Function Guide R BATS

IJsbrand Pool

2022-10-10

Contents

Introduction

This `function` guide focuses on explaining the functions of this package in detail. This way, users can fully understand what happens in the code and how the statistics are calculated.

Concepts

Recursion

The functions in this package use recursion a lot, and it can be quite a difficult concept to understand. In recursion, a function calls itself. This can be used to walk through an entire tree object and getting or setting its values. Here is a simple example of a recursing function:

```r
recursion <- function(x){
  if(x > 0){
    recursion(x - 1)
    print(x)
  }
}
```

This function needs an integer as input and will print all integers from 1 - the given number. The first time the function is called, it will check if the given input is higher than zero. If so, it will then call itself with the same input lowered by one. If the new integer is again higher than zero, it will call itself again. This will keep looping until the integer has reached zero. When this happens, the last function call will do nothing, but the previous call will print "1". This happens because in that function call the integer is 1. In the function call before that, the integer is "2", so it will print "2". The function call above prints "3", above that prints "4" etc. This idea of recursion can be used in the tree objects to walk through the tree and get or set values.

Nodes

As will be clarified in the function descriptions of the functions that make the tree objects, this package formats the trees in lists. The whole tree is a single list of the top node, with its two daughters in a list. These daughters are lists themselves too. There are two types of nodes, terminal and internal nodes. If a node is a leaf, it is a list with an empty daughters-list, and it is considered a terminal node. If a node has two daughters, it is considered an internal node. All nodes have a set of objects/attributes, see table 1.

Attribute	Type	Description
Distance	Both	The distance of the node given by the tree file
Position	Both	The position, used in the code to keep track of the position in the tree
Daughters	Both	The list of daughters, contains either two or zero daughters
state	Both	The state of the node
IsMono	Internal	A Boolean to mark the node as monophyletic or not
AI	Internal	The association index score of the node
PS	internal	The parsimony score of this node, 1 if its an intersection, 0 if not
Name	Terminal	The name of the leaf

Functions

In depth descriptions of all the functions.

Processing functions

The functions used to process the input files, and create the tree objects.

Process xml

```r
process_xml = function(input_file) {
  output_list <- list(name="List containing all taxons and their attributes", taxons=list())
  all_attributes = c()
  all_names <- c()
  attval <- list()
  attvalout <- list()
  inattr <- FALSE

  if(file.exists(input_file)){
    con = file(input_file, "r")
    on.exit(close(con))

    while(TRUE){
      line = readLines(con, n = 1)
      if(grepl("<taxon id=\"", line, fixed=TRUE)){
        name <- gsub("[[:space:]]", "", paste(strsplit(line, "")[[1]][
          14:(length(strsplit(line,"")[[1]])-2)], collapse=" "))
        all_names <- c(all_names, name)
      } else if(grepl("<attr name=\"", line, fixed=TRUE)){
        attribute <- toupper(gsub("[[:space:]]", "", paste(strsplit(line, "")[[1]][
          16:(length(strsplit(line,"")[[1]])-2)], collapse=" ")))
        all_attributes <- c(all_attributes, attribute)
        inattr <- TRUE
      } else if(inattr){
        value <- gsub("[[:space:]]", "", paste(strsplit(line, "")[[1]][
          5:length(strsplit(line,"")[[1]])], collapse=" "))
        attval[[attribute]] <- value
        inattr <- FALSE
      } else if(grepl("</taxon>", line, fixed=TRUE)){
        attvalout[[name]] <- attval
        attval <- list()
      } else if(grepl("</taxa>", line, fixed=TRUE) | length(line) == 0){
        break
      }
    }
  }
  output_list$names <- all_names
  output_list$attributes <- unique(all_attributes)
  output_list$states <- attvalout
  return(output_list)
}
```

This function processes the input xml file into a useful list containing a vector of all the taxon names, a vector of all unique attributes and a dictionary. This dictionary holds all attributes and their values for every taxon. The function opens the input xml file and starts reading all the lines. If it finds a line declaring a taxon, it will then look for the lines declaring its attributes and their values. These attribute value combinations are put in the dictionary under the name of the taxon, so it will look like this:

```
taxon_name = list(
  attribute1 = value1,
  attribute2 = value2,
  ...
)
```

Shuffle

```
shuffle <- function(xml, state_attribute){
  all_states <- c()
  for(taxon in xml$states){
    all_states <- c(all_states, taxon[[state_attribute]])
  }
  random_states <- sample(all_states)
  for(i in 1:length(xml$states)){
    xml$states[[i]][[state_attribute]] <- random_states[i]
  }
  return(xml)
}
```

This function will shuffle the order of the values of a given attribute. It does so by first getting a vector of all the values in the correct order. This vector is then shuffled. Then these shuffled values are put back into the dictionary in order of the shuffled vector. This shuffled dictionary can then be used for the shuffled trees.

