SIGTREE: AN AUTOMATED META-ANALYTIC APPROACH TO FIND SIGNIFICANT BRANCHES IN A PHYLOGENETIC TREE

by

Todd R. Jones

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Statistics

Approved:

Dr. John R. Stevens
Major Professor

Dr. Christopher D. Corcoran
Committee Member

Dr. Daniel Coster
Committee Member

Dr. Mark R. McLellen
Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah
2012
Abstract

SigTree: An Automated Meta-Analytic Approach to Find Significant Branches in a Phylogenetic Tree

by

Todd R. Jones, Master of Science
Utah State University, 2012

Major Professor: Dr. John R. Stevens
Department: Mathematics and Statistics

An experiment involving two treatment groups compared whole wheat diet and refined wheat diet on different sets of mice. Of interest were the differences by treatment of the levels of hundreds of bacteria in the guts of the mice. It was desired to determine the statistical significance of not only the individual bacteria, but also the families of bacteria. These family relationships are represented in a phylogenetic tree, and it was determined helpful to color the branches of bacteria based on the significance of their corresponding families. Calculating these p-values and coloring the branches by hand would not be a quick process. An automated method would greatly increase the efficiency of these calculations. To handle this problem, SigTree, an R package, was written. The p-values for individual bacteria (tips) are combined up the tree using meta-analysis methods, and significance is visualized on a color scale in a revised phylogenetic tree plot. SigTree is able to handle not only the motivating mouse diet experiment, but also experiments that fall into the general framework of having significance tests (and resulting p-values) on each tip in a phylogenetic tree.
Public Abstract

SigTree: An Automated Meta-Analytic Approach to Find Significant Branches in a Phylogenetic Tree

by

Todd R. Jones, Master of Science
Utah State University, 2012

Major Professor: Dr. John R. Stevens
Department: Mathematics and Statistics

An experiment was performed on two groups of mice, the first group receiving a refined wheat diet, and the second group receiving a whole wheat diet. The levels of abundance of hundreds of bacteria in the guts of the mice were measured. Of interest was whether the levels of these bacteria were different depending on type of diet. In addition, it was desired to examine differences by diet in all of the families of bacteria. These family relationships are represented in a tree, and it was determined helpful to color the branches of this tree based on the differences of their corresponding families. Performing the necessary statistical calculations and coloring the branches by hand would not be a quick process; an automated method would greatly increase the efficiency of these calculations. Therefore, SigTree, a piece of computer software (an R package), was written. It is able to create a customized, colored tree plot based on its calculations of the levels of difference of the individual bacteria, as well as the levels of difference of the families of bacteria. SigTree is able to handle not only the motivating mouse diet example, but also experiments that fall into the appropriate framework.
Acknowledgments

I am very grateful for the time, advice, and help that my advisor, Dr. John Stevens, has provided me.

Todd Jones
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Public Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivating Example</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Design</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Wilcoxon Test</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3 Goal</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Phylogenetic Trees and FigTree</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Tree Structure and Meaning</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Application</td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Software</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3.1 FigTree</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3.2 R</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3.3 TopiaryExplorer</td>
<td>8</td>
</tr>
<tr>
<td>1.3 P-value Combination</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 Stouffer’s Method</td>
<td>10</td>
</tr>
<tr>
<td>1.3.2 Fisher’s Method</td>
<td>12</td>
</tr>
<tr>
<td>1.3.3 Comparison</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Regular Expressions</td>
<td>14</td>
</tr>
<tr>
<td>1.4.1 Application to Phylogenetic Tree Data</td>
<td>15</td>
</tr>
<tr>
<td>1.4.2 Key Ideas and Functions</td>
<td>15</td>
</tr>
<tr>
<td>1.4.2.1 sub, gsub, grep, and grepl</td>
<td>15</td>
</tr>
<tr>
<td>1.4.2.2 Syntax</td>
<td>16</td>
</tr>
<tr>
<td>1.5 Multiple Hypothesis Testing</td>
<td>18</td>
</tr>
<tr>
<td>1.5.1 Per-comparison Error Rate</td>
<td>19</td>
</tr>
<tr>
<td>1.5.2 Family-wise Error Rate</td>
<td>19</td>
</tr>
<tr>
<td>1.5.3 False Discovery Rate</td>
<td>19</td>
</tr>
<tr>
<td>2 Methods: SigTree</td>
<td>21</td>
</tr>
<tr>
<td>2.1 Idea of Packages in R</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Tutorial</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Main Functions</td>
<td>23</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Tabular summary of errors in testing $m$ hypotheses.</td>
<td>19</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Histogram of Shapiro-Wilk p-values to assess normality in the OTU data (587 OTUs).</td>
</tr>
<tr>
<td>1.2</td>
<td>Plot of phylogenetic tree from the motivating experiment.</td>
</tr>
<tr>
<td>1.3</td>
<td>Phylogram plot of an example tree with labeled tips ($t_{1},...,t_{10}$), internal nodes (numbers $11,...,19$), and edges (boxes $1,...,18$).</td>
</tr>
<tr>
<td>1.4</td>
<td>Representations of the sample phylogenetic tree in different graphical forms.</td>
</tr>
<tr>
<td>1.5</td>
<td>Comparisons of Fisher’s and Stouffer’s Methods using different distributions of simulated p-values.</td>
</tr>
<tr>
<td>2.1</td>
<td>Plot of tutorial tree using Stouffer’s Method and BY-adjustment.</td>
</tr>
<tr>
<td>2.2</td>
<td>Plot of tutorial tree using Fisher’s Method and BY-adjustment.</td>
</tr>
<tr>
<td>2.3</td>
<td>Histogram of all of the p-values used in the tutorial example.</td>
</tr>
<tr>
<td>2.4</td>
<td>Screenshot of CSV file produced by <code>export.inherit</code> for the tutorial example.</td>
</tr>
<tr>
<td>2.5</td>
<td>Screenshot of tree opened in FigTree from file exported using <code>export.inherit</code>.</td>
</tr>
<tr>
<td>2.6</td>
<td>Screenshot of tree in FigTree with P-value Annotation box open. Note that the edge leading to tip $t_{7}$ (node 18 in Figs. 2.1 - 2.4) is selected.</td>
</tr>
<tr>
<td>3.1</td>
<td>Index matrix of tutorial example tree.</td>
</tr>
<tr>
<td>3.2</td>
<td>Edge matrix of tutorial example tree.</td>
</tr>
<tr>
<td>4.1</td>
<td>Plot of motivating example tree using Stouffer’s Method and BY-adjustment with 1-sided p-values.</td>
</tr>
<tr>
<td>4.2</td>
<td>Plot of motivating example tree using Fisher’s Method and BY-adjustment with 2-sided p-values.</td>
</tr>
<tr>
<td>4.3</td>
<td>Magnified section of interest of Stouffer’s-BY plot.</td>
</tr>
<tr>
<td>4.4</td>
<td>Magnified section of interest of Fisher’s-BY plot.</td>
</tr>
<tr>
<td>4.5</td>
<td>Portion of interest of Stouffers’s-BY CSV file created using <code>export.inherit</code>.</td>
</tr>
<tr>
<td>4.6</td>
<td>Portion of interest of Fisher’s-BY CSV file created using <code>export.inherit</code>.</td>
</tr>
</tbody>
</table>
Chapter 1

Background

1.1 Motivating Example

A 2011 experiment from the lab of Dr. Michael Lefevre, a Utah State University professor in nutrition, considered the profile of bacteria in murine intestines based on whether they ate a whole wheat diet or a refined wheat diet.

1.1.1 Design

The mice were divided into two treatment groups: whole wheat and refined wheat. There were 10 mice per group. Samples were taken from each mouse and the abundances of 587 strains of bacteria per mouse were measured. Thus, each of the 20 mice had 587 measurements taken. Each bacteria has an operational taxonomic unit (OTU) identifier label to identify it. An OTU is “a defined level which taxonomists use to discuss or compare organisms” [1].

Each mouse’s sample yielded whole integer-valued counts for every OTU. The OTU counts are obtained by taking a sample and counting the number of 16S rRNA sequences, each of which correspond to an OTU. Each sample was not exactly the same size, so the data were normalized, i.e., converted to a percentage of the total count of OTUs. For example (hypothetical and simplified), if the first OTU’s count were 10 and the total sample count were 100, then the first OTU’s value would be 0.1. There were more than 587 bacteria measured, but only 587 are of interest, so the rest are dropped after normalization. Thus, our normalized OTUs values add up to something less than 1 for each mouse.

The OTUs are related to each other in a known way. These OTUs belong to “family” units.
Fig. 1.1: Histogram of Shapiro-Wilk p-values to assess normality in the OTU data (587 OTUs).

1.1.2 Wilcoxon Test

Before we can examine the significance of the families of OTUs, we first need to calculate the significance of the individual OTUs themselves between diet groups. We first used a 2-sample T-test to try and do this. However, it was realized that the data for most of the OTUs were not normally distributed. For example, see Figure 1.1, which is a histogram of the p-values obtained from performing the Shapiro-Wilk test on the one-way ANOVA residuals from each OTU. The null hypothesis for the Shapiro-Wilk test is that the data are normally distributed. Eighty-three percent of the OTUs had a p-value less than .05. This is a problem because the one-way ANOVA model assumes normality.

A better approach is to use the nonparametric Wilcoxon rank-sum test for each OTU via taking the ten samples from the whole wheat diet and comparing their levels to the levels of the ten samples from the refined wheat diet. This is what we did to obtain a normal approximation p-value for each OTU. In order to determine the significance of the families of OTUs, we used a p-value combination method for the OTUs in the family, such as Stouffer’s Method [2] or Fisher’s Method [3]. These methods are explained in greater detail in Sections 1.3.1 and 1.3.2, respectively.
1.1.3 Goal

A major goal in the motivating experiment is to find the levels of statistical significance of diet for the OTUs as well as all of the families of OTUs. There are many different families and it would be an arduous, time-consuming, and possibly error-prone task to do this “by-hand.” Therefore, automating this process using computer algorithms would be very helpful. These algorithms could be written to handle not only the specific application of the motivating example, but also experiments in similar scenarios. I wrote a package, *SigTree* [4], in *R* [5] (explained in detail in Chapter 2) that is able to do this. It allows for easily adjusting various arguments, such as the type of p-value combination method to be used, thus contributing to its flexibility.

In order to be a more general framework, *SigTree* does not itself calculate the p-values of the individual OTUs, but requires them to be calculated already. This is due to the fact that there are countless experimental designs that can be used as well as many potential issues such as non-normality of the data. Thus, it is left to the user to handle the calculation of the p-values for raw, tip-level data in the phylogenetic tree. This allows for greater flexibility and applicability in *SigTree*. This generalizability issue is further discussed in Section 5.1.

1.2 Phylogenetic Trees and FigTree

The bacteria in the motivating experiment are represented in a phylogenetic tree, a graphical representation of the relationships of different entities to each other.

1.2.1 Tree Structure and Meaning

The phylogenetic tree from the motivating experiment is displayed in Fig. 1.2. In this case, the entities are the OTUs, which are shown on the graph as the tips of the tree. Each of the 587 tips is a bacteria species.

The tip is the outermost unit of a phylogenetic tree. It is connected via a branch (edge) to a parent node, which may be connected via another branch to another parent node, and so on until it reaches the root node. I will refer to these parent nodes (including the root) as internal nodes. Nodes include both internal nodes and tips. Each internal node has direct
The tips. The edges may be of different lengths. However, the SigTree package requires that each internal node has only two direct descendants. Each internal node is the ancestor to at least two tips, i.e., each internal node is connected through edges (and possibly internal node(s)) to at least two tips. The root node is the ancestor to all of the tips. The edges may be of different lengths.

For example, in the example tree of Fig. 1.3, node 18 is an internal node and is connected via branches 13 and 14 to tips $t8$ and $t7$, respectively. Node 11 is the root node. Each edge is labeled by a gray box. Each tips starts with a $t$. Each internal node is strictly a number.

Fig. 1.2: Plot of phylogenetic tree from the motivating experiment.
A phylogenetic tree can be represented graphically in a number of different ways. Examples include the phylogram, fan, cladogram, unrooted, and radial plots. The plot in Fig. 1.3 is a phylogram. Meanwhile, Fig. 1.4 shows an example of the same tree (discussed later in Section 2.2) plotted in the fan (Fig. 1.4A), cladogram (Fig. 1.4B), unrooted (Fig. 1.4C), and radial (Fig. 1.4D) styles.

1.2.2 Application

As mentioned in Section 1.1.3, we want to obtain the p-values for all of the families of
OTUs. In general, I define family as any of the nodes, but usually as any of the internal nodes. The members of a given node’s family are all of the tips that are the descendants of this node. For example, in Fig. 1.3, node 14 contains tips $t_6$, $t_2$, and $t_9$ in its family. To calculate the p-value of a family, we use a p-value combination method (see Section 1.3) on all of the tips in that family. For this example, we would perform the p-value combination method on tips $t_7$ and $t_8$ to obtain the p-value for family 18. I use the terms “node” (usually referring to an internal node) and “family” interchangeably. Note that the only member of the family of a given tip is that tip.

After we obtain the p-values for each family, we want to graphically represent the
significance of the nodes (as well as the tips) after adjusting for multiple hypothesis testing (see Section 1.5). The way we do this is by coloring the edges based on significance of the nodes. The color of the two descendant edges coming from a node corresponds to the significance of that node. For the plot in Fig. 1.3, edges 9 and 4 would be colored based on the p-value of family 13. One way to handle the coloring scheme is to color the highly significant nodes a dark shade of a color. Thus, if family 13 had a p-value of .002, we might color edges 9 and 4 dark red.

1.2.3 Software

In order to work with phylogenetic trees and to perform the necessary calculations to determine the significance of the families of p-values, etc., capable computer software is required.

1.2.3.1 FigTree

FigTree [6] is a freely-available software program that displays phylogenetic trees. It has many different options regarding the appearance of the trees, such as edge thickness and type of plot. In addition, this program has the capability of plotting trees as well as making their branches display as different colors. This is useful to accomplish the coloring described in Section 2.3.3 (see also Section 2.3.1).

FigTree is also able to handle annotations of branches. One can store information, such as the p-value of the node to the left (closer to the root) of the given branch, in an annotation. FigTree has support for both Newick [7] and NEXUS [8], two main phylogenetic tree formats. Newick is a way to represent a tree in text format. FigTree is able to load this file and convert it to a plot. NEXUS includes a Newick part in its code, but has additional information. FigTree uses a special kind of NEXUS file to store color information.

1.2.3.2 R

R [5] is also free software. It is a very powerful statistical software program. It also has support for phylogenetic trees. R is able to perform all of the necessary calculations
to determine the desired p-values and to plot and color the phylogenetic tree per the user’s specifications. *FigTree*, on the other hand, is not able to perform these calculations. However, it was desired to be able to view the tree in *FigTree* with the tree edges colored according to the significance levels of the families. We perform the necessary calculations in *R* and subsequently export a file which *FigTree* is able to read. This is accomplished by *SigTree*, the *R* package created for this thesis, discussed in Chapter 2.

1.2.3.3 *TopiaryExplorer*

*TopiaryExplorer* [9] is a recent software program that is capable of handling very large phylogenetic trees that contain over 100,000 tips. It provides for quick and efficient viewing and collapsing of the tree and contains many other innovative features.

It has very good support for tip-level metadata. This metadata includes tip metadata, defined as “metadata associated with tips (e.g., taxonomic assignments)”; a tip abundance matrix, defined as “a matrix mapping tips to their abundance in one or more samples”; and sample/environment data, defined as “a matrix mapping samples/environments to data about those environments” [10]. The main example presented in the literature on *TopiaryExplorer* involves using bacteria samples taken from individuals’ fingertips and keyboards to try and determine which individual typed on which keyboard. In this example, the tip metadata consists of a list of bacteria, denoted by OTU ID, with their corresponding Greengenes taxonomy and RDP taxonomy. The tip abundance matrix contains the sample IDs (fingertip and keyboard) in the columns with the corresponding measured abundances of OTUs in the rows. The sample data contains sample IDs (which are in the abundance matrix) and corresponding information, such as which key this sample was taken from or if it was from a fingertip.

