1;
  colormap["audio"]    = 2;
  colormap["document"] = 3;
  colormap["ebook"]    = 4;
  colormap["image"]    = 5;
  colormap["video"]    = 6;

  string typemap[string]; // maps extension to generic type
  // audio files
  typemap["aac"] = "audio";
  typemap["mp3"] = "audio";
  typemap["wav"] = "audio";
  // document files
  typemap["doc"] = "document";
  typemap["xls"] = "document";
  typemap["ppt"] = "document";
  // ebook files
  typemap["pdf"] = "ebook";
  typemap["chm"] = "ebook";
  typemap["azw"] = "ebook";
  // image files
  typemap["bmp"] = "image";
  typemap["gif"] = "image";
  typemap["jpg"] = "image";
  typemap["png"] = "image";
  typemap["svg"] = "image";
  // video files
  typemap["avi"] = "video";
  typemap["mp4"] = "video";
  typemap["wmv"] = "video";
}

N {
  // set default attributes
  style = "filled";
  colorscheme = "pastel19";

  // determine the file extension
  string ext = substr($.name,1+rindex($.name,"."));

  // map the file extension to a generic type
  string type = "";
  if(ext in typemap) { type = typemap[ext]; }

  // map the generic type to a color
  int color = 0;
  if(type in colormap) { color = colormap[type]; }

  // set the fillcolor attribute
  fillcolor = color;
}
```

Figure 26.16: *A* gvpr *script that color-codes the nodes of a treemap based upon the extension of the file each node represents.*

Lines 36 - 38 — assign the `style` and `colorscheme` attributes to each node.

Lines 40 & 41 — extract the file extension from the node's name (by taking the sub-string from the last occurrence of a `.` character to the end of the node's name).

Lines 43 - 45 — select the generic file type that is associated with the file extension (defaulting to an empty string (`""`) if the file extension isn't mapped to anything).

Lines 47 - 49 — select the color identifier that is associated with the generic file type (defaulting to `0` if the type isn't mapped to anything else).

Lines 51 & 52 — set the node's `fillcolor` attribute to the identified color.

This script is invoked by `gvpr` using the command:

```
> treemapper "My Documents" | gvpr -c -f color-by-type.gvpr
```

to generate the graph descriptor, and can be piped directly to `patchwork`:

```
> treemapper "My Documents" | gvpr -c -f color-by-type.gvpr | patchwork -…
  Tpng -o my-docs.png
```

to generate the treemap image immediately. See Figure 26.17 for an example of each.

```
> treemapper “My Documents” | \
  gvpr -c -f color-by-type.gvpr
graph {
  subgraph “cluster My Documents” {
    subgraph “cluster Media” {
      subgraph “cluster Video” {
        “a-video.avi” [area=47,
          colorscheme=pastel19,
          fillcolor=6,
          style=filled];
      }
      subgraph “cluster Music” {
        “song-1.mp3” [area=28,
          colorscheme=pastel19,
          fillcolor=2,
          style=filled];
        “song-2.mp3” [area=19,
          colorscheme=pastel19,
          fillcolor=2,
          style=filled];
      }
      subgraph “cluster Pictures” {
        “photo-2.jpg” [area=16,
          colorscheme=pastel19,
          fillcolor=5,
          style=filled];
        “photo-1.jpg” [area=15,
          colorscheme=pastel19,
          fillcolor=5,
          style=filled];
        “photo-3.jpg” [area=13,
          colorscheme=pastel19,
          fillcolor=5,
          style=filled];
      }
    }
    subgraph “cluster Documents” {
      “memo.doc” [area=7,
        colorscheme=pastel19,
        fillcolor=3,
        style=filled];
      “accounts.xls” [area=13,
        colorscheme=pastel19,
        fillcolor=3,
        style=filled];
    }
    “cal.pdf” [area=5,
      colorscheme=pastel19,
      fillcolor=4,
      style=filled];
  }
}
```

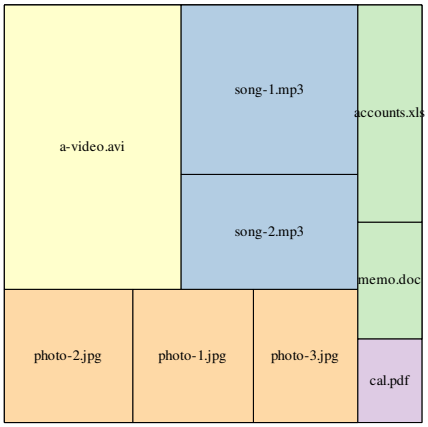

Figure 26.17: *A graph and corresponding* `patchwork` *rendering of a treemap representing the contents of an arbitrary directory, as reported by our* `treemapper` *script defined in Figure 26.16.*

Indicating sub-directories on the graph

Note that `patchwork` doesn't provide any explicit indicator of the clusters on the treemap. It's not obvious, for example, in Figure 26.14 that `accounts.xls` and `memo.doc` are located in a different sub-directory than `cal.pdf`.

We can remedy this in a couple of different ways.

Indicating sub-directories with borders

One remedy is to add a border around each cluster.

The graph attribute `penwidth` determines the width of the border that is drawn around the clusters within the graph. We could modify our AWK script to include this attribute, but we can also specify the attribute with an argument passed to `patchwork` on the command line:

