Setting Up and Running PHRAPL

by Nathan Jackson

Last updated on March 04, 2019

PHRAPL is a phylogeographic model selection method based on approximate likelihoods. This method estimates the probability of observing a set of gene trees under a model by calculating the frequency at which observed tree topologies occur in a distribution of expected tree topologies. The relative probability of models within a set can be assessed using Akaike information criterion (AIC). Because the method uses gene tree topologies only (excluding branch lengths), it can, relatively quickly, compare the fit of a broad range of models that include coalescence times, migration rates, and distinct/fluctuating population sizes, potentially all acting simultaneously.

The purpose of this vignette is to provide a brief tutorial for using this method. It will walk you through the necessary steps required for installing the package, preparing a dataset and model set using R, running an analysis, and summarizing results, all using a toy dataset. There are many options that one can persue when running a PHRAPL analysis, which could get a bit confusing for someone using the program for the first time. For this reason, we have highlighted with red asterisks those blocks of code that constitute the minimum steps that must be taken to run a PHRAPL analysis using the toy dataset.

This tutorial will assume that the reader already has a basic understanding of how PHRAPL works and the problems it is meant to address. The focus here will rather be on the practical issues surrounding the analysis of a dataset. Thus, prior to analyzing an empirical dataset, users should read the original paper that describes the principles and methodology behind PHRAPL:

```
Jackson, ND, AE Morales, BC Carstens, and BC O'Meara. 2017. PHRAPL: Phylogeographic Inference
Using Approximate Likelihoods. Systematic Biology (in press).
```

This tutorial assumes you are working in a Mac/Linux environment. We have yet to properly test the method using a PC, although we suspect a few (surmountable) issues will arise when the sufficiently adventurous try this out. For more detailed information about any of the tasks discussed below, take a look at the help files within PHRAPL (for a list of these, type `library(help=phrapl)` once the package is installed).

I. Installing PHRAPL

PHRAPL can currently be found on Github (`https://github.com/bomeara/phrapl`), although you don't actually need to navigate Github to install the program. You can simply install PHRAPL in R using the R package `devtools`. To do this, first install `devtools` from CRAN by typing

****

```
install.packages("devtools")
```

****

Then load the `devtools` library (i.e., `library(devtools)`) and type

****

```
devtools::install_github("bomeara/phrapl")
```

****

which grabs PHRAPL from Github and installs it.

Note that most of PHRAPL is written in R. However, the part of the code that matches observed trees to trees expected under a given model is currently written in Perl in order to speed up the method. If you are running PHRAPL on a Mac or Linux machine, Perl has most likely already been installed (to make sure, within R, you can type `system("perl -v")`, which will give you the version installed, if it exists). If Perl is not installed on your machine, you can download it from the Perl website (`http://www.perl.org/get.html`).

One final note. It has come to our attention that one of the R packages that PHRAPL imports, `diagram`, includes an error in it's DESCRPTION file such that `devtools` can't install it. Thus, if you are thrown an error upon trying to install PHRAPL using `devtools` (something along the lines of `Invalid comparison operator in dependency: >=`, try installing `diagram` first without `devtools`, by typing `install.packages("diagram")`. Then, try installing PHRAPL again using the `devtools` command.

Also, for Mac users, if you experience issues with `rgl` properly installing, another PHRAPL dependency, you should try updating your version of XQuartz (`https://www.xquartz.org/`), then reinstalling `rgl` separately in R (`install.packages("rgl")`). Then install PHRAPL.

Now PHRAPL is ready to run. Type `library(help=phrapl)` to get a list of functions with documentation. To open a help file for a particular function, type `?function_name`.

II. Importing your dataset into R

Three R objects must be initially specified to run a PHRAPL analysis:

1. A set of trees in newick format
2. A table that assigns tips of the trees to populations or species
3. A vector specifying the number of tips to subsample per population

1. Importing trees

If you are beginning with sequence data, note that PHRAPL includes a function for inferring gene trees from sequence data by calling up RAxML (type `?RunRaxml` for more information on using this function). If your sequence data is in nexus format, there is also a function for converting your data to phylip format, which is the required format for running RAxML (type `?RunSeqConverter`). This function calls up a Perl script written by Olaf R.P. Bininda-Emonds.

