Code Chunks – Ggraph

## Layouts

In very short terms, a layout is the vertical and horizontal placement of nodes when plotting a particular graph structure. Conversely, a layout algorithm is an algorithm that takes in a graph structure (and potentially some additional parameters) and return the vertical and horizontal position of the nodes. Often, when people think of network visualizations, they think of node-edge diagrams where strongly connected nodes are attempted to be plotted in close proximity. Layouts can be a lot of other things too though — e.g. hive plots and treemaps. One of the driving factors behind **ggraph** has been to develop an API where any type of visual representation of graph structures is supported. In order to achieve this we first need a flexible way of defining the layout…

### ggraph() and create\_layout()

As the layout is a global specification of the spatial position of the nodes it spans all layers in the plot and should thus be defined outside of calls to geoms or stats. In **ggraph** it is often done as part of the plot initialization using **ggraph()** — a function equivalent in intent to **ggplot()**. As a minimum **ggraph()** must be passed a graph object supported by **ggraph**:

library(ggraph)

library(igraph)

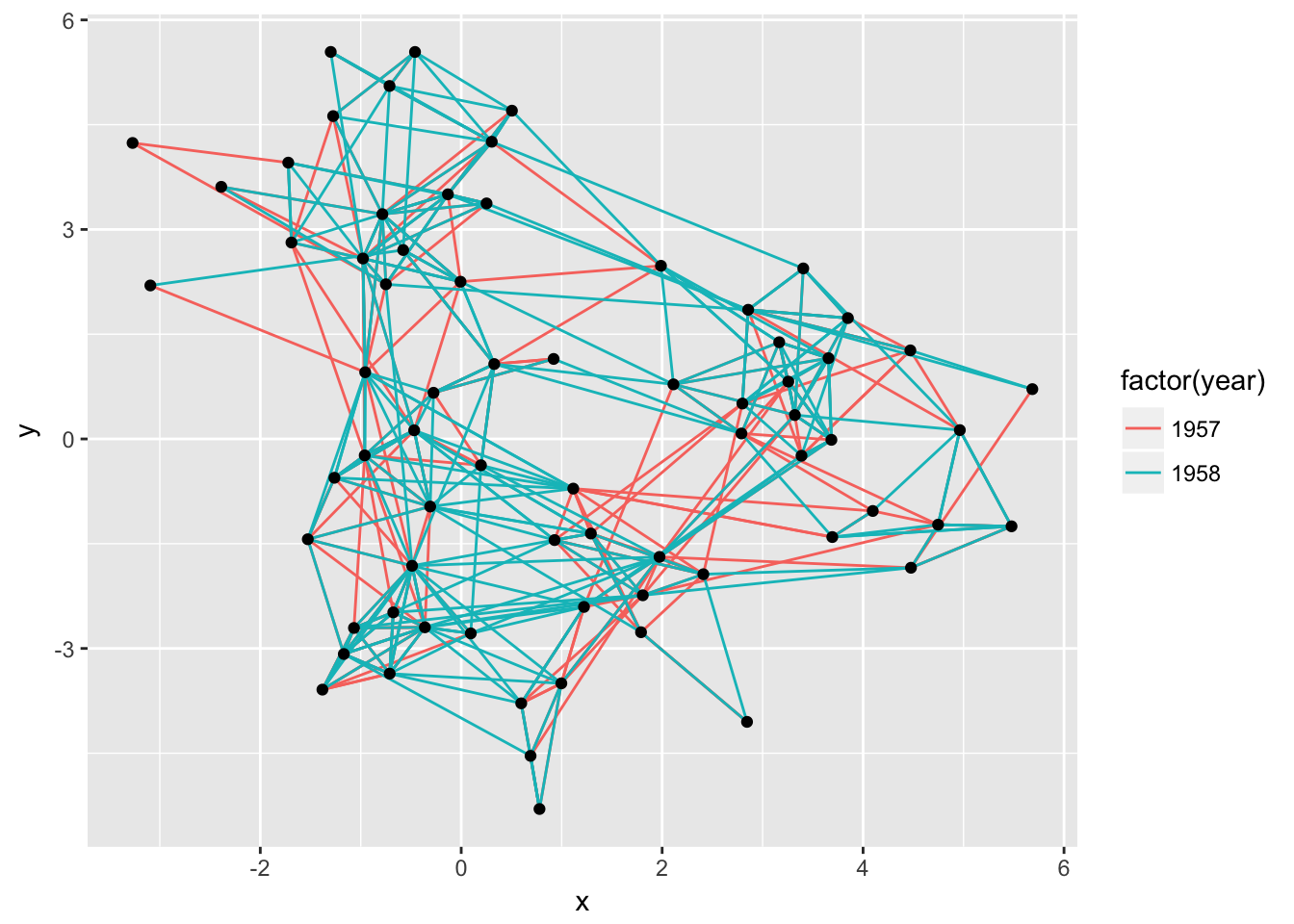
graph <- graph\_from\_data\_frame(highschool)

*# Not specifying the layout - defaults to "auto"*

ggraph(graph) +

geom\_edge\_link(aes(colour = factor(year))) +

geom\_node\_point()

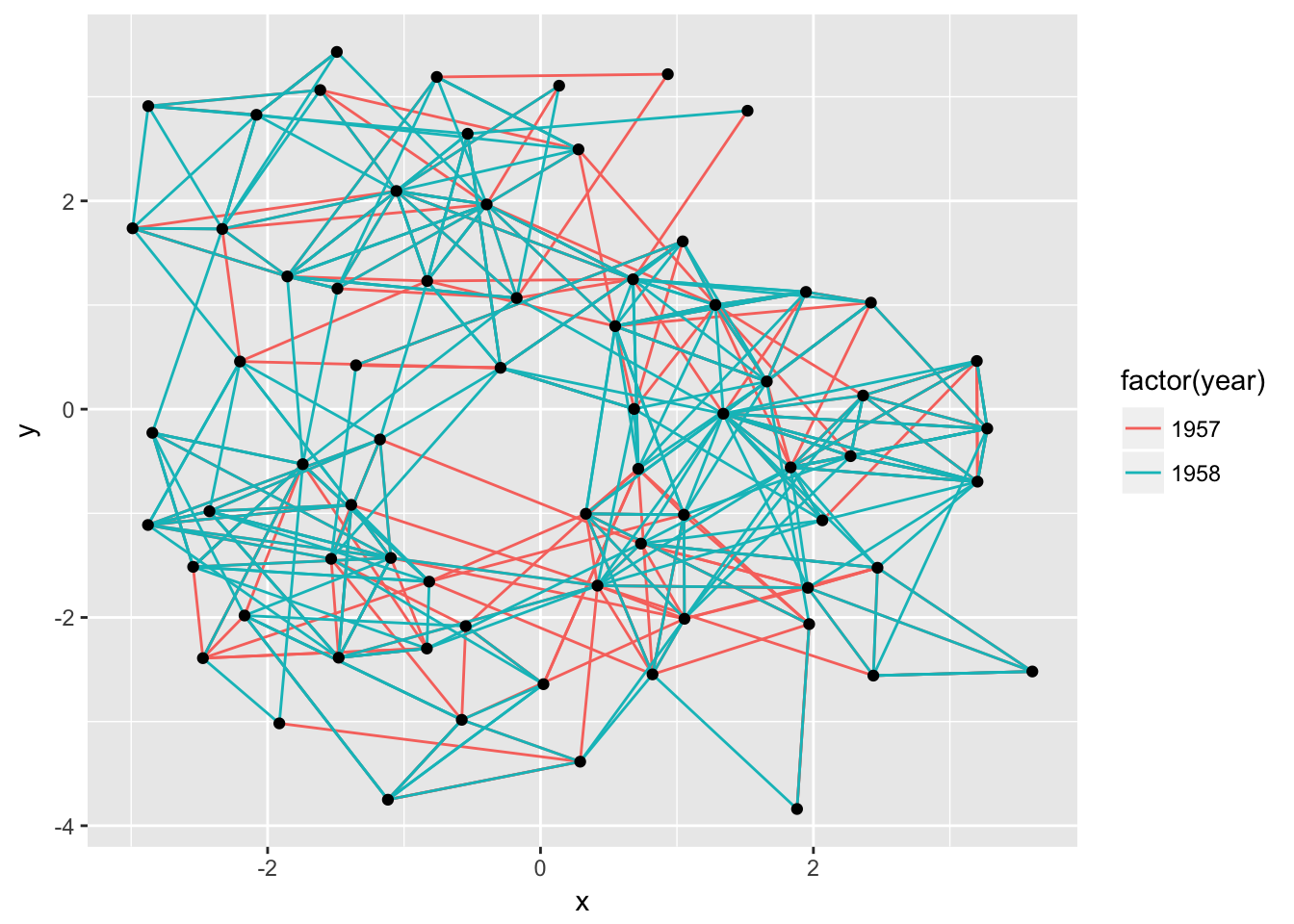


Not specifying a layout will make **ggraph** pick one for you. This is only intended to get quickly up and running. The choice of layout should be deliberate on the part of the user as it will have a great effect on what the end result will communicate. From now on all calls to **ggraph()** will contain a specification of the layout:

ggraph(graph, layout = 'kk') +

geom\_edge\_link(aes(colour = factor(year))) +

geom\_node\_point()