Make Tree

```
make_tree <- function(tree, statesdict, state_attribute) {
  treesplit <- strsplit(tree, "")[[1]]
  thisnode <- list(name="The tree", distance=0, ismono=FALSE, PS=0, daughters=list())

  i <- 1
  while(i<=length(treesplit)){
    char <- treesplit[i]
    if(char=="("){
      if(treesplit[i + 1] == "("){
        newnode <- make_internal_node(treesplit, i + 1, statesdict, state_attribute)
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      } else {
        newnode <- make_terminal_node(i + 1, treesplit, statesdict, state_attribute)
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      }
    } else if(char == ",") {
      if(treesplit[i+1] == "("){
        newnode <- make_internal_node(treesplit, i + 1, statesdict, state_attribute)
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      } else {
        newnode <- make_terminal_node(i + 1, treesplit, statesdict, state_attribute)
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      }
    }
    i <- i + 1
  }
  thisnode <- association_index(thisnode)
  return(thisnode)
}
```

This function makes the top node and makes its daughters by calling other functions to create terminal or internal nodes. When given a tree, it splits this tree so it can walk through all its characters. The first character will always be a “(”, so it then looks if the next character is also a “(”. If a “(” is found, it will create an internal node. If not, it means the daughter is a leaf and it will create a terminal node. After making its daughters, the association index of this top node is calculated. This function is not recursive.

Make Internal Node

```
make_internal_node <- function(treesplit, pos, statesdict, state_attribute) {
  thisnode <- list(daughters=list(), ismono=FALSE, PS=0)
  closeme <- FALSE

  i <- pos + 1
  while(!closeme){
    char <- treesplit[i]
    if(char=="("){
      newnode <- make_internal_node(treesplit, i, statesdict, state_attribute)
      lastnode <- newnode
      thisnode$daughters <- append(thisnode$daughters, list(newnode))
      i <- newnode$position - 1
    } else if(char==",") {
      if(treesplit[i + 1] == "("){
        newnode <- make_internal_node(treesplit, i + 1, statesdict, state_attribute)
        lastnode <- newnode
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      } else {
        newpos <- i + 1
        newnode <- make_terminal_node(newpos, treesplit, statesdict, state_attribute)
        thisnode$daughters <- append(thisnode$daughters, list(newnode))
        i <- newnode$position - 1
      }
    } else if(char == ")") {
      closeme <- TRUE
      thisnode$position <- i
      if(i < length(treesplit) & treesplit[i+1] == ":"){
        i <- i + 1
        distbuffer <- c()
        while(i < length(treesplit)){
          char <- treesplit[i]
          if(char=="," | char == ")"){
            break
          } else {
            distbuffer <- c(distbuffer, char)
          }
          i <- i + 1
        }
        distance <- paste(distbuffer, collapse = " ")
        distance <- gsub("[[:space:]]", "", distance)
        distance <- gsub(":", "", distance)
        thisnode$distance <- as.double(distance)
        thisnode$position <- i
      }
      thisnode <- association_index(thisnode)
      thisnode <- finish_node(thisnode)
      return(thisnode)
    } else {
      newpos <- i
      newnode <- make_terminal_node(newpos, treesplit, statesdict, state_attribute)
      thisnode$daughters <- append(thisnode$daughters, list(newnode))
      i <- newnode$position - 1
    }
    i <- i + 1
    if(i > length(treesplit)){
      break
      closeme <- FALSE
    }
  }
}
```

This function creates internal nodes. It is given a vector of the split tree, a position, the xml dictionary and the state attribute. Like the function to create the top node, this function walks through the vector of the split tree. If it finds a "(" a new internal node should be made. It then uses recursion until it finds an integer or alphabetical character, meaning its first daughter is a leaf. When it does it creates a terminal node of the name and distance. This terminal node has a position that indicates the character of the split tree vector after the distance. This position is then set as the new position in this node and the loop continues. After a terminal node is made, the next character will either be a ")" or a ",". If a "," is found, it will look at the next character which will either be a "(" or an integer/alphabetical character. It will then use recursion to create another internal node or another terminal node respectively. When it eventually finds a ")" it means that the node has two daughters and should be closed. Before closing the node, the set_monoweights function is called to set various attributes for this internal node, and its association index is calculated.

