

Vignette: Grammar of graphics of genealogy (ggenealogy)

Lindsay Rutter, Susan Vanderplas, Di Cook

ggenealogy version 0.1.0 , 2016-06-12

Contents

Summary	2
Introduction	2
Installation	2
Preprocessing pipeline	3
General (non-plotting) methods of genealogical data	4
Functions for individual vertices	4
Functions for pairs of vertices	8
Functions for the full genealogical structure	9
Plotting methods of genealogical data	10
Plotting the ancestors and descendants of a vertex	11
Plotting the shortest path between two vertices	14
Plotting shortest paths superimposed on full genealogical structure	17
Plotting pairwise distance matrices between a set of vertices	20
Interactive plotting methods of genealogical data	22
Conclusions	27

¹This L^AT_EX vignette document is created using the R function `Sweave` on the R package `ggenealogy`. It is automatically downloaded with the package and can be accessed with the R command `vignette("ggenealogy")`.

Summary

Description

The **ggenealogy** package provides tools to examine genealogical data, generating basic statistics on their graphical structures using parent and child connections, and displaying the results. The genealogy can be drawn in relation to additional variables, such as development year, and the shortest path distances between genetic lines can be determined and displayed. Production of pairwise distance matrices and phylogenetic diagrams constrained by generation count are also available in the visualization toolkit. This vignette is intended to walk readers through the different methods available in the **ggenealogy** package.

Caution

igraph must be used with version $\geq 0.7.1$

Introduction

Installation

R is a open source software project for statistical computing, and can be freely downloaded from the Comprehensive R Archive Network (CRAN) website. The link to contributed documentation on the CRAN website offers practical resources for an introduction to R , in several languages. After downloading and installing R , the installation of additional packages is straightforward. To install the **ggenealogy** package from R , use the command:

```
> install.packages("ggenealogy")
```

The **ggenealogy** package should now be successfully installed. Next, to render it accessible to the current R session, simply type:

```
> library(ggenealogy)
```

To access help pages with example syntax and documentation for the available functions of the **ggenealogy** package, please type:

```
> help(package="ggenealogy")
```

To access more detailed information about a specific function in the **ggenealogy** package, use the following help command on that function, such as:

```
> help(getChild)
```

The above command will return the help file for the `getChild` function. The help file often includes freestanding example syntax to illustrate how function commands are executed. In the case of the `getChild` function, the example syntax is the following three lines, which can be pasted directly into an R session.

```
> data(sbGeneal)
> getChild("Tokyo", sbGeneal)
> getChild("Essex", sbGeneal)
```

Preprocessing pipeline

In the `ggenealogy` package, there is an example dataset containing genealogical information on soybean varieties called `sbGeneal.rda`. It may be helpful to load that example file so that you can follow along with the commands and options introduced in this vignette. To ensure that you have uploaded the correct, raw `sbGeneal.rda` file, you can observe the first six lines of the file, and determine its dimension and structure:

```
> data(sbGeneal)
> head(sbGeneal)
```

	child	year	yield	year.imputed	parent
1	5601T	1981	NA	TRUE	Hutcheson
2	Adams	1948	2734	FALSE	Dunfield
3	A.K.	1910	NA	TRUE	<NA>
4	A.K. (Harrow)	1912	2665	FALSE	A.K.
5	Altona	1968	NA	FALSE	Flambeau
6	Amcor	1979	2981	FALSE	Amsoy 71

```
> dim(sbGeneal)
```

```
[1] 390  5
```

```
> str(sbGeneal)
```

```
'data.frame':      390 obs. of  5 variables:
 $ child      : chr  "5601T" "Adams" "A.K." "A.K. (Harrow)" ...
 $ year       : num  1981 1948 1910 1912 1968 ...
 $ yield      : int   NA 2734 NA 2665 NA 2981 2887 2817 NA NA ...
 $ year.imputed: logi   TRUE FALSE TRUE FALSE FALSE FALSE ...
 $ parent     : chr  "Hutcheson" "Dunfield" NA "A.K." ...
```

We see that the `sbGeneal` data file is a data frame structure with 390 rows (observations) and 5 columns (variables). Each row contains a `child` node character label and `parent` node character label. Each row also contains a numeric value corresponding to the `year` the child node was introduced, an

integer value of the protein `yield` of the child node, and a logical value `year.imputed`, which indicates whether or not the year of introduction of the child node was imputed.

Now that the `sbGeneal` file has been loaded as a data frame, it must next be converted into a graph object using the `dfToIG()` function. The `dfToIG()` function requires a data frame as input, and that data frame should be structured such that each row represents an edge with a child and parent relationship. For more information, try using the help command on the function:

```
> help(dfToIG)
```

We see that the function takes optional parameter arguments, such as `vertexinfo` (a list of columns of the data frame which provide information for the starting “child” vertex, or a separate data frame containing information for each vertex with the first column as the vertex name), `edgeweights` (a column that contains edge values, with a default value of unity), and `isDirected` (a boolean value that describes whether the graph is directed (true) or undirected (false); the default is false).

In this example, we want to produce an undirected graph object that contains all edge weight values of one, because our goal is to set an edge value of unity for every pair of vertices (individuals) that are related as parent and child. The `dfToIG()` function uses the software `igraph` to convert the data frame into a graph object. For clarity, we will assign the outputted graph object the name `ig` (for `igraph` object), and then examine its class type:

```
> ig <- dfToIG(sbGeneal)
> class(ig)
```

```
[1] "igraph"
```

Above, we confirmed that the `ig` object is of class type `igraph`. The `ig` object is required as input in many `ggenealogy` functions, which will be demonstrated below.

General (non-plotting) methods of genealogical data

The `ggenealogy` package offers several functions that result in useful information beside plots. Below is a brief introduction to some of the available non-plotting functions.

Functions for individual vertices

The `ggenealogy` package offers several functions that you can use to obtain information for individual vertices. First, the function `isParent()` can return a logical variable to indicate whether or not the second variety is a parent of the first variety.

```
> isParent("Young", "Essex", sbGeneal)
```

```
[1] TRUE
```

```
> isParent("Essex", "Young", sbGeneal)
```

```
[1] FALSE
```

We see that “Essex” is a parent of “Young”, and not vice-versa. Similarly, the function `isChild()` can return a logical variable to indicate whether or not the first variety is a child of the second variety.

```
> isChild("Young", "Essex", sbGeneal)
```

```
[1] TRUE
```

```
> isChild("Essex", "Young", sbGeneal)
```

```
[1] FALSE
```

We see that, as expected, “Young” is a child of “Essex”, and not vice-versa. It is also possible to derive the year of a given variety using the `getYear()` function:

```
> getYear("Young", sbGeneal)
```

```
[1] 1968
```

```
> getYear("Essex", sbGeneal)
```

```
[1] 1962
```

Fortunately, the returned year values are consistent, as the “Young” variety (1968) is a child to the “Essex” variety (1962) by an age difference of 6 years. In some cases, you may wish to obtain a complete list of all the parents of a given variety. This can be achieved using the `getParent()` function:

```
> getParent("Young", sbGeneal)
```

```
[1] "Davis" "Essex"
```

```
> getParent("Tokyo", sbGeneal)
```

```
character(0)
```

```
> getYear("Tokyo", sbGeneal)
```

```
[1] 1907
```

We learn from this that “Essex” is not the only parent of “Young”; “Young” also has a parent “Davis”. We also see that “Tokyo” does not have any documented parents in this dataset, and has an older year of introduction (1907) than other varieties we have examined thusfar. Likewise, in other cases, you may wish to obtain a complete list of all the children of a given variety. This can be achieved using the `getChild()` function:

```
> getChild("Tokyo",sbGeneal)
```

```
[1] "Ogden"      "Volstate"
```

```
> getChild("Ogden",sbGeneal)
```

```
[1] "C1069"      "C1079"      "D51-2427"    "D55-4090"    "D55-4159"
[6] "D55-4168"    "Kent"       "N44-92"      "N45-745"     "N48-1101"
[11] "Ogden x CNS"  "Ralsoy x Ogden"
```

We find that even though the “Tokyo” variety is a grandparent of the dataset, it only has two children, “Ogden” and “Volstate”. However, one of its children, “Ogden”, produced 12 children.