*TopiaryExplorer* also allows for the branches of the tree to be colored according to both tip data and sample data. When colored based on tip data, the tips are colored based on a selected category from the tip metadata, for example, the category of Greengenes taxonomy it falls into. Internal nodes are colored based on how the tips in their families are colored. (Note that in this section, when I talk about a tip being colored, I am referring
to the part of the branch that corresponds to the tip being colored. When I talk about an internal node being colored, I am referring to the parts of the branches that correspond to the internal node being colored.) For example, if all of the tips in a given internal node’s family were yellow, then that internal node would also be yellow. This is categorical coloring and the coloring scheme is customizable, i.e., the user can change which colors correspond to which categories. When collapsed into internal nodes (wedges), the internal nodes can be colored with weighted and/or majority coloring. In the keyboard example, they use weighted coloring to represent the percentage of tips (bacteria) in the wedges that were measured in a certain sample data group, with a darker blue corresponding to a higher percentage.

When coloring by sample data and in the simplest form (by sample ID), the tips are colored based on the sample in which they occur most abundantly. For example, if we colored by sample ID and there were 10 samples, each with its own color, then a given tip would be colored based on which of those 10 samples had the highest value (found in the tip abundance matrix). The coloring scheme here is also customizable and the internal nodes are also based on the coloring of the tips in their family. Note that coloring based on categories of sample data other than sample ID are also possible.

SigTree (see Chapter 2) is built to accomplish a different purpose than TopiaryExplorer. SigTree accepts a very specific type of tip metadata - tip name and tip p-value (obtained from performing a sample comparison test such as a 2-Sample T-Test) and combines p-values for the families and calculates colors corresponding to edges based on the level of significance. While TopiaryExplorer also colors families based on the information in the tips in these families, it does not combine the tip-level data to address significance as in SigTree.

1.3 P-value Combination

A p-value combination method is a way to determine a single p-value from a family of p-values [11]. For example, if we had five p-values from units that were somehow related (in a family), we could determine not only the significance of the five units individually, but
also the significance of all five collectively, i.e., the significance of this family. We do this by combining all of the individual p-values together in a meaningful way. In our application, if we wanted to see if a family of OTUs were significant, we could run a p-value combination method on all of the tips in the family and obtain a family-wide p-value, by which we could determine the significance of the group as a whole. Two main p-value combination methods are Stouffer’s Method [2] and Fisher’s Method [3].

### 1.3.1 Stouffer’s Method

Stouffer’s Method [2] is a consensus test [12,13] in the sense that it looks at the distribution of all of the p-values being combined, noted here as \( \{p_1, ..., p_n\} \). Stouffer’s Method uses the following formula:

\[
Z = \frac{\sum_{i=1}^{n} Z_i}{\sqrt{n}},
\]

where \( Z_i = \Phi^{-1}(p_i) \) and \( \Phi^{-1} \) is the inverse of the cumulative density function for the standard normal distribution. The test statistic \( Z \) follows a standard normal distribution. Therefore, the cumulative distribution function of the standard normal distribution is evaluated at \( Z \) to produce the Stouffer’s Method p-value.

Let us assume that the individual p-values we wish to combine are 1-sided p-values from a test of two treatment groups. For each individual treatment, our null hypothesis is that \( \text{Treatment 1} = \text{Treatment 2} \) (\( T1 = T2 \)) and the alternative hypothesis is that \( T1 < T2 \). Stouffer’s Method works under the assumption that the p-values have a uniform distribution (between 0 and 1) if each of the individual p-values corresponds to a true null hypothesis. Under this null, the mean of the p-values is .5. But if any of the p-values do not fall under the null, then the distribution of p-values will not be uniform. If the distribution is heavier on the left side (closer to 0), this is evidence that \( T1 < T2 \) (the alternative hypothesis). The heavier to the left side the distribution of p-values is, the more evidence we have that \( T1 < T2 \) (and the lower the p-value) for the family.

It is important to know what “side” the p-values are (i.e., what the alternative hypothesis is) so that one can interpret the combined p-value correctly. Stouffer’s Method is
designed for one-sided p-values (with the alternative hypothesis either that \( T1 < T2 \) or \( T1 > T2 \)). It is not, however, designed for two-sided p-values (with the alternative hypothesis that \( T1 \neq T2 \)). This issue of sidedness comes into play later on (in Section 2.3.1.7) when choosing the method of coloring significant branches and tips.

Stouffer’s Method is symmetric in the sense that if one runs Stouffer’s Method on a set of one-sided p-values, \( \{p_1,\ldots,p_n\} \), and gets back a combined p-value, \( \tilde{p} \), and then runs Stouffer’s Method on the set of p-values for the “opposite side” alternative, \( \{1 - p_1,\ldots,1 - p_n\} \), then they will get back the complement \( 1 - \tilde{p} \) [13]. This is a useful fact to know when interpreting results from Stouffer’s Method.

Assume the original p-values are one-sided with the alternative hypothesis that \( T1 < T2 \). Then if Stouffer’s Method for a given set of p-values yields a very small p-value (close to 0), this is strong evidence that \( T1 < T2 \). But if it yields a very large p-value (close to 1), then this is strong evidence that \( T1 > T2 \). If the same data were used to get the p-values, but the p-values were calculated using the alternative hypothesis that \( T1 > T2 \) (and assuming that the p-values were complements of the ones obtained using the alternative that \( T1 < T2 \), for example, obtained by using a 2-sided T-Test, but not a nonparametric Wilcoxon rank-sum test), then the above interpretation would hold, but be reversed - a very small Stouffer’s p-values would give strong evidence that \( T1 > T2 \), while a very large Stouffer’s p-value would give strong evidence that \( T1 < T2 \).

This is illustrated by running Stouffer’s Method on the following set of p-values: .2, .4, .3, and .99. The result is 0.6381. But if we run it on the complements of these values, i.e., .8, .6, .7, and .01, we obtain 0.3619. This is the complement of .6381.

Note that if one of the p-values in the vector of p-values that is passed to the Stouffer’s function in SigTree has value 0, then Stouffer’s function does not work. In SigTree, there is code (in the plot.color - Section 2.3.1, export.inherit - Section 2.3.2, and export.figtree - Section 2.3.3 functions) that converts p-values which are less than a certain value very close to 0 to be a value that is still very close to 0, but a value that R recognizes as non-zero. A similar procedure is done for values that are very close to (but less than) or equal to 1.
This same conversion happens (if necessary) when using Fisher’s Method as well.

The \( R \) implementation in \textit{SigTree} is the same as found in a working paper [14].

1.3.2 Fisher’s Method

Fisher’s Method [3], on the other hand, operates under the null hypothesis that there are no false nulls in the family, with the alternative hypothesis that there is at least one false null [12, 13]. Very small p-values can have a lot of influence in Fisher’s Method. The formula Fisher’s Method uses for a set of p-values, \( \{p_1, \ldots, p_n\} \), is

\[
\chi^2 = -2 \sum_{i=1}^{n} \log(p_i).
\]

Fisher recommended using one-sided p-values where possible [15]. The test statistic \( \chi^2 \) follows a chi-square distribution with \( 2n \) degrees of freedom. One minus the cumulative distribution function of \( \chi^2_{2n} \) evaluated at \( \chi^2 \) yields the Fisher’s p-value.

The complementary sidedness issue discussed in Section 1.3.1 does not apply to Fisher’s Method, i.e., Fisher’s Method is not a symmetric test [13]. This is due to using the chi-square distribution in the calculation of the Fisher’s p-value. If we ran Fisher’s Method on the same p-values (.2, .4, .3, and .99) from Section 1.3.1, then we get 0.4859. However, the result we get from running Fisher’s on the complement of the p-values (.8, .6, .7, and .01) is 0.1805. This is nowhere close to being the complement of 0.4859.

1.3.3 Comparison

Stouffer’s and Fisher’s Methods will often, but not always, agree. The situations where they fail to agree are potentially very meaningful and are related to the distribution of p-values to be combined. This concept is illustrated using simulated p-values (from Beta distributions), as shown in Fig. 1.5. Each of these plots contains 50 p-values (corresponding to a hypothetical family of 50 tips in a phylogenetic tree) following different distributions. Let us again assume that the null hypothesis is that \( T1 = T2 \) and that the alternative hypothesis is that \( T1 < T2 \).
Fig. 1.5: Comparisons of Fisher’s and Stouffer’s Methods using different distributions of simulated p-values.

Fig. 1.5A contains p-values that are roughly uniformly distributed (using Beta(1,1)). We obtain nonsignificant p-values of .574 using Stouffer’s Method and .601 using Fisher’s Method and therefore do not reject the null hypothesis, and conclude that the two treatments are not significantly different. The mean of the individual p-values is roughly .5, so Stouffer’s Method calls this insignificant. There are not enough low p-values for Fisher’s to call this significant.

Fig. 1.5B is skewed right (using Beta(.5, 2)). Both Stouffer’s (p-value of 9.83e-14) and Fisher’s (p-value of 2.8e-12) give very strong evidence that $T1 < T2$. The mean of the
p-values is .236, quite a bit less than .5, so Stouffer’s produces a small p-value. There are quite a lot of small p-values, so Fisher’s also returns a small p-value. If Fig. 1.5B were instead skewed left, then the logic and conclusion here would be the reverse of that in the skewed right case.

Fig. 1.5C is bimodal (using Beta(.5, .5)), with only a slight difference in the heights of the two peaks. Stouffer’s finds no significant consensus evidence of treatment effects (with a p-value near .5). In this case, the two modes would in effect cancel each other out. There are enough small p-values to cause Fisher’s to find significant evidence (with a p-value of 2.1e-06) that $T_1 < T_2$.

The distribution in Fig. 1.5D is also bimodal (using Beta(.5, .4)), but with a clearly higher concentration of p-values on the right side than on the left. Stouffer’s yields a p-value very close to 1, suggesting that $T_1 > T_2$. Fisher’s disagrees with Stouffer’s and gives strong evidence that $T_1 < T_2$ with a p-value very close to 0. There are enough small p-values close to 0 to make the Fisher’s p-value close to 0, despite the peak of p-values near 1. This follows Rice’s observation that “Because the logarithm of the geometric mean P-value is influenced more heavily by smaller P-values, Fisher’s statistic is more sensitive to smaller, as compared to larger, P-values” [12].

1.4 Regular Expressions

A regular expression [16] is a method of matching a string of text. For example, a regular expression could be used to find a telephone number in a text string. There are different functions that utilize regular expressions to match strings. After these functions have matched the strings, they do different things, such as noting the location of the matched strings or replacing the matched string with another string. For example, we could replace the phone number with the same phone number, but with an area code added.

Regular expressions are highly customizable and have advantages over a common string-matching method, the standard Search facility in a word-processing or browser software. For example, regular expressions can be used to find different variations of the same word, while the traditional Search can only find one spelling at a time. Regular expressions can
handle situations where it would be impossible or not feasible to use the standard Search. An example of this is finding an unknown email address or telephone number. An experienced user of regular expressions would have little trouble writing regular expressions code to handle any phone number (as long as it falls within a known syntax). Using the standard Search, one can only search for one phone number at a time. In a document with millions of items, regular expressions can prove to be very valuable in text mining. For example, if one needs to find and save all of the email addresses in a document that is 1000 pages long, it would likely be much quicker to do so using regular expressions than the standard Search.

1.4.1 Application to Phylogenetic Tree Data

I use regular expressions to bridge the gap between R and FigTree. In the export.figtree function (see Section 2.3.3), after performing the calculations based on the tree structure and p-values, I get p-values and edge color information. I need to put this information into a NEXUS file that FigTree can recognize. I first create a NEXUS string in R with the correct information, but not in the correct FigTree NEXUS format. I get it in the correct format by utilizing regular expressions to modify the NEXUS string from R.

1.4.2 Key Ideas and Functions

A brief discussion of regular expressions will follow, for the sake of completeness. In R, commonly used regular expressions functions are sub, gsub, grep, and grepl.

1.4.2.1 sub, gsub, grep, and grepl

sub attempts to match text with a given regular expression pattern. If a match is found, the match is replaced with given text. The main part of the syntax is as follows: sub(pattern, replacement, x). pattern is the pattern to attempt to match. replacement is what to replace pattern with if it is found. x is the text object that will be searched. gsub is similar to sub, with the exception that sub only replaces the first match in the text object, while gsub replaces all of the matches. For example, if our string were “aaa”, our pattern
were “a”, and our replacement were “b”, then sub would match the pattern “a” with the first “a” in “aaa” and replace it with “b”. It would return “baa”. But gsub would return “bbb”. sub would stop after matching the first “a”, but gsub would keep going after this match.

    grep returns the indices of all of the matches. The main syntax is grep(pattern, x), where pattern and x mean the same thing as pattern and x above. For example, if we had a vector defined in R as c(“a”, “b”, “ab”) and we wanted to find the indices of all of the matches for “a”, we could do grepl(“a”, c(“a”, “b”, “ab”). R would then indicate that it had matches in positions 1 and 3, meaning the first and third elements of the vector. grepl is similar to grep, except it returns TRUE values when it finds a match and FALSE if it does not. Continuing the last example, grepl would return a vector with values TRUE, FALSE, and TRUE.

1.4.2.2 Syntax

    Oftentimes, the regular expression pattern will be more complicated than a simple character, such as the “a” in the example above. The pattern can be multiple letters, such as “abc”, or can be things such as any combination of three letters in a row. There are many possibilities. It can also treat a character or groups of characters as optional. For example, we could make an area code in a telephone number optional. This is one of the reasons regular expressions are so useful.

    There is a syntax, or language, to the matching in regular expressions. There are special operators. Some of the most important include +, *, and ?. These deal with how many of the preceding character or group of characters to match. + attempts to match the preceding character or group of characters one or more times, * attempts to match the preceding character or group of characters zero or more times, and ? makes the preceding character or group of characters optional (meaning 0 or 1 in practice).

    For example, if we used sub on “aab” with the replacement “$”, then the patterns of “a+” and “a*” both give us “$b” because they start trying to find the “a” in “aab”, find a match with the first “a”, and then keep going because they both keep trying to find “a”’s
until they cannot anymore. They replace this with "\". But running sub with the pattern of "a?" gives us "\ab" because it finds one match, "a", then stops trying to match anymore (due to the ? stopping after finding 0 or 1 of what precedes it).

But if our text to examine were "baab" (again with the replacement of "\"), then the patterns "a?" and "a*" both return "\ab", but the pattern "a+" returns "b\b". This is because the ? in "a?" means that there is an optional "a". "baab" starts with "b" and not "a", so there are zero "a"s found, which is still a match for the ?. Therefore, the replacement "\" is inserted at the first part of "baab". Because we are using sub (not gsub), it stops once there is a match. The "a*" is similar. But the "a+" does not initially find a match at the "b" because the + needs to find at least one "a" for a match. It keeps going and finds an "a". The +’s rule is 1 or more, so it keeps trying to find "a"s until it can’t anymore. Once it stops finding "a"s, it replaces the match of "aa" with the "\".

Another common technique used in regular expressions is to utilize the square brackets, [ ] and ]. Inside of these, one can put different characters and any of them will work for a match. For example, "[0-9a-z]" will match any number and any lowercase (unless the ignore.case argument is set to TRUE, which makes the function ignore the case of alphabetic characters) letter. Note that the - inside ] and [ makes "[0-9]" include the range from 0 to 9 (all of the numbers) and "[a-z]" include the range from a to z (all of the letters). We can use the +, *, and ? operators after the square brackets, ] and [. For example, gsub("[0-9]?", "\", "123") will return "\\$". "[0-9]?" attempts to match 0 or 1 number. In this case, it matches "1" and replaces it with "\". Because we are using gsub and not sub, it keeps trying to match after the first match and replaces the "2" with a "\". It does the same with the "\".

It is often desired to be able to use captured text in the replacement in sub or gsub. This is done by using the parentheses ( and ). (...) captures whatever is inside of it and saves it for use later. To access it (in replacement), we write \1 for the first instance, \2 for the second, and so on. For example, if we knew that we had a variable called
text containing text that started with an unknown number of numbers that was followed by an unknown number of letters (that had at least one number and one letter) and we wanted to insert a dash in between the numbers and letters, we could use a command such as `gsub("([0-9]+)(([a-z]+)\", text, ignore.case=TRUE). If text were “111AAA”, this command would return “111-AAA”. The number match would be found by the “([0-9]+)” portion of the regular expression. The (...) surrounding it captures the match for use later. The letters are found and captured by “([a-z]+)”. Then the original string is replaced by “\1-\2”, “\1” is the string of numbers. Then comes the inserted “-”. This is followed by the “\2”, which is the string of letters.