Make Terminal Node

```
make_terminal_node <- function(newpos, treesplit, statesdict, state_attribute) {
  thisnode <- list(name = "empty", position=newpos, daughters=list())

  isdist <- FALSE
  newchar <- treesplit[newpos+1]
  if(newchar == "," | newchar == ")"){
    thisnode$position <- newpos + 1
  } else {
    buffer <- c()
    i <- newpos
    while(i<=length(treesplit)){
      char <- treesplit[i]
      if(char == ":"){
        nodename <- paste(buffer, collapse = " ")
        nodename <- gsub("[[:space:]]", "", nodename)
        nodename <- gsub("\\(", "", nodename)
        thisnode$name <- nodename
        buffer <- c()
        isdist <- TRUE
      } else if(char == ")" | char == ","){
        if(isdist){
          distance <- paste(buffer, collapse = " ")
          distance <- gsub("[[:space:]]", "", distance)
          thisnode$distance <- as.double(distance)
          break
        } else {
          thisnode$name <- buffer
          break
        }
      } else {
        buffer <- c(buffer, char)
      }
      i <- i + 1
    }
    thisnode$position <- i
  }
  thisnode$state <- statesdict$states[[thisnode$name]][[state_attribute]]
  return(thisnode)
}
```

This function creates terminal nodes. It is given a vector of the split tree, a position, the xml dictionary and the state attribute. The position indicates the first character in the split tree vector of the name of this leaf. This funcion also loops through the split tree vector, starting at the first character of the name. It will add the characters it finds to a buffer until it finds a “)”, a “,” or a “:”. In most if not all cases it will find a “:” first, meaning the leaf has a distance. At this point the buffer contains all characters of the name of this node, so they are pasted together and set as the name. After that the buffer is cleared and the function marks that the following characters are the distance of this node. These characters are added to the buffer until a “)” or “,” is found. Then the buffer characters are pasted together and set as the distance. The next character should always be a “)” or a “,”. In both cases this terminal node is done and the loop is stopped. At last the function looks in the xml dictionary for the taxon matching the name of the node, and sets it state to the value of the state attribute of the taxon.

Finish Node

```
finish_node <- function(thisnode){
  statelist <- c()
  dmono <- c()
  for(daughter in thisnode$daughters){
    if(length(daughter$daughters)>0){
      dmono <- c(dmono, daughter$ismono)
    } else {
      statelist <- c(statelist, daughter$state)
    }
  }
  if(length(statelist)==2){
    if(length(unique(statelist))==1){
      thisnode$ismono <- TRUE
      thisnode$state <- unique(statelist)
    } else {
      thisnode$state <- unique(statelist)
      thisnode$ismono <- FALSE
      thisnode$PS <- 1
    }
  } else if(length(statelist)==1 && TRUE %in% dmono){
    daughterstates <- c()
    for(daughter in thisnode$daughters){
      daughterstates <- c(daughterstates, daughter$state)
    }
    if(daughterstates[1] == daughterstates[2]){
      thisnode$ismono <- TRUE
      thisnode$state <- toString(statelist)
    } else {
      thisnode$ismono <- FALSE
      for(daughter in thisnode$daughters){
        if(length(daughter$daughters)>0){
          thisnode$monoweights[[daughter$state]] <- count_leaves(daughter)
        }
      }
      thisnode$state <- unique(daughterstates)
      thisnode$PS <- 1
    }
  } else if(length(dmono)==2){
    if(dmono[1] && dmono[2]){
      daughterstates <- c()
      for(daughter in thisnode$daughters){
        daughterstates <- c(daughterstates, daughter$state)
      }
      if(length(unique(daughterstates)) == 1){
        thisnode$state <- unique(daughterstates)
        thisnode$ismono <- TRUE
      } else {
        thisnode$ismono <- FALSE
        thisnode$state <- unique(daughterstates)
        thisnode$PS <- 1
        for(daughter in thisnode$daughters){
          thisnode$monoweights[[daughter$state]] <- count_leaves(daughter)
        }
      }
    } else if(!dmono[1] && !dmono[2]){
      thisnode$ismono <- FALSE
      daughterstates <- list()
      for(i in 1:length(thisnode$daughters)){
        daughterstates[[i]] <- daughter$state
      }
      if(length(intersect(daughterstates[[1]], daughterstates[[2]])) > 0){
        thisnode$state <- intersect(daughterstates[[1]], daughterstates[[2]])
      } else {
        thisnode$state <- c(daughterstates[[1]], daughterstates[[2]])
        thisnode$PS <- 1
      }
    } else {
      for(daughter in thisnode$daughters){
        if(daughter$ismono){
          thisnode$monoweights[[daughter$state]] <- count_leaves(daughter)
        }
      }
      daughterstates <- list()
      for(i in 1:length(thisnode$daughters)){
        daughterstates[[i]] <- thisnode$daughters[[i]]$state
      }
      if(length(intersect(daughterstates[[1]], daughterstates[[2]])) > 0){
        thisnode$state <- intersect(daughterstates[[1]], daughterstates[[2]])
      } else {
        thisnode$state <- unique(daughterstates)
        thisnode$PS <- 1
      }
    }
  } else {
    internal_states <- c()
    for(daughter in thisnode$daughters){
      if(length(daughter$daughters)>0){
        internal_states <- daughter$state
      }
    }
    if(length(intersect(internal_states, statelist))>0){
      thisnode$state <- intersect(internal_states, statelist)
    } else {
      thisnode$state <- c(internal_states, statelist)
      thisnode$PS <- 1
    }
  }
  return(thisnode)
}
```