For the purposes of this tutorial, I will assume that you already have a set of trees in newick format sitting in a text file called "trees.tre". To bring these trees into R, first load the `ape` library:

****

```
library(ape)
```

****

Then read the trees into R and create a multiPhylo tree object called `trees`:

```
trees<-read.tree("trees.tre")
```

PHRAPL comes packaged with a toy dataset that we will use during this tutorial. This dataset consists of 10 trees, each with 61 tips: 20 tips from each of three populations (“A”, “B”, and “C”), plus one outgroup individual. To bring this dataset into R, type

****

```
trees<-read.tree(paste(path.package("phrapl"),"/extdata/trees.tre",sep=""))
```

****

Note that multiPhylo objects behave as R lists. Thus to access the first tree within the object, type

```
trees[[1]]
```

To plot this tree, type

```
plot.phylo(trees[[1]])
```

2. Importing an assignment table

PHRAPL assumes that the assignment of tips to populations or species is known. Thus these assignments must be specified upfront in the form of a table. This population assignment table must consist of two columns: the first column lists the individuals in the dataset, whose names must match those at the tips of the trees. Note that not all the individuals listed in the table must exist on every tree (i.e., missing data/unique tip names for each tree are fine). The second column should provide the population or species name to which each individual is assigned (e.g., “A”, “B”, “C”). If there is an outgroup taxon, it MUST be listed as the last population in the table and the first letter in the population name should also come last alphanumerically (e.g., `"Z.outgroup"`). A header row of some sort must also be included.

You can of course create this table directly in R as a data.frame. If you’ve created the assignment table as a tab-delimited text file (e.g., “cladeAssignments.txt”), you can import it into R as a data.frame by typing

```
assignFile<-read.table("cladeAssignments.txt",header=TRUE,stringsAsFactors=FALSE)
```

To import an assignment table for our toy dataset, type

****

```
assignFile<-
read.table(paste(path.package("phrapl"),"/extdata/cladeAssignments.txt",sep=""),
    header=TRUE,stringsAsFactors=FALSE)
```

****

3. Specifying the number of tips to subsample (popAssignments)

Rather than fitting datasets to models using all the tree tips simultaneously, PHRAPL uses iterative subsamples of tips, allowing for the method to work in a manageable tree space. Thus, the number of tips to subsample per population must be specified by the user.

A `popVector` is a vector, whose length is equal to the number of populations in the dataset and whose values are equal to the number of subsampled tips. So, for example, if you are subsampling 3 tips from a dataset that contains 3 populations, then `popVector = c(3,3,3)`. If subsampling 4 tips from 2 populations, `popVector = c(4,4)`.

Because PHRAPL possesses the ability to analyze a dataset under a series of different subsampling regimes, `popAssignments`, which is simply a list of `popVectors`, is the object that users actually specify (although `popAssignments` will typically be a list of one `popVector`). So, if the desired `popVector = c(3,3,3)`, then `popAssignments` should be defined by typing

****

```
popAssignments<-list(c(3,3,3))
```

****

which is the value we will use for our toy dataset.

We recommend generally subsampling 3 or 4 tips per population. However, subsampling 2 tips may be necessary with more than 5 populations while subsampling 5 tips may work fine with 3 or fewer populations. As a rule of thumb, we recommend subsampling in a way that keeps the sum of `popAssignments` at 16 or less. With greater than 16 tips, the number of possible trees becomes so large that PHRAPL has a difficult time simulating enough trees under a given model to find any matches to the empirical trees, rendering log-likelihood (lnL) estimation challenging.

III. Subsampling trees

As discussed above, approximate likelihood calculation as currently implemented in PHRAPL requires that trees not be too large (~16 tips or fewer) such that there exists a measurable chance of observing the empirical tree within a distribution of simulated trees. Since most phylogeographic datasets are considerably larger than this, replicate subsamples of trees must typically be analyzed. Trees are subsampled (randomly, with replacement) using the function `PrepSubsampling`, which requires the following arguments:

1. `assignmentsGlobal` (the population assignments table)
2. `observedTrees` (the original trees)
3. `popAssignments` (the number of tips subsampled per population)
4. `subsamplesPerGene` (the number of replicate subsamples to take per locus)
5. `outgroup` (whether an outgroup is present in the dataset, i.e., `TRUE` or `FALSE`)
6. `outgroupPrune` (whether an outgroup, if present, should be excluded from the subsampled trees).