If we want to obtain a list that contains more than just one generation past or previous to a given variety, then we can use the `getAncestors()` and `getDescendants()` functions, where we specify the number of generations we wish to view. This will return a data frame to us with the labels of each ancestor or descendant, along with the number of generations each one is from the given variety.

If we only look at one generation of ancestors of the “Young” variety, we should see the same information we did earlier when we used the `getParent()` function of the Young variety:

```
> getAncestors("Young",sbGeneal,1)
```

```
  label gen
2 Davis   1
1 Essex   1
```

Indeed, we consistently see that the “Young” variety has only 2 ancestors within one generation, “Davis” and “Essex”. However, if we view the first five generations of ancestors of the “Young” variety, we can view four more generations of ancestors past simply the parents:

```
> getAncestors("Young",sbGeneal,5)
```

```
          label gen
27      Davis   1
26      Essex   1
```

25	Ral soy x Ogden	2
24	Roanoke x (Ogden x CNS)	2
23	Lee	2
22	S55-7075	2
21	Ogden	3
20	Ral soy	3
19	Ogden x CNS	3
17	CNS	3
18	Roanoke	3
16	S 100	3
15	N48-1248	3
14	Perry	3
10	Ogden	4
13	PI 54610	4
12	Tokyo	4
11	CNS	4
9	Clemson	4
6	Roanoke	4
8	Illini	4
7	N45-745 x (Ogden x CNS)	4
4	PI 54610	5
3	Tokyo	5
1	Ogden x CNS	5
5	Clemson	5
2	A.K.	5

```
> nrow(getAncestors("Young",sbGeneal,5))
```

```
[1] 27
```

In the second line of code above, we determined the dimensions of the returned data frame, and see that there are 27 ancestors within the first five ancestral generations of the “Young” variety.

Similarly, if we only look at the first generation of descendants of the “Ogden” variety, we should see the same information as we did earlier when we used the `getChild()` function on the “Ogden” variety:

```
> getDescendants("Ogden",sbGeneal,1)
```

	label	gen
12	C1069	1
11	C1079	1
10	D51-2427	1
9	D55-4090	1
8	D55-4159	1
7	D55-4168	1
6	Kent	1
5	N44-92	1
4	N45-745	1

```

3      N48-1101    1
2      Ogden x CNS 1
1  Ralsoy x Ogden  1

```

Indeed, we see again that “Ogden” has 12 children. Additionally, if we want to view not only the children, but also the grandchildren, of the “Ogden” variety, then we can use this function, only now specifying two generations of descendants:

```
> getDescendants("Ogden",sbGeneal,2)
```

	label	gen
28	C1069	1
27	C1079	1
26	D51-2427	1
25	D55-4090	1
24	D55-4159	1
23	D55-4168	1
22	Kent	1
21	N44-92	1
20	N45-745	1
19	N48-1101	1
18	Ogden x CNS	1
17	Ralsoy x Ogden	1
16	Columbus	2
15	Cutler	2
14	C1266R	2
13	Semmes	2
11	D60-7965	2
12	D60-7965	2
10	D59-9289	2
9	Beeson	2
8	Calland	2
7	Hood	2
6	N48-1867	2
5	D52-810	2
4	N45-745 x (Ogden x CNS)	2
3	R54-168	2
2	Roanoke x (Ogden x CNS)	2
1	Davis	2

We see that variety “Ogden” has 16 grandchildren from its 12 children.

Functions for pairs of vertices

Say you have a pair of vertices, and you wish to determine the degree of the shortest path between them, where edges represent parent-child relationships. You can accomplish that with the `getDegree()` function.


```
> getDegree("Tokyo", "Ogden", ig, sbGeneal)
```

```
[1] 1
```

```
> getDegree("Tokyo", "Holladay", ig, sbGeneal)
```

```
[1] 7
```

As expected, the shortest path between the “Tokyo” and “Ogden” varieties has a value of 1, as we already determined that they have a direct parent-child relationship. However, the shortest path between “Tokyo” and one of its descendants, “Holladay”, has a much higher degree of 7.

Note that degree calculations in this case are not limited to one linear string of parent-child relationships; cousins and siblings and products thereof will also have computable degrees via nonlinear strings of parent-child relationships.

Functions for the full genealogical structure

There are many parameters about the full genealogical structure that you may wish to know that cannot easily be obtained through images and tables. The function `getBasicStatistics()` will return graph theoretical measurements of the full genealogy. For instance, is the full genealogy connected? If not, how many separated components does it contain? In addition to these parameters, the `getBasicStatistics()` function will also return the number of nodes, the number of edges, the average path length, the graph diameter, among others:

```
> getBasicStatistics(ig)
```

```
$isConnected
```

```
[1] FALSE
```

```
$numComponents
```

```
[1] 11
```

```
$avePathLength
```

```
[1] 5.333746
```

```
$graphDiameter
```

```
[1] 13
```

```
$numNodes
```

```
[1] 230
```

```
$numEdges
```

```
[1] 340
```

```
$logN
```

```
[1] 5.438079
```

In this case, we learn that our full genealogical structure is not all connected by parent-child edges. Instead, it is composed of 11 separate components. The average path length of the full genealogy is 5.333746, that the graph diameter is 13, and that the logN value is 5.438079. We also see that the number of nodes in the full genealogy is 230, and the number of edges in the full genealogy is 340.

But can we view a list of these nodes and edges? To do so, we can call the `getNodes()` and `getEdges()` commands to obtain lists of all the unique nodes and edges in the full genealogical structure. Here, we obtain a list of the 340 edges (with each row containing the names of the two connected vertices, and an edge weight, if existent). We will simply view the first six rows of the object, and determine the number of edges by counting the number of rows (340):

```
> eList = getEdges(ig, sbGeneal)
> head(eList)
```

	child	parent
[1,]	"5601T"	"Hutcheson"
[2,]	"Adams"	"Dunfield"
[3,]	"A.K. (Harrow)"	"A.K."
[4,]	"Altona"	"Flambeau"
[5,]	"Amcor"	"Amsoy 71"
[6,]	"Amsoy"	"Adams"

```
> nrow(eList)
```

```
[1] 340
```

We then obtain a list of the 230 nodes. Again, we only view the first six rows of the object, and determine the number of nodes by counting the number of indices (230).

```
> nList = getNodes(sbGeneal)
> head(nList)
```

[1]	"5601T"	"Adams"	"A.K."	"A.K. (Harrow)"	"Altona"
[6]	"Amcor"				

```
> length(nList)
```

```
[1] 230
```

Plotting methods of genealogical data

Until this point, the vignette has introduced functions that return lists, data frames, and statistics about the genealogical dataset. However, the `ggenealogy` package also contains visualization tools for genealogical datasets. Access to various types of visual plots and diagrams of the lineage can allow genealogical researchers to more efficiently and accurately explore an otherwise complicated data structure. Below, we introduce functions in `ggenealogy` that produce visual outputs of the dataset.

Plotting the ancestors and descendants of a vertex

One visualization tool, `plotAncDes()`, allows the user to view the ancestors and descendants of a given variety. The inputted variety is highlighted in the center of the plot, ancestors are displayed to the left of the center, and descendants are displayed to the right of the center. The further left or right from the center, the larger the number of generations that particular ancestor/descendant is from the inputted and centered variety.

As such, this plotting command does not provide visual information about specific years associated with each related variety (as is done in some of the visualization tools introduced later), but it does group all varieties from each generation group onto the same position of the horizontal axis. Here, we specify that we want to plot 5 ancestor generations and 4 descendant generations of the variety “Lee”:

```
> plotAncDes("Lee", sbGeneal, 5, 4)
```

We immediately see in Figure 1 that this visual representation of the ancestors and descendants of a given variety can often provide enhanced readability compared to the list output provided in the previous functions, `getAncestors()` and `getDescendants()`. We notice that even though we specified for 5 generations of ancestors, the extent of documented ancestors of “Lee” includes only 3 generations.