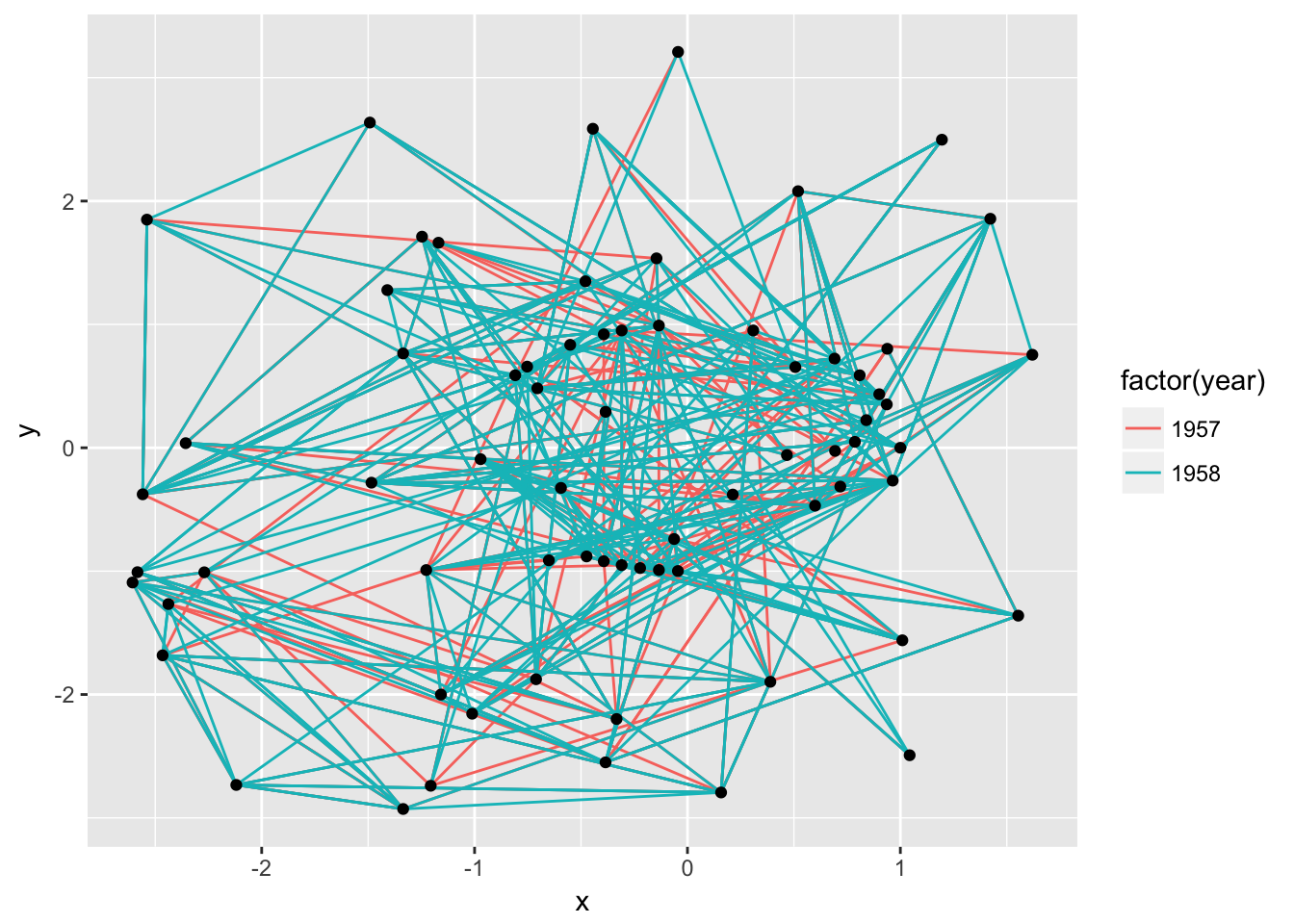


If the layout algorithm accepts additional parameters (most do), they can be supplied in the call to **ggraph()** as well:

ggraph(graph, layout = 'kk', maxiter = 100) +

geom\_edge\_link(aes(colour = factor(year))) +

geom\_node\_point()



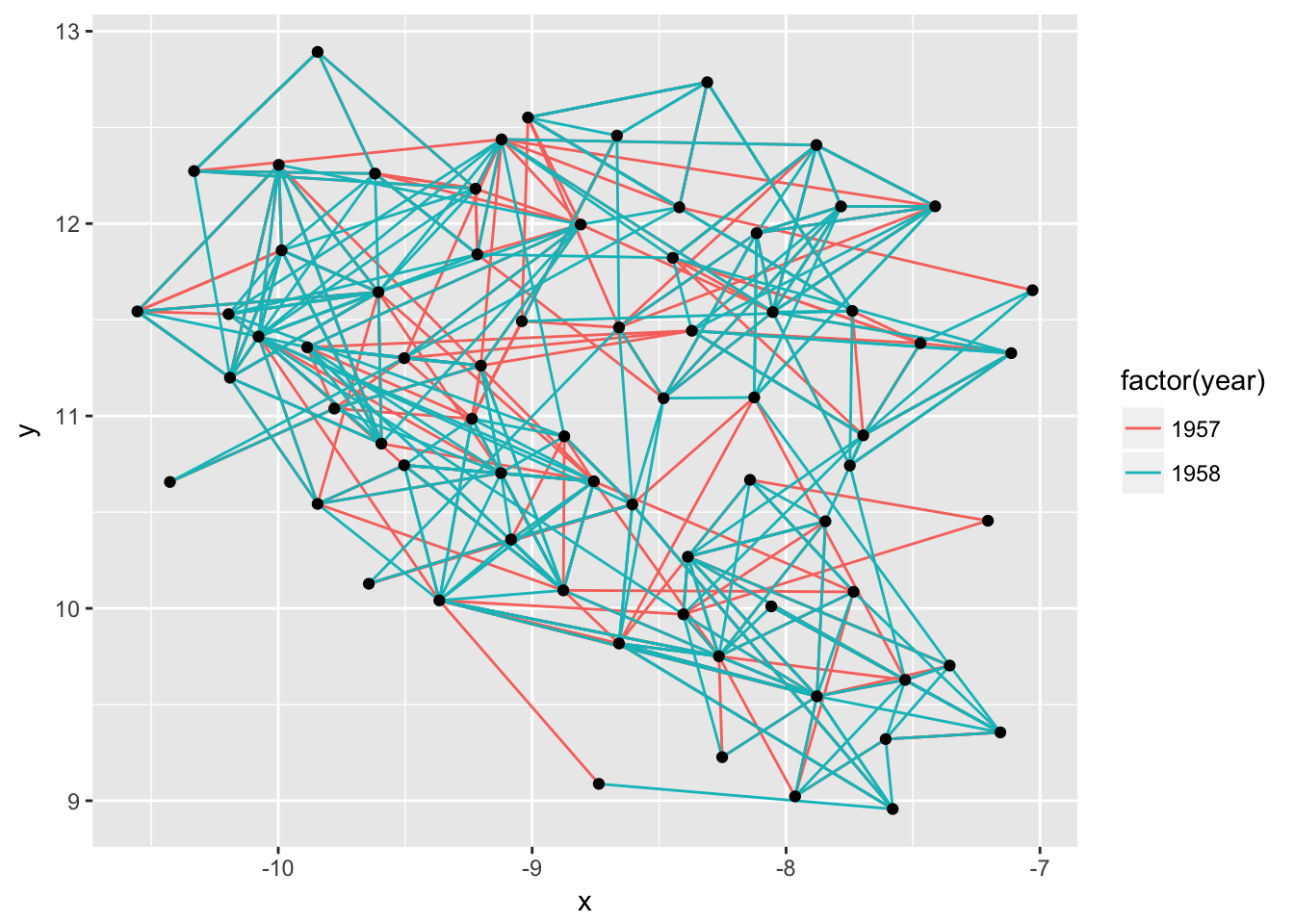
In addition to specifying the layout during plot creation it can also happen separately using **create\_layout()**. This function takes the same arguments as **ggraph()** but returns a **layout\_ggraph** object that can later be used in place of a graph structure in ggraph call:

layout <- create\_layout(graph, layout = 'drl')

ggraph(layout) +

geom\_edge\_link(aes(colour = factor(year))) +

geom\_node\_point()



Examining the return of **create\_layout()** we see that it is really just a **data.frame** of node positions and (possible) attributes. Furthermore the original graph object along with other relevant information is passed along as attributes:

head(layout)

*#> x y name ggraph.orig\_index circular ggraph.index*

*#> 1 -7.734004 10.085789 1 1 FALSE 1*

*#> 2 -8.251559 9.226503 2 2 FALSE 2*

*#> 3 -7.205127 10.455535 3 3 FALSE 3*

*#> 4 -7.113050 11.326465 4 4 FALSE 4*

*#> 5 -7.748919 10.742258 5 5 FALSE 5*

*#> 6 -7.355531 9.702643 6 6 FALSE 6*

attributes(layout)

*#> $names*

*#> [1] "x" "y" "name"*

*#> [4] "ggraph.orig\_index" "circular" "ggraph.index"*

*#>*

*#> $row.names*

*#> [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23*

*#> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46*

*#> [47] 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69*

*#> [70] 70*

*#>*

*#> $class*

*#> [1] "layout\_igraph" "layout\_ggraph" "data.frame"*

*#>*

*#> $graph*

*#> IGRAPH c187beb DN-- 70 506 --*

*#> + attr: name (v/c), ggraph.orig\_index (v/n), year (e/n)*

*#> + edges from c187beb (vertex names):*

*#> [1] 1 ->14 1 ->15 1 ->21 1 ->54 1 ->55 2 ->21 2 ->22 3 ->9 3 ->15 4 ->5*

*#> [11] 4 ->18 4 ->19 4 ->43 5 ->19 5 ->43 6 ->13 6 ->20 6 ->22 7 ->17 8 ->14*

*#> [21] 8 ->17 9 ->12 9 ->20 9 ->21 9 ->22 9 ->51 11->19 11->50 11->52 11->53*

*#> [31] 12->20 12->21 12->22 13->17 13->20 13->21 13->22 14->21 14->22 15->20*

*#> [41] 16->18 16->41 16->43 17->7 17->8 18->11 18->16 18->19 19->4 19->11*

*#> [51] 19->16 19->18 19->27 20->6 20->12 20->21 20->22 20->38 21->22 21->51*

*#> [61] 21->54 21->55 22->20 22->21 22->38 22->51 23->40 23->43 23->50 23->52*

*#> [71] 23->53 23->60 23->62 23->65 23->68 24->51 26->32 26->35 26->36 26->40*

*#> + ... omitted several edges*

*#>*

*#> $circular*

*#> [1] FALSE*

As it is just a **data.frame** it means that any standard **ggplot2** call will work by addressing the nodes. Still, use of the **geom\_node\_\*()** family provided by **ggraph** is encouraged as it makes it explicit which part of the data structure is being worked with.

