Modelling Evolutionary Trees
CSC8622

Alex Kell Pedro Silva

November 6, 2016

1 Simulating a Yule Process

Question (i)

Below we present our functions for generating a Yule tree. The algorithm for building the structure of the tree is straightforward:

1. Initialize a vector of size $2n - 2$ with zeros.
2. For each value j in 1 to $n - 1$:
   (a) Find the non-zero elements of the vector whose index (represent extant species).
   Special case for $j = 1$.
   (b) Sample one value from the set of candidates (parent).
   Special case for $j = 1$ where parent is the root with value $2 \times n - 1$.
   (c) Sample up to two positions in the vector with value zero (childs).
   (d) Set the value of childs in the vector as the index of the parent.

In the end of the for loop we have a single numeric vector v containing the structure of tree. Species or edge childs are given by the indexes of v whereas the parent of a species stored at i is obtained by simply accessing $v[i]$. Children of species i are obtained by searching the vector for the element whose value is equal to i. If v does not contain such value then i is an extant species. The root is represented by value $2 \times n - 1$. Although this was a neat and resource-wise cheap way to store and represent the tree, we are also required to store the length of each edge. Therefore we augmented this basic representation of the tree to a matrix that also includes the birth, termination and length of each species. Even though that for a given edge the value of the child is given by the index of the row, we added a column ”Child” to the generated matrix in order to increase readibility (as we are not worried about the space tradeoff).

However, we did not label the nodes according to the example given, i.e. the parent node labelled as $n + 1$, the inner nodes between $n + 2$ and $2xn - 1$ and the leaf nodes between 1 and n. To that extent, we perform on 2 a simple reordering of the labels which does not affect the structure of the tree. Finally we wrapped our method in a function that returns a data frame with further information, namely whether the species (or child) is extant or not.

```r
buildTree = function(n=10, lambda=0.5) {
  nspecies = (2*n - 2)
  cols = c("parent", "child", "birth", "termination", "length")

  tree = matrix(NA, nrow = nspecies, ncol = length(cols))
  colnames(tree) = cols
  tree[,c("parent", "child")] = 0
  t = 0
  for(k in 1:(n-1)) {
    if(k == 1) {
      parent = 2*n - 1
    } else {
      # A candidate edge is an edge of which the child is an extant
      # species. In this case we look for children which do not
      # have an entry in the vector of parents.
      candidates = which( ! (tree[, "child"] %in% tree[, "parent"]))
      # Length of candidates is always > 1 otherwise we would
      #  have to be careful with the behavior of sample
      #  (undesired behavior for length == 1)
      parent = sample(candidates, 1)
      t = t + rexp(1, rate = k*lambda)
    }

    childs = sample(which(tree[,"parent"]==0), 2)
    tree[childs, "child"] = childs
    tree[childs, "parent"] = parent
    tree[childs, "birth"] = t
    if(k > 1) {
      tree[parent, "termination"] = t
    }
  }
  t = t + rexp(1, rate = n*lambda)
  tree[is.na(tree[, "termination"]), "termination"] = t

  tree[, "length"] = tree[, "termination"] - tree[, "birth"]
  return(tree)
}

isExtant = function(tree, index=1:nrow(tree)) {
  ! (tree[index, "child"] %in% tree[, "parent"])
}

loadSpecies = function(path="../aux/species.txt") {
  species = read.table(path, header=FALSE, sep = "+", stringsAsFactors = FALSE)$V1
  species[-which(species=="unavailable")]
}

# Yet Another Yule (YAY)
yay = function(n=10, lambda=0.5) {
  tree = buildTree(n, lambda)
  species = loadSpecies()
  if(length(species) > 0) {
    nomes = species[sample(1:length(species), nrow(tree)+1)]
  } else {
    nomes = paste("poney", 1:(nrow(tree)+1), sep="")
  }

  yule = data.frame(Parent      = tree[, "parent"],
                    ParentName  = nomes[tree[, "parent"]],
                    Child       = tree[, "child"],
                    ChildName   = nomes[tree[, "child"]],
                    isExtant    = isExtant(tree),
                    Birth       = tree[, "birth"],
                    Termination = tree[, "termination"],
                    Length      = tree[, "length"])
  yule[yule$Parent == 2*n-1, ]$ParentName = nomes[2*n-1]
  return(yule)
}

yule = yay()
head(yule, 1)

##   Parent          ParentName Child           ChildName isExtant     Birth
## 1     16 Callipepla gambelii     1 Fregata magnificans    FALSE 0.3011882
##   Termination    Length
## 1    0.982732 0.6815438

yule[, -c(2,4)]

##    Parent Child isExtant     Birth Termination     Length
## 1      16     1    FALSE 0.3011882   0.9827320 0.68154378
## 2      11     2    FALSE 0.4307695   0.6703504 0.23958089
## 3       4     3     TRUE 0.4083406   1.9761293 1.56778866
## 4      16     4    FALSE 0.3011882   0.4083406 0.10715242
## 5      19     5     TRUE 0.0000000   1.9761293 1.97612931
## 6       7     6     TRUE 1.0553830   1.9761293 0.92074626
## 7       2     7    FALSE 0.6703504   1.0553830 0.38503267
## 8      18     8     TRUE 0.6962738   1.9761293 1.27985552
## 9      11     9    FALSE 0.4307695   1.7750047 1.34423520
## 10      1    10     TRUE 0.9827320   1.9761293 0.99339730
## 11      4    11    FALSE 0.4083406   0.4307695 0.02242883
## 12      7    12     TRUE 1.0553830   1.9761293 0.92074626
## 13      9    13     TRUE 1.7750047   1.9761293 0.20112463
## 14      1    14     TRUE 0.9827320   1.9761293 0.99339730
## 15      9    15     TRUE 1.7750047   1.9761293 0.20112463
## 16     19    16    FALSE 0.0000000   0.3011882 0.30118822
## 17     18    17     TRUE 0.6962738   1.9761293 1.27985552
## 18      2    18    FALSE 0.6703504   0.6962738 0.02592341
```