1.5 Multiple Hypothesis Testing

In situations like the motivating example (see Section 1.1), many significance tests (corresponding to tips and families in the phylogenetic tree) are of simultaneous interest. In statistical hypothesis testing, if one wants to determine the significance of a single p-value, one simply needs to compare this p-value to an α level. But if there is more than one hypothesis that needs to be tested, then an adjustment to all of the p-values needs to be made in order to have a meaningful type I error rate and control this rate. The functionality to adjust the (possibly dependent) p-values to account for multiple hypothesis testing (across tips and families) is included in `SigTree`

In the single p-value framework, α is the probability of a Type I error, or rejecting the null hypothesis when the null hypothesis is true. In the multiple hypothesis testing framework, however, we likely want to adjust our α (or equivalently adjust our p-values, which is what we do in practice) in order to control either the the family-wise error rate (FWER) or the false discovery rate (FDR) [17] (see Sections 1.5.2 and 1.5.3, respectively).

Table 1.1 [17] makes multiple hypothesis testing easier to explain. V is the number of Type I errors and S is the number of correct decisions where we reject a false null hypothesis. $V + S = R$ (the total number of rejection decisions based on some decision rule, using some (or no) multiple adjustment). m is the total number of tests.
Table 1.1: Tabular summary of errors in testing $m$ hypotheses.

<table>
<thead>
<tr>
<th>Decision</th>
<th>Fail to</th>
<th>Reject</th>
<th>Total:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null True</td>
<td>$U$</td>
<td>$V$</td>
<td>$m_0$</td>
</tr>
<tr>
<td>Null False</td>
<td>$T$</td>
<td>$S$</td>
<td>$m - m_0$</td>
</tr>
<tr>
<td>Total:</td>
<td>$m - R$</td>
<td>$R$</td>
<td>$m$</td>
</tr>
</tbody>
</table>

### 1.5.1 Per-comparison Error Rate

The per-comparison error rate (PCER) [17] is the error rate when there is no adjustment for multiple hypothesis testing. It is defined as $E(V/m)$. Then $E(V/m) \leq \alpha$. For example, if we performed $m=100$ tests (without adjustment) at level $\alpha$, then the expected total number of Type I errors would be less than or equal to 5.

### 1.5.2 Family-wise Error Rate

The family-wise error rate is defined as $P(V \geq 1)$, in other words, the probability that we commit one or more Type I errors. A common FWER-controlling method is Bonferroni. This method rejects a null if its p-value is less than $\frac{\alpha}{m}$, where $m$ is the total number of tests. This is equivalent to rejecting if $m \cdot p_i < \alpha$, where $p_i$ is the $i$th raw (2-sided) p-value. Then $\tilde{p}_i = \min\{m \cdot p_i, 1\}$ can be considered the Bonferroni-adjusted p-value. For example, if $p_i$ were .03 and we had 5 total tests in the family, our adjusted p-value would be $\tilde{p}_i=.15$, which we could compare to an $\alpha$-level. When using Bonferroni, $P(V \geq 1) \leq \alpha$, which is much more desirable than the PCER. A problem with Bonferroni is that if there are many tests (large $m$), then our adjustment is too conservative, meaning the $\frac{\alpha}{m}$ threshold is too severe.

### 1.5.3 False Discovery Rate

The false discovery rate [17] is the expected value of $\frac{V}{R}$. A common FDR-controlling method is called Benjamini-Hochberg (BH) [17]. This approach assumes that all of the p-values are independent of each other. This is a necessary consideration in phylogenetic trees - p-values of tips and parent nodes are clearly not independent, so the BH method
is not ideal. The Benjamini-Yekutieli (BY) method [18] allows for dependence among p-values. BY has a more severe adjustment cost as compared to BH. In other words, BY is less likely to find a given p-value significant than BH. The number of p-values and the shape of the unadjusted p-values’ distribution affect the level of the adjustment severity.
Chapter 2

Methods: SigTree

2.1 Idea of Packages in R

R allows for the creation of packages, a sort of program-within-a-program. I created an R package called SigTree that uses functions to perform the tasks necessary to handle the determination of significance of the families in a tree based on the methods discussed in Chapter 1. It plots the tree in R with edges colored based on the level of significance of their corresponding family. It also exports a NEXUS file that FigTree is able to open that produces a colored tree similar to the one in R. The plot in R has colored tips; the plot in FigTree does not, but does have p-value annotations.

I created numerous internal functions to be used in the background in order for the user functions to work. I define user functions as functions that the user of the package is likely to use. R code for all SigTree functions is included in the Appendix. The remainder of this chapter provides a tutorial on the use and an overview of the structure of this SigTree package.

2.2 Tutorial

In order to better demonstrate the functionality of SigTree, I have created an example tree, r.tree, with a corresponding p-value data frame, r.pvalues. It is the same tree as shown in Fig. 1.3 and Fig. 1.4. I apply the functions described in Section 2.3 on this tree and discuss the results. I also include the code used for these functions.

This tree is a tree object (see Section 2.3.1.1). It has 10 tips and was created using the rtree function from the ape package [19]. This function creates a random tree with a given number of tips. Because I wanted to have a reproducible example, I used the set.seed
function to set the seed to be a known value in the random number generator R uses. The tips in this tutorial tree are labeled “t1”, “t2”,..., “t10”.

In addition to a tree, I created \texttt{r.pvalues}, an \texttt{unsorted.pvalues} object (see Section 2.3.1.2) containing tip labels and p-values. This object is a data frame with tip labels in the first column and p-values in the second column. The tip labels here are the same as the tip labels in the tree, but are in a different order. I used the \texttt{sample} function to randomize the order. For the p-values, I used the \texttt{rbeta} function to get random values from 0 to 1 that follow a Beta-distribution. The shape parameters I used were .1 and .1. I also used \texttt{set.seed} here and when using \texttt{sample}, again for reproducibility.

In this tutorial, we assume that our p-values are 1-sided p-values (see \texttt{side} in Section 2.3.1.4) in a two-sample test. We also assume that the null hypothesis is that \textit{Treatment 1 = Treatment 2 (T1 = T2)} and that the alternative hypothesis is that \textit{T1 < T2}. Therefore, we use a divergent set of colors to contrast low vs. high p-values in the visualized tree (see \texttt{pal} in Section 2.3.1.7 for more details). We use a vector of colors that is similar to the \texttt{RColorBrewer “RdBu”} (Red-Blue) palette [20], with the exception that the middle color is a gray from the \texttt{RColorBrewer “Greys”} palette. The “RdBu” palette gives us dark red for low p-values and dark blue for high p-values, with lighter shades in between. Therefore, dark reds correspond to p-values in support of \textit{T1 < T2} while dark blues are in support of \textit{T1 > T2}.

The code to create the tree, data frame, and vector of colors is as follows:

```r
## Code for random tree and data frame

node.size <- 10
seed <- 109

# Create tree
set.seed(seed);
library(ape)
r.tree <- rtree(node.size)

# Create p-values data frame
```
set.seed(seed)
r.pval <- rbeta(node.size, .1, .1)
# Randomize the order of the tip labels
set.seed(seed)

r.tip.label <- sample(r.tree$tip.label, size=length(r.tree$tip.label))
r.pvalues <- data.frame(label=r.tip.label, pval=r.pval)
# Define new vector of colors based on "RdBu" palette with added gray
library(RColorBrewer)
RdBu1 <- brewer.pal(7, "RdBu")

2.3 Main Functions

The three user functions in SigTree are entitled plot.color, export.inherit, and export.figtree and are explained in detail in Sections 2.3.1, 2.3.2, and 2.3.3, respectively.

2.3.1 plot.color

The first of the user functions in SigTree is called plot.color. It uses the function plot.phylo from the ape package to plot the tree. To this plot, I add color to the tips and edges based on the corresponding p-values (tips according to their own p-value and edges according to the combined p-value of the family or node to their left/node closer to the root) as well as node labels. These node labels can be referenced to the CSV file that is created in the export.inherit function (see Section 2.3.2). There are a number of different arguments that are passed to plot.color. They are tree, unsorted.pvalues, adjust, side, method, p.cutoffs, pal, test, node.label, tip.color, edge.color, tip.label.size, node.label.size, type, and use.edge.length. The default values that these arguments take are shown in the following definition of plot.color (the default values are the values that follow the "=") after each argument, excluding tree and unsorted.pvalues, both of which require objects to be passed to them):
plot.color <- function(tree, unsorted.pvalues, adjust=TRUE, side=1,
method="BY", p.cutoffs=ifelse(rep(side==1, ifelse(side==1, 6, 3)),
c(.01, .05, .1, .9, .95, .99), c(.01, .05, .1)),
pal=ifelse(rep(side==1, ifelse(side==1, 1, length(p.cutoffs1)+1)),
"RdBu", rev(brewer.pal(length(p.cutoffs1)+1,"Reds")),
test="Stouffers", node.label=TRUE, tip.color=TRUE, edge.color=TRUE,
tip.label.size=.1, node.label.size=.1, type="fan",
use.edge.length=TRUE)

These arguments are described in the following sections:

2.3.1.1 tree

*tree* should be an object of class *phylo* (a class of phylogenetic tree from the *ape* package). Files exported from *FigTree* (using File: Export Trees) can be opened in the *phylo* class in *R*. If the exported *FigTree* file is in the Newick format, the *read.tree* function in *R* can be used to create a tree of class *phylo*. For example, if the exported file from *FigTree* is called “newicktree.tre” and ... is the file path of the folder this file is in, the command

```
newick.tree <- read.tree(".../newicktree.tre")
```

creates the *newick.tree* tree, which is of class *phylo*. If the exported *FigTree* file is in the NEXUS format, then the *R* function *read.nexus* creates a tree of class *phylo*. Both functions are from the *ape* [19] package in *R*.

2.3.1.2 unsorted.pvalues

The argument *unsorted.pvalues* should be a data frame (or matrix) that contains the tip labels in the first column and their corresponding p-values in the second column. It is very important that the tip labels in *tree* (accessed via *tree$tip.label*) all correspond exactly to the tip labels in *unsorted.pvalues* (i.e. have the exact same name). They may, however, be in a different order. For example, if *tree* had the tip labels *t1*, *t2*, and *t3*,

...
then $t_1$, $t_2$, and $t_3$ would also need to be in the first three rows of `unsorted.pvalues`, but perhaps in a different order. It will not work if there are more tip labels in either `tree` or `unsorted.pvalues`. “unsorted” in `unsorted.pvalues` simply refers to the fact that the tip labels of `unsorted.pvalues` and the tip labels of `tree` need not be in the same order.

These p-values cannot be less than 0 or greater than 1. If a p-value is 0 or 1 (or essentially 0 or 1), then it gets adjusted to something very close to 0 or 1, respectively, to avoid computational errors in the p-value combination methods. This is discussed in Section 1.3.1.

### 2.3.1.3 adjust

`adjust` is a logical argument (taking values `TRUE` and `FALSE`) that indicates whether a p-value adjustment should take place (`TRUE`) or not (`FALSE`). If `FALSE`, the arguments `side` and `method` (see Sections 2.3.1.4 and 2.3.1.5, respectively) do not affect p-value adjustment because the p-value adjustment doesn’t happen in this case. Because we are performing multiple hypothesis tests simultaneously, our p-values need to be adjusted accordingly (see Section 1.5). They are adjusted using the `p.adjust` function in R with the type of test determined by the `method` argument (see Section 2.3.1.5). `side` also affects how the adjustment is performed (see Section 2.3.1.4).

### 2.3.1.4 side

The argument `side` refers to whether the p-values in `unadjusted.pvalues` are 1-sided or 2-sided. Thus, `side` takes on values 1 and 2. If `side` is 1, then `SigTree` converts these 1-sided p-values to 2-sided, performs the p-value adjustment using `p.adjust`, then converts them back to 1-sided p-values. This is necessary because p-value adjustment methods in `p.adjust` are constructed for 2-sided p-values. If `side` is 2, then the p-values get adjusted without first being converted.

### 2.3.1.5 method

`method` refers to the p-value adjustment method to be utilized (using `p.adjust`). See
Section 1.5. The available methods are “holm” [21], “hochberg” [22], “hommel” [23], “bonferroni”, “BH” (Benjamini and Hochberg; “fdr” is equivalent), and “BY” (Benjamini and Yekutieli).

2.3.1.6 p.cutoffs

The argument p.cutoffs takes as its value a vector of ascending values greater than 0 and less than 1. These values are the desired p-value cutoff levels to define different colors in the visualized phylogenetic tree (see Section 2.3.1.7). For example, if p.cutoffs = c(.01, .05, .1, .9, .95, .99), then the cutoffs would be .01, .05, .1, .9, .95, and .99. These cutoffs are used in the coloring of the tips and edges in our plot. Note that the intervals include the right point, but not the left (except for the left-most interval which includes 0). For example, the range in between .05 and .01 is (.05, 1].

The default value for p.cutoffs is c(.01, .05, .1, .9, .95, .99) if side is 1 and c(.01, .05, .1) if side is 2. This is due to the 1-sided vs. 2-sided p-value issue discussed in Section 2.3.1.7.

2.3.1.7 pal

The next argument is pal. This is a color palette from the RColorBrewer package [20]. It may also be a vector of colors that are valid colors in R. To see available RColorBrewer palettes, enter display.brewer.all(), ?brewer.pal, or brewer.pal.info in R. The palette corresponds to p.cutoffs and how the plot will be colored. For example, if the “RdBu” palette is chosen, and if we use the .01, .05, .1, .9, .95, and .99 p-value cutoffs, then the darkest shade of red will be the color of the edges with p-values in the [0, .01] range. A lighter shade of red will be used for the edges with p-values in the (.01, .05] range, with an even lighter shade for the (.05, .1) range. The “RdBu” palette is a divergent palette, emphasizing differences from the center, ranging from dark red to dark blue. Edges in the (.99, 1] range will have a dark blue color, while those in the (.95, .99] range will be a lighter shade of blue, with an even lighter shade for the (.9, .95] range. The (.1, .9] range has a very light shade, which can be difficult to see. That is why, in the tutorial example (see Section 2.2), we modify
the “RdBu” palette to include an easily visible color (gray) for this range.

`pal` can also contain a vector of RGB colors in a hexadecimal format with a “#” sign in front of it. For example, one color in the vector may be “#CA0020”. The number of colors needs to match the number of coloring regions (determined by the `p.cutoffs` argument). This number is one greater than the number of elements in `p.cutoffs`. Then, if using the aforementioned `p.cutoffs`, we would need 7 colors in our vector of colors.

It is also instructive to discuss the choice of a divergent (e.g., “RdBu”) vs. sequential (e.g., “Reds”) color palette as well as the values for `p.cutoffs`. If the p-values in `unsorted.pvalues` are 1-sided, it would likely make more sense to use a divergent palette. If the null hypothesis for these p-values is that $T_1 = T_2$ and the alternative hypothesis is that $T_1 < T_2$, a p-value of .001 is highly significant and provides strong evidence against the null hypothesis (i.e. evidence that $T_1 < T_2$). But a p-value of .999 provides evidence that $T_1 > T_2$. Thus, we might consider having both low (close to 0) and high (close to 1) cutoff levels, and also a divergent palette so that values close to 0 are a dark color as are values close to 1 (probably a different color). Note that this interpretation is more meaningful when using Stouffer’s Method, which is a symmetric test, than Fisher’s Method, which is not (see Sections 1.3.1 and 1.3.2, respectively).

If our p-values are 2-sided, then it probably makes more sense to only have p-value cutoffs close to 0 and to have the palette be sequential so that the values in the range closest to 0 are the darkest shade and the values farthest away are the lightest shade. The low p-values give evidence for the null hypothesis that the two treatments are equal, while high p-values provide evidence that they are not equal. P-values close to 1 have no other special significance (as they do for the 1-sided p-values discussed above). Remember also that Stouffer’s Method is not intended for 2-sided p-values (see Section 1.3.1). Of course, the selection of these arguments and p-value cutoff levels should make sense in the context of whatever experiment is being analyzed.

Unfortunately, the sequential palettes in `RColorBrewer` start from light and go to dark. This means that the lightest colors correspond to the most significant (i.e., small-
est) p-values. This is solved by setting pal using the rev function in R. For example, 
\texttt{rev(brewer.pal(4, "Reds"))} yields the colors from the “Reds” palette (with 4 colors) in re-
verse so that the darkest shade corresponds to the smallest p-values.

The default value of pal is “RdBu” if side is 1 and the reverse of “Reds” if side is 2. 
This is due to the 1-sided vs. 2-sided p-value issue discussed in Section 2.3.1.7.

2.3.1.8 \textit{test}

The argument test can be either “Stouffers” (Stouffer’s Method) or “Fishers” (Fisher’s 
Method). The selection of this argument determines the p-value combination method that 
will be used. (See Section 1.3).

2.3.1.9 \textit{node.label}

The argument node.label is a logical argument that, when TRUE, enables the internal 
nodes to be labeled. They are labeled starting at the number after the number of tips. If 
there were 587 tips (as in the motivating example), then the node labels start at 588. One 
can view the p-values corresponding to each of these internal nodes (as well as the tips) by 
looking at the CSV file obtained from the export.inherit function (see Section 2.3.2). This 
file also shows which tips belong to which family.