```
treemapper "My Documents" | \
  patchwork -Gpenwidth=4 -Tpng -o my-documents.png
```

This yields a treemap in which the clusters are more easily identified:

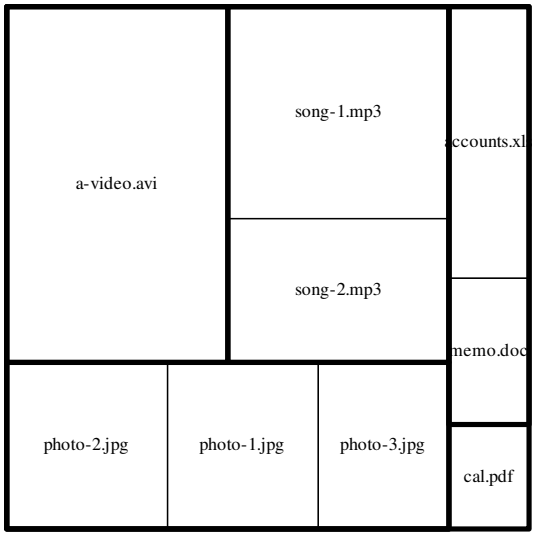

Figure 26.18: *A treemap rendered by* `patchwork` *with cluster borders emphasized by virtue of the* `penwidth` *graph/cluster attribute.*

Indicating sub-directories with colors

Another way to make sub-directories within the treemap stand out is to assign a unique background color to each cluster.

The graph attribute `bgcolor` determines the fill-color of each cluster. We *could* modify our AWK script to assign a unique color to each cluster, but Graphviz's `gvpr` application offers an easier way.

Recall that `gvpr` will process a DOT file invoking user-defined *actions* when a node, edge or graph is encountered. We can use `gvpr` to step through each cluster in the graph, assigning a `bgcolor` attribute as needed.

Rather than hard-coding a specific list of colors to assign, we'll apply the `colorscheme` technique (Chapter 19) and assign colors as indexes within a given palette rather than by name.

Since `gvpr` doesn't provide a direct *action* for matching clusters and subgraphs, we'll need to walk through the clusters manually.

The `gvpr` script in Figure 26.19 is one implementation of this strategy.