We also see now that some node labels are repeated. For instance, the “5601T” variety appears twice, once as a great-grandchild (third generation descendant) of “Lee”, and once as a great-great-grandchild (fourth generation descendant) of “Lee”. This is because there are two separate parent-child pathways between “Lee” and “5601T”, one pathway with only two nodes (“Essex” and “Hutchson”) between them, and one pathway with three nodes (“Essex”, “T80-69”, and “TN89-39”) between them.

Why does this happen? In this visual tool, we are constraining the horizontal axis to generation count. Without allowing nodes to repeat, this data information cannot be clearly and succinctly presented. Most graph visualization software that genealogists might use to view their datasets do not allow for repeated nodes, as per the definition of a graph. Hence, the `plotAncDes()` function is one of the more unique visual tools of the `ggenealogy` package.

It should be noted that the `plotAncDes()` function, by default, highlights the centered variety label in pink. However, the user can alter this color, as we will show next. Furthermore, the user can specify additional grammar of graphics plotting tools (from the `ggplot2` package) to tailor the output of the `plotAncDes()` function, which we also show below.

For example, we will now change the color of the center variety label `vColor` to be highlighted in blue. Also, we will add a horizontal axis label called “Generation index”, using the `ggplot2` syntax. Note that this time we do not specify the generational count for ancestors and descendants, and so the default value of three generations is applied to both cases. Remember, to determine such default values, as well as all function parameters, simply run the help command on the function of interest.

```
> plotAncDes("Tokyo", sbGeneal, vColor = "blue") + ggplot2::labs(x="Generation index", y="")
```

We verify immediately from Figure 2 that the “Tokyo” variety does not have any ancestors in this dataset, an observation consistent with what we discovered earlier. We also see the “Tokyo” variety only has two children, but has many more grandchildren, and great-grand children.

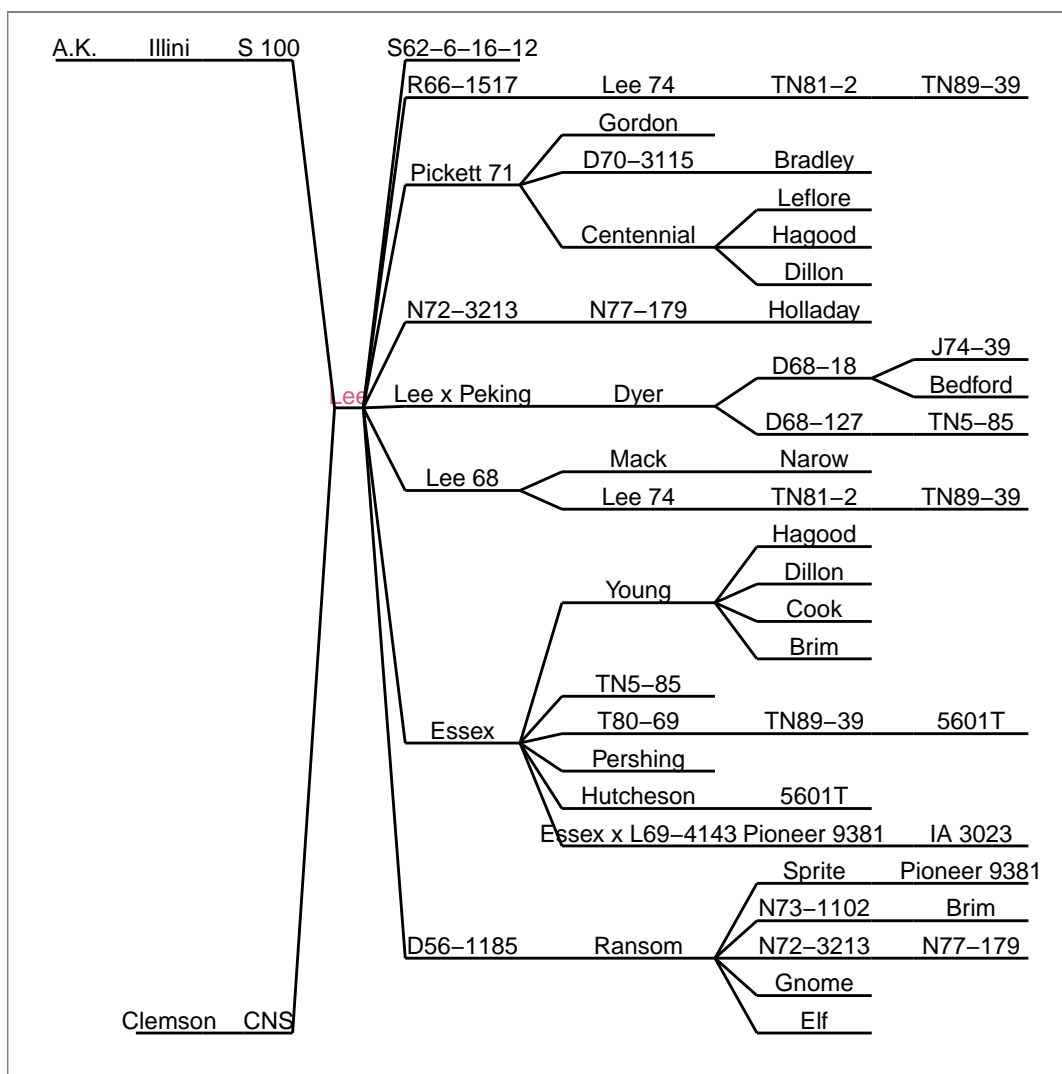


Figure 1: Ancestors and descendants of the "Lee" variety, constrained on the horizontal axis by generational separation from "Lee".