Question (ii)

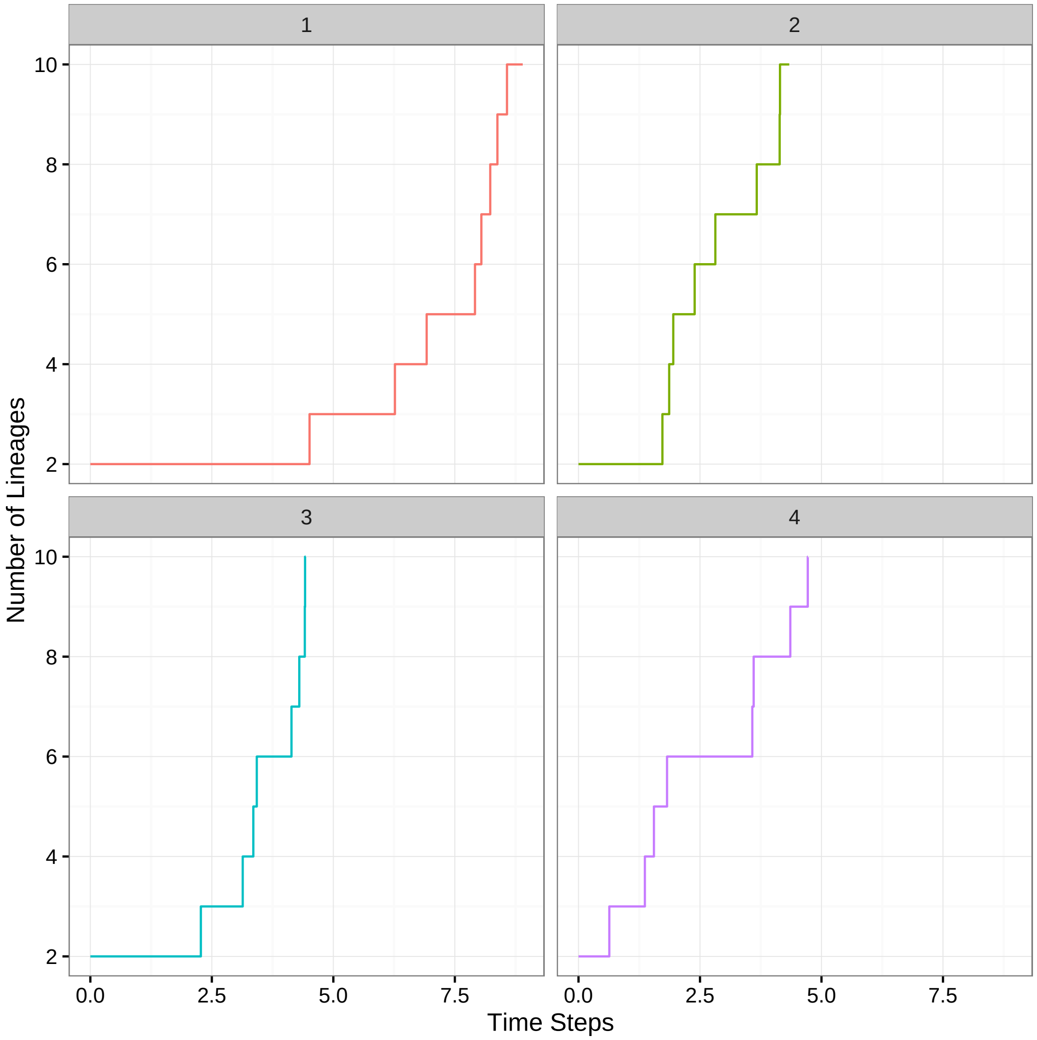
Below is the function for computing the time-step changes in the number of extant lineages:

```
yuleSteps = function(yule) {
  tstep = unique(sort(yule$Birth))
  tstep = c(tstep, max(yule$Termination))
  return(data.frame(tstep=tstep, nlineages=c(2:length(tstep), length(tstep))))
}

yuleSteps(yule)

##        tstep nlineages
## 1  0.0000000         2
## 2  0.3011882         3
## 3  0.4083406         4
## 4  0.4307695         5
## 5  0.6703504         6
## 6  0.6962738         7
## 7  0.9827320         8
## 8  1.0553830         9
## 9  1.7750047        10
## 10 1.9761293        10
```