This function sets various attributes for an internal node. It receives the internal node it should finish, and it returns the same internal node with the added attributes. This function starts by getting the states of its terminal daughter(s) and/or the monophyletic state(s) of its internal daughters. This will result in two vectors. Depending on these lists, the state, monophyletic state and the PS score of this node are set according to one of these cases:

- If the node has **two terminal** daughters and they have the **same state**, this node will set its own state to that state and mark itself as monophyletic.
- If the node has **two terminal** daughters and they have **differing states**, it will set its state to a vector of the two states and mark itself as non-monophyletic.
- If the node has **two monophyletic** internal daughters, it will check if they are monophyletic for the same state.
  - If so, set this node's state to that state and mark it as monophyletic.
  - If not, set this node's state to a vector of the two daughters' states and mark it as non-monophyletic. Also calculates the monophyletic clade for both states and set this node's PS to 1
- If the node has a **monophyletic and non-monophyletic** daughter, calculate the monophyletic clade for the state of the monophyletic node, mark this node as non-monophyletic and check if the state-list of the non-monophyletic daughter contains the state of the monophyletic daughter.
  - If so, set this node's state to that state
  - If not, set this node's state to a vector of all its daughters states and set its PS to 1
- If the node has **two non-monophyletic** internal daughters, mark this node as non-monophyletic and check if the daughters' state-lists have one or more matching states.
  - If so, set this node's state to that/those state(s)
  - If not, set this node's state to a vector of all its daughters' states and set its PS to 1
- If the node has a **terminal and an monophyletic** internal daughter, it will check if the internal daughter is monophyletic for the terminal daughters' state.
  - If so, set this node's state to that state and mark it as monophyletic.
  - If not, set this node's state to a vector of both states, mark it as non-monophyletic, calculate the monophyletic clade and set its PS to 1.
- If the node has a **terminal and a non-monophyletic** internal daughter, mark this node as non-monophyletic and check if the terminal daughters' state is in the internal daughters state-list.
  - If so, set this node's state to that state
  - If not, set this node's state to a vector of all states

Other functions

These functions are meant to support other functions, so certain things dont have to be calculated or looped through multiple times.

Get Possible States

```
get_possible_states <- function(xmldict, attribute){
  statelist <- c()
  for(taxon in xmldict$states){
    statelist <- c(statelist, taxon[[attribute]])
  }
  return(unique(statelist))
}
```

Uses the xml dictionary to get all the possible states for a given attribute.