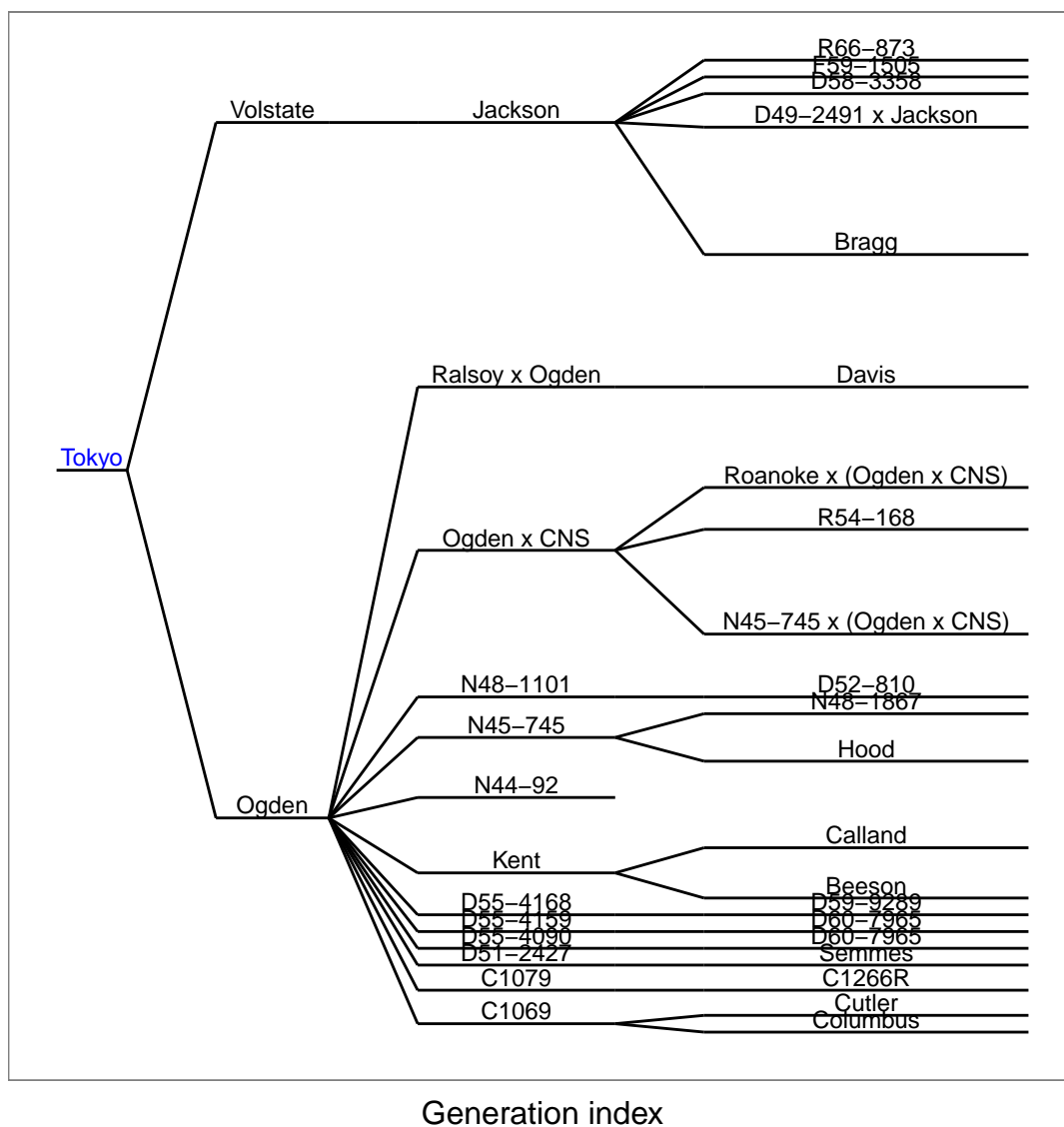


Figure 2: Ancestors and descendants of the “Tokyo” variety, constrained on the horizontal axis by generational separation from “Tokyo”.

Plotting the shortest path between two vertices

As this data set deals with soybean lineages, it may be useful for agronomists to track how two varieties are related to each other via parent-child relationships. Then, any dramatic changes in protein yield, SNP varieties, and other measures of interest between the two varieties can be tracked across their genetic timeline, and pinpointed to certain paths within their historical lineage.

The `ggenealogy` software allows users to select two varieties of interest, and determine the shortest pathway of parent-child relationships between them, using the `getPath()` function. This will return a list path object that contains the variety names and their years in the path. The returned path object can then be plotted using the `plotPath()` function, which we now demonstrate.

The `getPath()` function determines the shortest path between the two inputted vertices, and takes into account whether or not the graph is directed with the parameter `isDirected`, which defaults to `false`. The `getPath()` function will check both directions and return the path if it exists:

```
> getPath("Brim", "Bedford", ig, sbGeneal, isDirected=FALSE)

$pathVertices
[1] "Brim"      "Young"     "Essex"     "T80-69"    "J74-40"    "Forrest"   "Bedford"

$yearVertices
[1] "1977" "1968" "1962" "1975" "1975" "1973" "1978"
```

We see that there is a path between “Brim” and “Bedford” varieties, with 5 varieties separating them. We are not considering direction, however, because the `ig` object is undirected. However, to demonstrate the importance of direction, we will recompute the path where the direction matters. We first produce a directed `igraph` object `dirIG`, and then try to determine the path between the same two vertices, “Brim” and “Bedford”.

```
> dirIG = dfToIG(sbGeneal, isDirected = TRUE)
> getPath("Brim", "Bedford", dirIG, sbGeneal, isDirected = TRUE)
```

Now that we are considering the direction, we are only considering paths where each edge represents a parent-child relationship in the same direction as the one before it. We now would receive an error warning that we cannot compute a directed path on an undirected graph. We next try to reverse the input order of the vertices, as shown below, but we will receive the same error message:

```
> getPath("Bedford", "Brim", dirIG, sbGeneal, isDirected=TRUE)
```

We can derive from the errors returned in the last two commands that the varieties “Brim” and “Bedford” are not connected by a linear sequence of parent-child relationships. Rather, the path between them branches at some point, involving siblings and/or cousins.

Hence, unless you are working with a dataset that must be analyzed as a directed graph, it is best to use the `getPath()` function with the default third parameter indicating lack of direction, and to use an `igraph` object without direction, such as our original `ig` object. We do just that, and save the path between these two varieties to a variable called `path`:

```
> pathBB = getPath("Bedford", "Brim", ig, sbGeneal, isDirected=FALSE)
```

Now that we have a non-empty `pathBB` object that consists of two lists (for variety names and years), we can plot the relationship between the two using the `plotPath()` function.

```
> plotPath(pathBB)
```

This produces a neat visual (see Figure 3) that informs us of all the varieties involved in the shortest path between “Brim” and “Bedford”. In this plot, the years of all varieties involved in the path are indicated on the horizontal axis, while the vertical axis has no meaning other than to simply to display the labels evenly spaced vertically.

Although a call to the `ggenealogy` function `getYear()` indicates that “Bedford” was developed in 1978 and “Brim” in 1977, we quickly determine from the plot that “Brim” is not a parent, grandparent, nor

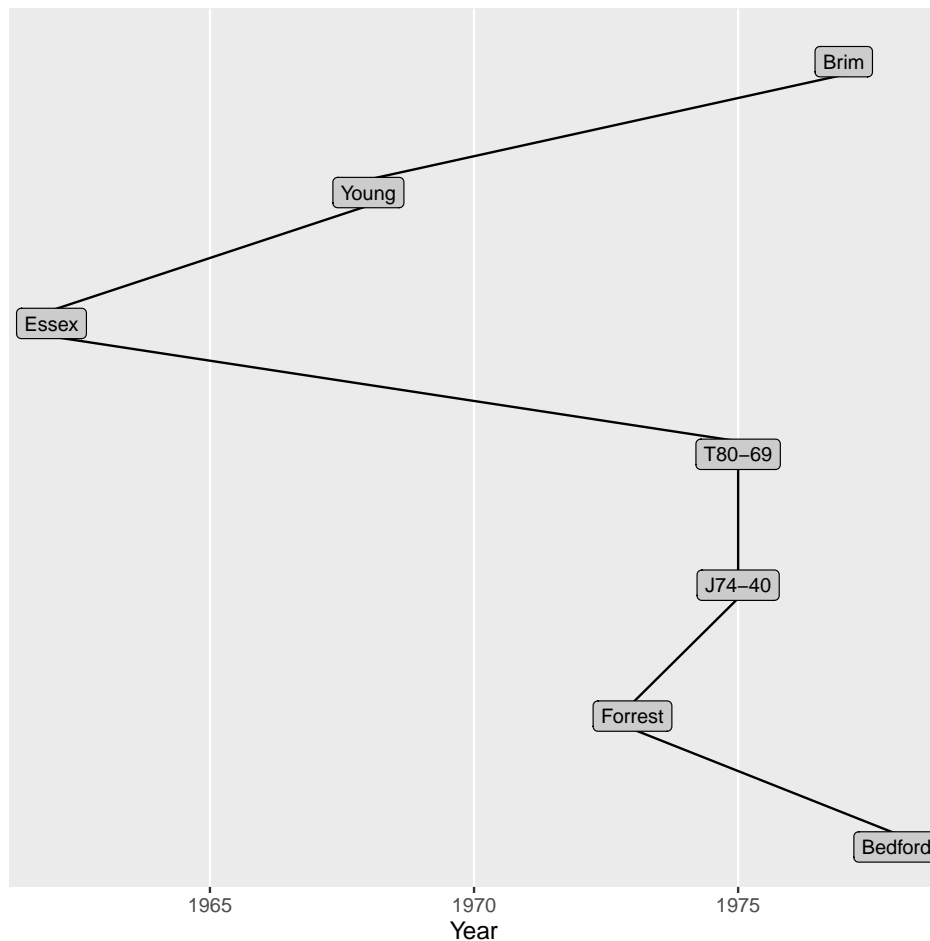


Figure 3: The shortest path between varieties “Brim” and “Bedford” is not strictly composed of unidirectional parent-child relationships, but instead, includes cousin-like relationships.

any great-grandparent of “Bedford”. Instead, we see that these two varieties are not related through a unidirectional parent-child lineage, but have a cousin-like relationship. The oldest common ancestor between “Bedford” and “Brim” is the variety “Essex”, which was developed in 1962.