```
1  BEGIN {
2
3    string palette = "accent8";
4    int max_color = 8;
5    int next_color = 0;
6
7    void assign_colors(graph_t sg) {
8      sg.colorscheme = palette;
9      sg.bgcolor = 1+(next_color%max_color);
10     next_color++;
11   }
12
13   void visit_subgraph(graph_t sg) {
14     if(sg) {
15       assign_colors(sg);
16       visit_subgraph(fstsubg(sg));
17       visit_subgraph(nxtsubg(sg));
18     }
19   }
20
21 }
22
23 N { }
24
25 END_G { visit_subgraph(fstsubg($G)); }
```

Figure 26.19: *A* `gvpr` *script that color-codes the tiels of a treemap based upon the sub-directory in which the corresponding file is contained. See Figure 26.20 for an example of the output from this script.*

Let's examine that script more closely.

Lines 1 - 21 — define an action that is invoked exactly once, at the beginning of `gvpr`'s processing. We use this block to define some functions and initialize some variables we'll use later on in the processing.

Lines 4 - 5 — define some variables we use to keep track of the colors that have been (and will be assigned). `palette` is the name of our `colorscheme`. **FIX ME:** *Currently,* `accent8`*, one of the Brewer color schemes.* `max_color` is variable representing the number of colors in that palette. `max_color` tells us when the color index needs to "roll-over" and start again from the top. `next_color` is a counter that tells us the index of the next color to assign.

Lines 7 - 11 — declare a function (named `assign_colors`) that, given a subgraph, will set the `colorscheme` and `bgcolor` attributes appropriately. In line 8 we use the modulus operator (`%`) to calculate the *remainder* of `next_color` divided by `max_color`. This prevents us from accidentally referencing a color index larger than the maximum. (We add `1` to this value since the index of the first color in a Brewer palette is `1`, not `0`.

Lines 13 - 19 — declare a function (named `visit_subgraph`) that will recursively visit the subgraphs in the tree, processing each as necessary.

Line 14 — tests that the given subgraph (`sg`) actually exists before proceeding. If it doesn't this method jumps to line 18 and exits.

Line 15 — invokes the `assign_colors` function to assign a color to the current subgraph (`sg`).

Line 16 — uses the built-in `fstsubg` (*first subg*raph) function to select the first "child" of the current subgraph (if any), and passes it to `visit_subgraph` to begin walking further down the directory tree.

Line 17 — uses the built-in `nxtsubg` (*next subg*raph) function to select the first "sibling" of the current subgraph (if any), and passes it to `visit_subgraph` to continue walking the directory tree.

Line 23 — declares an empty node action. This simply ensures that the nodes in the input graph are included in the output graph.

Line 25 — declares an *action* that is fired once, at the end of graph processing. Here we invoke the `visit_subgraph` method on the first subgraph of the root graph (`$G`) in order to initiate the processing.

This script is invoked by `gvpr` using the command:

```
> treemapper "My Documents" | gvpr -c -f color-by-subdir.gvpr
```

to generate the graph descriptor, and can be piped directly to `patchwork`:

```
> treemapper "My Documents" | \
  gvpr -c -f color-by-subdir.gvpr | \
  patchwork -Tpng -o my-docs.png
```

to generate the treemap image immediately. Note that, as before, we can add the `-Gpenwdith` option to also draw a border around each cluster. See Figure 26.20 for an example.