Count Leaves

```
count_leaves <- function(node){
  total_leaves <- 0
  if(length(node$daughters)>0){
    for(daughter in node$daughters){
      total_leaves <- total_leaves + count_leaves(daughter)
    }
  } else {
    total_leaves <- 1
  }
  return(total_leaves)
}
```

Counts all leaves/terminal nodes under the given internal node. It uses recursion to loop through all the daughters and adds 1 for every terminal node it finds.

Set State Attribute

```
set_state_attribute <- function(xmldict, userinput){
  warn <- FALSE
  if(!is.null(userinput)){
    state_attribute <- toupper(userinput)
  } else if(length(xmldict$attributes) == 1) {
    state_attribute <- xmldict$attributes
  } else {
    warn <- TRUE
    state_attribute <- smart_pick(xmldict)
  }
  statelist <- get_possible_states(xmldict, state_attribute)
  if(warn == TRUE){
    warning("There are multiple attributes and none was specified to be the state attribute.\"",
            state_attribute, "\" was automatically chosen to be the state attribute.
            If this attribute is not the state attribute, please specify which attribute it is.\n",
            "The found attributes are: \n", paste(xmldict$attributes, collapse=", "))
  }
  if(length(unique(statelist)) == 1){
    warning("All taxons have the same state for this attribute, this means that this
            attribute is not fit to be the state attribute")
  } else if(length(unique(statelist)) == length(xmldict$states)){
    warning("This attribute has only unique states, this means that this attribute is
            not fit to be the state attribute")
  }
  return(state_attribute)
}
```

Sets one of the attributes in the xml file as the state attribute. If the user specified an an attribute to be used as the state attribute, that attribute will be the state attribute. If the user did not specify an attribute and there are multiple attributes in the xml file, it will call the smart_pick function to pick one of the attributes. If no suitable attribute was found, the function then also warns the user about this and gives a list of all attributes. If there is only one attribute in the xml file, that attribute will be used as the state attribute. This function also checks if the given/set state attribute is equal or unique for all taxons, in which case it should not be used as the state attribute.

Smart Pick

```r
smart_pick <- function(xmldict){
  for(attribute in xmldict$attributes){
    states <- c()
    for(taxon in xmldict$states){
      states <- c(states, taxon[[attribute]])
    }
    if(!length(unique(states))==1 &&
       !length(unique(states))==length(states) &&
       !min(table(states))==1){
      return(attribute)
    }
  }
  return(xmldict$attributes[[1]])
}
```

Attempts to pick a suitable attribute as state attribute. By looping through the xml dictionary, this function checks for each attribute if it can be used as a state attribute and returns either that attribute or the first.

Get Daughter States

```
get_daughter_states <- function(node){
  all_states <- c()
  if(length(node$daughters)>0){
    for(daughter in node$daughters){
      all_states <- c(all_states, get_daughter_states(daughter))
    }
  } else {
    all_states <- c(all_states, node$state)
  }
  return(all_states)
}
```

This function creates a vector of all the states of a given internal node's leaves. It uses recursion to loop through all the daughters and keeps track of a vector of states. for every terminal node it finds, it adds it's state to this list.

Print Start

```
print_start <- function(treefile, apetrees, possible_states, state_attribute, reps){
  if(length(apetrees) == 1){
    filename <- strsplit(treefile, "/")[[1]][length(strsplit(treefile, "/")[[1]])]
    cat("\nAnalysing 1 tree from file ", filename, " with ", reps, " shuffled trees.\n")
    cat("Using state attribute \"", state_attribute, "\" with ", length(possible_states),
        " states. ", "(", paste(possible_states, collapse=", "), ")\n")
  } else if(length(apetrees) > 1){
    filename <- strsplit(treefile, "/")[[1]][length(strsplit(treefile, "/")[[1]])]
    cat("\nAnalysing", length(apetrees) , "trees from file ", filename, " with ",
        reps * length(apetrees), " shuffled trees.\n")
    cat("Using state attribute \"", state_attribute, "\" with ", length(possible_states),
        " states. ", "(", paste(possible_states, collapse=", "), ")\n")
  }
}
```

This function prints some details about the tree and the states before after the tree file has been read, before the calculating process starts.

Calculating functions

These functions use the tree object that was generated to calculate nine statistics for every tree.