To subsample our toy dataset using 10 replicates and 3 tips per population, do the following:

****

```
   assignmentsGlobal<-assignFile
   observedTrees<-trees
   popAssignments<-list(c(3,3,3))
   subsamplesPerGene<-10
   outgroup=TRUE
   outgroupPrune=TRUE

   observedTrees<-
 PrepSubsampling(assignmentsGlobal=assignmentsGlobal,observedTrees=observedTrees,
      popAssignments=popAssignments,subsamplesPerGene=subsamplesPerGene,outgroup=outgroup,
      outgroupPrune=outgroupPrune)
```

****

Note that this function produces a list of subsampled tree sets to match the format of popAssignments (i.e., one set per popVector). Thus, if you have a single popVector, the PrepSubsampling function will output a list with a single set of trees. This set of subsampled trees first contains all the subsample iterations for the first locus, then all the iterations for the second locus, and so forth. Thus, for example, for our observedTrees we just generated, we can access the first subsample iteration of the second locus by typing observedTrees[[1]][[11]].

Note also that the more tips that exist in the original tree, the larger the number of subsample iterations that must be taken in order to capture a representative sample of the tree. One rule of thumb is to take AT LEAST as many subsample iterations that would be required to sample each tip once if subsampling were being done without replacement. So, for our current dataset, with three tips being taken per population, this would require that subsamplesPerGene be at least 7 (3 subsampled tips x 7 iterations > 20 total tips for a given population in the toy tree). However, increasing the number of subsamples to 2, 3, or more times this number will likely reduce the error in the likelihood estimate for a given tree. We suggest taking as many subsample iterations as is logistically feasible, although keeping in mind that there will be a point of diminishing returns that should be weighed against other specifications of the analysis (e.g., the number of models to run, the number of parameters to specify, the number of trees to simulate, etc.).

Finally, keep in mind that PHRAPL requires that trees be rooted, but not that the outgroup be included in the analysis. The purpose of including an outgroup taxon is merely to ensure that the subsampled trees are properly rooted. Thus, typically if an outgroup is present, outgroupPrune should be set to TRUE. If an outgroup is not available, one could midpoint root the subsampled trees. One way to do this in R is to use the midpoint function within the phangorn package. For example,

```
library(phangorn)
observedTreesMidpoint<-lapply(observedTrees[[1]],midpoint)
class(observedTreesMidpoint)<-"multiPhylo"
observedTreesMidpoint<-list(observedTreesMidpoint)
```

IV. Calculating degeneracy weights for subsampled trees

As a way of reducing the tree space that PHRAPL faces when calculating the probability of a gene topology, all tip labels **within** populations are essentially ignored when assessing matches between simulated and observed trees. If the only differences between two trees consist of intra-population discrepancies, then these trees are considered to be "matches". To adjust lnLs in a way that accounts for this degeneracy of intra-population tip labels, topological weights are calculated based on the proportion of times that the intra-population permuting of labels across a tree results in the same topology. These weights can be calculated during the PHRAPL search analysis - to do this, when running GridSearch (described below), set subsampleWeights.df = NULL and doWeights = TRUE. However, when there are more than a few tips in the subsampled trees, this can be time consuming, and thus it is helpful to calculate these upfront. To get these weights beforehand, you can use the function GetPermutationWeightsAcrossSubsamples, which takes as input popAssignments and the set of subsampled trees. For our toy dataset, type

****

```
subsampleWeights.df<-GetPermutationWeightsAcrossSubsamples(popAssignments=popAssignments,
    observedTrees=observedTrees)
```

****

As with PrepSubsampling, this function produces a list of weights tables with the same length as popAssignments. Within a weights table, each weight corresponds to a subsampled tree.