```
> treemapper "My Documents" | \
  gvpr -c -f color-by-subdir.gvpr
graph {
  subgraph "cluster My Documents" {
    graph [bgcolor=1,
      colorscheme=accent8
    ];
    subgraph "cluster Media" {
      graph [bgcolor=2,
        colorscheme=accent8
      ];
      subgraph "cluster Video" {
        graph [bgcolor=3,
          colorscheme=accent8
        ];
        "a-video.avi" [area=47];
      }
      subgraph "cluster Music" {
        graph [bgcolor=4,
          colorscheme=accent8
        ];
        "song-1.mp3" [area=28];
        "song-2.mp3" [area=19];
      }
      subgraph "cluster Pictures" {
        graph [bgcolor=5,
          colorscheme=accent8
        ];
        "photo-2.jpg" [area=16];
        "photo-1.jpg" [area=15];
        "photo-3.jpg" [area=13];
      }
    }
    subgraph "cluster Documents" {
      graph [bgcolor=6,
        colorscheme=accent8
      ];
      "memo.doc" [area=7];
      "accounts.xls" [area=13];
    }
    "cal.pdf" [area=5];
  }
}
```

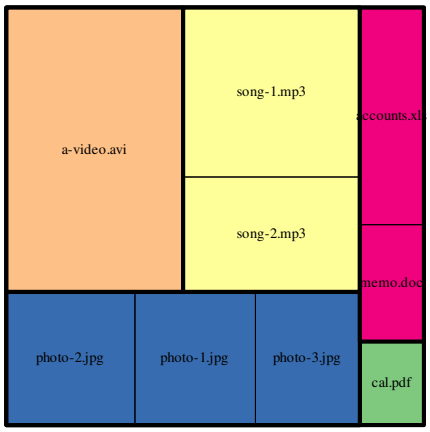

Figure 26.20: *A DOT language graph descriptor and corresponding* `patchwork` *rendering of a treemap generated by the* `gvpr` *command listed in Figure 26.19.*

Adding sub-directory labels

Building on the previous extension, suppose we'd like to label each cluster with the name of the corresponding directory.

To label sub-directories on the treemap it would be sufficient, in theory, to add a `label` attribute to each cluster. `patchwork` however ignores cluster `label`s, so simply setting the `label` attribute has no effect on the resulting treemap.

Hence adding sub-directory labels to the treemap is a two part problem:

1. Finding a way to draw cluster `label`s on the treemap image, and
2. Programatically assigning `label`s to the clusters in the first place.

For the first challenge, we can make use of the multiple-layout technique (described in Chapter 15) to work around the fact that `patchwork` ignores cluster

labels.

Specifically, we'll use patchwork to perform the core layout, and then run the resulting graph through neato to draw the cluster labels on the final image. With the appropriate command line parameters, we can tell neato not to make any changes to the positioning of nodes or clusters, so patchwork's layout can pass through unscathed.

```
1 > treemapper "My Documents" | \
2   patchwork | \
3   neato -Tpng -o my-docs.png \
4     -n -Nfixedsize=true \
5     -Nfontsize=11 -Gfontsize=12 \
6     -Glabeljust="r" -Glabelloc="b" \
7     -Gpenwidth=4
```

Figure 26.21: *Using* neato *to add cluster labels to a graph laid-out by* patchwork*. The* -n *parameter prevents* neato *from changing the* pos *attribute assigned to each input node.*

In detail:

Line 1 — invokes our treemapper script to generate the base DOT file.

Line 2 — invokes patchwork, requesting the default DOT format output rather than an image of some sort.

Line 3 — invokes neato, with our typical arguments to create a PNG image file.

Line 4 — adds a couple of options to the neato, command line to keep neato from changing patchwork's layout. The -n argument tells neato not to change the existing node and cluster positions. The -Nfixedsize=true argument adds a default fixedsize=true attribute to each node. This ensures that the node stays the exact size specified by patchwork, even if that means the node's label stretches beyond the boundaries of the node itself.

Lines 5 & 6 — set some attributes that tweak the style and placement of the cluster and node labels.

Line 7 — adds a border around each cluster (as above).

For the second challenge, assigning labels to the clusters in the first place, we'll once again turn to gvpr.

Since gvpr doesn't have a built-in event that is fired for each subgraph or cluster, we must define functions that "manually" traverse the clusters within the graph. The gvpr script in Figure 26.22 is one implementation of this strategy.

In detail:

Lines 1 - 38 — define an action that is invoked exactly once, at the beginnning of gvpr's processing. We use this action to define some variables and our cluster-walking algorithm.