Calculate All Stats

```r
calculate_all_stats <- function(treelist, utree, possible_states,
                                state_attribute, xmldict, all_stats){
  NTI_and_NRI <- NTI_NRI(utree, possible_states, state_attribute, xmldict)

  all_stats$Total_distance <- c(all_stats$Total_distance, get_total_distance(treelist))
  all_stats$Internal_distance <- c(all_stats$Internal_distance,
                                   get_internal_distance(treelist))
  all_stats$Association_index <- c(all_stats$Association_index, calculate_AI(treelist))
  all_stats$Parsimony_score <- c(all_stats$Parsimony_score, get_parsimony(treelist))
  all_stats$UniFrac_score <- c(all_stats$UniFrac_score, get_UniFrac(treelist))
  all_stats$Nearest_taxa_index <- c(all_stats$Nearest_taxa_index, NTI_and_NRI$NTI)
  all_stats$Net_relatedness_index <- c(all_stats$Net_relatedness_index, NTI_and_NRI$NRI)
  all_stats$Phylogenetic_distance <- c(all_stats$Phylogenetic_distance,
                                       get_phylogenetic_distance(treelist))
  for(state in possible_states){
    stat_name <- paste0("Monophylteic_clade_", state, collapse="")
    all_stats[[stat_name]] <- c(all_stats[[stat_name]],
                                highest_mono(treelist, possible_states)[[state]])
  }
  return(all_stats)
}
```

This function calls other functions to calculate the statistics and adds them to a list. This list contains a vector for each statistic, so all the statistics of the given tree will be added to the vectors of the statistics of all previously calculated trees. It does this for the tree and the xml dictionary it is given. This xml dictionary can be either the normal xml dictionary or a shuffled dictionary, in which cases it the list will be either the list of all normal trees statistics or the list of all shuffled trees statistics respectively.

Get Total Distance

```r
get_total_distance <- function(node){
  totaldist <- 0
  if(length(node$daughters) > 0){
    for(daughter in node$daughters){
      newdist <- get_total_distance(daughter)
      totaldist <- totaldist + newdist
    }
    totaldist <- totaldist + node$distance
  } else {
    totaldist <- node$distance
  }
  return(totaldist)
}
```

This function calculates the total distance of a given tree. It uses recursion to get the distance of all internal and terminal nodes, and adds these together to get the total distance.

Get Internal Distance

```r
get_internal_distance <- function(node){
  totaldist <- 0
  if(length(node$daughters) > 0){
    for(daughter in node$daughters){
      newdist <- get_internal_distance(daughter)
      totaldist <- totaldist + newdist
    }
    totaldist <- totaldist + node$distance
  }
  return(totaldist)
}
```

This function calculates the total internal distance of a given tree. It uses recursion to get the distances of all internal nodes, and adds these together to get the total internal distance.

Association Index

```r
association_index <- function(thisnode){
  daughter_states <- get_daughter_states(thisnode)
  daughter_count <- count_leaves(thisnode)
  frequency <- max(table(daughter_states)) / daughter_count

  AI <- (1-frequency)/(2^(daughter_count - 1))

  thisnode$AI <- AI
  return(thisnode)
}
```

This function calculates the association index of a given node. It first gets all the states of all the leaves, and the amount of leaves under this node. Then it calculates the frequency of the most frequent state. These parameters are then used in this formula:

$frequency = max(daughterStates)/daughterCount$

$AI = (1 - frequency)/(2^{daughterCount-1})$

Calculate AI

```r
calculate_AI <- function(node){
  total_ai <- 0
  if(length(node$daughters)>0){
    for(daughter in node$daughters){
      total_ai <- total_ai + calculate_AI(daughter)
    }
    total_ai <- total_ai + node$AI
  }
  return(total_ai)
}
```

This function calculates the total association index of a given tree. It uses recursion to get the AI score of all internal nodes and adds these together to get the total association index of the tree.

Get Parsimony

```r
get_parsimony <- function(node){
  total_parsimony <- 0
  if(length(node$daughters) > 0){
    for(daughter in node$daughters){
      total_parsimony <- total_parsimony + get_parsimony(daughter)
    }
    total_parsimony <- total_parsimony + node$PS
  }
  return(total_parsimony)
}
```

This function calculates the total parsimony score of a given tree. It uses recursion to get the PS of all internal nodes and adds these together to get the total parsimony score of the tree.