V. Generating models

The last thing we need prior to analyzing the dataset is a set of models to test. This set of models can be constructed “one at a time”, if you have specific **a priori** histories to compare, or a set of “all possible models” can be generated automatically given a specified list of criteria. Or of course you can combine these approaches. This section first describes the structure of models in PHRAPL and then outlines how to go about generating a model set using these two approaches.

1. Some background on model structure

A model analyzed by PHRAPL is called a migrationIndividual, which contains matrices that define the unique coalescence (collapseMatrix), population size (n0multiplierMap), population growth (growthMap), and migration (migrationArray) parameters included in that model. A set of models (confusedly) is also called a migrationArray, which is an R list of migrationIndividuals. All four parameter matrices must be included within a migrationIndividual, even if a parameter type is excluded from the model.

The parameter units used in PHRAPL are the same as those used in the program ms:

1. Coalescence time (**t**) is relative to the present time and is in units of 4**N** (where **N** = the diploid population size)
2. The population size scalar (**n**) is a multiplier of **N**
3. Exponential growth rate (**g**) is in units of α, as defined in the equation $\mathbf{Ne}^{-\alpha \mathbf{t}}$, where a positive α produces population growth (moving toward the tips) and a negative α produces population contraction.
4. Migration rate (**M**) is in units of 4**Nm**, where **m** equals the number of migrants per generation.

Zero, one, or multiple free parameter values can be established for each parameter type within a model, where parameter indices are labeled consecutively as 1, 2,…**K** for **K** free parameters being specified for a given type of parameter. For example, if the collapseMatrix is

```
     [,1] [,2]
[1,]    1    2
[2,]    1   NA
[3,]    0    2
```

where rows are populations and columns are coalescence events, chronologically ordered (tip to root) from left to right, two free coalescence time parameters exist: **t1** for coalescence between populations 1 and 2, and **t2**, for coalescence between ancestral populations 1-2 and 3.

Similarly, if the corresponding migration matrices are

```
, , 1

     [,1] [,2] [,3]
[1,]   NA    1    0
[2,]    2   NA    0
[3,]    0    0   NA

, , 2
```

```
     [,1] [,2] [,3]
[1,]   NA   NA    3
[2,]   NA   NA   NA
[3,]    3   NA   NA
```

where matrix 1 is for the present generation and matrix 2 is for the ancestral generation, three free migration parameters exist: **M1**, from population 1 into population 2, **M2**, from population 2 into population 1, and **M3**, which defines symmetrical migration rates between ancestral populations 1-2 and 3. The number of migration matrices must always match the number of columns in the `collapseMatrix`.

Note that the matrix dimensions and population histories for population size and growth matrices must also always match those of the `collapseMatrix`. For example, if there are no population growth or population size differences within the model specified above, the `n0multiplierMap` would be

```
     [,1] [,2]
[1,]    1    1
[2,]    1   NA
[3,]    1    1
```

and the `growthMap` would be

```
     [,1] [,2]
[1,]    0    0
[2,]    0   NA
[3,]    0    0
```

2. Generating a set of models

One advantage of PHRAPL is that it can generate and analyze a large set of models relatively quickly. A model set (`migrationArray`) consisting of all possible models, given certain constraints such as the number of free parameters, **K**, available overall (set by `maxK`), or available for a particular parameter (set by `maxN0K`, `maxGrowthK`, and `maxMigrationK`), can be generated using the `GenerateMigrationIndividuals` function. This generates a model set that includes all possible combinations of parameter indices, given the constraints.

For example, a list of models for 3 populations with

1. An overall maximum K of 3 (`maxK = 3`)
2. A maximum number of migration parameters of 1 (`maxMigrationK = 1`)
3. No variation in population size among populations (`maxN0K = 1`)
4. Only fully resolved trees (`forceTree = TRUE`)
5. Migration set to be symmetrical between populations (`forceSymmetricalMigration = TRUE`)

can be generated for use with our toy dataset as follows:

****

```
popVector<-popAssignments[[1]]
maxK<-3
maxMigrationK=1
maxN0K=1
maxGrowthK=0
forceTree=TRUE
forceSymmetricalMigration=TRUE
```

```
migrationArray<-GenerateMigrationIndividuals(popVector=popVector,maxK=maxK,
    maxMigrationK=maxMigrationK,maxN0K=maxN0K,maxGrowthK=maxGrowthK,
    forceTree=forceTree,forceSymmetricalMigration=forceSymmetricalMigration)
```

****

This yields a `migrationArray` containing 48 models. If desired, one can also specify a particular set of fixed parameter indices (using the `collapseList`, `n0multiplierList`, `growthList`, and `migrationList` arguments) such that only a subset of parameters will be varied in the outputted `migrationArray`. For more on this, see the R documentation (`?GenerateMigrationIndividuals`).