However, there are other cases of pairs of varieties that are connected by a linear, unidirectional combination of parent-child relationships, as we see below:

```
> pathNT = getPath("Narow", "Tokyo", ig, sbGeneal, isDirected=FALSE)
> plotPath(pathNT)
```

From the output, shown in Figure 4, we see that the variety “Tokyo” is an ancestor of “Narow” via four linear parent-child relationships. Because of this, we can still view the pathway, even when we use an `igraph` object `dirIG` that is directed, and set the boolean `isDirected` variable to true.

Either ordering of the two varieties will produce the exact same result. In other words, the following

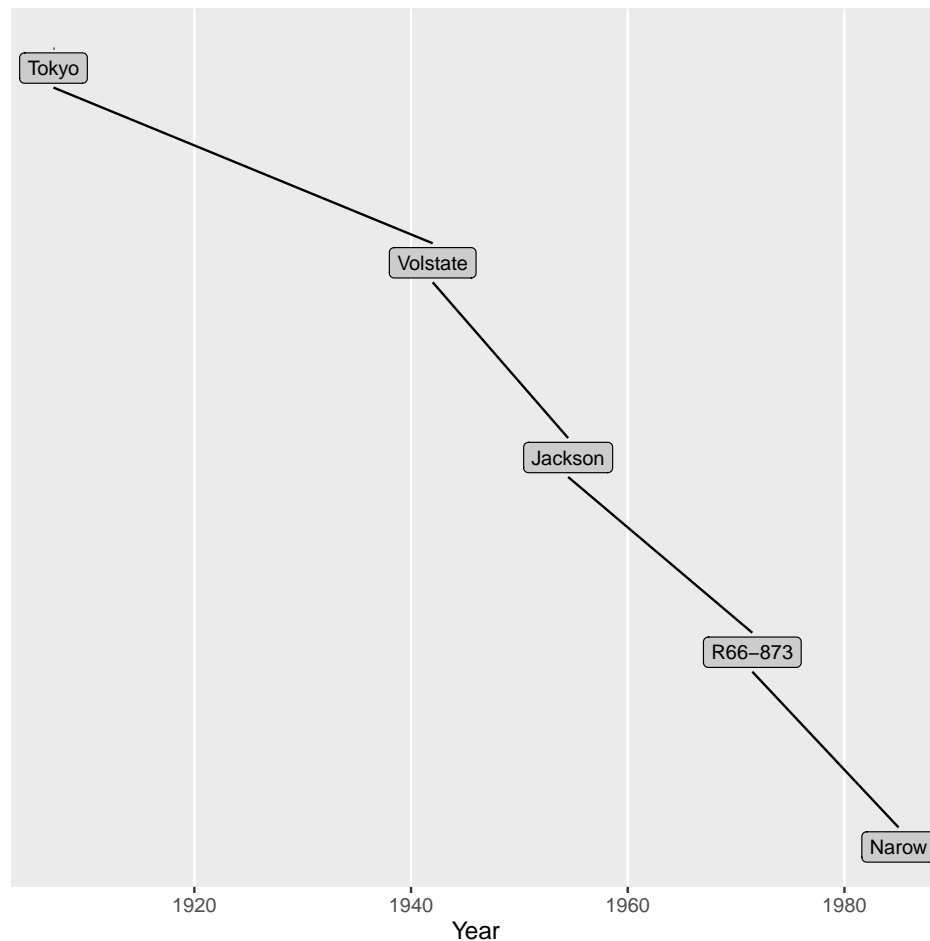


Figure 4: The shortest path between varieties “Narow” and “Tokyo” is strictly composed of a unidirectional sequence of parent-child relationships.

two code blocks would produce the same figure we saw in Figure 4:

```
> pathNT = getPath("Narrow", "Tokyo", dirIG, sbGeneal, isDirected=TRUE)
> plotPath(pathNT)

> pathTN = getPath("Tokyo", "Narrow", dirIG, sbGeneal, isDirected=TRUE)
> plotPath(pathTN)
```

Plotting shortest paths superimposed on full genealogical structure

Now that we can create and plot path objects, we may wish to know how those paths are positioned in comparison to the genealogical lineage of the entire data structure. For instance, of the documented soybean cultivar lineage varieties, where does the shortest path between two varieties of interest exist? Are these two varieties comparatively older compared to the overall data structure? Are they newer? Or, do they span the entire structure, and represent two extreme ends of documented time points?

There is a function available in the **ggenealogy** package, `plotPathOnAll()`, that allows users to quickly visualize their path of interest superimposed over all varieties and edges present in the whole data structure. Here we will produce a plot of the previously-determined shortest path between varieties “Tokyo” and “Narrow” across the entire dataset (in this particular dataset, some edges are not plotted, as they contain NA values):

```
> plotPathOnAll(pathNT, sbGeneal, ig, binVector = 1:3)
```

The resulting plot is shown in Figure 5.

While the first three explicit parameters to the function `plotPathOnAll()` have been introduced earlier, the fourth parameter (`binVector`) requires some explanation. The motivation of the `plotPathOnAll()` function is to write variety text labels on a plot, with the center of each variety label constricted on the horizontal axis to its developmental year. As is the case for the plots before, the vertical axis has no specific meaning. Unfortunately, for large datasets, this motivation can be a difficult task because the text labels of the varieties can overlap if they are assigned a similar y coordinate, have a similar year (x coordinate), and have labels with large numbers of characters (width of x coordinate).

For each variety, the x coordinate (year) and width of the x coordinate (text label width) cannot be altered, as they provide useful information. However, the vertical coordinate is arbitrary. Hence, in an attempt to mitigate text overlap, the `plotPathOnAll()` function does not randomly assign the vertical coordinate. Instead, it allows users to specify the number of bins (`binVector`), which partially controls the vertical positions.

If the user determines to produce a plot using three bins, as in the example code above, then the varieties are all grouped into three bins based on their years of development. In other words, there will be bin 1 (the “oldest bin”) which includes the one-third of all varieties with the oldest developmental years, bin 2 (the “middle bin”), and bin 3 (the “youngest bin”).

Then, in order to decrease text overlap, consecutively increasing vertical positions are alternatively assigned to the three bins (For example: bin 1, then bin 2, then bin 3, etc.) repeatedly until all

of years (1950-1970) of which the most varieties had development years.

Plotting pairwise distance matrices between a set of vertices

It may also be of interest to generate matrices where the cell colors indicate the magnitude of a variable (such as the degree of the shortest path) between all pairwise combinations of inputted varieties. The package `ggenealogy` also provides a function `plotDegMatrix()` for that purpose.