Get UniFrac

```r
get_UniFrac <- function(node){
  totaldist <- 0
  if(length(node$daughters) > 0){
    for(daughter in node$daughters){
      newdist <- get_UniFrac(daughter)
      totaldist <- totaldist + newdist
    }
    if(node$ismono == TRUE){
      totaldist <- totaldist + node$distance
    }
  }
  if(!is.null(node$name)){
    if(node$name == "The tree"){
      totaldist <- totaldist / get_internal_distance(node)
    }
  }
  return(totaldist)
}
```

This function calculates the UniFrac of a given tree. It uses recursion to get the distance of all internal nodes that are marked as monophyletic and adds these together to get the total UniFrac. When it finishes its recursion and ends at the top node, it divides this total UniFrac by the total internal distance of the tree.

NTI and NRI

```
NTI_NRI <- function(utree, possible_states, userinput, xmldict){
  distmatrix <- data.frame(ape::dist.nodes(utree))
  node_names <- utree$tip.label

  distmatrix[utree$Nnode+2 : ncol(distmatrix)] <- list(NULL)
  distmatrix <- distmatrix[0:utree$Nnode+1,]

  colnames(distmatrix) <- node_names
  rownames(distmatrix) <- node_names


  total_NTI <- 0
  total_NRI <- 0
  for(state in possible_states){
    names_by_state <- c()
    for(name in utree$tip.label){
      if(xmldict$states[[name]][[userinput]] == state){
        names_by_state <- c(names_by_state, name)
      }
    }
    statematrix <- distmatrix[names_by_state,names_by_state]
    for(col in statematrix){
      total_NTI <- total_NTI + min(col[col>0])
      total_NRI <- total_NRI + sum(col)
    }
  }
  NTI <- total_NTI / length(possible_states)
  NRI <- total_NRI / length(possible_states) / 2
  return(list(NTI=NTI,NRI=NRI))
}
```

This function calculates the nearest taxon index and the net relatedness index. These statistics both need a state-matrix and are therefore calculated in the same function. First, the ape package is used to generate a distance matrix. Then all rows and columns containing the distances of the internal nodes are removed, and the column and row names are set to the names of the taxons. The distance matrix is split in two state matrices, so each matrix only contains the rows and columns of one state. Using theses matrices, the NTI and NRI are calculated for each state. The NTI is calculated by adding all the minimums of every column together and the NRI is calculated by adding the sum of every column together. Both total scores are then divided by the total amount of unique states. However, the distance matrix is a symmetrical matrix. This does not matter for calculating the NTI, but for the NRI it means that the calculated score is exactly twice as big. Therefore, the total NRI is further divided by two.

Get Phylogenetic Distance

```r
get_phylogenetic_distance <- function(node){
  totaldist <- 0
  if(length(node$daughters) > 0){
    for(daughter in node$daughters){
      newdist <- get_phylogenetic_distance(daughter)
      totaldist <- totaldist + newdist
    }
    totaldist <- totaldist + node$distance
    if(!node$ismono){
      totaldist <- totaldist + node$distance
    }
  } else {
    totaldist <- totaldist + node$distance
  }
  return(totaldist)
}
```

This function calculates the phylogenetic distance of a given tree. It uses recursion to get the distance of all the internal and terminal nodes, but it counts the distances of all monophyletic internal nodes double. It then adds these together to get the total phylogenetic distance.

Highest Mono

```
highest_mono <- function(treelist, possible_states){
  statelist <- list()
  for(state in possible_states){
    statelist[[state]] <- c(1)
  }

  statelist <- topmono(treelist, statelist)

  for(name in names(statelist)){
    statelist[[name]] <- max(statelist[[name]])
  }
  return(statelist)
}
```

This function calculates the monophyletic clade for a given tree. It creates a dictionary of all the states and sets them equal to one at first. Then it calls a function to calculate all monophyletic clade scores for each state. Out of these scores, all but the highest score are removed and the dictionary is returned. This way the dictionary will have the highest monophyletic clade score for each state with a minimum of one.

Top Mono

```
topmono <- function(node, statelist){
  if(length(node$daughters)>0){
    for(daughter in node$daughters){
      statelist <- topmono(daughter, statelist)
      if(!is.null(daughter$monoweights)){
        for(name in names(daughter$monoweights)){
          statelist[[name]] <- c(statelist[[name]], daughter$monoweights[[name]])
        }
      }
    }
  }
  return(statelist)
}
```

This function adds all monophyletic clade scores of all internal nodes to the list of monophyletic clade scores for each state. It uses recursion to loop through all the nodes, and checks if the node has a monophyletic clade score. If it does, it adds this score to the list of the state of the node and continues.