Question (iii)

Using the function previously defined we created a time series step-plot for four different yule trees. It's worth noting that we coded this task so that more plots could be generated using the same code (only required to change the value of the variable *nPlots*).

```r
n = 10
lambda=0.5
nPlots = 4

yays = lapply(1:nPlots, function(i) yuleSteps(yay(n, lambda)))
yays = rbind.fill(yays)
yays$group = ((as.numeric(rownames(yays)) - 1) %/% n) + 1

ggplot(yays, aes(x = tstep, y = nlineages))   +
  geom_step(aes(colour=factor(group))) +
  facet_wrap(~ group, ncol=nPlots %/% 2) +
  ylab("Number of Lineages") +
  xlab("Time Steps") +
  theme_bw() +
  scale_colour_discrete(guide = FALSE)
```



2 Represeting Trees as phylo Objects

Question (i)

At this stage, we introduced a new function to relabel the edges as required and return a phylo object instead. The relabelling, aparently tricky at first, consisted in a simple re-order of parent and child labels based on whether the edge represented an extant or extinct species.

We created a new function instead of changing our previously defined *yay* function, because each returns a different representation of the yule tree, even though the underlying structure is the same. Consider that we might be interested later in creating our own class for representing phylogentic trees, called *yay*. Then, separating concerns now across different functions without altering behavior would prove beneficial later.

```r
# Yet Another Phylo
yaPhylo = function(n=10, lambda=0.5) {
  yule = yay(n, lambda)

  # Relabelling the nodes
  yule[yule$isExtant==TRUE, ]$Child = 1:n
  yule[yule$isExtant==FALSE, ]$Child = (n+2):(2*n-1)
  yule$Parent = yule$Child[yule$Parent]
  yule[is.na(yule$Parent), ]$Parent = n + 1

  phylo = list(edge = matrix(c(yule$Parent, yule$Child), ncol = 2),
               edge.length = yule$Length,
               tip.label = paste("t", 1:n, sep=""),
               Nnode = n - 1)
  class(phylo) = "phylo"
  return(phylo)
}
```