Here, we plot a distance matrix for a set of 8 varieties, defining both the x- and y- axes titles as “Soybean label”, and the legend label as “Degree”. Syntax from the `ggplot2` package can be appended to tailor the output of the `plotDegMatrix()` function. In this case, we denote pairs with small degrees to be colored white, and pairs with large degrees to be colored dark green, using `scale_fill_continuous()`:

```
> varieties=c("Brim", "Bedford", "Calland", "Narrow", "Pella", "Tokyo", "Young", "Zane")
> p = plotDegMatrix(varieties, ig, sbGeneal, "Soybean label", "Soybean label", "Degree")
> p + ggplot2::scale_fill_continuous(low="white", high="darkgreen")
```

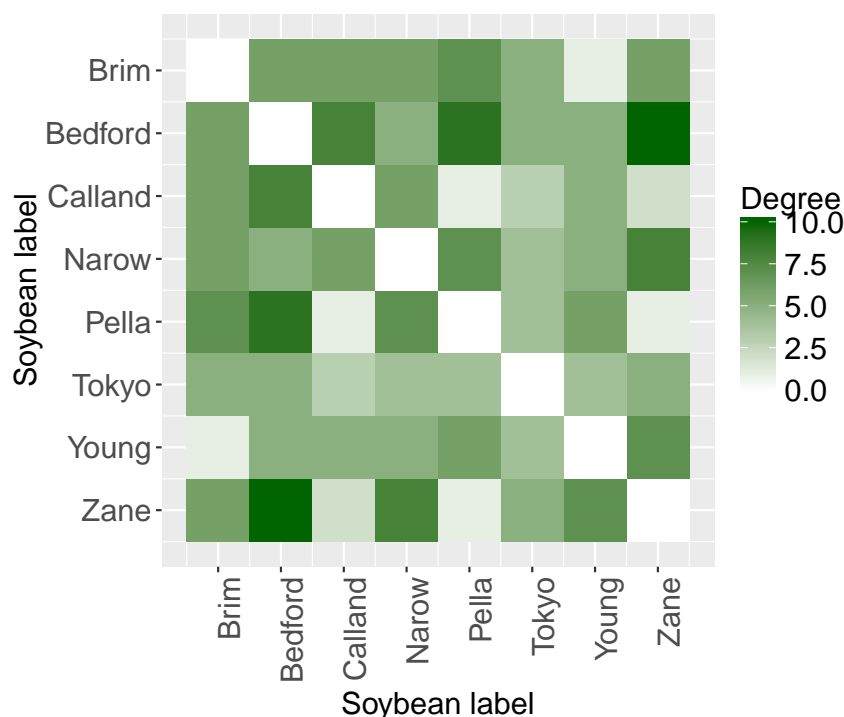


Figure 7: Colored matrix plot showing the degrees of the shortest paths between all pair combinations from a set of eight varieties of interest.

Figure 7 shows that the degree of the shortest path between varieties “Bedford” and “Zane” seems to be the largest in the dataset, which should be around 10. We can verify this simply with:

```
> getDegree("Bedford", "Zane", ig, sbGeneal)
```

```
[1] 10
```

Indeed, the degree of the shortest path between “Bedford” and “Zane” is 10. The distance matrix plot provides us additional information: The degree of 10 may be a comparatively large degree within the given soybean dataset `sbGeneal`, seeing that the degrees of the shortest paths for the other 27 pairwise combinations of the eight varieties that we explored here are less than 10.

In a similar function `plotYearMatrix()`, the difference in years between all pairwise combinations of vertices can be constructed and viewed:

```
> varieties=c("Brim", "Bedford", "Calland", "Narrow", "Pella", "Tokyo", "Young", "Zane")
> plotYearMatrix(varieties,sbGeneal)
```

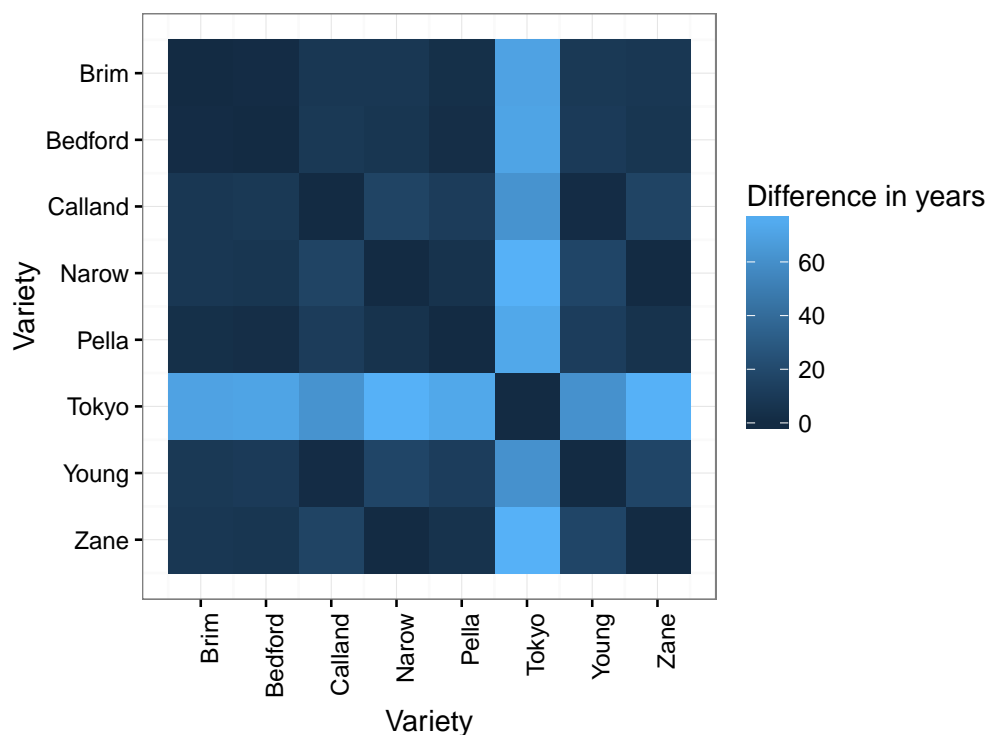


Figure 8: Colored matrix plot showing the year differences between all pair combinations from a set of eight varieties of interest.

Here, we did not change any defaults. As such, the resulting plot in Figure 8 contains the default values of “Variety” for the x-and y-axis labels, and “Difference in years” for the legend label. It also uses the default colors of dark blue for small year difference and light blue for large year difference.

Running this function on this particular set of eight vertices suggests that most combinations of varieties are only one or two decades apart in year introduction, with the exception of the “Tokyo” variety, which appears to be separated from each of the other seven varieties by about six decades. This is not surprising, because we have seen throughout the tutorial that the “Tokyo” variety is the oldest variety in the dataset.

Interactive plotting methods of genealogical data

There is a second example dataset included in the `ggenealogy` package of the academic genealogy of statisticians. More information about this example dataset can be found in the `R/data-statGeneal.R` file. We can load the example dataset of academic genealogy of statisticians (`statGeneal`) and examine its structure.

```
> data("statGeneal")
> dim(statGeneal)

[1] 8165    6

> colnames(statGeneal)

[1] "child"    "parent"   "year"     "country"  "school"   "thesis"
```

As this example academic genealogy dataset is much larger than the example soybean dataset, we can begin by creating a plot of ancestors and descendants. The ability to plot ancestors and descendants by generation was demonstrated using the plant breeding genealogy in Figure 1 and 2. As we believe this is the most novel plotting tool in the `ggenealogy` package, we will test it again here using the academic genealogy.

We need to choose a central individual of interest in order to create this plot. Perhaps we can use the academic statistician in the dataset that has the largest number of “descendants”. To determine the name of this individual, below we use the `ggenealogy` function `getNode()` to create a vector `indVec` that contains the names of all individuals in the dataset. We then use the `dplyr` package to apply the `ggenealogy` function `getDescendants()` on each individual in the `indVec` vector (?). We set the parameter `gen` to a conservatively large value of 100 as this dataset is unlikely to have any individuals with more than 100 generations of “descendants”.