Get Output Dataframe

```r
get_output <- function(all_stats, shuffled_stats){
  output_list <- list()
  all_stats <- lapply(all_stats, sort)
  shuffled_stats <- lapply(shuffled_stats, sort)
  medians <- lapply(all_stats, median)

  output_frame <- t(data.frame(lapply(all_stats, mean)))
  output_frame <- cbind(output_frame, t(data.frame(lapply(all_stats, function(x) {
    return(x[round(length(x)*0.95)])}))))
  output_frame <- cbind(output_frame, t(data.frame(lapply(all_stats, function(x) {
    return(x[max(c(round(length(x)*0.05), 1))])}))))
  output_frame <- cbind(output_frame, t(data.frame(lapply(shuffled_stats, mean))))
  output_frame <- cbind(output_frame, t(data.frame(lapply(shuffled_stats, function(x) {
    return(x[round(length(x)*0.95)])}))))
  output_frame <- cbind(output_frame, t(data.frame(lapply(shuffled_stats, function(x) {
    return(x[max(c(round(length(x)*0.05), 1))])}))))

  for(stat_num in 1:length(names(medians))){
    stat <- names(medians)[[stat_num]]
    count <- 0
    for(value in shuffled_stats[[stat]]){
      if(value<medians[[stat]] && stat_num <= 8){
        count <- count + 1
      } else if(value<medians[[stat]] && stat_num >= 9){
        count <- count + 1
      }
    }
    significance <- 1 - (count / length(shuffled_stats[[stat]]))
    output_list$significance[[stat]] <- significance
  }
  output_frame <- cbind(output_frame, t(data.frame(output_list$significance)))
  colnames(output_frame) <- c("Normal_mean", "Normal_Upper_CI", "Normal_Lower_CI",
                              "Random_mean", "Random_Upper_CI", "Random_Lower_CI",
                              "Significance")
  output_frame <- data.frame(output_frame)
  return(output_frame)
}
```

This function calculates various statistics of the statistics lists. It calculates for each statistic the mean and the upper and lower confidence intervals. It does so for both the normal trees and the trees with shuffled states. All these statistics are put in a dataframe. Of all these statistics, the significance is calculated by calculating the amount of values of the shuffled trees that are higher than the median of the normal trees.

Main function

```r
bats <- function(treefile, xmlfile, reps=1, userinput=NULL){
  apetrees <- treeio::read.beast(treefile)
  xmldict <- process_xml(xmlfile)
  state_attribute <- set_state_attribute(xmldict, userinput)
  possible_states <- get_possible_states(xmldict, state_attribute)
  print_start(treefile, apetrees, possible_states, state_attribute, reps)
  all_normal_stats <- list()
  all_shuffled_stats <- list()
  if(length(apetrees)==1){
    apetrees <- c(apetrees)
  }
  for(tree in apetrees){
    tree <- ape::as.phylo(tree)
    utree <- tree
    tree <- TreeTools::NewickTree(tree)
    treelist <- make_tree(tree, xmldict, state_attribute)
    all_normal_stats <- calculate_all_stats(treelist, utree, possible_states,
                                            state_attribute, xmldict, all_normal_stats)

    for(i in 1:reps){
      shuffled_xmldict <- shuffle(xmldict, state_attribute)
      shuffled_treelist <- make_tree(tree, shuffled_xmldict, state_attribute)
      all_shuffled_stats <- calculate_all_stats(shuffled_treelist, utree, possible_states,
                                                state_attribute, shuffled_xmldict, all_shuffled_stats)
    }
  }
  output <- get_output(all_normal_stats, all_shuffled_stats)
  return(output)
}
```

This function starts the whole process. It requires a path to a .trees or .tre file and the path to the .xml file. The arguments “reps” and “userinput” are optional. “reps” is the amount of shuffled trees that should be made. If the user inputs 1 or leaves this argument blank, 1 shuffled tree will be generated for each normal tree. If the number is higher than one, reps amount of shuffled trees will be generated for each normal tree. “userinput” is a string that indicates the attribute that should be use as the state attribute. Leaving this blank will make the program automatically choose a state attribute.