2.3.1.10 \textit{tip.color} and \textit{edge.color}

\textit{tip.color} and \textit{edge.color} are logical arguments. When \textit{tip.color} and \textit{edge.color} are 
TRUE, the tips and the edges, respectively, are colored based on their significance. Other-
wise, they are black.

2.3.1.11 \textit{tip.label.size} and \textit{node.label.size}

\textit{tip.label.size} controls the (cex - a graphical parameter) size of the labels on the tips. 
The default value is 0.1, but 1 may be better for small trees. Similarly, \textit{node.label.size} sets 
the size of the node labels.
2.3.1.12 \textit{type}

The \textit{type} changes the type, or style, of plot of the tree. The default value is “fan”, which produces a circular graph. The other possible options are “phylogram”, “cladogram”, “unrooted”, and “radial”. See Section 1.2.1.

2.3.1.13 \textit{use.edge.length}

The final argument for \textit{plot.color} is \textit{use.edge.length}. When \text{TRUE}, the plot is drawn using the tree’s defined edge lengths. When \text{FALSE}, the edge lengths are ignored and \text{R} makes its own. This may be useful to obtain a more uniformly-spaced plot. This has no effect if there are no edge lengths defined in the tree.

2.3.1.14 Examples

Fig. 2.1 shows the tutorial tree (see Section 2.2) plotted using Stouffer’s Method. The code to produce this plot follows. Because of our choice of the \textit{RdBu1} vector of colors for the \textit{pal} argument (see Section 2.2), all of the nonsignificant edges and tips are colored gray. Tips that are red provide evidence that $T1 < T2$, with darker reds providing more evidence than lighter reds. The interpretation of the blues is the same, except that the blues provide evidence that $T1 > T2$, with higher p-values providing more evidence. For example, the edges coming from family 14 are a dark red. As we can see from the code, we did not set the \textit{p.cutoffs} argument. Therefore, it goes to the default value of \text{c(.01, .05, .1, .9, .95, .99)}. Then family 14 has a p-value that is less than .01 (highly significant). Note that we applied p-value adjustment to the p-values. This defaults to Benjamini-Yekutiel adjustment. We also default to 1-sided p-values.

\begin{verbatim}
plot.color(r.tree, r.pvalues, node.label.size=1, tip.label.size=1,
edge.color=TRUE, tip.color=TRUE, test="Stouffers", pal=RdBu1,
adjust=TRUE)
\end{verbatim}

Fig. 2.2 shows the tutorial tree plotted using all of the same setting used in the previous plot with the exception that we now use Fisher’s Method. The tree looks largely the same,
but there are some differences. Family 16 is not significant anymore. Family 17 has become significant in this new plot. In addition, family 11 (which is the root) and family 12 have also become significant. Stouffer’s and Fisher’s do not agree in family 11 and 12 because we are encountering a bimodal p-value situation as described in Section 1.3.3 (see Fig. 2.3 for a histogram of all of the p-values of \textit{r.pvalues} (i.e., all of the members (tips) of family 11); note that these p-values are unadjusted at this point). The interpretation of the blue tips and edges is not the same due to the nonsymmetry of Fisher’s Method. The code is as follows:
plot.color(r.tree, r.pvalues, node.label.size=1, tip.label.size=1,
    edge.color=TRUE, tip.color=TRUE, test="Fishers", pal=RdBu1,
    adjust=TRUE)

Fig. 2.2: Plot of tutorial tree using Fisher’s Method and BY-adjustment.

2.3.2 export.inherit

export.inherit is the second of the three user functions in SigTree. It produces a CSV file (that can be opened by a program such as Microsoft Excel) that contains the p-values for each node as well as the tips that belong to the family of each node. Excluding the
column labels, there are N rows, where N is the total number of nodes, i.e., number of tips+number of internal nodes. After the row of column labels, the first n.tips (number of tips) rows have their tip labels in the first column. After the tips, the internal nodes come. These are labeled with node numbers, starting at n.tips+1 (root). These numbers are the labels \( R \) gives the internal nodes and can be referenced to the node labels in the plot produced by \( plot.color \). The second column is the p-value corresponding to the tip or internal node. If a p-value adjustment has been performed, the p-values for the tip will be different than the original p-values from \( unsorted.pvalues \). Otherwise, they will be the same. The p-value for each internal node is the family-wide p-value (i.e., obtained by performing Stouffer’s or Fisher’s Method on all of the p-values of the tips that are in its family and adjusted if \( adjust \) is \textit{TRUE}).

For a given row, columns 3 through column n.tips+2 contain either a tip label or are left blank. They represent the tips that are in the family of a given node. Note that the “X” in the column header in these columns is meaningless. The number following the “X” is a number from 1 to n.tips. For a given row/family, to find out how many tips are in this family, one would look at the number in the column header of the last column containing a tip label. Of course, the only tip in a given tip’s family is itself. Therefore, each tip has
no other tip in any of the columns after column 3. The column header for column 3 is “X1”, meaning that this family has only one member. But all of the internal nodes will have at least two columns (from column 3 through column n.tips+2) filled. The root node (the n.tips+1 row of tips) contains all of the tips in its family and thus will have all of its columns filled.

`export.inherit` takes `tree`, `unsorted.pvalues`, `adjust`, `side`, `method`, `file`, and `test` as arguments. The default values that these arguments take are shown in the following definition of `export.inherit`:

```r
export.inherit <- function(tree, unsorted.pvalues, adjust=TRUE, side=1, method="BY", file="", test="Stouffers")
```

All arguments except `file` have been discussed in Section 2.3.1.

### 2.3.2.1 file

`file` is a file path for R to save the CSV file to. If left blank, the default value is "". This will simply print the CSV output to the screen.

### 2.3.2.2 Example

Fig. 2.4 shows a screenshot of the CSV file produced by `export.inherit` using the tutorial tree and the data frame of p-values. It uses Stouffer’s Method and default arguments (other than `file`). Column A lists the tips, then the internal nodes. Column B has their corresponding p-values. Columns C-L have all of the tips that are members of each node’s family. (Note that the only member in each tip’s family is the tip itself (see Section 1.2.2).) For example, family 14 has a p-value that is essentially 0 (hence its dark red color in Fig. 2.1) and has tips `t9`, `t2`, and `t6` in its family. The final column containing a tip label for this row is column E, which has column header “X3”, meaning that this family has three tips in it. Node 11 is the root. The code to export to the file “ExportInherit1.csv” is as follows:

```r
export.inherit(r.tree, r.pvalues, test="Stouffers",
               file="ExportInherit1.csv")
```
Fig. 2.4: Screenshot of CSV file produced by `export.inherit` for the tutorial example.

2.3.3 `export.figtree`

`export.figtree` is the third and final user function of `SigTree`. It provides an interface between R and FigTree and makes use of regular expressions (see Section 1.4). Through this function, R calculates combined p-values and color assignments and exports a NEXUS file that can then be opened in FigTree. Upon opening the file in FigTree, the tree is plotted with colored branches (but not colored tips as in the plot produced by `plot.color` in R). In addition, the p-values for the nodes are accessible via annotations (provided this option has been selected in `export.figtree` - see Section 2.3.3.2). Note that the plot produced in FigTree by opening the exported NEXUS file includes an extra parent branch that is coming out of the root node. This is colored based on the root node's p-value. Thus, it has the same color and p-value annotation as the two children branches coming out of the root node.

The arguments for `export.figtree` are `tree`, `unsorted.pvalues`, `adjust`, `side`, `method`, `p.cutoffs`, `file`, `pal`, `test`, and `edge.label`. The default values that these arguments take are shown in the following definition of `export.figtree`:

```r
export.figtree <- function(tree, unsorted.pvalues, adjust=TRUE, side=1,
                           method, p.cutoffs, file, pal, test, edge.label)
```

```csharp
Fig. 2.4: Screenshot of CSV file produced by `export.inherit` for the tutorial example.

2.3.3 `export.figtree`

`export.figtree` is the third and final user function of `SigTree`. It provides an interface between R and FigTree and makes use of regular expressions (see Section 1.4). Through this function, R calculates combined p-values and color assignments and exports a NEXUS file that can then be opened in FigTree. Upon opening the file in FigTree, the tree is plotted with colored branches (but not colored tips as in the plot produced by `plot.color` in R). In addition, the p-values for the nodes are accessible via annotations (provided this option has been selected in `export.figtree` - see Section 2.3.3.2). Note that the plot produced in FigTree by opening the exported NEXUS file includes an extra parent branch that is coming out of the root node. This is colored based on the root node's p-value. Thus, it has the same color and p-value annotation as the two children branches coming out of the root node.

The arguments for `export.figtree` are `tree`, `unsorted.pvalues`, `adjust`, `side`, `method`, `p.cutoffs`, `file`, `pal`, `test`, and `edge.label`. The default values that these arguments take are shown in the following definition of `export.figtree`:

```r
export.figtree <- function(tree, unsorted.pvalues, adjust=TRUE, side=1,
                           method, p.cutoffs, file, pal, test, edge.label)
```

method="BY", p.cutoffs=ifelse(rep(side==1, ifelse(side==1, 6, 3)),
c(.01, .05, .1, .9, .95, .99), c(.01, .05, .1)), file="",
pal=ifelse(rep(side1==1, ifelse(side1==1, 1, length(p.cutoffs1)+1)),
"RdBu", rev(brewer.pal(length(p.cutoffs1)+1,"Reds"))),
test = "Stouffers", edge.label=TRUE, ignore.edge.length=FALSE)

All but edge.label and ignore.edge.length have been previously discussed in Sections 2.3.1 and 2.3.2. However, pal needs additional explanation for FigTree compatability.

2.3.3.1 pal

The argument of pal can either be a valid RColorBrewer palette or a vector of hexadecimal colors as discussed previously in Section 2.3.1.7. Note that the “#” at the beginning of the color is now optional. Formats of colors other than hexadecimal will likely give unwanted results in the edges of the tree produced in FigTree, such as all-black edges or the edges being colored in a meaningless way. This is because the color conversion assumes hexadecimal colors (see Section 3.2.3). The default value of pal is the same as in Section 2.3.1.7 - “RdBu” if side is 1 and the reverse of “Reds” if side is 2.

2.3.3.2 edge.label

edge.label is a logical argument that allows the p-values of the internal nodes to be available in FigTree. These node p-values are accessed via highlighting a branch and clicking on the Annotate button (with a picture or a paper clip above Annotate) near the top of the window and then changing the value of Annotation to P-value. The p-value for the node to the left of this branch, i.e., the node closer to the root, shows up in the Value: field.

2.3.3.3 ignore.edge.length

ignore.edge.length is similar to use.edge.length in Section 2.3.1.13, but has the opposite meaning. When TRUE, the edge lengths of tree are ignored and FigTree makes its own edge
lengths. When \texttt{FALSE}, the original edge lengths are used. This may be useful to obtain a more uniformly-spaced plot. This has no effect if there are no edge lengths to begin with.

2.3.3.4 Example

Fig. 2.5 is a screenshot of the Stouffer-BY tree using \texttt{r.tree} and \texttt{r.pvalues} (as in Fig. 2.1) opened in \textit{FigTree}. Other than the file name, all of the arguments are the default \texttt{export.figtree} arguments. Tip font and line weight options in \textit{FigTree} have been changed to improve visibility in this thesis.

Fig. 2.6 is another screenshot of the same tree that highlights the p-value annotation. The highlighted edge corresponds to family 18. Its p-value is 0.9782 (as in Fig. 2.4).
Fig. 2.6: Screenshot of tree in *FigTree* with *P-value* Annotation box open. Note that the edge leading to tip *t7* (node 18 in Figs. 2.1 - 2.4) is selected.

The code to create a NEXUS file called *ExportFigtree.tre* that *FigTree* can open is as follows:

```r
graph figure
element edge length 0.6

element annotation
  label "P-value"
  value 0.978171051940522

export.figtree(r.tree, r.pvalues, test="Stouffers", file="ExportFigtree.tre",
               ignore.edge.length=TRUE, pal=RdBu1)
```
Chapter 3

Methods: Noteworthy Challenges and Solutions

Creating the *SigTree* package took a lot of time and thought. Some of the challenges that came up during this process are discussed in the following sections.

3.1 Index Matrix

In order to make my package work, I needed to create a so-called *index* matrix. I created an internal function, *index.matrix*, to do so. *index* is used to indicate which tips belong to which family and is used in the creation of the CSV file in the *export.inherit* function. It is also used in the combination calculation of the p-values which are used to determine the coloring of the branches. For a given internal node, it is necessary to know which tips belong to this node’s family so that these tips’ p-values can be used in this calculation. It was decided that I need to create an *index* matrix. The columns of *index* correspond to all of the nodes, with the tips first, followed by the internal nodes. The rows correspond to all of the tips. Note that the meanings of the rows and columns are essentially reversed from that of *export.inherit* (see Section 2.3.2). The idea is that, for a given node (column), all of the tips (rows) in this node’s family will have the value of 1, while those not belonging to this family will have a value of 0. For example, if node 10 had tips 1 and 2 in its family, column 10 would contain 0’s everywhere except for rows 1 and 2, which would contain 1’s.

An example of an index matrix is shown in Figure 3.1. This is the index matrix from the tutorial tree, introduced in Section 2.2. The tips (rows) are not labeled with their tip labels. Here is how they correspond: row 1=tip *t10*, 2=*t9*, 3=*t2*, 4=*t6*, 5=*t3*, 6=*t8*, 7=*t7*, 8=*t1*, 9=*t5*, and 10=*t4*. The tip labels appear in the order that *R* internally orders them. Family (column) 11 is the root node and contains all of the tips. Family 14 has tips 2 (*t9*),
3 (t2), and 4 (t6) in it.

How to fill this matrix in correctly and efficiently was a challenge. I wrote code that could correctly fill it in, but it was prohibitively slow. A great deal of background is necessary to explain the process and issues that relate to the index matrix.

The idea was to use the edge matrix to fill the index matrix in. The edge matrix is a matrix that holds information about the tree’s structure - how all of the nodes are connected to each other. Each phylogenetic tree in R of class phylo has an edge matrix associated with it. The row number represents an edge number. Each edge connects two nodes (both internal nodes and tips) together. For example, Figure 3.2 is the edge matrix for the tutorial tree. Edge 13 connects internal node 18 with tip 6 (t8) Each node has a number from 1 to N, where N is the total number of nodes (number of tips + number of internal nodes). The tips are numbered from 1 to T, where T is the number of tips. The internal nodes are numbered T+1 to N. The root node is numbered T+1.

Each internal node follows the rule that its number is smaller than all of its descendants (excluding tips). In the edge matrix, for a given edge, the first column is the node closer to the root and the second column is the node closer to the tips. The tips will always be in the second column, and if, for a given edge, the second column is an internal node, then
the first column will always have a smaller number. All of the above statements about node numbering and the arrangement of the edge matrix are assumed to be true when using this package. This appears to be the case when a phylo tree object is created using the read.tree and read.nexus functions (see Section 2.3.1.1).

In constructing the index matrix and partially filling it in, I appended a matrix full of 0’s with S columns and T rows to an identity matrix of size T, where S is the number of internal nodes and T is the number of tips. The result is a matrix with T rows and N columns, where N is the total number of nodes. The first T columns are complete because the tips contain only themselves in their family. Filling in the remaining S columns is the problem.

I came up with a very inefficient method that utilized a sorted edge matrix (this sorting was unnecessary to do) to fill in the index matrix. For a given edge connecting nodes i (column 1) to j (column 2), the algorithm determined whether each row of j had a 1 value, and if so, assigned 1 to that row of column i. It cycled through all of the rows and also all of the edges. On top of that, I did all of this an arbitrary number of times (e.g. 100) to attempt to completely fill in this matrix. If this number was not high enough, it would not have been completely filled in. (Note that the logic behind this method is roughly similar to that explained in the following paragraph.) This method worked, but was slow and could definitely be improved upon. This method used three for-loops and an if-statement.
A better method (the method I used for SigTree) involved using only one for-loop. I utilized the above assumption that, excluding tips, an ancestor node always has a lower node number than its descendants. I sorted the edge matrix in descending order by the first column. Then I used the for-loop to cycle through the sorted edge matrix. For each edge (connecting nodes i (column 1) to j (column 2)), column j (of the index matrix) is added to column i. The logic is to use the edge matrix to see which nodes are connected, and then to add the tips (1’s) in the descendant node to the tips in the parent node. For example, the first iteration will add the column in the index matrix of some tip to the column of its parent. Because the tip has one 1 in it, this 1 will end up in the parent’s column, which will now have one 1 in it. The 0’s are also added, but don’t change anything. Later, this same parent column will get another 1 in it from its other child tip. Later still, this parent’s two 1’s will be added to it’s parent and so on. The last iteration will involve the root. Then the index matrix will be complete.

3.2 Exporting Tree Structure with Color from R to FigTree

A main goal of this project was to be able to obtain a colored plot in FigTree based on the p-value and other calculations performed in R. Getting this to work proved to be far from trivial. I needed to figure out how FigTree exported color data with the idea that FigTree could read a file it exported, so if I could figure out, in R, how to mimic the exported FigTree file, I could export this from R and read it into FigTree to get a tree like the one that had originally been exported from FigTree. I also needed to figure out how to add p-value annotations that FigTree could recognize as well as deal with phylogenetic trees with no edge lengths defined.

3.2.1 Creating NEXUS Files with Annotation Data and Coloring Trees in R

FigTree can read (and subsequently plot) the Newick and NEXUS formats of phylogenetic trees. NEXUS contains a Newick component along with additional information. The Newick component defines the structure of the tree. In addition, FigTree can export both Newick and NEXUS, but can only include annotations when exporting NEXUS. Annota-
tions are important for us because the information pertaining to the tree's coloring scheme is included in the annotations. Annotations are also where we can store the p-value information. It seemed like NEXUS was the format for us to use in order to get the branches colored in the way we needed.

A major problem was that I did not know how to add the annotations for the colors in a NEXUS file. While R is able to export NEXUS files, it was not obvious to me how to first of all even include annotations, but also to include their annotations in the way I needed. I found possible solutions to the problem of how to add annotations, but none of these proved successful. I corresponded (through a mailing list and email) with several people with expertise in this area in hopes of solving this problem.

Initially, I was unsure as to whether phylogenetic trees could be colored in R, but I found that they could be (one person suggested coloring the tree in R - this may have been when I realized you could do this). plot.phylo (from the ape package in R) plots phylogenetic trees and colors them based on their edges (and it can also color the tip labels). I had written my code to give me the p-values of all of the nodes. I wanted to color a given edge based on the p-value of the attached node (family) closer to the root. Therefore, I needed to write code to determine these colors for the edges, which I did.

FigTree also colors the edges and not the nodes (but not directly as in R - see below). I was able to get a NEXUS file exported from R, open it in a text-editor, add color code to it (which I copied from an exported NEXUS file from FigTree), and open it in FigTree (after also deleting a small part of the code) and have the resulting tree have color. Obviously, instead of doing this manually, I needed R to be able to add the color information in a suitable manner.

I learned about different phylogenetic tree classes in R through correspondences with people, resources on the internet, and experimenting in R. One of these classes is called phlo4d_ext. It is from the package phyext [24]. Trees of class phlo4d_ext have the capability of having extra information attached to each node that a basic tree (of class phylo) does not have. The function tdata (originally from the phylobase package [25]) allows one to access
this extra information. This functionality proves vital in the process of exporting a file from 
R that can be opened in FigTree with colored edges. It is a way to add annotations to a 
tree in R.

I thought that I needed to assign these annotations to the edges in order to get a plot 
the way I wanted in FigTree. This proved to be an incorrect thing to do. I spent a lot 
of time trying to figure this out. But a person I emailed said that he thought that the 
annotation information for a given edge in FigTree was stored in the node to the edge’s 
right (closer to the tips). It turned out that he was correct - this is how FigTree handles its 
annotations. This was a very important realization. I modified my code and became more 
familiar with things such as the $tdata$ command. For some reason, there was a problem 
with assigning information using the $ operator (for example, \texttt{tdata(tree)$name < - “a”}) 
with the motivating example. This was solved by creating a new data frame and assigning 
it to the \texttt{phylo4d$ext} tree directly.

### 3.2.2 Modifying NEXUS With Regular Expressions

I was able to get R to export a NEXUS file with color annotation, but I still needed to 
modify this exported file to be in a state that FigTree could use. The method I used utilized 
regular expressions (see Section 1.4). In R, I captured the text generated when exporting 
the NEXUS file and then accessed the portion I needed to modify (the Newick portion). On 
this portion, I used the regular expression command \texttt{gsub} to find what I needed to modify 
and replaced it with roughly the same basic information, but in a format that FigTree could 
make use of. For example, the following is a snippet of the NEXUS-format text from the 
example tree (see Section 2.2) obtained using the \texttt{write.nexus.simmap} function from the 
\texttt{phyext} package (using default argument settings):