### Adding support for new data sources

Out of the box **ggraph** supports **dendrogram** and **igraph** objects natively as well as **hclust** and **network** through conversion to one of the above. If there is wish for support for additional classes this can be achieved by adding a set of specific methods to the class. The **ggraph** source code should be your guide in this but I will briefly describe the methods below:

#### create\_layout.myclass()

This method is responsible for taking a graph structure and returning a **layout\_ggraph** object. The object is just a **data.frame** with the correct class and attributes added. The class should be **c('layout\_myclass', 'layout\_ggraph', 'data.frame')** and it should at least have a **graph** attribute holding the original graph object as well as a **circular** attribute with a logical giving whether the layout has been transformed to a circular representation or not. If the graph structure contains any additional information about the nodes this should be added to the **data.frame** as columns so these are accessible during plotting.

#### getEdges.layout\_myclass()

This method takes the return value of **create\_layout.myclass()** and returns the edges of the graph structure. The return value should be in the form of an edge list with a **to** and **from** column giving the indexes of the terminal nodes of the edge. Furthermore, it must contain a **circular** column, again indicating whether the layout should be considered circular. If there are any additional data attached to the edges in the graph structure these should be added as columns to the **data.frame**.

#### getConnection.layout\_myclass()

This method is intended to return the shortest path between two nodes as a list of node indexes. This method can be ignored but will result in lack of support for **geom\_conn\_\*** layers.

#### layout\_myclass\_\*()

Any type of layout algorithm that needs to be available to this class should be defined as a separate **layout\_myclass\_layoutname()** function. This function will be called when **'layoutname'** is used in the **layout** argument in **ggraph()** or **create\_layout()**. At a minimum each new class should have a **layout\_myclass\_auto()** defined.

## Layouts abound

There’s a lot of different layouts in **ggraph** — first and foremost because **igraph** implements a lot of layouts for drawing node-edge diagrams and all of these are available in **ggraph**. Additionally, **ggraph** provides a lot of new layout types and algorithms for your drawing pleasure.

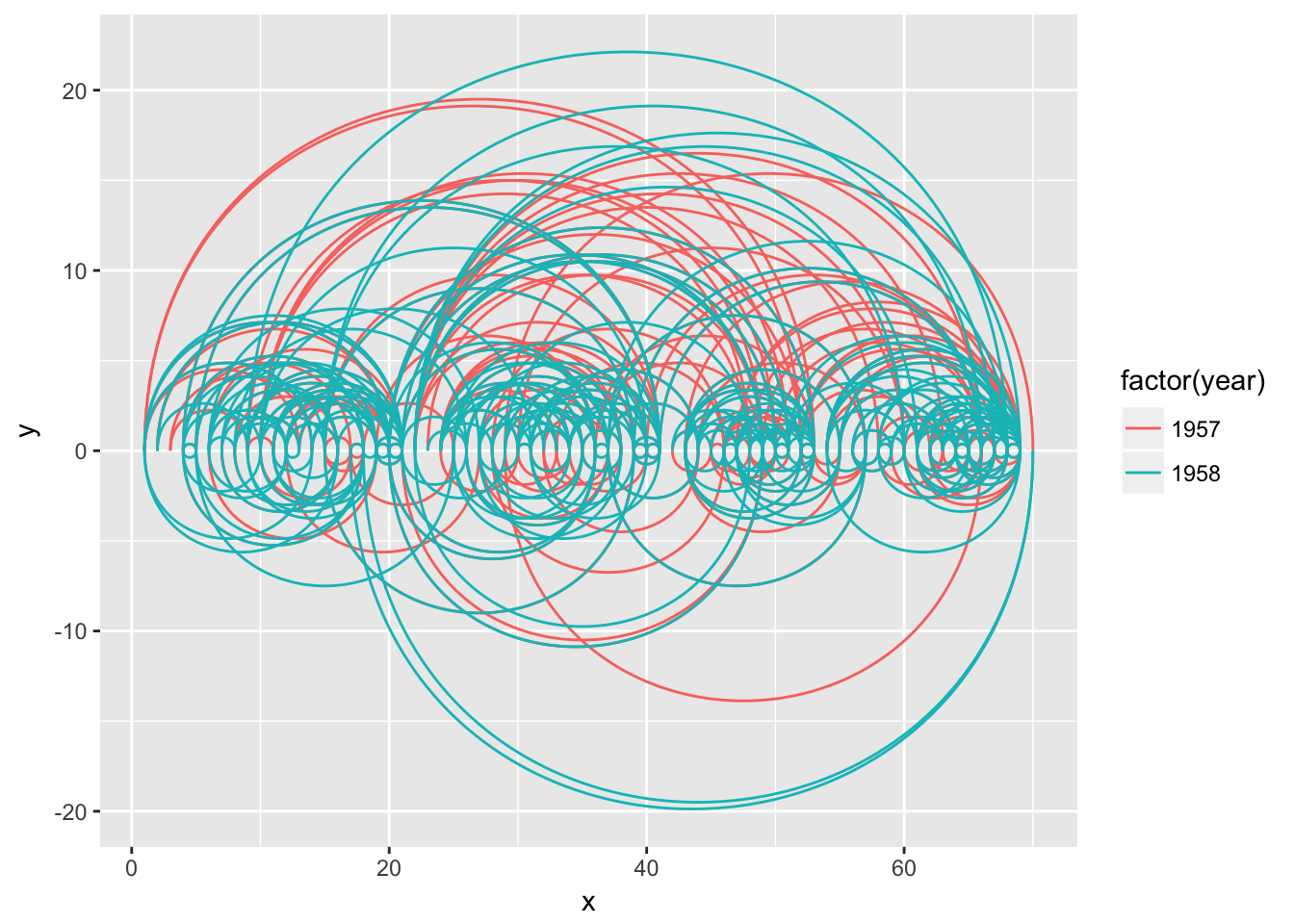
### A note on circularity

Some layouts can be shown effectively both in a standard Cartesian projection as well as in a polar projection. The standard approach in **ggplot2** has been to change the coordinate system with the addition of e.g. **coord\_polar()**. This approach — while consistent with the grammar — is not optimal for **ggraph** as it does not allow layers to decide how to respond to circularity. The prime example of this is trying to draw straight lines in a plot using **coord\_polar()**. Instead circularity is part of the layout specification and gets communicated to the layers with the **circular** column in the data, allowing each layer to respond appropriately. Sometimes standard and circular representations of the same layout get used so often that they get different names. In **ggraph** they’ll have the same name and only differ in whether or not **circular** is set to **TRUE**:

*# An arc diagram*

ggraph(graph, layout = 'linear') +

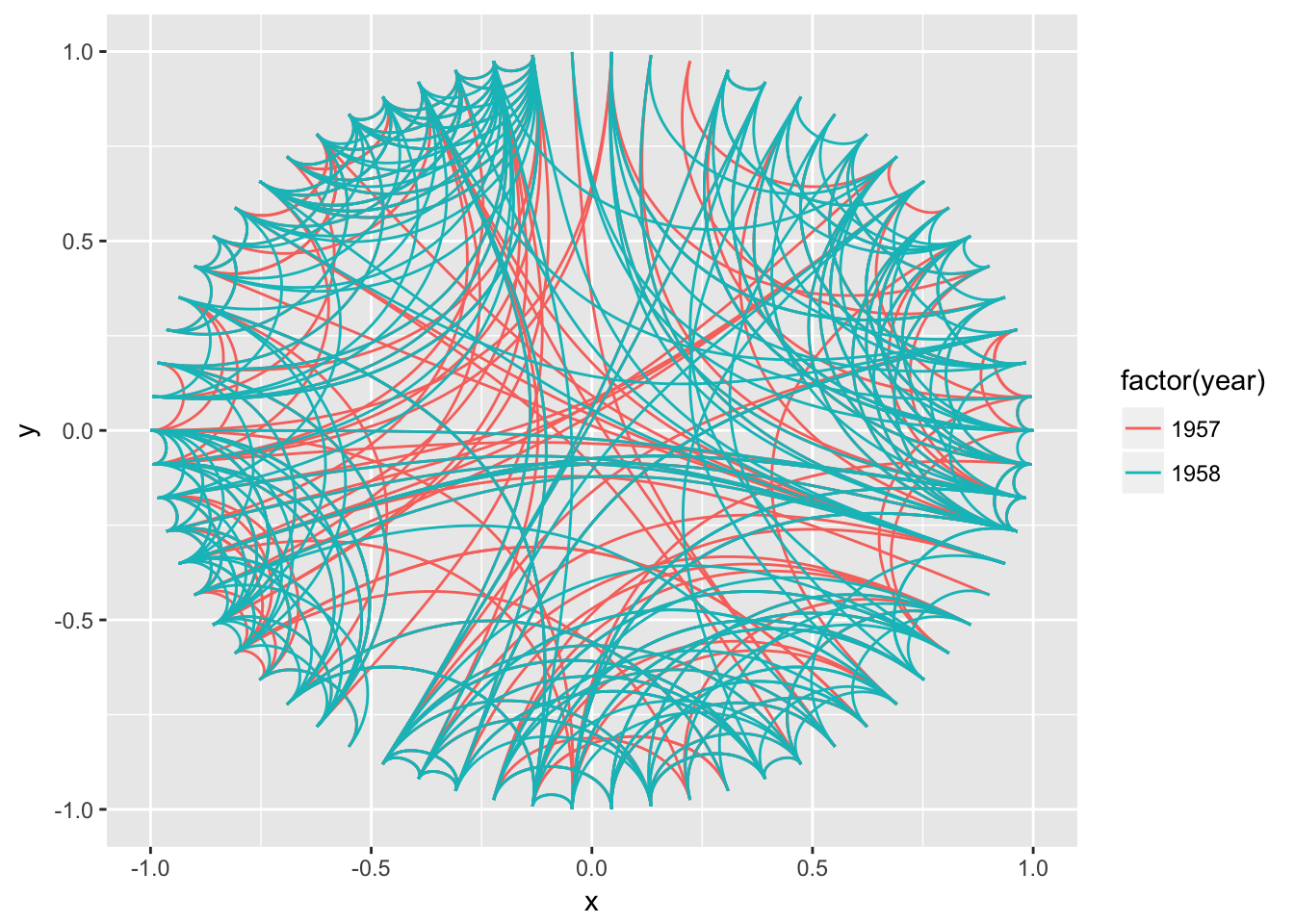
geom\_edge\_arc(aes(colour = factor(year)))