Question (ii)

The *length* function for the phylo class consists in summing the lengths of all phylo edges:

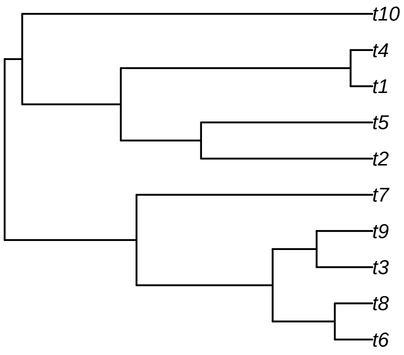
```r
length.phylo = function(phylo) {
  sum(phylo$edge.length)
}
```

Question (iii)

Using the plot method for the phylo class we created following four tree plots:

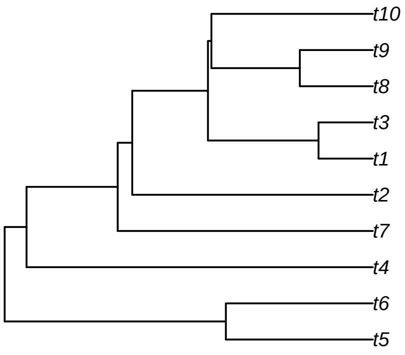
```r
n = 10
lambda =  0.5
par(mfrow=c(2,2))

for (i in 1:4) {
  phylo = yaPhylo(n, lambda)
  plot(phylo)
  title(main=paste("Tree", i),
        sub=paste("Phylogenetic Diversity:", round(length(phylo),2)),
        line = 1)
}
```

Tree 1



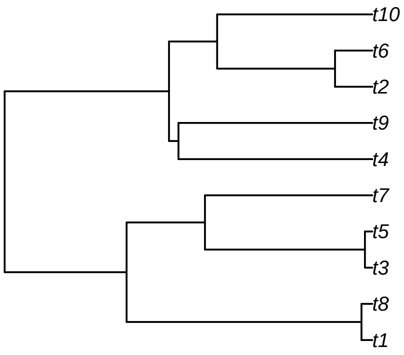
Phylogenetic Diversity: 27.04

Tree 2



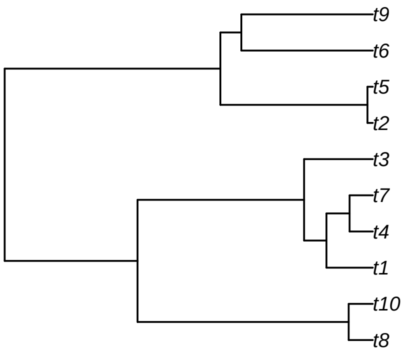
Phylogenetic Diversity: 31.22

Tree 3



Phylogenetic Diversity: 20.41

Tree 4



Phylogenetic Diversity: 18.42

3 Investigating Properties of Trees

Question (i)

The following function is used to calculate how many extant species are descendants of each of the two lineages which emerge from the root.

```
degreeOfBalance = function(tree){
  num_of_leaves = length(tree$tip.label)
  # Returns the number of leaves within a particular tree
  root_node = num_of_leaves+1
  # Finds the number that the root node has been assigned to
  c1 = tree$edge[which(tree$edge==root_node)[1],2]
  c2 = tree$edge[which(tree$edge==root_node)[2],2]
  # Finds the two descendents of the root node

  if((c1<=num_of_leaves && c2<=num_of_leaves+1)  ||
     ((c2<=num_of_leaves && c1<=num_of_leaves+1))){
    # Check to see whether the descendents of the root are both leaf nodes
    # ie. Tree with total of 2 leaf nodes
    ret = c(1,1)
    return(ret)
  }else if(c1<=num_of_leaves || c2<=num_of_leaves){
    # Check to see whether a descendent of root is a leaf node
    desc1 = 1
    desc2 = num_of_leaves-1
    # For this condition to be true,
    # all of the leaf nodes must descend from other root descendent
    ret = c(desc1, desc2)
    return(ret)
  }else{
    sub_tree1 = extract.clade(tree, c1)
    sub_tree2 = extract.clade(tree, c2)
    # Creates two trees where the root is either descendent of the root
    desc1 = length(sub_tree1$tip.label)
    desc2 = length(sub_tree2$tip.label)
    # Calculate number of leaves on each of the newly created trees
    ret = c(desc1,desc2)
    return(ret)
  }
}
```