After that, we can generate a table to examine all values of “descendant” counts in the dataset, along with the number of individuals who have each of those values of “descendant” counts. Of the 8165 individuals in this dataset, 6252 of them have zero “descendants”, 322 of them have one “descendant”, and 145 of them have two “descendants”. There are only 17 individuals who have more than 30 “descendants”, and there is one individual who has the largest value of 159 “descendants”. We determine that this individual is the prominent British statistician Sir David Cox, who is known for the Box-Cox transformation and Cox processes, as well as for mentoring many younger researchers who later became notable statisticians themselves.

```
> library("dplyr")
> indVec <- getNode(statGeneal)
```

```

> indVec <- indVec[which(indVec != "", )]
> dFunc <- function(var) nrow(getDescendants(var, statGeneal, gen = 100))
> numDesc <- sapply(indVec, dFunc)
> table(numDesc)

```

```

numDesc
  0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18
6251 322 145  88  58  36  31  22  23  14  17  13  14  10   9   6   4   3   2
 19   20   21   22   23   24   25   26   27   29   30   34   37   38   40   41   44   45   48
   5    7    4    3    3    2    2    6    1    1    3    2    1    1    1    1    1    1    1
 49   60   61   75   77   84  159
   2    1    1    1    1    1    1

```

```

> which(numDesc == 159)

```

```

David Cox
  1980

```

We can now visualize how these 159 “descendants” are related to Sir David Cox by calling the `plotAncDes()` function of `ggenealogy`. As such, we create Figure 9 using the code below.

```

> plotAncDes("David Cox", statGeneal, mAnc = 6, mDes = 6, vCol = "blue")

```

We see from Figure 9 that Sir David Cox had 42 “children”, many of them becoming notable statisticians themselves, such as Basilio Pereira, Valerie Isham, Gauss Cordeiro, Peter McCullagh, and Henry Wynn. Of his “children”, the one who produced the most “children” of their own was Peter Bloomfield, who has 26 “children” and 49 “descendants”. In total, Sir David Cox had five generations of academic statistics mentees in this dataset.

```

> length(getChild("Peter Bloomfield", statGeneal))

```

```

[1] 26

```

```

> nrow(getDescendants("Peter Bloomfield", statGeneal, gen = 100))

```

```

[1] 49

```

At this point, it would be insightful to examine a more detailed view of one of the longest strings of “parent-child” relationships between Sir David Cox and one of the two individuals who are his fifth generation “descendants”. We do so with the code below, choosing his fifth generation “descendant” to be Petra Buzkova. We set the `fontFace` variable of the `plotPath()` to a value of 4, indicating we wish to boldface and italicize the two varieties of interest.

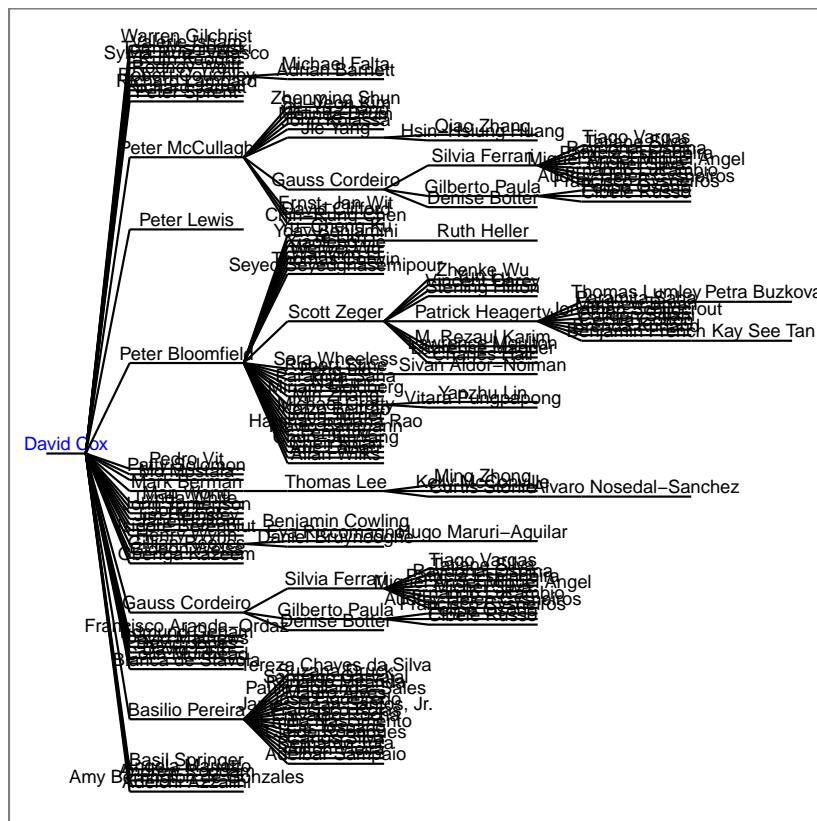


Figure 9: The 159 academic statistician “descendants” of Sir David Cox.

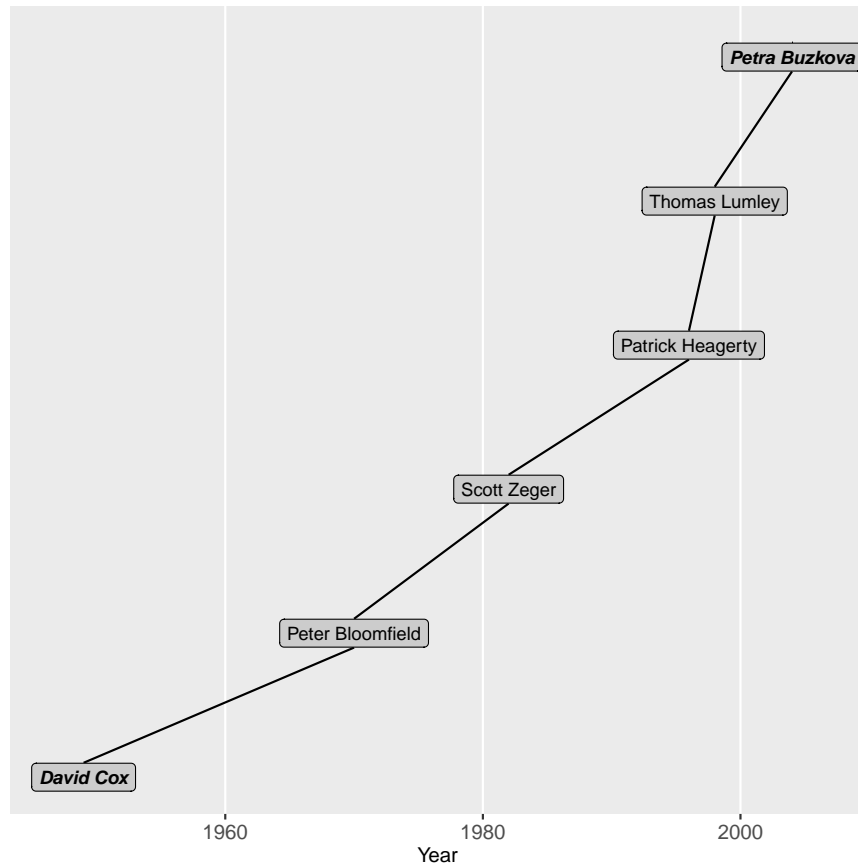


Figure 10: The shortest path between Sir David Cox and one of his fifth generation “descendants”, Petra Buzkova.

```
> statIG <- dfToIG(statGeneal)
> pathCB <- getPath("David Cox", "Petra Buzkova", statIG, statGeneal,
+   isDirected = FALSE)
> plotPath(pathCB, fontFace = 4) + ggplot2::theme(axis.text =
+   ggplot2::element_text(size = 10), axis.title =
+   ggplot2::element_text(size = 10)) + ggplot2::scale_x_continuous(expand
+   = c(.1, .2))
```

This code results in Figure 10. We see that the shortest path between Sir David Cox and Petra Buzkova is strictly composed of five unidirectional “parent-child” relationships that span about 55 years. We see that the time difference between when an advisor and student earned their degrees is not consistent across this path: The three statisticians who earned their degrees earliest in this path span more than 30 years in degree acquisition, whereas the three statisticians who earned their degrees later in this path only span less than ten years in degree acquisition.