*# A coord diagram*

ggraph(graph, layout = 'linear', circular = TRUE) +

geom\_edge\_arc(aes(colour = factor(year)))

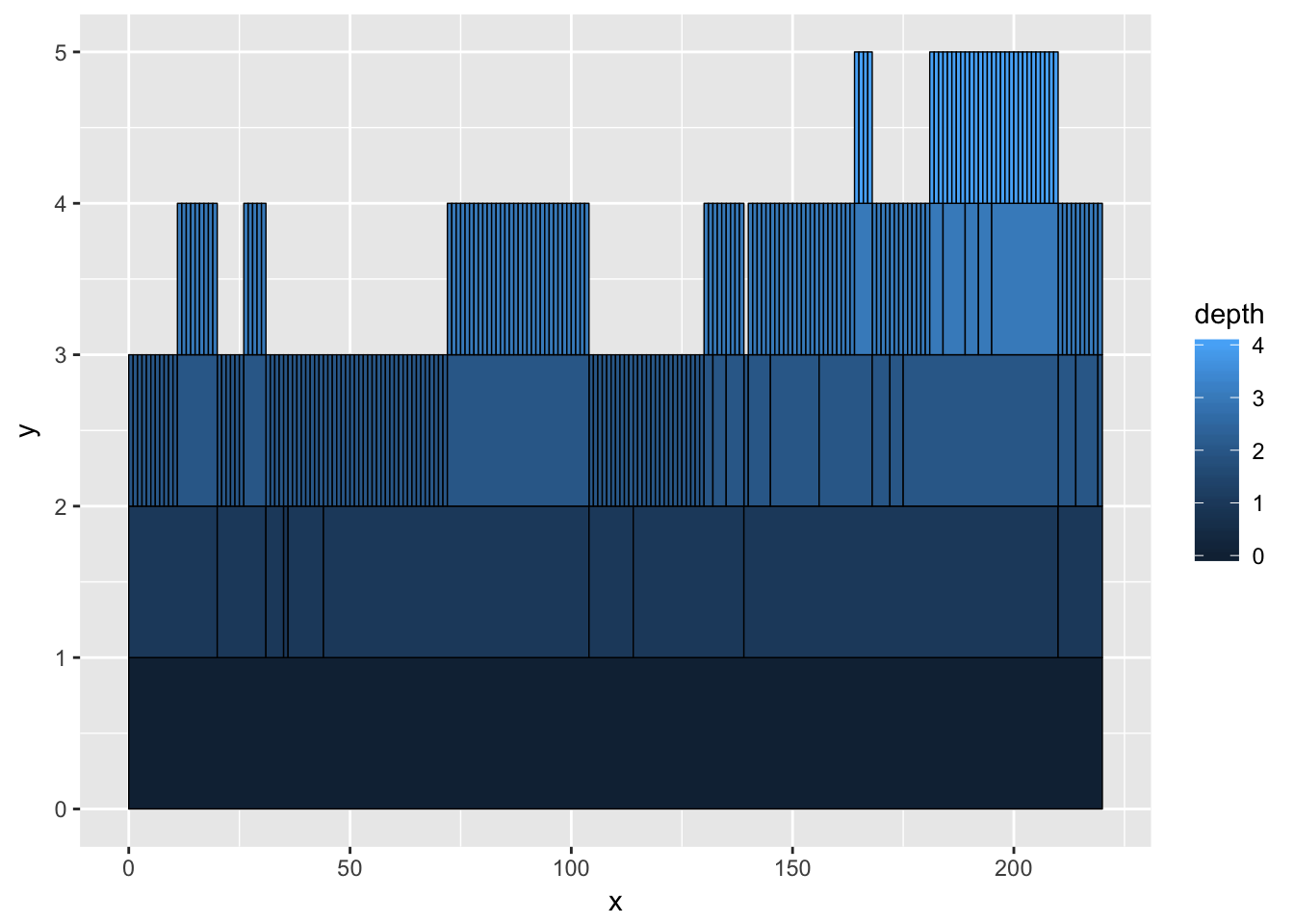


graph <- graph\_from\_data\_frame(flare$edges, vertices = flare$vertices)

*# An icicle plot*

ggraph(graph, 'partition') +

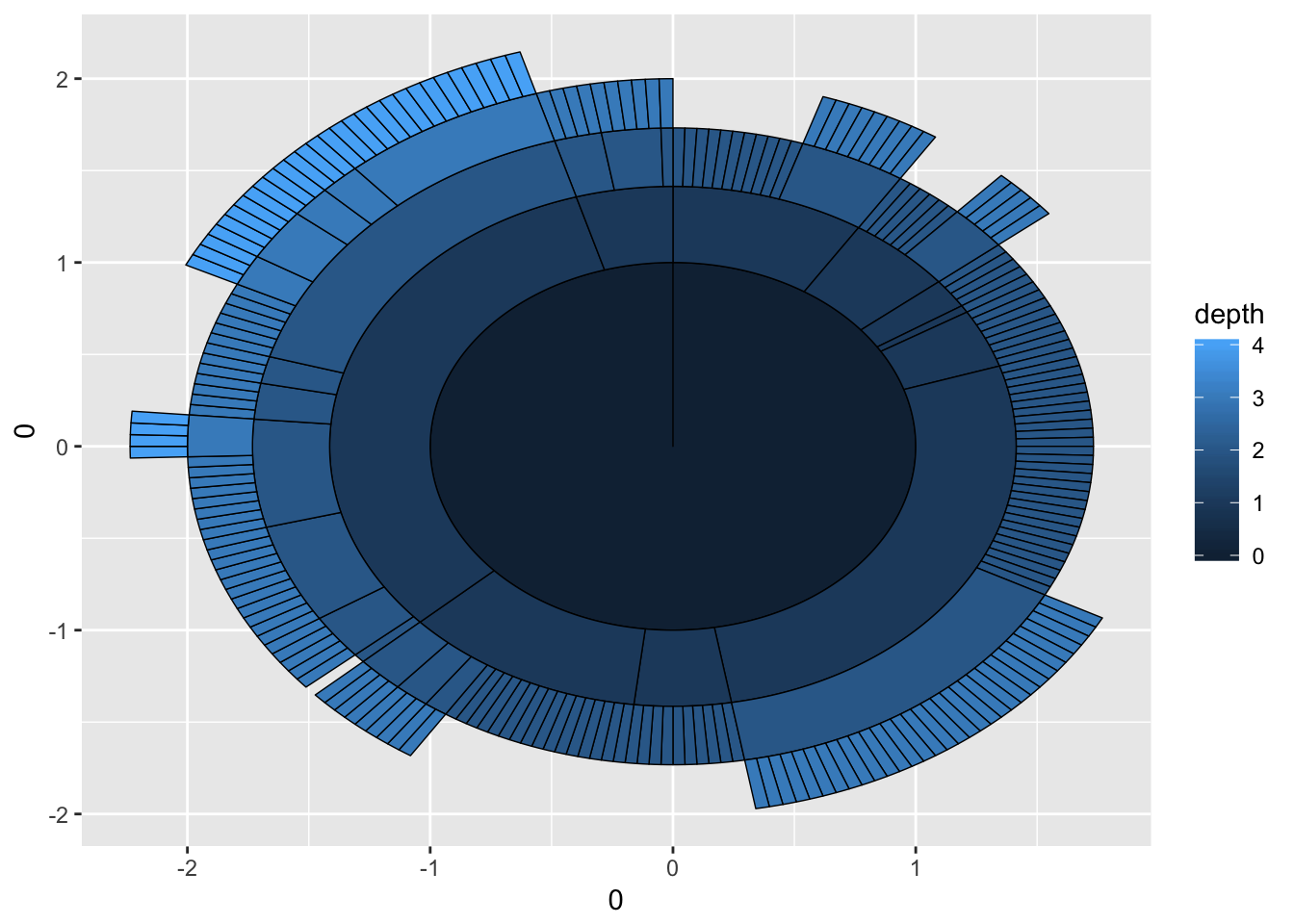
geom\_node\_tile(aes(fill = depth), size = 0.25)



*# A sunburst plot*

ggraph(graph, 'partition', circular = TRUE) +

geom\_node\_arc\_bar(aes(fill = depth), size = 0.25)



Not every layout has a meaningful circular representation in which cases the **circular** argument will be ignored.