The following function counts the number of branches there are between each extant species and the root

```
numOfBranches = function(tree){
  num_of_leaves=length(tree$tip.label)
  # Calculates the total number of leaves on the tree
  root = num_of_leaves+1
  # Finds the number that the root node has been assigned to
  iterateBranch = function(tree, up){
    # Function which takes the tree and node number to count
    count = 0
    while(up!=root){
      # Continues to traverse towards the root via parent nodes until
      # the root node has been reached
      up = tree$edge[which(tree$edge[,2]==up)]
      # Returns the parent node of the node we are currently on
      count = count + 1
      # Iterates counter to calculate number of parents nodes we have traveresed through
    }
    return(count)
  }
  store = sapply(1:num_of_leaves, function(i) iterateBranch(tree, i))
  # Calculates number of branches between each species and the root for each leaf
  names(store) = tree$tip.label[1:num_of_leaves]
  # Attributes vector names to the result of number of branches for each species
  return(store)
}
```

The following code calculates the length of each pendant edge

```
lengthOfPendants = function(tree){
  num_of_leaves=length(tree$tip.label)
  # Calculates the number of leaves for the given tree
  penLength = rep(NA, num_of_leaves)
  penLength = tree$edge.length[1:num_of_leaves]
  # Extracts the lengths of the pendant edges
  return(penLength)
}
```

Question (ii)

The following code calculates three properties for a simulated yule tree. This is done using the functions defined in **part (ii)**

```
tree = yaPhylo(n = 16, lambda = 2.5)
# Generates a tree using the user defined function
balance = degreeOfBalance(tree)
# Calculates the number of extants species that are descendents of the two
# lineages emerging from the root
sample(balance, 1)

## [1] 6

# Samples the number of extant species from one of the descendents from the root
extants = numOfBranches(tree)
# Calculates number of branches between extant species and root
sample(extants, 1)

## t10
##   3

# Samples number of branches between itself and root for one of the extant species
penEdge = lengthOfPendants(tree)
# Calculates the length of the pendant for each extant species
sample(penEdge, 1)

## [1] 0.1793468

# Samples one of the extant species and returns its length
```