```
"(1:[@nodecolor={16250871},pvalue={0.5}]0.43944978271611,(((2: 
[@nodecolor={11671595},pvalue={0.00190853897848085}]
0.257429021410644"
```
The following is regular expression code in R that gets the above snippet in a format
FigTree can use:

```r
gsub(":\[&nodecolor=\{([0-9\-])\},pvalue=\{([0-9\.\-\.]\}\}],(1:\[&nodecolor={16250871}, pvalue={0.5}\]0.43944978271611,((((2:\[&nodecolor={11671595}, pvalue={0.00190853897848085}\]0.257429021410644")
```

This is what is returned:

```
"(1[&!color=#16250871, P-value="0.5"]:0.43944978271611,(((2[&!color=#11671595, P-value="0.00190853897848085"]):
0.257429021410644"
```

This is in a format the FigTree can read. Note that it is only a snippet of the file, so
FigTree cannot meaningfully use only this part without additional text. For more informa-
tion on the portions of the above code which involve colors and p-values, see Sections 3.2.3
and 3.2.4, respectively.

### 3.2.3 Coloring Tree

Another issue I came up against involves colors. In general, there are different ways to
represent the same color, such as in RGB or hexadecimal formats. To try and figure out
what format of color FigTree uses, I opened a NEXUS file in FigTree that I had exported
from R. This file had color information. I then exported this file and opened it in a text-
editing program. The colors were displayed in a format such as “#5105621”. It is not clear
what color format this is, but it is a red color. (I also looked up colors such as this on the
internet, but I am still unsure as to what exactly is going on with their format.) I used the
RColorBrewer package in my functions export.figtree and plot.color. This package (using the
function brewer.pal) produces colors in the hexadecimal format. The color from the original
exported R file (mentioned above) is represented as “#B2182B”. This is hexadecimal. In
other words, *FigTree* converts “#B2182B” to “#-5105621”. As far as I can tell, this is the same color.

I assumed that I needed to get the hexadecimal colors and convert them to decimal in *R* (I previously thought that decimal was the format of color *FigTree* exports with - this may be, but I do not know) so that when exported, *FigTree* could open them. (I later realized that *FigTree* is also capable of handling hexadecimal.) Because the hexadecimal colors obtained via *brewer.pal* had a “#” in front of them, I used regular expressions to get rid of this “#”. I subsequently used the `strtoi(color,16)` command in *R* to convert to color (hexadecimal) to decimal.

### 3.2.4 Adding P-value Annotations

Later, I added the functionality to be able to export p-value information that can be viewed in *FigTree*. If one highlights a branch, then clicks on the *Annotation* button, the p-value for the attached node closer to the root is visible (see Section 2.3.3.2 and Fig. 2.6). The problem was that in *R*, I needed to put this annotation information along with the node closer to the tips in the exported .tre file obtained from `export.figtree` (see Section 2.3.3). For example, if we had parent node 1 connected by an edge to child node 2, then in order to get the p-value annotation for the edge (corresponding to node 1), we would put this annotation information along with node 2. I was able to do this. I also had to write new regular expression code to handle this extra information because I now had two columns of annotations. It involved using a different option in `write.nexus.simmap`.

### 3.2.5 Handling Trees Without Defined Edge Lengths

I later tested the `export.figtree` function with a tree that had no edge lengths defined. I opened the exported file in *FigTree* and there were no errors, but there was also no visible tree. It turned out that if there were not any edge lengths defined for the tree, then the edge lengths were coded as NA in the `tree4d_ext` version of this same tree, but showed up as 0 in the text generated by running the `write.nexus.simmap` function. I had written my regular expression code to handle edge lengths, but if the edge lengths were 0, then the
NEXUS file that FigTree opened produced no visible edges. I fixed this by checking to see if the edges have lengths (using the hasEdgeLength function originally from the phylobase package; I assume that if any edge does not have an edge length, then all edges do not). If not, I did not include the edge lengths in the file FigTree imports. In this case of no edge lengths, FigTree automatically assigns its own edge lengths to the edges.

Note that, in the phylo tree, there is a difference between a tree having no edge lengths defined (for a tree named tree, tree$edge.length will return NULL) and a tree having edges of 0 length. The first case is discussed in the above paragraph. In the second case, in the plots in FigTree (using export.figtree) and R (using plot.color), the edges with 0 length will have no length.

I also added an option in SigTree to ignore edge lengths (if the tree has edge lengths to begin with) so that a more uniformly-spaced tree is easy to obtain in FigTree.
Chapter 4

Results: Motivating Example

4.1 Application to Motivating Example

The inspiration behind this thesis and the SigTree package was the motivating example of Section 1.1. I analyzed the motivating example using SigTree. Using raw data, nonparametric Wilcoxon rank-sum p-values (see Section 1.1.2) were calculated. These p-values came in two varieties - one-sided and two-sided. For the one-sided, the alternative hypothesis is that refined wheat (RW) diet is less than whole wheat (WW) diet. I used these one and two-sided p-values and their corresponding OTU tip labels to construct two unsorted.pvalues (see Section 2.3.1.2) data frames. The first data frame is called pvalues.1sided and uses the 1-sided p-values. The other uses the 2-sided p-values and is called pvalues.2sided. The phylogenetic tree (see Section 2.3.1.1), tree, was created by using the read.tree function on a Newick file provided by the collaborating laboratory. It is the same tree as in Fig. 1.2. This tree has tip labels (OTU IDs) that match the tip labels in both pvalues.1sided and pvalues.2sided.

4.2 Setup

I used the plot.color function (see Section 2.3.1) from SigTree to create two plots of the tree of interest (Figs. 4.1 and 4.2, corresponding to Stouffer’s Method and Fisher’s Method, respectively). A magnified portion of Fig. 4.1 is shown in Fig. 4.3, and a magnified portion of Fig. 4.2 is shown in Fig. 4.4. Screenshots of two CSV files, created using the export.inherit function, are shown in Figs. 4.5 and 4.6, which correspond to Figs. 4.3 and 4.4, respectively.
4.2.1 Stouffer’s-BY

The first plot (Fig. 4.1) uses Stouffer’s Method to combine the p-values. The pvalues.Isided data frame with 1-sided p-values is used because Stouffer’s Method is not made for 2-sided p-values (see Section 1.3.1). I adjust the p-values using the Benjamini-Yekutieli Method (BY). This is more appropriate than the Benjamini-Hochberg Method, for example, due to the fact that BY does not assume independence among p-values (see Section 1.5.3) and because dependence is inherent in the tree structure. For the p-value cutoff argument, I use the value of $c(0.01, 0.05, 0.1, 0.9, 0.95, 0.99)$. This is appropriate in this case because 1-sided
p-values are used (see Section 2.3.1.7). I use a modified version of the divergent “RdBu” palette from the RColorBrewer palette. This vector of colors is called RdBu1 and is the same RdBu1 as described in Section 2.2. RdBu1 is the same as “RdBu”, with the exception that the middle color is now gray. RdBu1 ranges from dark red (corresponding to very low p-values and providing strong evidence that RW < WW) to dark blue (corresponding to very high p-values and providing strong evidence that RW > WW). The (insignificant) range from .1 to .9 is colored gray. All other arguments assume their default values. The code used for this plot is:
The code to produce the CSV file (named "StouffersBY.CSV") is:

```r
export.inherit(tree, pvalues.1sided, test="Stouffers", adjust=TRUE, side=1, method="BY", p.cutoffs=c(.01, .05, .1, .9, .95, .99), pal=RdBu1)
```

The plot produced is shown in Fig. 4.1 and part of the CSV file is shown in Fig. 4.5. A magnified portion of Fig. 4.1 is shown in Fig. 4.3. Discussion will follow the setup of the second plot.

### 4.2.2 Fisher’s-BY

The second plot (Fig. 4.2) uses Fisher’s Method. It uses the `pvalues.2sided` data frame (with 2-sided p-values). I again use BY adjustment. The value I use for the p-value cutoffs argument is `c(.01, .05, .1)` because of the 2-sided nature of the p-values (see 2.3.1.7). In
addition, because I use 2-sided p-values, it is more appropriate to use a sequential palette. As discussed in Section 2.3.1.7, a reversed version of a sequential RColorBrewer palette makes more sense so as to make the darkest colors correspond to the lowest p-values and provide the strongest visual evidence that RW ≠ WW. I reverse the “Reds” palette as in Section 2.3.1.7, but also add the gray color, as in Section 2.2 to obtain Reds1, the modified version of the “Reds” palette. Gray corresponds to p-values above .1. All of the remaining arguments take on their default values. The code used for this plot is:

```r
plot.color(tree, pvalues.2sided, test="Fishers", adjust=TRUE, side=2, method="BY", p.cutoffs=c(.01, .05, .1), pal=Reds1)
```

The code to produce the CSV file (named “FishersBY.CSV”) is:

```r
export.inherit(tree, pvalues.2sided, test="Fishers", adjust=TRUE, side=2, method="BY", file="FishersBY.CSV")
```
Fig. 4.5: Portion of interest of Stouffers's-BY CSV file created using export.inherit.

The plot produced is shown in Fig. 4.2 and part of the CSV file is shown in Fig. 4.6. A magnified portion of Fig. 4.2 is shown in Fig. 4.4.

4.3 Results

At first glance, Fig. 4.1 is largely blue and gray and Fig. 4.2 is largely red and gray. Fig. 4.1, however, does contain some red branches and tip labels. Both plots share many of the same significant edges. Fig. 4.1’s edges would be colored roughly the same as Fig. 4.2 if the blues were converted to red. The section in the upper right is uncolored in Fig. 4.2 and mostly uncolored in Fig. 4.1. The large, interior branches in the middle of both plots are largely highly significant.

For Fig. 4.1, the red branches and tips correspond to low p-values and provide evidence that the OTU levels from a refined wheat diet are lower than the OTU levels from a whole wheat diet. Conversely, blue branches and tips provide evidence that the OTU levels from a
refined wheat diet are greater than the OTU levels from a whole wheat diet. Gray tips and branches provide evidence that there is no significant difference between diets. Thus, many tips and branches are not significant. A small number are significant in the RW < WW direction, while a greater number are significant in the RW > WW direction.

For Fig. 4.2, the red branches provide evidence that the levels of OTUs from a refined wheat diet are not equal to the levels of OTUs from a whole wheat diet, or that RW ≠ WW. The gray branches indicate no significant difference. There are many branches colored red and many of these are a dark color of red.

For a specific example, refer to Fig. 4.3 and Fig. 4.4. Fig. 4.3 is a magnified section of Fig. 4.1, the Stouffer’s-BY plot. Fig. 4.4 is a magnified section of Fig. 4.2, the Fisher’s-BY plot. Tips 259148, 271734, and 308269 are significant in both Figures. Tips 14093, 260752, and 304065 are significant in the Stouffer’s-BY portion, but not in the Fisher’s-BY portion. The remaining tips in both graphs are all gray. The Fisher’s-BY portion has more branches that are significant. The branches coming from node 1013 in this plot are more highly significant than the same branch in the Stouffer’s-BY portion. This is likely due to the fact
that Fisher’s Method is not a consensus test and the highly significant p-values influence the family more than they do in Stouffer’s Method, which is a consensus test. See Section 1.3.

Fig. 4.5 shows the p-values (column B) for the tips (column A) in Fig. 4.1, and Fig. 4.6 show the p-values for the tips in Fig. 4.2. For example, tip 259148 has p-value .996 in the Stouffer’s-BY plot and .007 in the Fisher’s-BY plot. Both of these are highly significant. Column C shows that the only member of each tips’ family is itself. For these rows (and all other rows of tips), column C is the last (and only) column that contains a tip.

The results presented in this chapter are subject to further interpretation by the collaborating laboratory.

4.4 Context

For the purposes of demonstration, I included plots (and CSV output) using both Stouffer’s Method (with 1-sided p-values) and Fisher’s Method (using 2-sided p-values). In doing so, I illustrate both divergent and convergent sets of colors.

However, for this application, the Stouffer’s Method with 1-sided p-values is probably the best choice. The experimental design used better lends itself to analysis using Stouffer’s Method. We are more interested in using a consensus test (Stouffer’s) than a non-consensus test (Fisher’s). Stouffer’s Method is also a symmetric test (and we are using 1-sided p-values), so we can more easily interpret the coloring of the blue branches and tips (corresponding to p-values close to 1) than we could if we were to use 1-sided p-values with Fisher’s Method. (Note that we calculated our p-values here using the Wilcoxon test. This gives us p-values that are not symmetric between competing alternative hypotheses, but are close.) Fisher’s Method with 2-sided p-values, as used in Section 4.2.2, does not indicate the direction of the significance of the tips and edges, i.e., it does not indicate whether there is evidence in favor of RW < WW or RW > WW. For these reasons, more weight should be placed on the output produced using Stouffer’s Method.
Chapter 5

Discussion

5.1 Generalizability/Utility of SigTree

*SigTree* is able to handle the motivating experiment of Section 1.1, but it is not limited to analyzing abundances of bacteria species found in the guts of mice. It can be used in any application that has meaningful p-values for each tip in a known phylogenetic tree structure. Part of this flexibility in *SigTree* is due to the fact that it does not calculate the p-values of the tips of the tree. This is left to the user. As seen in Section 1.1.2, we can run into problems such as non-normality of the data. There are many different designs of experiments that are possible as well as many ways to calculate p-values for these different experiments. In addition, data comes in many different shapes and sizes. Thus, it is not feasible to include functionality in *SigTree* to calculate p-values from raw data for every possible application. Therefore, it is left to the user to create the data frame with the tip labels and corresponding p-values (the argument `unsorted.pvalues` in *SigTree* user functions - see Section 2.3.1.2).

*SigTree* also allows the user to use 1-sided or 2-sided p-values. In addition, the user can either use Stouffer’s Method or Fisher’s Method in order to combine the p-values. *SigTree* allows for the selection of various p-value adjustment methods to account for multiple hypothesis testing. It allows the user to customize the way the trees are plotted and to have the ability to choose the p-value cutoff ranges and the colors they wish to use.

5.2 Possible Additions

As with software in general, improvements to *SigTree* are possible. Two of these possible improvements are discussed.
5.2.1 Subset of Master Tree

The tree in the motivating example contains over 500 tips, but is only a subset of a larger phylogenetic tree of bacteria [26, 27]. This larger tree is more-or-less unchanging and contains the relationships of many different bacteria. In the future, the OTUs of interest could change and so one may want to use a different subset of this “master” tree. Other researchers may be interested in a completely different subset of this same tree. Possible “master” trees for various taxonomic ranks can be obtained from, for example, greengenes [27] or the R packages PhyloOrchard [28] and treebase [29].

A possible future addition to SigTree is to require the user to only pass in the p-value data frame (see Section 2.3.1.2), and not a tree argument (see Section 2.3.1.1). In place of the tree argument, the user could perhaps pass in a string indicating that they want to use the master tree. Assuming the tip labels of the data frame match a subset of the tip labels in the master tree, then the three main functions in SigTree could be modified to create a trimmed tree based on the tip labels in the data frame. I do not think this would be too difficult to do. It would likely involve using the extract.clade function from the ape package [19]. This function accepts as its main arguments a tree and a vector of tips to keep. It makes a trimmed tree with only these tips. With this smaller tree, we could proceed with the user functions as before.

Another approach would be to make a new function that accepts a data frame as its argument and returns a tree (that is, a subset of the master tree) that contains only the tips of interest based on the tip of the data frame. This tree could then be passed to the three main user functions along with the data frame. This accomplishes the same thing as described above, but would not involve modifying the main functions.

5.2.2 Zooming In

Another possible addition involves zooming in on a portion of the tree (in R). The intent is not to recalculate the p-values and colors, but simply to zoom in on a tree that is already colored to better see the subtree of interest. This would be especially useful for very large trees. One possible approach to doing this involves the subtreeplot function from
the `ape` package [19]. This function brings up a graphics window with two plots of the same tree - one on the left and one on the right. If the user clicks on a node on the left tree, then the tree on the right is redrawn with the clicked-on node as the root. In the `plot.color` user function of SigTree, I use the `plot.phylo` function (from the `ape` [19] package) to plot my tree and pass it vectors of colors for the coloring of the tips and edges. I think that replacing `plot.phylo` with `subtreeplot` might work after some modification.
References


Appendix
#SignTree Code

#Functions that accept the tree and return the number of edges=num.edges; number of
#tips=num.tips
#number of internal nodes=num.internal.nodes; total number of nodes (internal + tip) =
#num.nodes

#Internal functions
#Arguments: One or more of the functions plot.color, export.inherit, and
#export.figtree call these functions (directly or indirectly). Find
#argument descriptions in plot.color, export.inherit, or export.figtree.

num.edges <- function(tree) length(tree$edge[,1])
num.tips <- function(tree) length(tree$tip.label)
num.internal.nodes <- function(tree) tree$Nnode
num.total.nodes <- function(tree) length(tree$tip.label)+tree$Nnode

sort.pvalues <- function(tree, unsorted.pvalues)
{
  #The order of p value labels is likely different than the order of the tip labels in
  #the tree.
  #This sorts the original p values according to the order of the tips in the tree using
  #the merge function.
  #It returns the p values in sorted order (sorted in the same way as the order of the
  #tree tips).
  #Internal function
  #Arguments: One or more of the functions plot.color, export.inherit, and
  #export.figtree call this function (directly or indirectly). Find
  #argument description in plot.color, export.inherit, or export.figtree.
  merge(tree$tip.label, unsorted.pvalues, by.x=1, by.y=1, all=TRUE, sort=FALSE) ->
  sorted.pvalues
  return(sorted.pvalues)
}

stouffers <- function(x)
{
  #Function that accepts a vector of values and calculates a family-wide p value based on
  #Stouffer’s method
  #Internal function
  #Argument: x is a vector of p-values
\[ \text{pnorm} \left( \frac{\sum (\text{qnorm}(x))}{\sqrt{\text{length}(x)}} \right) \]

```r
fishers <- function(y)
{
# Function that accepts a vector of values and calculates a family-wide p value based on
# Fisher's method
# Internal function
# Argument: y is a vector of numbers
  pchisq((-2)*sum(log(y)), 2*length(y), lower.tail=FALSE)
}
```