We also notice in Figure 10 that Sir David Cox received his statistics degree in about 1950, and Petra Buzkova received her statistics degree in about 2005. This genealogy only contains historical information about obtained degrees, and does not project into the future. Hence, we can be assured that

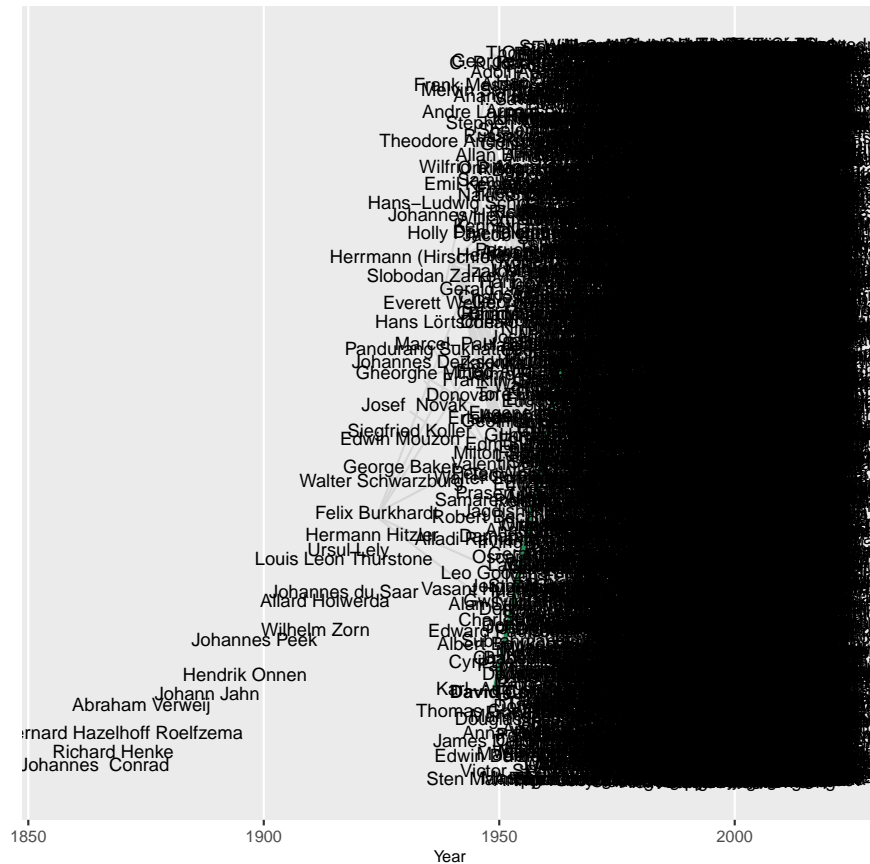


Figure 11: The shortest path between Sir David Cox and Petra Buzkova, superimposed over the data structure, using a bin size of 200.

Petra Buzkova is one of the younger individuals in the dataset, at least in the sense that the youngest individual could only have received his or her degree ten years after Petra Buzkova. However, we cannot be assured that Sir David Cox is one of the oldest individuals in the dataset. As such, it would be informative to superimpose this path of interest onto the entire dataset, using the `plotPathOnAll()` function of the `ggenealogy` package, as we did for the soybean genealogy in Figures 5 and 6.

We can achieve this using the below code. After trial and error, we use a `binVector` of size 200, and append `ggplot2` syntax to define suitable x-axis limits. The output of this process is illustrated in Figure 11.

```
> plotPathOnAll(pathCB, statGeneal, statIG, binVector = 1:200) +
+   ggplot2::theme(axis.text = ggplot2::element_text(size = 8), axis.title =
+   ggplot2::element_text(size = 8)) + ggplot2::scale_x_continuous(expand = c(.1, .2))
```

We see from the resulting Figure 11 that almost all text labels for individuals who received their graduate-level statistics degrees between 1950 and 2015 are undecipherable. We also see that the year Sir David Cox acquired his statistics degree is somewhere in the later half of the variable year for this dataset, as the oldest dates for acquisition of statistics degrees in this dataset occur around 1860.

However, the number of individuals who are documented as receiving their statistics degrees between 1860 and 1950 are few enough so that their text labels are somewhat readable.

The text labels are so numerous in Figure 11 that simply trying different values for the input parameter `binVector` will not solve the text overlapping problem. Instead, one approach we can try is to reconstruct the plot using the same `ggenealogy` function `plotPathOnAll()`, only now specifying variables to render the size (2.5) and color (default of black) of the text for nodes that are on the path of interest to be more noticeable than the size (0.5) and color (dark gray) of the text for nodes that are not on the path of interest. Moreover, we can make the edges that are not on the path of interest to be represented in a less noticeable color (light gray) than the edges that are on the path of interest (default of dark green). The variable names and options for these aesthetics is further detailed in the help manual of the function. We provide one example code that alters the defaults of the text color and sizes of nodes and edges below, which results in Figure 12.

```
> plotPathOnAll(pathCB, statGeneal, statIG, binVector = 1:200, nodeSize = .5,  
+   pathNodeSize = 2.5, nodeCol = "darkgray", edgeCol = "lightgray") +  
+   ggplot2::theme(axis.text = ggplot2::element_text(size = 8), axis.title =  
+   ggplot2::element_text(size = 8)) + ggplot2::scale_x_continuous(expand = c(.1, .2))
```

Conclusions

The `ggenealogy` package offers various plotting tools that can assist those studying genealogical lineages in the data exploration phases. As each plot comes with its advantages and disadvantages, we recommend for users to explore several of the available visualization tools.

This vignette briefly introduced some of the capabilities of the `ggenealogy` package. Inevitably, new approaches will necessitate new features in subsequent versions and might reveal unforeseen bugs. Please send comments, suggestions, questions, and bug reports to `lrutter@iastate.edu`.

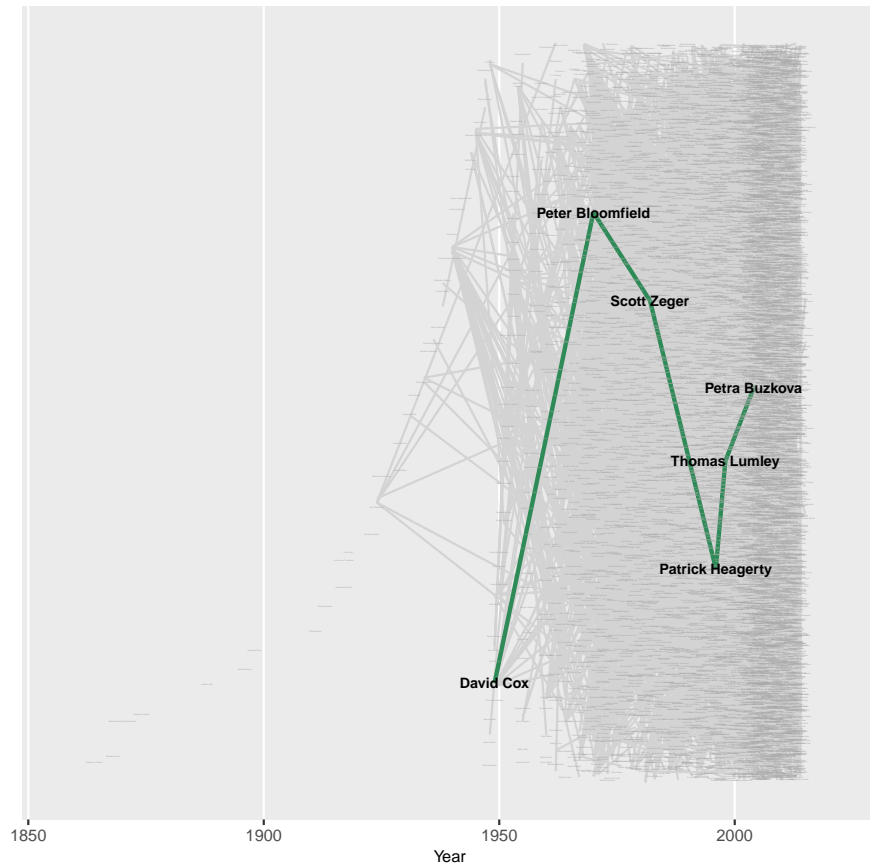


Figure 12: The shortest path between Sir David Cox and Petra Buzkova, superimposed over the data structure, using a bin size of 200. Individuals on the shortest path are labeled in large and black text and connected by dark green edges; all other individuals are labeled in small and gray text and connected by light gray edges.