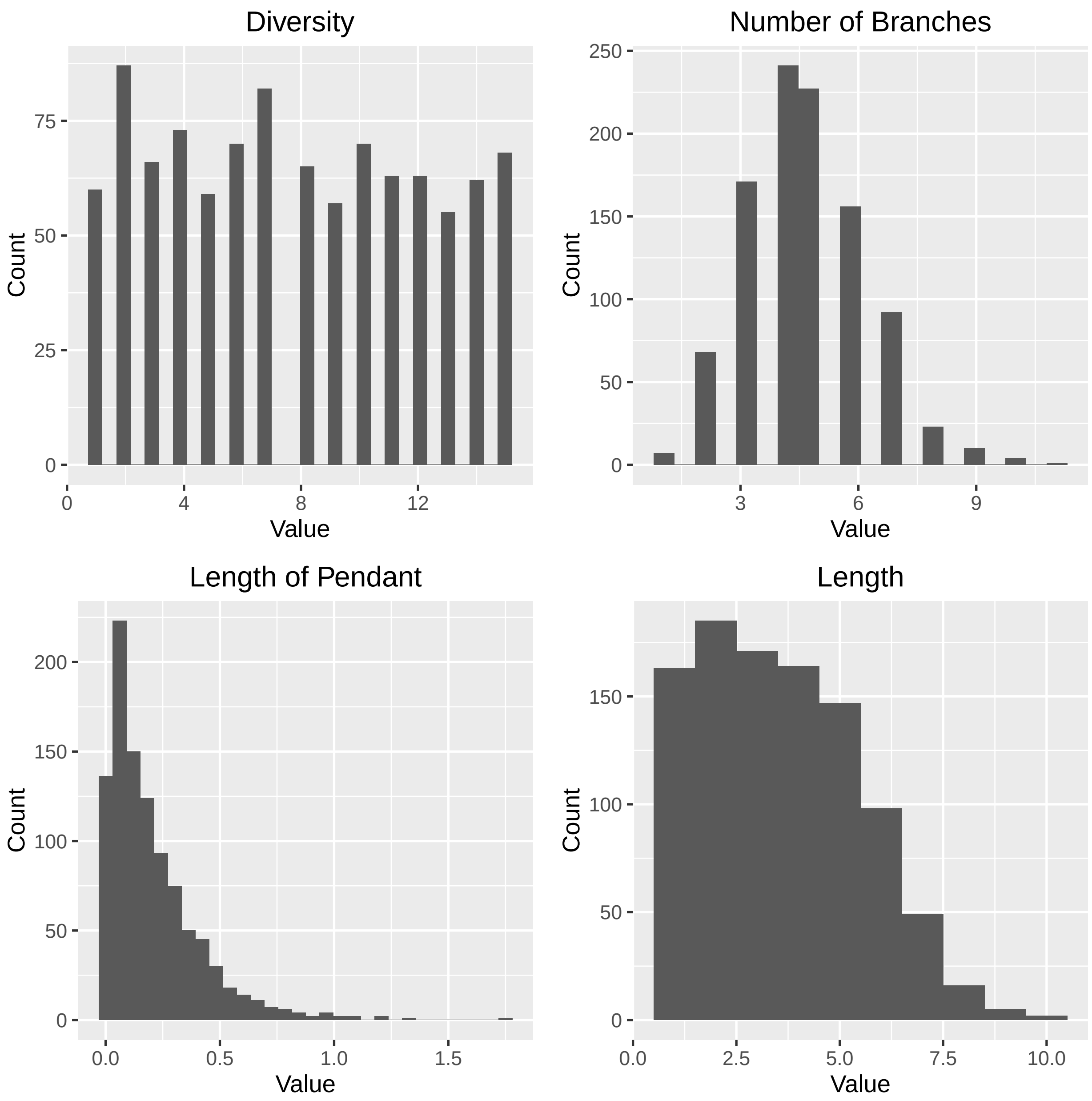
The following function generates a tree, and calculates different properties on each of the trees. Including diversity, number of branches from extant species to root, length of pendant, and length of tree.

```
calc_properties = function(dummy){
  tree = yaPhylo(n = 16, lambda = 2.5)
  # Generates a tree with 16 extant species and a lambda of 2.5
  d1 = sample(degreeOfBalance(tree),1)
  d2 = sample(numOfBranches(tree),1)
  d3 = sample(lengthOfPendants(tree),1)
  d4 = sample(length(tree),1)
  # Sample each of the properties, as described above
  ret = c(d1,d2,d3,d4)
  # Returns the samples for each tree
  return(ret)
}
```

```r
cl = makeCluster(4) # Makes a cluster on all cores of the computer
clusterExport(cl, c("isExtant","loadSpecies","buildTree","yay","yaPhylo",
                    "calc_properties","degreeOfBalance","numOfBranches",
                    "lengthOfPendants","length.phylo","extract.clade"))
# Exports all of the objects which will be required when making the calculations
  n = parSapply(cl, 1:1000, calc_properties)
  # Uses parSapply to run the function 1000 times, returning all values to n
stopCluster(cl) # Stops the cluster after use
```

The following code uses ggplot to plot the properties in histograms.

```r
n=as.data.frame(t(n))
# Converts n to a dataframe to be used by ggplot
p1 = ggplot(n, aes(n[,1])) + geom_histogram(bins = 30) + ggtitle("Diversity") +
  labs(x="Value", y="Count")
# Use of ggplot to plot histograms, with 30 bins
p2 = ggplot(n, aes(n[,2])) + geom_histogram(bins = 20) + ggtitle("Number of Branches")+
  labs(x="Value", y="Count")
p3 = ggplot(n, aes(n[,3])) + geom_histogram(bins = 30) + ggtitle("Length of Pendant") +
  labs(x="Value", y="Count")
p4 = ggplot(n, aes(n[,4])) + geom_histogram(bins = 10) + ggtitle("Length") +
  labs(x="Value", y="Count")
grid.arrange(p1,p2,p3,p4) # Use of gridExtra package to arrange plots in 2 columns and 2 rows
```



The plots show varying distributions for each of the tree properties.