```
BEGIN {
  int next_color = 1;
  int max_color = 6;

  void set_colors(graph_t sg) {
    sg.colorscheme = "paired12";
    sg.bgcolor = 1 + 2*(next_color % max_color);
    sg.fontcolor = sg.bgcolor + 1;
    next_color++;
  }

  // Determines the path to the directory corresponding
  // to the given subgraph by parsing the directory name
  // out of the containing clusters.
  string get_path(graph_t sg) {
    if(sg.parent) {
      return sprintf("%s%s/",
                     get_path(sg.parent),
                     substr(sg.name,8));
    } else {
      return "";
    }
  }

  // Process an individual subgraph, and then
  // recurse to visit any children or siblings.
  void visit_subgraph(graph_t sg) {
    if(sg) {
      set_colors(sg);
      if(fstsubg(sg)) {
        visit_subgraph(fstsubg(sg));
      } else {
        sg.label = get_path(sg);
      }
      visit_subgraph(nxtsubg(sg));
    }
  }
}


N { } // A no-op action to copy all nodes into output graph.

// Visit the first subgraph (and recurse).
END_G { visit_subgraph(fstsubg($G)); }
```

Figure 26.22: *A* `gvpr` *script that walks subgraph tree assigning* `label`*s and colors to clusters.*

Lines 2 & 3 — define variables that track the index of the next and last color.

Lines 5 - 10 — define a function that, given a subgraph `sg`, assign the appropriate `color` and `colorscheme` attributes.

Lines 12 - 23 — define a function that, given a subgraph `sg`, will return a `string` representing the path to the directory that subgraph represents (by parsing the directory names from the cluster identifers).

Lines 25 - 38 — define a function that, given a subgraph `sg`, will assign color (via `set_colors` and label (via `get_path`) attributes, and recursively visit all subgraphs that are below or after `sg`.

Line 41 — defines a "no-op" node action that ensures all the nodes in the input graph are copied into the output graph.

Lines 44 & 44 — define an action that is invoked at the end of the root graph processing. Here, we invoke the `visit_subgraph` method to start the process of walking through the graph's clusters.

When rendered with `patchwork`, the graph generated by the script in Figure 26.22 generates an graph descriptor and image like that found in Figure 26.23.

```
> treemapper “My Documents” | \
  gvpr -c -f label-subdir.gvpr
graph {
 subgraph “cluster My Documents” {
  graph [bgcolor=3,
   colorscheme=paired12,
   fontcolor=4
  ];
  subgraph “cluster Media” {
   graph [bgcolor=5,
    colorscheme=paired12,
    fontcolor=6
   ];
   subgraph “cluster Video” {
    graph [bgcolor=7,
     colorscheme=paired12,
     fontcolor=8,
     label=“My Documents/Media/Video/”
    ];
    “a-video.avi” [area=47];
   }
   subgraph “cluster Music” {
    graph [bgcolor=9,
     colorscheme=paired12,
     fontcolor=10,
     label=“My Documents/Media/Music/”
    ];
    “song-1.mp3” [area=28];
    “song-2.mp3” [area=19];
   }
   subgraph “cluster Pictures” {
    graph [bgcolor=11,
     colorscheme=paired12,
     fontcolor=12,
     label=“My Documents/Media/Pictures/”
    ];
    “photo-2.jpg” [area=16];
    “photo-1.jpg” [area=15];
    “photo-3.jpg” [area=13];
   }
  }
  subgraph “cluster Documents” {
   graph [bgcolor=1,
    colorscheme=paired12,
    fontcolor=2,
    label=“My Documents/Documents/”
   ];
   “memo.doc” [area=7];
   “accounts.xls” [area=13];
  }
  “cal.pdf” [area=5];
 }
}
```

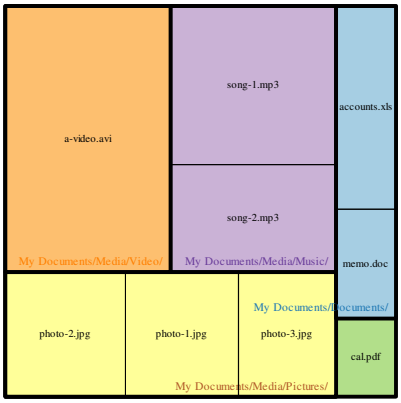

Figure 26.23: *A* `patchwork`*-friendly, DOT format graph descriptor generated by* `treemapper` *and the* `gvpr` *script listed in Figure 26.22, and a* `patchwork`*-generated treemap rendering.*