### Node-edge diagram layouts

**igraph** provides a total of 13 different layout algorithms for classic node-edge diagrams (colloquially referred to as hairballs). Some of these are incredibly simple such as randomly, grid, circle, and star, while others tries to optimize the position of nodes based on different characteristics of the graph. There is no such thing as “the best layout algorithm” as algorithms have been optimized for different scenarios. Experiment with the choices at hand and remember to take the end result with a grain of salt, as it is just one of a range of possible “optimal node position” results. Below is an animation showing the different results of running all applicable **igraph** layouts on the highschool graph.

library(tweenr)

igraph\_layouts <- c('star', 'circle', 'gem', 'dh', 'graphopt', 'grid', 'mds',

'randomly', 'fr', 'kk', 'drl', 'lgl')

igraph\_layouts <- sample(igraph\_layouts)

graph <- graph\_from\_data\_frame(highschool)

V(graph)$degree <- degree(graph)

layouts <- lapply(igraph\_layouts, create\_layout, graph = graph)

layouts\_tween <- tween\_states(c(layouts, layouts[1]), tweenlength = 1,

statelength = 1, ease = 'cubic-in-out',

nframes = length(igraph\_layouts) \* 16 + 8)

title\_transp <- tween\_t(c(0, 1, 0, 0, 0), 16, 'cubic-in-out')[[1]]

for (i in seq\_len(length(igraph\_layouts) \* 16)) {

tmp\_layout <- layouts\_tween[layouts\_tween$.frame == i, ]

layout <- igraph\_layouts[ceiling(i / 16)]

title\_alpha <- title\_transp[i %% 16]

p <- ggraph(graph, 'manual', node.position = tmp\_layout) +

geom\_edge\_fan(aes(alpha = ..index.., colour = factor(year)), n = 15) +

geom\_node\_point(aes(size = degree)) +

scale\_edge\_color\_brewer(palette = 'Dark2') +

ggtitle(paste0('Layout: ', layout)) +

theme\_void() +

theme(legend.position = 'none',

plot.title = element\_text(colour = alpha('black', title\_alpha)))

plot(p)

}

#### Hive plots

A hive plot, while still technically a node-edge diagram, is a bit different from the rest as it uses information pertaining to the nodes, rather than the connection information in the graph. This means that hive plots, to a certain extend is more interpretable as well as less vulnerable to small changes in the graph structure. They are less common though, so use will often require some additional explanation.

V(graph)$friends <- degree(graph, mode = 'in')

V(graph)$friends <- ifelse(V(graph)$friends < 5, 'few',

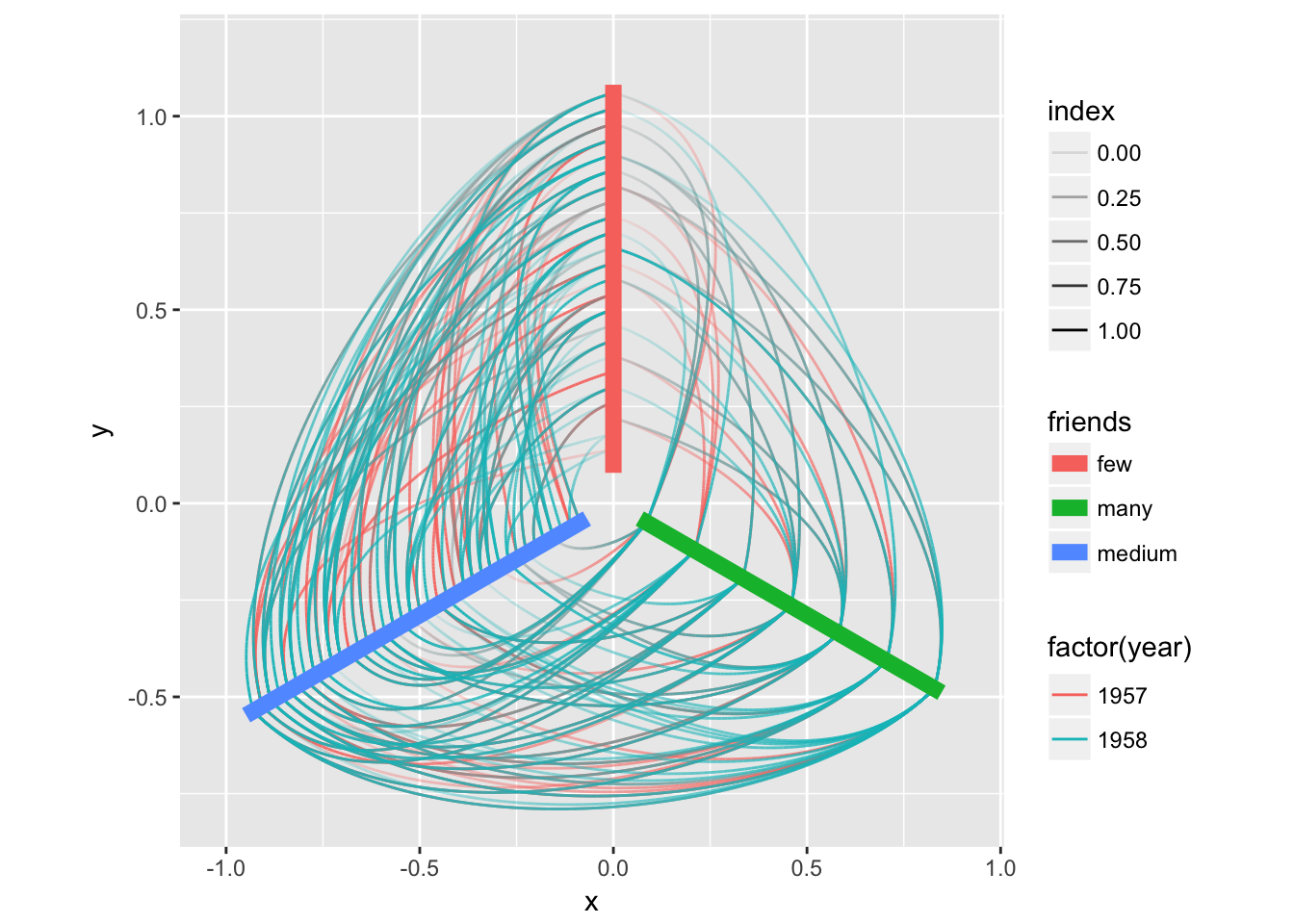
ifelse(V(graph)$friends >= 15, 'many', 'medium'))

ggraph(graph, 'hive', axis = 'friends', sort.by = 'degree') +

geom\_edge\_hive(aes(colour = factor(year), alpha = ..index..)) +

geom\_axis\_hive(aes(colour = friends), size = 3, label = FALSE) +

coord\_fixed()



### Hierarchical layouts

Trees and hierarchies are an important subset of graph structures, and **ggraph** provides a range of layouts optimized for their visual representation. Some of these uses enclosure and position rather than edges to communicate relations (e.g. treemaps and circle packing). Still, these layouts can just as well be used for drawing edges if you wish to:

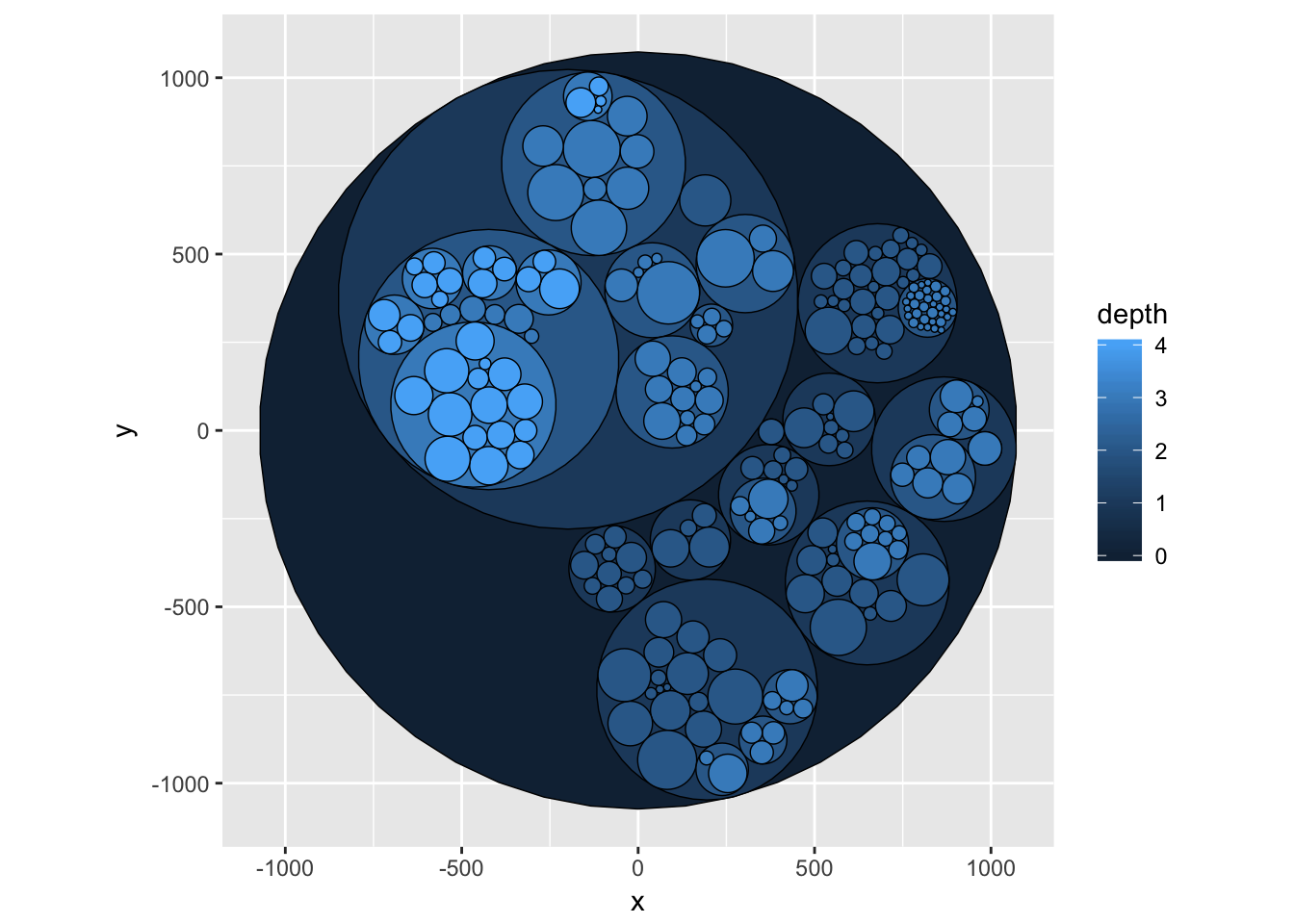
graph <- graph\_from\_data\_frame(flare$edges, vertices = flare$vertices)