The diversity of the tree seems to be roughly uniform. Whilst there seems to be variation between each value, there does not seem to be a statistically significant value, this remains true with repeated sampling. This is due to the random nature at which nodes give rise to children. With any node having an equally likely change to undergo a branch splitting.

The number of branches between an extant and the root seems to follow a normal distribution, with the mode being 4 branches. There seem to be very little counts where there is just 1 branch between the extant and root, and also where there are over 8 branches. This distribution can be explained by the number of extant species set at the generation of the tree. The algorithm stops at 16 species, and therefore does not leave time for successive branches to be made.

The length of the pedant seems to follow an exponential distribution. This is due to the fact that these timings are generated from an exponential distribution when the tree is generated.

The total length of a tree shows an exponential distribution, with a mode of value at around 2.5.

4 Simulating a DNA Alignment

Question (i)

Below we present our functions for simulating an alignment under the Jukes Cantor model. The first function computes the transition of nucleotides from one parent to a child. It receives as input the parent's DNA sequence along with α and the child's length, returning a new or equal dna sequence for the child.

The alignment matrix is built using the function named $yaJC$. Inside this, we implemented a $lambda-type$ function that traverses the $phylo$ tree and computes all species dna sequences recursevely.

```r
# Function to compute the transition of nucleotides from parents to childs
mutateSequence = function(sequence, alpha, ts){
  p = (1/4) * (1 - exp(-4 * alpha * ts))
  p1 = (1/4) * (1 + 3 * exp(-4 * alpha * ts))
  newSeq = rep(NA, length(sequence))

  for(ip in 1:length(sequence)){
    if(sequence[ip] == 1){
      newSeq[ip] = sample(1:4, size=1, prob = c(p1,p,p,p))
    } else if(sequence[ip] == 2){
      newSeq[ip] = sample(1:4, size=1, prob = c(p,p1,p,p))
    } else if(sequence[ip] == 3){
      newSeq[ip] = sample(1:4, size=1, prob = c(p,p,p1,p))
    } else{
      newSeq[ip] = sample(1:4, size=1, prob = c(p,p,p,p1))
    }
  }
  return(newSeq)
}

# Yet Another Jukes Cantor Model - returns Alignment Matrix
yaJC = function(tree, alpha = 0.5, ncol = 5, ntypes = 4) {
  n = length(tree$tip.label)
  root = n + 1
  rootSequence = sample(ntypes, ncol, replace=TRUE)

  # Allocate the alignment matrix and define root dna sequence
  alignment = matrix(data=NA,
                     nrow = length(tree$edge.length) + 1,
                     ncol = ncol + 2)
  alignment[1, -c(1,2)] = rootSequence
  alignment[,1] = c(root, tree$edge[,2])

  # Function to recurse on the tree
  treeFall = function(phylo, parent, parentSequence, alpha) {
    # Find the childs of parent
    childs = phylo$edge[phylo$edge[,1]==parent , 2]
    # If no child, return
    if(length(childs) == 0) {
      return()
    }

    for (child in childs) {
      # Get the length of this edge
      t = phylo$edge.length[phylo$edge[,2]==child]
      ## Compute child mutated sequence
      childSeq = mutateSequence(parentSequence, alpha, t)
      ## Act on matrix
      alignment[alignment[,1] == child, -c(1,2)] <<- childSeq
      ## Spawn new treeFall
      treeFall(phylo, child, childSeq, alpha)
    }
  }
  treeFall(tree, root, rootSequence, alpha)
  alignment = alignment[as.numeric(alignment[,1]) <= n,]
  # Get labels for second column
  labels = tree$tip.label[tree$edge[,2]]
  alignment[,2] = labels[!is.na(labels)]
  return(alignment)
}

(jc = yaJC(rtree(5)))

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,] "1"  "t3" "4"  "3"  "4"  "3"  "1"
## [2,] "2"  "t2" "3"  "1"  "1"  "2"  "3"
## [3,] "3"  "t1" "3"  "4"  "2"  "3"  "1"
## [4,] "4"  "t5" "2"  "3"  "2"  "2"  "1"
## [5,] "5"  "t4" "2"  "3"  "4"  "1"  "2"
```