```r
index.matrix <- function(tree)
{
# Create an index data frame called index. We do this so we can know which descendants
# belong to each node, or family (used later to calculate family wide p values later).
# Each column corresponds to a node (family). The rows correspond to the tips. If
# there is a 1 in a cell, then the row # (tip) belongs to the column # (node/family).
# For example, if column 2, row 2 had value 1, then tip 2 (tree$tip.label[2]) belongs
# to node/family 2. If it had value 0, it doesn't belong to node/family 2.
# This assumes that each internal nodes all have smaller numbers than each of their
# descendants, not including tips.
# Internal function
# Argument: One or more of the functions plot.color, export.inherit, and
#          export.figtree call this function (directly or indirectly). Find
#          argument description in plot.color, export.inherit, or export.figtree.
  # Call num.tips, num.internal.nodes, and num.total.nodes to get us the # of edges, #
  # of internal nodes, and # of total nodes, respectively.
  n.tips <- num.tips(tree)
  n.internal.nodes <- num.internal.nodes(tree)
  n.total.nodes <- num.total.nodes(tree)
  n.edges <- num.edges(tree)
# Create an identity matrix, tipmatrix, corresponding to the first (n.tips) columns.
# This is done because the first columns (columns 1 to n.tips) correspond only to
# tips and not internal nodes and will only include themselves in their family.
# For example, if there were 25 tips, then the index matrix would contain 25 rows
# and 25 columns. Column 12 would only have a 1 in row 12, with the rest being
# 0's. After (n.tips), the nodes are internal nodes (not tips), and hence will

```
# have more than one tip in their family. We create internalnodesmatrix, simply a 
# matrix full of 0's with (n.tips) rows and (n.internal.nodes) columns. Then 
# append internalnodesmatrix to tipmatrix to get index, a (currently) blank matrix 
# of size (n.tips) rows and (n.tips + n.internal.nodes) columns

diag(n.tips) -> tipmatrix
mat.or.vec(n.tips, n.internal.nodes) -> internalnodesmatrix
index <- cbind(tipmatrix, internalnodesmatrix)

#Sort the tree$edge matrix by the first column (in descending order) and call it 
# tree$edgesort in order for the following for loop to capture all of what we need. 
# *This assumes that our tree has its nodes (and edge matrix) assigned the way we 
# expect. This is that the root is the (n.tips+1) node and that an internal node 
# always has a lower number than any of its descendants. This is important so that 
# our sort in the following line of code will work. The edge matrix is a matrix 
# that identifies how all of the nodes are connected to each other. The row number 
# corresponds to the edge number. For examples, if in our edge matrix, row 2 has 
# value 12 in column 1 and value 13 in column 2, then edge 2 connects node 12 to 
# node 13. For a given row, we assume that the edge matrix always has a higher 
# number in column 2 if column 2 is a tip. If column 2 is a tip, we assume column 
# 2 will have a higher number. We also assume that tips can only be in column 2.

tree$edge[order(-tree$edge[,1]),] -> tree$edgesort

#Fill in the index matrix with 1's if the tip corresponding to the row belongs to 
# the family corresponding to the column. The way tree$edgesort is sorted allows 
# us to use the following for loop to do this. Our tree$edgesort matrix still holds 
# the information about which nodes are connected together. The edges are just 
# sorted in a different way now. Because our index matrix has the identity matrix as 
# the first (n.tips x n.tips) rows x columns, and because the identity matrix contains 
# the information that the tips belong to their own family, and because the 
# tree$edgesort matrix is sorted high to low, our for loop fills in the index matrix 
# with 1's for all of the rows/tips that are descenants of a given column. During the 
# first iteration, the tree$edgesort[1,1]th (1) column of index will be added to the 
# tree$edgesort[1,2]th (2) column of index and assigned to the tree$edgesort[1,1]th 
# column of index. Because (2) is a tip, it will have exactly one '1.' Now, (1), 
# which is an ancestor of (2), will have one '1.' During the second iteration, (1) 
# will get another '1,' bringing its total to 2. Thus (1) is an ancestor of both (2) 
# and (3): the tree$edgesort[2,2]th (1) column of index. This will continue to work 
# because of the way it is sorted and the assumption that for non-tips, the ancestor 
# has a smaller node number than the descendant.

#also assumes that each node only has 2 children

for(i in 1:(n.edges)) #changed from n.total.nodes-1 to n.edges 
{
index[,tree$edgesort[i,1]] <- index[,tree$edgesort[i,1]] + index[,tree$edgesort[i,2]]

return(index)

p.p2.ADJ.p1 <- function(p, method)
{
#Function to convert a vector of one-tailed p-values to two-tailed, perform p-value
# adjustment on the p-values (for multiple hypothesis testing), and then go back to
# the one-tailed [adjusted] scale.
#This assumes null Mean2=Mean1 and alt: Mean2>Mean1
#P-value adjustment methods are found in ?p.adjust
#Argument: p is a vector of one-tailed p-values
#Arguments: One or more of the functions plot.color, export.inherit, and
#          export.figtree call this function (directly or indirectly). Find
#          argument descriptions in plot.color, export.inherit, or export.figtree.
  t <- p <= 0.5 # T/F of left tail
  p1 <- p2 <- rep(NA,length(p))
  p2[t] <- 2*p[t]
  p2[!t] <- 2*(1-p[!t])
  p.ADJ <- p.adjust(p2,method=method)
  p1[t] <- p.ADJ[t]/2
  p1[!t] <- 1-p.ADJ[!t]/2
  return(p1)
}

result <- function(tree, unsorted.pvalues, test, adjust, side, method)
{
#Calculate pvalues based on Stouffer's or Fisher's tests. Return reults, a data frame
# that contains this information.
#Internal function
#Arguments: One or more of the functions plot.color, export.inherit, and
#          export.figtree call this function (directly or indirectly). Find
#          argument descriptions in plot.color, export.inherit, or export.figtree.
#Call other functions to get n.total.nodes, the index matrix, sorted pvalues, and
# create an empty data frame called results
  n.total.nodes <- num.total.nodes(tree)
  index <- index.matrix(tree)
results <- data.frame()
sorted.pvalues <- sort.pvalues(tree, unsorted.pvalues)

# Fill in results data frame, which contains the p-values for all of the
# nodes/families for the selected test (Stouffer's or Fisher's). temp contains the
# p-values (obtained from the adjusted.pvalues data frame) for the jth column (node).
# temp contains only the p-values for the rows where index[,j]==1, ie. the tips
# belonging to the jth column. We then call the Stouffer's and Fisher's functions
# on temp and assign the values we obtain to the first column of results. Each row
# of results represents a different node.

if(test=="Stouffers")
{
    for(j in 1:n.total.nodes)
    {
        temp <- sorted.pvalues[index[,j]==1,2]
        results[j,1] <- stouffers(temp)
        names(results)[1]<-"Stouffer's"
    }
}else
{
    for(j in 1:n.total.nodes)
    {
        temp <- sorted.pvalues[index[,j]==1,2]
        results[j,1] <- fishers(temp)
        names(results)[1]<-"Fisher's"
    }
}

# Now we need to apply the p-value adjustment if the adjust argument is TRUE. If
# not, this step is skipped. Afterwards, results is returned.

if(adjust==TRUE)
{
    if(side==1)
    {
        results[,1] <- p.p2.ADJ.pl(results[,1], method)
    }else
    {
        results[,1] <- p.adjust(results[,1], method=method)
    }
}

return(results)
tip.colors<-function(tree, unsorted.pvalues, p.cutoffs, pal, test, adjust, side, method)
{
  #Create tipcolor, which is a matrix of color values that determines how the tips/OTUs
  # are to be colored. It is of (n.tips) length and contains color values. Each row
  # represents a tip.
  #Internal function
  #Arguments: One or more of the functions plot.color, export.inherit, and
  # export.figtree call this function (directly or indirectly). Find
  # argument descriptions in plot.color, export.inherit, or export.figtree.
  #get results from the result function and create tipcolor, a matrix.
  results <- result(tree, unsorted.pvalues, test, adjust, side, method)
  matrix() -> tipcolor
  #create p.cutoffs.new, which is just p.cutoffs with 1 appended to the end. n.cutoffs
  # is the length of p.cutoffs.new.
  p.cutoffs.new<-c(p.cutoffs, 1)
  n.cutoffs <- length(p.cutoffs.new)
  #Test to see if pal is of length 1. If it is, it will be a valid RColorBrewer
  # palette due to the error checking in the function that called it (assuming that
  # either plot.color or export.figtree called it). If it is a valid palette, we
  # create cols and assign it a vector of hexadecimal colors from RColorBrewer based
  # on n.cutoffs and pal. If it is not of length 1, then it is a vector of (we
  # assume) valid colors. Then we simply assign pal to cols.
  if(length(pal)==1)
  {
    cols <- brewer.pal(n.cutoffs, pal)
  }
  else
  {
    cols <- pal
  }
  #n.tips is the number of tips in tree.
  n.tips <- num.tips(tree)
  #for loop that works its way backwards from n.cutoffs to 1. for each k, results[,1]
  # <= p.cutoffs.new[k] is a vector of TRUE/FALSE values. For all of the TRUE values,
  # cols[k] is assigned to tipcolor. cols is a vector of colors. For example, the
  # first iteration is when k=n.cutoffs. This is the length of p.cutoffs.new. If our
  # cutoffs were (.01, .05, .10, .90, .95, .99), then p.cutoffs.new would be (.01, .05,
  # .10, .90, .95, .99, 1) and n.cutoffs would be 7. Then k=7. cols would have 7
}
# colors in it. cols[7], the 7th color, would be assigned to all of the rows of
tipcolor where the corresponding p-value was less than or equal to 1. As a matter
of fact, all rows would be assigned this color because all of the p-values are by
definition less than or equal to 1. The second iteration would assign cols[6] to
all of the rows where the p-values were <= .99 and so on. The last iteration would
only assign cols[1] to values that were <= .01. Note that with the way we did our
logic in this for loop, we needed to append 1 to p.cutoffs (and create
p.cutoffs.new). If we did not, we would have not assigned values to our top
interval, for example (.99,1].

for(k in n.cutoffs:1)
{
    tipcolor[results[,1] <= p.cutoffs.new[k]]<cols[k]
}

#Take only the first (n.tips) values of tipcolor and assign to tipcolor. We do this
# because tipcolor is originally too long. It should only be (n.tips) long because
# there are only (n.tips) tips.
tipcolor <- tipcolor[1:n.tips]
return(tipcolor)

edge.colors<-function(tree, unsorted.pvalues, p.cutoffs, pal, test, adjust, side, method)
{
#Create edgecolor, which is a matrix that determines how the edges/families are to be
# colored. It is of length (n.edges)-I think and contains color values. Each row
# represents an edge.

#Internal function

#Arguments: One or more of the functions plot.color, export.inherit, and
#          export.figtree call this function (directly or indirectly). Find
#          argument descriptions in plot.color, export.inherit, or export.figtree.

#get results from result function; create edgecolor, a matrix
results <- result(tree, unsorted.pvalues, test, adjust, side, method)
matrix() -> edgecolor

#create p.cutoffs.new, which is just p.cutoffs with 1 appended to the end. n.cutoffs
# is the length of p.cutoffs.new. cols is a vector of hexadecimal colors from
# RColorBrewer based on n.cutoffs and pal

p.cutoffs.new <- c(p.cutoffs, 1)
n.cutoffs <- length(p.cutoffs.new)

#Test to see if pal is of length 1. If it is, it will be a valid RColorBrewer
# palette due to the error checking in the function that called it (assuming that
# either plot.color or export.figtree called it). If it is a valid palette, we
# create cols and assign it a vector of hexadecimal colors from RColorBrewer
# based on n.cutoffs and pal. If it is not of length 1, then it is a vector of
# (we assume) valid colors. Then we simply assign pal to cols.

if(length(pal)==1)
{
  cols <- brewer.pal(n.cutoffs, pal)
}
else
{
  cols <- pal
}

#For loop that works its way backwards from n.cutoffs to 1.
# (1) results[tree$edge[,1],1]<=p.cutoffs.new[k] is a vector of TRUE/FALSE
# values. We use results[tree$edge[,1],1] in our comparison because we want to
# find the p-value corresponding to the left node (if looking at the tree where
# the root is on the left and the tips on the right; otherwise the more interior
# node; the ancestor) of the edge to determine the coloring for that edge. This
# is why we choose tree$edge[,1] instead of tree$edge[,2]. (1) will be in the
# order of the edge matrix. Thus edgecolor will be in the correct order.
#For example, on the first iteration, k=n.cutoffs. This is the length of
# p.cutoffs.new. If our cutoffs were (.01, .05, .10, .90, .95, .99), then
# p.cutoffs.new would be (.01, .05, .10, .90, .95, .99, 1) and n.cutoffs would be
# 7. Then k=7. Cols would have 7 colors in it. cols[7], the 7th color, would be
# assigned to all of the rows (edges) of edgecolor where the corresponding p-value
# (from results[tree$edge[,1],1]) was less than or equal to 1. As a matter of
# fact, all rows would be assigned this color because all of the p-values are by
# definition less than or equal to 1. The second iteration would assign cols[6]
# to all of the rows where the corresponding p-value were <=.99. The last
# iteration would only assign cols[1] to values that were <=.01. Note that with
# the way we did our logic in this for loop, we needed to append 1 to p.cutoffs
# (and create p.cutoffs.new). If we did not, we would have not assigned values to
# our top interval, for example (.99,1).

for(k in n.cutoffs:1)
{
  edgecolor[results[tree$edge[,1],1]<=p.cutoffs.new[k]] <- cols[k]
}

return(edgecolor)
plot.color <- function(tree, unsorted.pvalues, adjust=TRUE, side=1, 
method="BY", p.cutoffs=ifelse(rep(side==1, ifelse(side==1, 6, 3)), 
c(.01, .05, .1, .95, .99), c(.01, .05, .1)), 
pal=ifelse(rep(side==1, ifelse(side==1, 1, length(p.cutoffs)+1)), 
"RdBu", rev(brewer.pal(length(p.cutoffs)+1,"Reds"))), 
test="Stouffers", node.label=TRUE, tip.color=TRUE, edge.color=TRUE, 
tip.label.size=.1, node.label.size=.1, type="fan", 
use.edge.length=TRUE)
{
#Create plot.color, a function that plots our tree with optional edge colors and
# tip colors.
#User function
#Argument: tree is a phylogenetic tree of class phylo.
#Argument: unsorted.pvalues (can be sorted as well) is a data frame with the tip
# identifier in column 1 and the pvalue to be used in column 2
#Argument: adjust is a logical argument controlling p-value adjustment. If FALSE, then
# no p-value adjustment will take place. If true, then side and method will determine
# the type of adjustment.
#Argument: side is either 1 or 2. This corresponds to whether the p-values are 1-sided
# or 2-sided.
#Argument: method is one of the methods found in p.adjust.methods and is the method to
# be used in the p.adjust function.
#Arguments: p.cutoffs is a vector of numbers, x, where 0<x<1 (not including 0 and 1) and
# sorted in ascending order. These are the p value cutoffs. They affect how the edges
# are colored. If pal is a divergent palette (such as RdBu; probably use a divergent
# palette if p-values are 1-sided), the smallest and the largest p values are colored
# the darkest. For example, if p.cutoffs = c(.01, .05, .10, .90, .95, .99), then
# the ranges are [0-.01), (.01-.05], ..., (.99-1.] When using the "RdBu" pallette, the
# range 0-.01 is the darkest red shade and .99-1 is the darkest blue shade. A
# sequential palette would probably be used when p-values are 2-sided. The default
# argument for p.cutoffs is c(.01, .05, .10, .90, .95, .99) if side is 1 and
# c(.01, .05, .1) if side is 2.
#Argument: pal is a pallete from the package RColorBrewer. Find a list of palettes by
# display.brewer.all() or ?brewer.pal or brewer.pal.info. pal may also be a vector of
# colors. The length needs to be one longer than the length of p.cutoffs (because this
# is how many ranges of colors there are). Colors in this vector need to be in
# hexadecimal format. For example, "#E2182B". pal=ifelse(side==1, "RdBu", "Reds")
# assigns the default argument of pal to be "RdBu" if side is 1 and "Reds" if side is 2.
# This is because it is probably better to use a divergent palette if using 1-sided
# p-values and a sequential palette if using 2-sided p-values. The default
# argument for pal is "RdBu" when side is 1 and "Reds" (but in reverse) when side is 2.
# Note that the sequential palettes in RColorBrewer range from light to dark. This
# means that the light colors correspond to low p-values. This is why the default
# palette when side is 2 is a reversed version of "Reds", so that the darker reds
# correspond to the lower p-values.
#Argument: test is either "Stouffers" or "Fishers." This is the p-value combination to
# be used.
#Argument: node.label is a logical parameter that displays numerical node labels when
# TRUE and doesn't when FALSE. These labels can be used to cross-reference to the file
# obtained from the export.inherit function.
#Argument: tip.color is a logical argument that enables the tips to be colored when TRUE
# and not be colored when FALSE.
#Argument: edge.color is a logical argument that enables the edges to be colored when TRUE
# and not be colored when FALSE.
#Argument: tip.label.size is a numerical argument that controls the (cex) size of the text
# of the tip labels.
#Argument: node.label.size is a numerical argument that controls the (cex) size of the text
# of the node labels.
#Argument: type controls which type of plot will be produced. Available options are
# "phylogram," "cladogram," "fan," "unrooted," and "radial."
#Argument: use.edge.length is a logical argument. If TRUE, then the plot uses the defined
# edge lengths as usual. But if FALSE, then the plot ignores these edge lengths. This
# can be useful to produce a more uniformly-spaced tree.

#Error checking.
if(class(tree)!="phylo")
{
    return(cat("Error: Class of tree must be phylo.","\n"))
}
if(length(tree$tip.label)!=length(unsorted.pvalues[,1]))
{
    return(cat("Error: There must be the same number of tip labels in tree as in
unsorted.pvalues.","\n"))
}else if(mean(sort(as.character(unique(unsorted.pvalues[,1])))==sort(as.character(  
unique(tree$tip.label))))!=1)
{
    return(cat("Error: The tip labels in tree must have the same labels as the tip
labels in unsorted.pvalues.","\n"))
}
if(min(unsorted.pvalues[,2])<0 | max(unsorted.pvalues[,2]>1)
{  
    return(cat("Error: P-values in unsorted.pvalues must be between greater
            than or equal to 0 and less than or equal to 1."","\\n"))  
}  
else  
{  
    f.one <- 1-100*.Machine$double.eps  
    f.zero <- 100*.Machine$double.eps  
    t <- unsorted.pvalues[,2]>=f.one  
    unsorted.pvalues[t,2] <- f.one  
    t <- unsorted.pvalues[,2]<=f.zero  
    unsorted.pvalues[t,2] <- f.zero  
}  
if(adjust!=TRUE & adjust!=FALSE)  
{  
    return(cat("Error: Value of adjust must be either TRUE or FALSE."","\\n"))  
}  
if(side!=1 & side!=2)  
{  
    return(cat("Error: Value of side must be either 1 or 2."","\\n"))  
}  
if(is.numeric(method))  
{  
    return(cat("Error: Value of method should be one of those found in
            p.adjust.methods"))  
}  
if(length(p.cutoffs)==0)  
{  
    return(cat("Error: There must be at least one value of p.cutoffs."","\\n"))  
}  
if(max(p.cutoffs)>=1 | min(p.cutoffs)<=0 | is.unsorted(p.cutoffs))  
{  
    return(cat("Error: The values of p.cutoffs must be greater than 0, less than
            1, and in ascending order."","\\n"))  
}  
if(length(pal)==1 & is.character(pal))  
{  
    if(!is.element(pal, rownames(brewer.pal.info)))  
    {  
        return(cat("Error: pal must be either a palette from RColorBrewer(to
                see a list: rownames(brewer.pal.info)) or a vector of colors."","\\n"))  
    }  
}
} else if (length(pal) != length(p.cutoffs) + 1)
{
    return(cat("Error: The numbers of colors in pal must be one more than the
number of values in p.cutoffs." ,"n"))
}

if(test != "Stouffers" & test != "Fishers")
{
    return(cat("Error: Value of test must be either "Stouffers" or
"Fishers"." ,"n"))
}

if(test == "Stouffers" & side == 2)
{
    cat("Caution: Stouffer's Method is designed for 1-sided p-values.", "n")
}

if(test == "Fishers" & side == 1)
{
    cat("Caution: For Fisher's Method applied to one-tailed p-values, significance
thresholds for small p-values (near 0) are more meaningful than for large
p-values (near 1).", "n"
}

if(node.label != TRUE & node.label != FALSE)
{
    return(cat("Error: Value of node.label must be either TRUE or FALSE." ,"n"))
}

if(tip.color != TRUE & tip.color != FALSE)
{
    return(cat("Error: Value of tip.color must be either TRUE or FALSE." ,"n"))
}

if(edge.color != TRUE & edge.color != FALSE)
{
    return(cat("Error: Value of edge.color must be either TRUE or FALSE." ,"n"))
}

if(!is.numeric(tip.label.size))
{
    return(cat("Error: Value of tip.label.size must be numeric"))
}

if(!is.numeric(node.label.size))
{
    return(cat("Error: Value of node.label.size must be numeric"))
}
if(use.edge.length!=TRUE & use.edge.length!=FALSE)
{
    return(cat("Error: Value of use.edge.length must be either TRUE or FALSE.\n"))
}

#Get tipcolor and edgecolor from tip.colors and edge.colors, respectively. These
# are matrices of color values used in the plot.phylo function

tipcolor <- tip.colors(tree, unsorted.pvalues, p.cutoffs, pal, test, adjust,