set.seed(1)

ggraph(graph, 'circlepack', weight = 'size') +

geom\_node\_circle(aes(fill = depth), size = 0.25, n = 50) +

coord\_fixed()



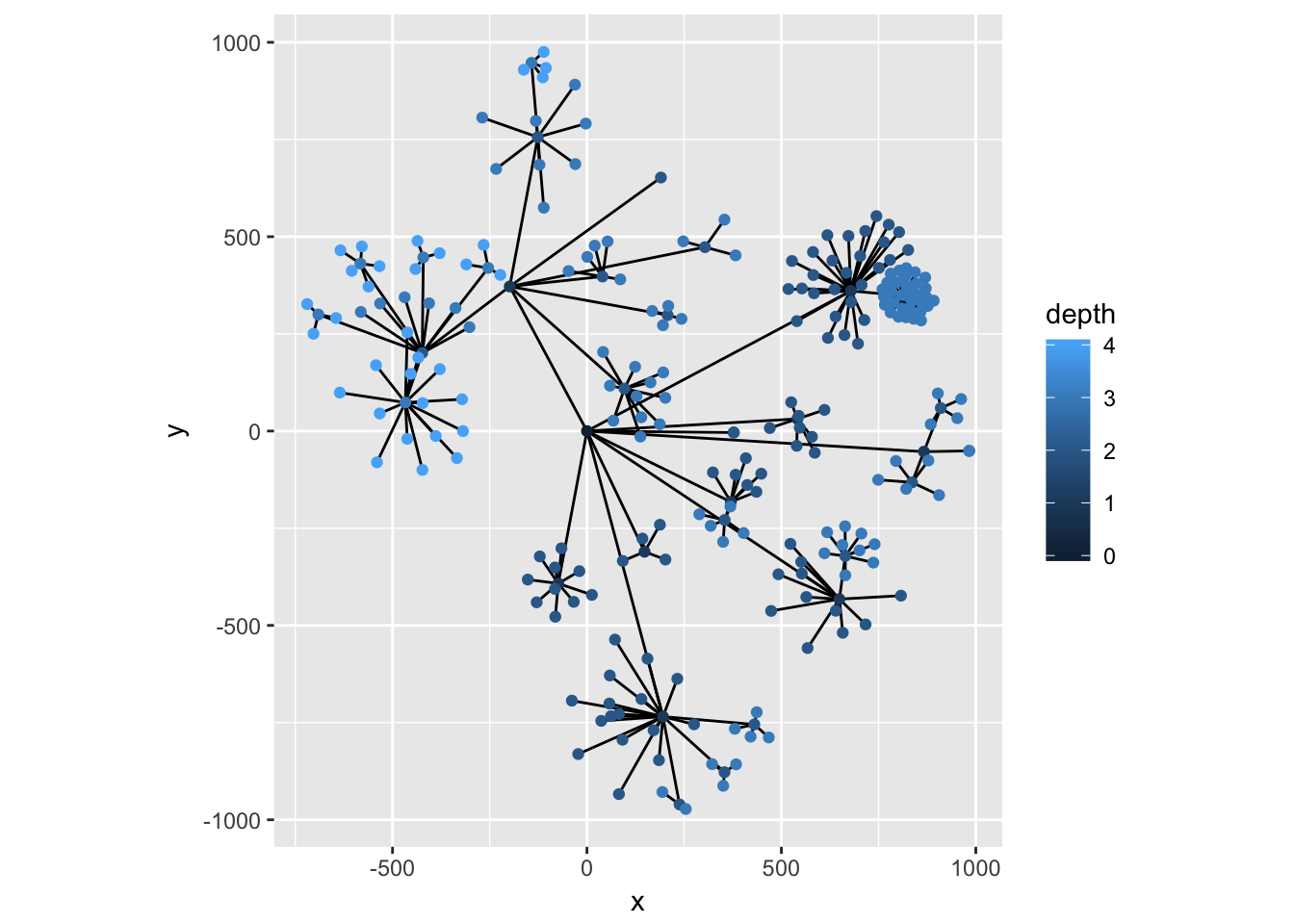
set.seed(1)

ggraph(graph, 'circlepack', weight = 'size') +

geom\_edge\_link() +

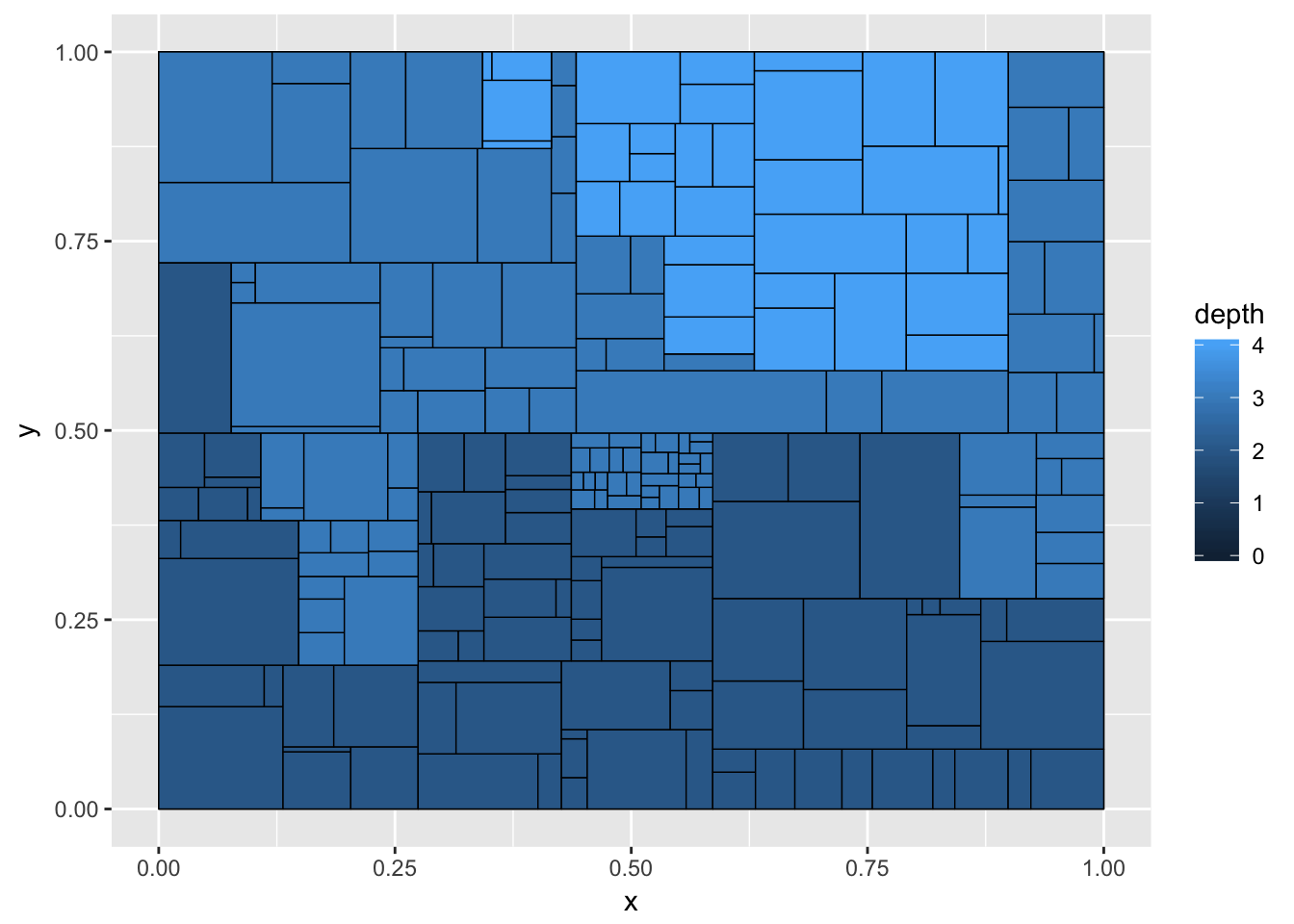
geom\_node\_point(aes(colour = depth)) +