Question (ii)

The Hamming Distance between two gene sequences consists simply in the number of nucleotides which are different. Our function receives as input an alignment matrix with n rows and outputs a n by n matrix with the hamming distances between all species' dna sequences in the tree.

```
# Function to compute hamming distance
hammingDistance = function(alignmentMatrix){
  hamDistMat = matrix(data=NA, ncol=nrow(alignmentMatrix), nrow=nrow(alignmentMatrix))
  dimnames(hamDistMat) = list(alignmentMatrix[,2],alignmentMatrix[,2])
  for(i in 1:nrow(alignmentMatrix)){
    for(j in 1:nrow(alignmentMatrix)){
      s = sum(alignmentMatrix[i,-c(1,2)] != alignmentMatrix[j,-c(1,2)])
      hamDistMat[i,j] = s
    }
  }
  diag(hamDistMat) = NA
  return(hamDistMat)
}

hammingDistance(jc)

##    t3 t2 t1 t5 t4
## t3 NA  5  3  3  3
## t2  5 NA  4  4  5
## t1  3  4 NA  3  5
## t5  3  4  3 NA  3
## t4  3  5  5  3 NA
```

Question (iii)

In this question we apply the previously defined functions to the bird.orders dataset. α was set to 0.001 and the number of columns to 10000. After computing the aligment matrix and respective hamming distance matrix, we find the 3 values that maximize and the 3 values that minimize the hamming distance.

```
data("bird.orders")
birdAlignment = yaJC(bird.orders, alpha=0.001, ncol=10000)
birdHam = hammingDistance(birdAlignment)

# Upper and lower triangular parts of hamming distance matrix are equal
# Find the indices of maximum and minimum elements per column
birdHam[!upper.tri(birdHam)] = NA
maxs = order(birdHam, decreasing = TRUE, na.last = NA)[1:3]
mins = order(birdHam, decreasing = FALSE, na.last = NA)[1:3]

top3 = lapply(maxs, function(i) which(birdHam == birdHam[i], arr.ind = TRUE))
bot3 = lapply(mins, function(i) which(birdHam == birdHam[i], arr.ind = TRUE))

names(top3) = sapply(top3, function(i) colnames(birdHam)[i[1,2]])
names(bot3) = sapply(bot3, function(i) colnames(birdHam)[i[1,2]])

# 3 Pairs for which Hamming Distance is maximized
top3

## $Coliiformes
##              row col
## Anseriformes   5  13
##
## $Upupiformes
##              row col
## Anseriformes   5  10
##
## $Galbuliformes
##              row col
## Anseriformes   5   8

# 3 Pairs for which Hamming Distance is minimized
bot3

## $Ciconiiformes
##            row col
## Gruiformes  21  22
##
## $Passeriformes
##               row col
## Columbiformes  20  23
##
## $Gruiformes
##               row col
## Columbiformes  20  21
## Columbiformes  20  22
```

The pairs of species that minimize the Hamming distance are:

- $Upupiformes, Bucerotiformes$
- $Ciconiiformes, Gruiformes$
- $Strigiformes, Musophagiformes$

While the pairs of species that maximize the Hamming distance are:

- *Galbuliformes, Craciformes*
- *Apodiformes, Craciformes*
- *Coraciiformes or Gruiformes, Craciformes*

As expected, the pairs of species with lowest Hamming distance are the ones that speciated from the same parent species and hence diverged from the same gene pool. On the contrary, the pairs of species with largest Hamming distance are further away from the tree and its closest common ancestor is several speciation events apart. Curiously, the species *Craciformes* is present in all pairs that maximize the Hamming distance. In fact, the most recent common ancestor for these pairs is the root of the tree itself. This suggests that either the species *Craciformes*, the opposite branch of the tree or both underwent a set of gene mutations that provoked a considerable shift in the alignment of their gene sequences.