    side, method)

edgecolor <- edge.colors(tree, unsorted.pvalues, p.cutoffs, pal, test, adjust,
    side, method)

#Test whether the tip.color and edge.color parameters are true. If tip.color (the
# parameter from our function) is true, then tip.color=tipcolor in plot (this is
# actually plot.phylo). If false, then tip.color="black." If edge.color (the
# parameter from our function) is true, then edge.color=edgecolor. If false, then
# edge.color="black." We plot our tree. We set show.node.label to false. If the
# user wants node labels, we handle it by the nodelabels() function below. type
# means the type of plot. cex is the size of the tip labels and is set by the
# parameter tip.label.size.

{if(tip.color & edge.color)    plot(tree, tip.color=tipcolor, show.node.label=FALSE,
    edge.color=edgecolor, show.tip.label=TRUE, type=type, cex=tip.label.size,
    use.edge.length=use.edge.length)
else if(tip.color & !edge.color) plot(tree, tip.color=tipcolor,
    show.node.label=FALSE, edge.color="black", show.tip.label=TRUE, type=type,
    cex=tip.label.size, use.edge.length=use.edge.length)
else if(!tip.color & edge.color) plot(tree, tip.color="black",
    show.node.label=FALSE, edge.color=edgecolor, show.tip.label=TRUE,
    type=type, cex=tip.label.size, use.edge.length=use.edge.length)
else    plot(tree, tip.color="black", show.node.label=FALSE, edge.color="black",
    show.tip.label=TRUE, type=type, cex=tip.label.size,
    use.edge.length=use.edge.length))

#If the parameter node.label is true, then we add labels to the internal nodes.
# cex is the size of the node labels and is set by the parameter node.label.size.
# We do not want to have frames around these labels, so set frame to "none."
if(node.label==TRUE)
{
    nodelabels(cex=node.label.size, frame="none")
}
export.inherit <- function(tree, unsorted.pvalues, adjust=TRUE, side=1, method="BY", file="", test="Stouffers")
{
# This is what one can reference the graph to in order to see which of the internal nodes
# contain which tips. It exports a csv file that has the node number in column 1, its
# pvalue in column 2, and then the names of the tips that are in its family in columns
# 3 and on. The first (n.tips) rows will have only 1 member of their row - themselves
# (because these are all tips).
# User function
# Argument: tree is a phylogenetic tree of class phylo.
# Argument: unsorted.pvalues (can be sorted as well) is a data frame with the tip
# identifier in column 1 and the pvalue to be used in column 2
# Argument: adjust is a logical argument controlling p-value adjustment. If FALSE, then
# no p-value adjustment will take place. If true, then side and method will determine
# the flavor of the adjustment.
# Argument: side is either 1 or 2. This corresponds to whether the p-values are 1-sided
# or 2-sided.
# Argument: method is one of the methods found in p.adjust.methods and is the method to
# be used in the p.adjust function.
# Argument: file is the file path to export to.
# Argument: test is either "Stouffers" or "Fishers." This is the p-value combination
# to be used.
# Error checking
if(class(tree)!="phylo")
{
    return(cat("Error: Class of tree must be phylo.","\n"))
}
if(length(tree$tip.label)!=length(unsorted.pvalues[,1]))
{
    return(cat("Error: There must be the same number of tip labels in tree as in
unsorted.pvalues.","\n"))
}else if(mean(sort(as.character(unique(unsorted.pvalues[,1])))==sort(as.character(
    unique(tree$tip.label))))!=1)
{
    return(cat("Error: The tip labels in tree must have the same labels as the tip
    labels in unsorted.pvalues.","\n"))
}
if(min(unsorted.pvalues[,2])<0 | max(unsorted.pvalues[,2])>1)
{
return(cat("Error: P-values in unsorted.pvalues must be between greater than
or equal to 0 and less than or equal to 1.","\n"))
}

else
{
  f.one <- 1-100*.Machine$double.eps
  f.zero <- 100*.Machine$double.eps
  t <- unsorted.pvalues[,2]>=f.one
  unsorted.pvalues[t,2] <- f.one
  t <- unsorted.pvalues[,2]<=f.zero
  unsorted.pvalues[t,2] <- f.zero
}

if(adjust!=TRUE & adjust!=FALSE)
{
  return(cat("Error: Value of adjust must be either TRUE or FALSE.","\n"))
}

if(side!=1 & side!=2)
{
  return(cat("Error: Value of side must be either 1 or 2.","\n"))
}

if(is.numeric(method))
{
  return(cat("Error: Value of method should be one of those found in
  p.adjust.methods"))
}

if(test!="Stouffers" & test!="Fishers")
{
  return(cat("Error: Value of test must be either \"Stouffers\" or
  \"Fishers\"","\n"))
}

if(test=="Stouffers" & side==2)
{
  cat("Caution: Stouffer's Method is designed for 1-sided p-values.", "\n\n")
}

if(test=="Fishers" & side==1)
{
  cat("Caution: For Fisher's Method applied to one-tailed p-values, significance
  thresholds for small p-values (near 0) are more meaningful than for large
  p-values (near 1).","\n")
}

#Get n.tips, n.total.nodes, index through functions. Create inherit, a data frame
# with n.total.nodes rows and n.tips columns. results is obtained through the
# result function.

n.tips <- num.tips(tree)
n.total.nodes <- num.total.nodes(tree)
index <- index.matrix(tree)
inherit <- data.frame(matrix(nrow=n.total.nodes, ncol=n.tips))
results <- result(tree, unsorted.pvalues, test, adjust, side, method)

# for loop that goes through each row in order to assign all of the tips belonging
# to each row (node). tips is obtained by using the indices of all of the 1's from
# a given column of index to get all of the correct tip labels
# (from tree$tip.label). length.tips is the length of tips. tips is assigned to
# the correct row of inherit. We use 1:length.tips to make sure that we assign the
# correct number of columns (and thus don’t repeat). For example, if the 2nd tips
# had 4 values, we would assign these 4 values to inherit[4,]'s first four columns
# and none after. Each row of inherit represents a node and each column a tip.
for(i in 1:n.total.nodes)
{
    tips <- tree$tip.label[index[,i]==1]
    length.tips <- length(tips)
    inherit[i,1:length.tips] <- tips
}

# create a new data frame names new.inherit that has tree$tip.label, ie. the names of
# the tips in the first column and then the numbers from (n.tips+1) to n.total.nodes
# as the rest of the values of the first column. We do this because we want the tip
# labels to be displayed instead of just 1:(n.tips+1) (what R knows them as).
# Afterwards, we want the numbers from (n.tips+1) to n.total.nodes. This way, we can
# reference them to the node labels in the plot. In the second column, we use
# results[,1], which is the pvalues corresponding to the first column. In the third
# column and on, we have the inherit data frame we just created which contains all of
# the tips that each node has in its family.
new.inherit <- data.frame(c(tree$tip.label,(n.tips+1):n.total.nodes),results[,1],
inherit)

# label the first column of new.inherit as "Values"; label the second column as
# "Stouffer's p-value" or "Fisher's p-value" depending on the value of test.
names(new.inherit)[1] <- "Values"
if(test=="Stouffers") names(new.inherit)[2] <- "Stouffer's p-value"
else names(new.inherit)[2] <- "Fisher's p-value"

# write new.inherit to the filepath. na=' ' makes all of the NA's become blank spaces.
write.csv(new.inherit, file, na=' ', row.names=FALSE)
export.figtree <- function(tree, unsorted.pvalues, adjust=TRUE, side=1, 
method="BY", p.cutoffs=ifelse(rep(side==1, ifelse(side==1, 6, 3)), 
c(.01, .05, .1, .9, .95, .99), c(.01, .05, .1)), file="", 
pal=ifelse(rep(side==1, ifelse(side==1, 1, length(p.cutoffs)+1)), 
"RdBu", rev(brewer.pal(length(p.cutoffs)+1,"Reds"))), 
test = "Stouffers", edge.label=TRUE, ignore.edge.length=FALSE)
{
# Creates a file that can be loaded in figtree. The branches are colored in figtree and
# the edges are annotated with the p-value (if edge.label is TRUE)
# User function
# Argument: tree is a phylogenetic tree of class phylo.
# Argument: unsorted.pvalues (can be sorted as well) is a data frame with the tip
# identifier in column 1 and the p-value to be used in column 2
# Argument: adjust is a logical argument controlling p-value adjustment. If FALSE, then
# no p-value adjustment will take place. If true, then side and method will determine
# the type of adjustment.
# Argument: side is either 1 or 2. This corresponds to whether the p-values are 1-sided
# or 2-sided.
# Argument: method is one of the methods found in p.adjust.methods and is the method to
# be used in the p.adjust function.
# Arguments: p.cutoffs is a vector of numbers, x, where 0<x<1 (not including 0 and 1) and
# sorted in ascending order. These are the p value cutoffs. They affect how the edges
# are colored. If pal is a divergent palette (such as RdBu; probably use a divergent
# palette if p-values are 1-sided), the smallest and the largest p values are colored
# the darkest. For example, if p.cutoffs<-c(.01, .05, .10, .90, .95, .99), then the
# ranges are [0-.01), (.01-.05], ..., (.99-1] When using the "RdBu" palette, the
# range 0-.01 is the darkest red shade and .99-1 is the darkest blue shade. A
# sequential palette would probably be used when p-values are 2-sided. The default
# argument for p.cutoffs is c(.01, .05, .10, .90, .95, .99) if side is 1 and
# c(.01, .05, .1) if side is 2.
# Argument: file is the file path to export to.
# Argument: pal is a palate from the package RColorBrewer. Find a list of palettes by
# display.brewer.all() or ?brewer.pal or brewer.pal.info. pal may also be a vector of
# colors. The length needs to be one longer than the length of p.cutoffs (because
# this is how many ranges of colors there are). Colors in this vector need to be in
# hexadecimal format. For example, "#B2182B". #pal=ifelse(side==1, "RdBu", "Reds")
# assigns the default argument of pal to be "RdBu" if side is 1 and "Reds" if side is
# 2. This is because it is probably better to use a divergent palette if using 1-sided
# p-values and a sequential palette if using 2-sided p-values. The default
# argument for pal is "RdBu" when side is 1 and "Reds" (but in reverse) when side is 2.
# Note that the sequential palettes in RColorBrewer range from light to dark. This
# means that the light colors correspond to low p-values. This is why the default
# palette when side is 2 is a reversed version of "Reds", so that the darker reds
# correspond to the lower p-values.

#Argument: test is either "Stouffers" or "Fishers." This is the p-value combination to
# be used.

#Argument: edge.label is a logical argument. When TRUE, the edges in FigTree will have
# annotations with P-values corresponding to the p-value for the node they connect
# that is closer to the root.

#Argument: ignore.edge.length only has an effect if the original tree had edge lengths
# defined. If it did, then setting this argument to FALSE ignores these edge lengths.

#Error checking
if(class(tree)!="phylo")
{
  return(cat("Error: Class of tree must be phylo.
","\n"))
}

if(length(tree$tip.label)!=length(unsorted.pvalues[,1]))
{
  return(cat("Error: There must be the same number of tip labels in tree as in
  unsorted.pvalues.
","\n"))
}else if(mean(sort(as.character(unique(unsorted.pvalues[,1])))==sort(as.character(unique(
tree$tip.label))))!=1)
{
  return(cat("Error: The tip labels in tree must have the same labels as the tip
  labels in unsorted.pvalues.
","\n"))
}

if(min(unsorted.pvalues[,2])<0 | max(unsorted.pvalues[,2])>1)
{
  return(cat("Error: P-values in unsorted.pvalues must be between greater than or
  equal to 0 and less than or equal to 1.
","\n"))
}else
{
  f.one <- 1-100*.Machine$double.eps
  f.zero <- 100*.Machine$double.eps
  t <- unsorted.pvalues[,2]>=f.one
  unsorted.pvalues[t,2] <- f.one
  t <- unsorted.pvalues[,2]<f.zero
  unsorted.pvalues[t,2] <- f.zero
if(adjust!=TRUE & adjust!=FALSE)
{
    return(cat("Error: Value of adjust must be either TRUE or FALSE.","\n"))
}
if(side!=1 & side!=2)
{
    return(cat("Error: Value of side must be either 1 or 2.","\n"))
}
if(is.numeric(method))
{
    return(cat("Error: Value of method should be one of those found in
        p.adjust.methods"))
}
if(max(p.cutoffs)>=1 | min(p.cutoffs)<=0 | is.unsorted(p.cutoffs))
{
    return(cat("Error: The values of p.cutoffs must be greater than 0, less than 1,
        and in ascending order."","\n"))
}
if(length(p.cutoffs)==0)
{
    return(cat("Error: There must be at least one value of p.cutoffs."","\n"))
}
if(length(pal)==1 & is.character(pal))
{
    if(!is.element(pal, rownames(brewer.pal.info)))
    {
        return(cat("Error: pal must be either a palette from RColorBrewer(to see
            a list: rownames(brewer.pal.info)) or a vector of colors."","\n"))
    }
} else if (length(pal)!=length(p.cutoffs)+1)
{
    return(cat("Error: the numbers of colors in pal must be one more than the number
        of values in p.cutoffs."","\n"))
}
if(test!="Stouffers" & test!="Fishers")
{
    return(cat("Error: Value of test must be either "Stouffers"
        or "Fishers"."","\n"))
}
if(test=="Stouffers" & side==2)
{
    cat("Caution: Stouffer's Method is designed for 1-sided p-values.", "\n\n")
}

if(test=="Fishers" & side==1)
{
    cat("Caution: For Fisher’s Method applied to one-tailed p-values, significance
thresholds for small p-values (near 0) are more meaningful than for large
p-values (near 1).", "\n\n")
}

if(edge.label!=TRUE & edge.label!=FALSE)
{
    return(cat("Error: Value of edge.label must be either TRUE or FALSE.","\n"))
}

if(ignore.edge.length!=TRUE & ignore.edge.length!=FALSE)
{
    return(cat("Error: Value of ignore.edge.length must be either TRUE
or FALSE.","\n"))
}

#Use tree to create tree4d, a tree of class phylobase.
phylo4d(tree)->tree4d

#Use tree4d to create tree4dext, a tree of class phyext.
new("phylo4d_ext", tree4d)->tree4dext

#Get n.total.nodes, n.tips, results, and edgecolor from functions.
n.total.nodes <- num.total.nodes(tree)
n.tips <- num.tips(tree)
results <- result(tree, unsorted.pvalues, test, adjust, side, method)
edgecolor <- edge.colors(tree, unsorted.pvalues, p.cutoffs, pal, test, adjust,
    side, method)

#Make a new column (column 2) in results that gives us the edge to the left of the
# current node (ie. the row number). Edge to the left refers to the edge
# connecting to the immediate ancestor. We use the which.edge function to find
# this immediate ancestor. Then label this column "Edge to Left."
for(i in 1:n.total.nodes)
{
    results[i,2] <- which.edge(tree,i)
}

names(results)[2]<-"Edge to Left"

#Create a data frame called nodecolor filled with NA's and having n.total.nodes rows.
nodecolor<-data.frame(matrix(NA, nrow=n.total.nodes))
#it doesn't look like i need to include t here.
#t <- !is.na(results[,2])
#nodecolor[,1] <- edgecolor[results[,2]]

#Assign edgecolor[results[,2]] to the first column of nodecolor. results[,2] is the
# edge to the left of the current node (ie. row number)
nodecolor[,1] <- edgecolor[results[,2]]

#Create p.cutoffs.new, which is p.cutoffs with a 1 appended to the end. Create
# n.cutoffs, which is the length of p.cutoffs.new. cols is a vector of color
# values from RColorBrewer based on n.cutoffs and pal.
p.cutoffs.new <- c(p.cutoffs, 1)
n.cutoffs <- length(p.cutoffs.new)
if(length(pal)==1)
{
  cols <- brewer.pal(n.cutoffs, pal)
} else cols <- pal

#This is only for the branch that is to the left(away from the tips) of the root. This
# has not yet been handled, and will be black in figtree unless we do this for loop.
# I am assigning the same color to this branch as to the two edges that are connected
# to the root node. This part is similar to the for loop in the edgecolor function.
for(k in n.cutoffs:1)
{
  if(results[n.tips+1,1]<=p.cutoffs.new[k]) nodecolor[n.tips+1,1]<-cols[k]
}

#to get p-value for node to the right of current node. only happens if they set
# edge.label=TRUE i.e., they want a p-value annotation for each edge.
#If edge.label=TRUE, we do this. Create a data frame called node.pval full of
# NA's and with n.total.nodes rows. Set results$tree$edge[results[,2],1] to be
# the first column. results[,2] is the edge to the left of the current row (node).
# For a given row, tree$edge[results[,2],1] is the node in the first column (i.e.,
# parent node) of tree$edge that corresponds to the edge from results[,2], i.e., it
# is the parent node of the current node(row). Then
# results$tree$edge[results[,2],1,1] is the p-value of the direct
# ancestor/parent/left node of the given direct descendant/child/right node (the
# current row number). Assign this to node.pval's first column. We do all of
# this because of how figtree handles its labels. Figtree's annotations are
# attached to the edges, not the nodes. We want to be able to highlight an edge
# and find the p-value of the node to the left, ie the parent node. Because of
# the way figtree's annotations work, we need to assign this p-value to the child
# node of the edge. That is why we need to assign node.pval this way.
if(edge.label)
node.pval <- data.frame(matrix(NA, nrow=n.total.nodes))
node.pval[,1] <- results[tree$edge[results[,2],1],1]
# fill in p-value for the branch coming out of the root (away from the tips).
# The p-value here is the same as the p-value for the two children branches
# from the root.
node.pval[n.tips+1,1] <- results[n.tips+1,1]
names(node.pval)[1] <- "pvalue of node to right"

# Use gsub and regular expressions to get rid of the "#" in front of the hexadecimal
# color. Place in column 2.
gsub("#", "", nodecolor[,1]) -> nodecolor[,2]
# Use strtoi to convert the hexadecimal color to a decimal color. Place in third
# column of nodecolor.
strtoi(nodecolor[,2], 16) -> nodecolor[,3]
# Label the columns of nodecolor
names(nodecolor)[1]<-"Hexadecimal color"
names(nodecolor)[2]<-"Hex. color w/o #"
names(nodecolor)[3]<-"Decimal color"

# Put information in tdata(tree4dext), which is extra information about our tree. If
# edge.label==FALSE, we only put nodecolor[,3] (the decimal color information) in
# tdata(tree4dext). But if edge.label==TRUE, we first create a new data frame
# called df which contains nodecolor[,3] in the first column and node.pval[,1] in
# the second column and name those columns "nodecolor" and "pvalue." Then we assign
# df to tdata(tree4dext).
if(edge.label)
{

df <- data.frame(nodecolor[,3], node.pval[,1])
names(df)[1:2] <- c("nodecolor", "pvalue")
tdata(tree4dext) <- df
}
else tdata(tree4dext) <- nodecolor[,3]}

# This code with op.warn is to suppress the warning messages. After we have run the
# line to get temptext, we reset the warn option to what it originally was. Similar
# with op.scipen. Save the original parameter as op.scipen. It has to do with when
# R prints things in scientific notation. We need it to not use scientific notation
# in order for the regular expressions to work.
op.warn <- options()$warn
options(warn=-1)
op.scipen <- options()$scipen
options(scipen=1000)
# Create temptext, which is the text of write.nexus.simmap of tree4dext (in a vector # format). If edge.label is TRUE, then we use version 1.5. This is because # tdata(tree4dext) contains two files and version 1.5 handles this. If edge.label is # FALSE, we only have one column of tdata(tree4dext) and we can use the older # version. The regular expression code is different based on the value of edge.label.

if(edge.label) capture.output(write.nexus.simmap(tree4dext, vers=1.5), file = NULL, 
  append = FALSE) -> temptext
else capture.output(write.nexus.simmap(tree4dext), file = NULL, 
  append = FALSE) -> temptext

# Set the options back to what they were before we changed them.
options(warn=op.warn, scipen=op.scipen)
# Create length.temptext, which is simply the length of temptext.
length(temptext) -> length.temptext

# If edge.label==TRUE, we use this regular expression statement, which evaluates # temptext[length.temptext-1]. This is the only part of temptext that we need to use # regular expressions on. We make it be in a format that figtree can read and use # our color information. We also need to test to see if tree4dext has edge lengths # (if not, they are all NA's and show up as 0's). If they do, we need to include this # in the exported file for Figtree. If not, we don't include the 0. If # ignore.edge.length is true, we go to the else statement that doesn't include the # edge lengths.

if(edge.label)
{
  if(hasEdgeLength(tree4dext) & ignore.edge.length==FALSE)
  {
    temptext[length.temptext-1] <-
    gsub("\[(\d+)\], \d+", 
      \[(\d+)\], \d+",
      ">
    ">
    ">
    temptext[length.temptext-1]
  }
  else
  {
    temptext[length.temptext-1] <-
    gsub("\[(\d+)\], \d+", 
      \[(\d+)\], \d+",
      ">
    ">
    ">
    temptext[length.temptext-1]
  }
}
If `edge.label==FALSE`, then we go to this else statement, which uses regular expressions to change `temptext[length.temptext-1]` to be in a format that Figtree can read and use the color information. We also need to test to see if `tree4dext` has edge lengths (if not, they are all `NA's` and show up as `0's`). If they do, we need to include this in the exported file for Figtree. If not, we don't include the `0`. If `ignore.edge.length` is true, we go to the else statement that doesn't include the edge lengths.

```r
else {
  if(hasEdgeLength(tree4dext) & ignore.edge.length==FALSE) {
    temptext[length.temptext-1] <-
    gsub("\{([0-9.]+),\([0-9.]*\)\}\", "[&!color=\1]:\2", temptext[length.temptext-1])
  }
  else {
    temptext[length.temptext-1] <-
    gsub("\{([0-9.]+),\([0-9.]*\)\}\", "[&!color=\1]", temptext[length.temptext-1])
  }
}
write(temptext, file=file)
```