coord\_fixed()



ggraph(graph, 'treemap', weight = 'size') +

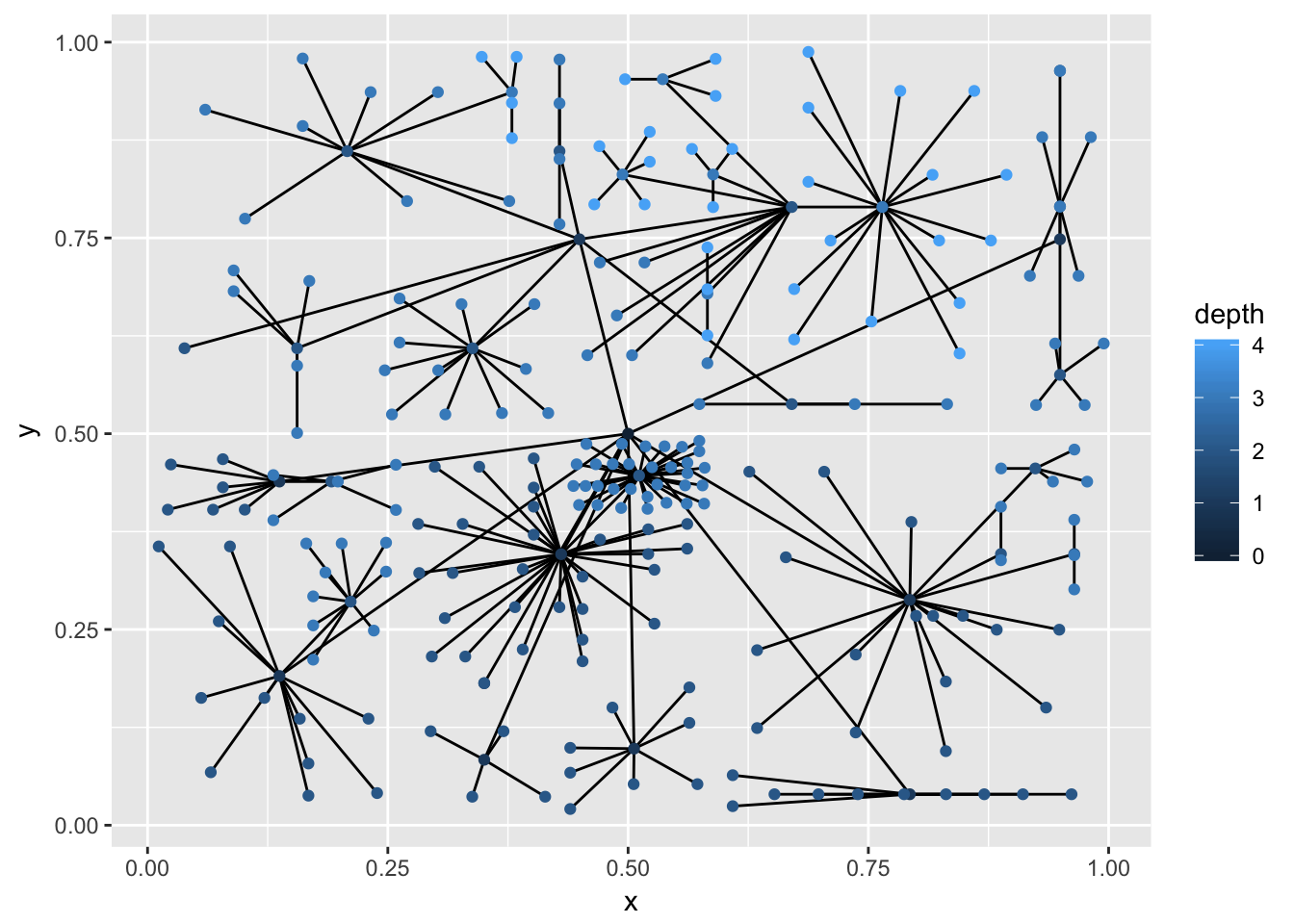
geom\_node\_tile(aes(fill = depth), size = 0.25)



ggraph(graph, 'treemap', weight = 'size') +

geom\_edge\_link() +

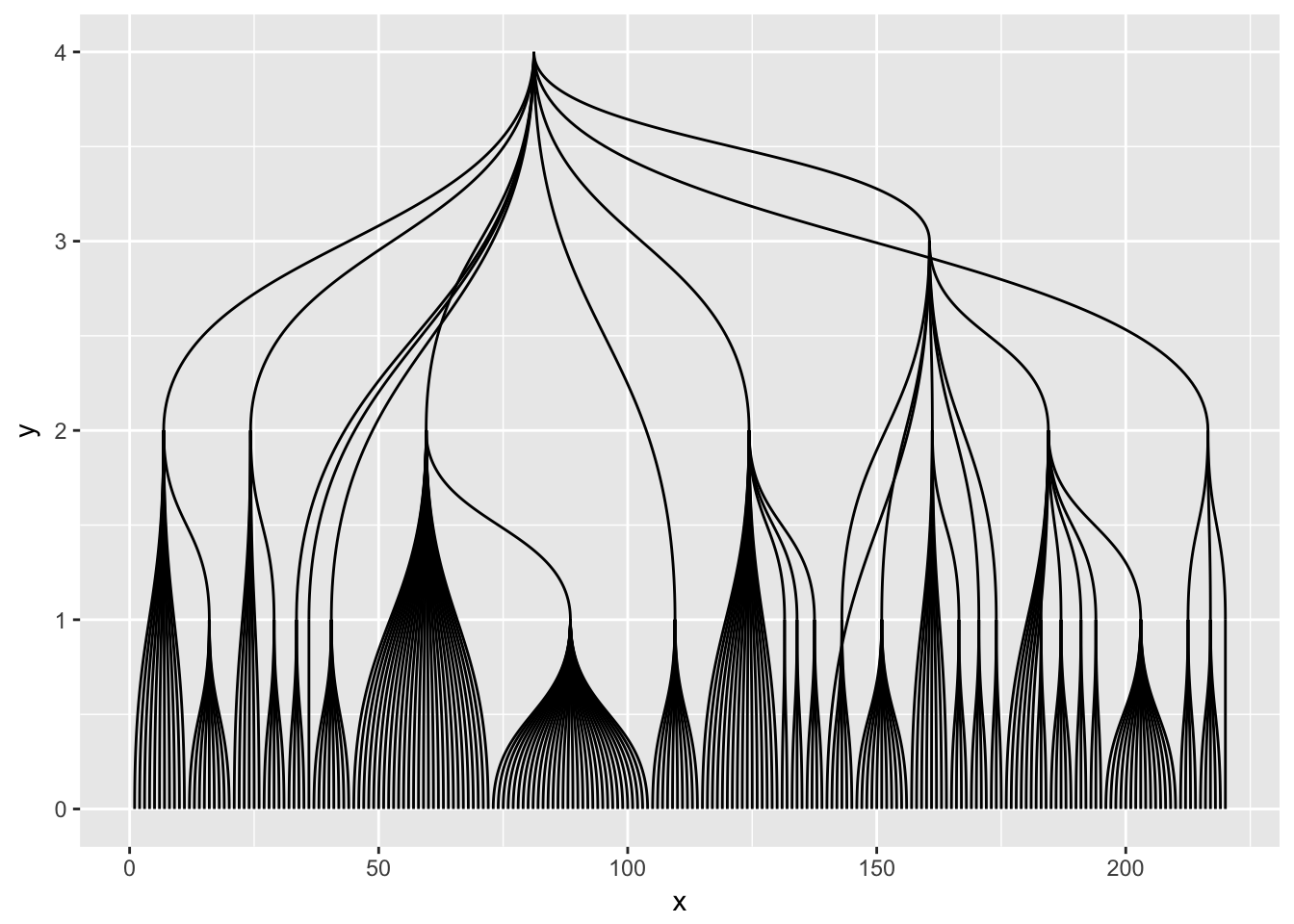
geom\_node\_point(aes(colour = depth))



The most recognized tree plot is probably dendrograms though. Both **igraph** and **dendrogram** object can be plotted as dendrograms, though only **dendrogram** objects comes with a build in height information for placing the branch points. For igraph objects this is inferred by the longest ancestral length:

ggraph(graph, 'dendrogram') +

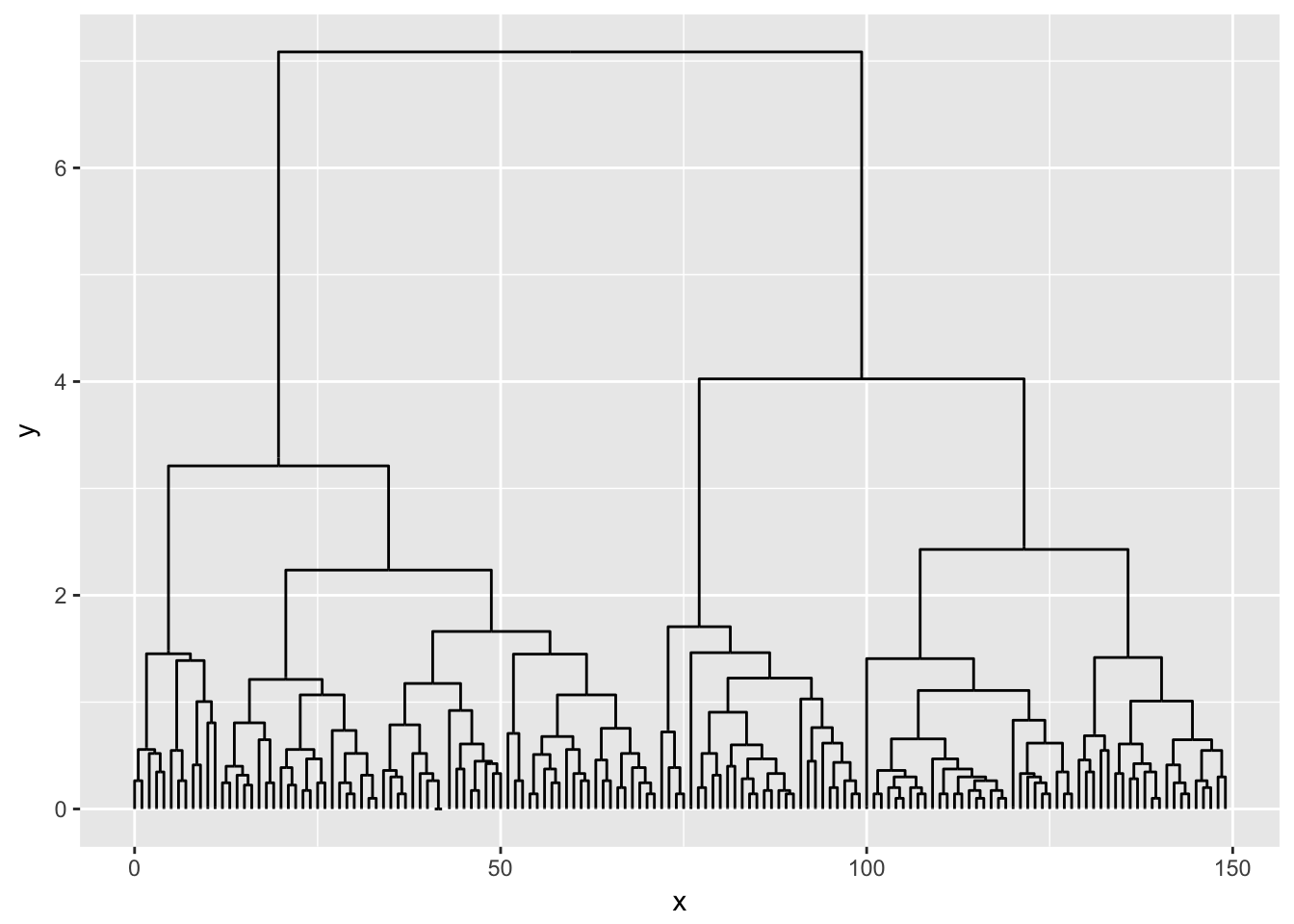
geom\_edge\_diagonal()



dendrogram <- as.dendrogram(hclust(dist(iris[, 1:4])))

ggraph(dendrogram, 'dendrogram') +

geom\_edge\_elbow()

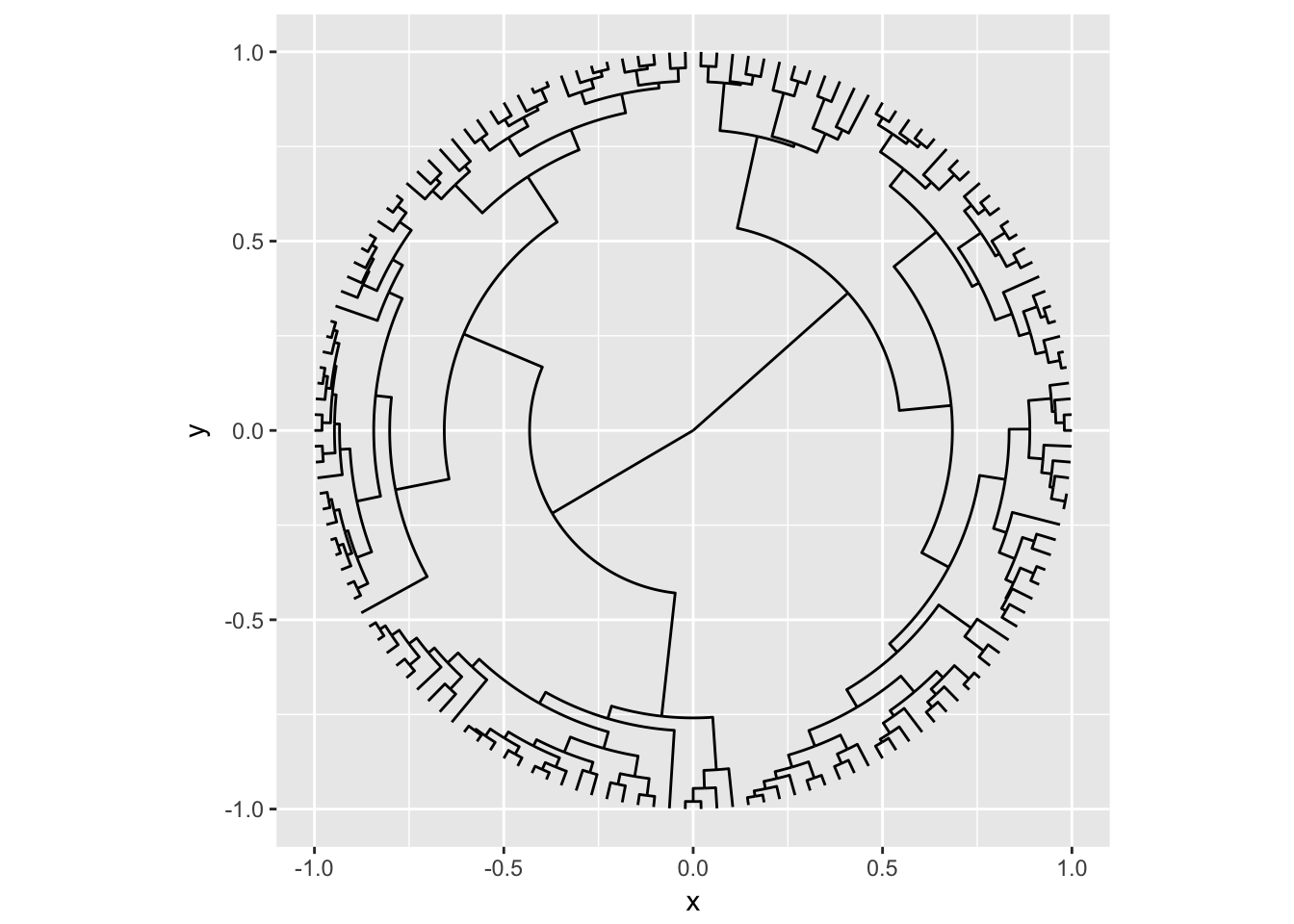


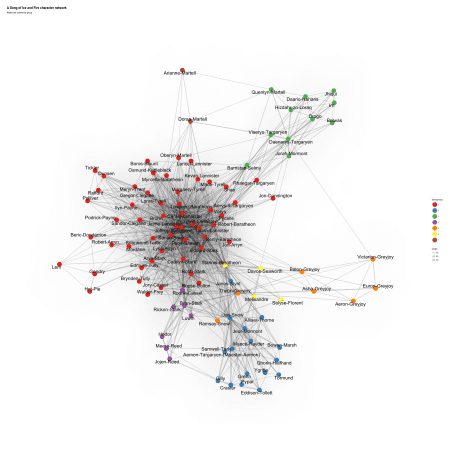
Dendrograms are one of the layouts that are amenable for circular transformations, which can be effective in giving more space at the leafs of the tree at the expense of the space given to the root:

ggraph(dendrogram, 'dendrogram', circular = TRUE) +

geom\_edge\_elbow() +

coord\_fixed()





Interestingly, many of the groups reflect the narrative perfectly: the men from the Night’s Watch are grouped together with the Wildlings, Stannis, Davos, Selyse and Melisandre form another group, the Greyjoys, Bran’s group in Winterfell before they left for the North, Dany and her squad and the Martells (except for Quentyn, who “belongs” to Dany – just like in the books ;-)). The big group around the remaining characters is the only one that’s not split up very well.

For the next graphs, I want specific colors form the RColorBrewer palette “Set1”:

cols <- RColorBrewer::brewer.pal(3, "Set1")

ggraph(layout) +

geom\_edge\_density(aes(fill = weight)) +

geom\_edge\_link(aes(width = weight), alpha = 0.2) +

geom\_node\_point(aes(color = factor(center), size = dist\_to\_center)) +

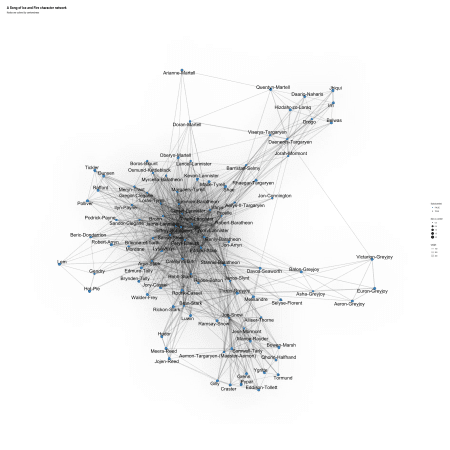
geom\_node\_text(aes(label = name), size = 8, repel = TRUE) +

scale\_colour\_manual(values = c(cols[2], cols[1])) +

theme\_graph() +

labs(title = "A Song of Ice and Fire character network",

subtitle = "Nodes are colored by centeredness")



In the next graph I plotted the center-most characters in red and the distance to center as node size. The two center characters across all books are Robert Baratheon and Tyrion Lannister. I had not expected Robert, since he dies pretty much right away but I guess he and his rebellion following Lyanna’s “abduction” is the main trigger for most of what happens in the books, so why not… And that Tyrion is the best character