A word of warning: if significant constraints are not placed on the number of parameters and populations specified when using `GenerateMigrationIndividuals`, then you can easily generate a list of models that will overwhelm your hard disk space, let alone your ability to analyze them. For example, with just four populations, assuming you are only considering fully resolved topologies (`forceTree=TRUE`), there are 18 possible coalescence histories. Taking just one of these histories and adding a single migration rate parameter yields a list of 1,024 models. Multiply this by 18 and you have over 18,000 models to analyze, with just `maxMigrationK=1` (and with `maxGrowthK` and `maxN0K = 0`). By just adding in two different possible population sizes (setting `maxN0K = 2`), this number of models jumps up to > 82,944,000, which would be difficult to analyze to say the least.

Given that `migrationArray`s can take a while to generate when they are large, we have pre-generated some that come packaged with PHRAPL. To view the `migrationArray`s available, type

```
unlist(strsplit(grep("MigrationArray",list.files(paste(path.package("phrapl"),
    "/data/",sep="")),value=TRUE),".rda"))
```

Then to load one of these `migrationArray`s into your current environment, type

```
data("Name_of_migrationArray")
```

You can type `?CachedMigrationArrays` to get more information about the specifications for each `migrationArray`.

Note that the imported object will always be called `migrationArray` and will thus override any previous `migrationArray` that exists in your current R session.

If you have a `migrationArray` produced under a previous version of PHRAPL that did not model population growth, you will need to add growth matrices to these models prior to running them with the current version. To add a null growth statement (i.e., zero growth) to each model, use the function `AddGrowthToAMigrationArray`, e.g.:

```
migrationArray<-AddGrowthToAMigrationArray(migrationArray)
```

---

3. Generating a single a priori model

What if, instead of or in addition to generating a list of possible models, you would like to create a specific model for which you have some **a priori** reason to consider alone, or to add to a larger `migrationArray` as generated above? To do this, you can use the function `MakingMigrationIndividualsOneAtATime`, whose purpose is to create a single **a priori** `migrationIndividual` for a specified coalescence, n0multiplier, growth, and migration history. The four corresponding arguments for this function are `collapseList`, `n0multiplierMapList`, `growthList` and `migrationList`.

`collapseList` is a list of coalescence history vectors, one vector for each coalescent event in the tree (i.e., for each column in the `collapseMatrix`. So,

```
collapseList = list(c(1,1,0),c(2,NA,2))
```

means that there are two coalescence events: in the first event, populations 1 and 2 coalesce while population 3 does not; in the second event, ancestral population 1-2 coalesces with population 3.

`n0multiplierList` and `growthList` are given in the same format as `collapseList`, and the available parameters for this must match the splitting history depicted in the `collapseList`. Remember that the `n0multiplier` parameter is not a parameter for population size, but is rather a population size scalar. So, a model in which population sizes are equal across populations (contemporary and ancestral) would be given as

```
n0multiplierList = list(c(1,1,1),c(1,NA,1))
```

If you would like to model one growth rate for tip populations and another growth rate for ancestral populations, this would be given as

```
growthList = list(c(1,1,1),c(2,NA,2))
```

Finally, `migrationList` is a list of migration matrices for the model. There will be one migration matrix for the present generation, plus any historical matrices that apply (there will be as many matrices as there are collapse events). So, in the three population scenario, if there is symmetrical migration between populations 1 and 2 and no historical migration, migrationList will be:

```
migrationList<-list(
t(array(c(
NA, 1, 0,
1, NA, 0,
0, 0, NA),
dim=c(3,3))),

t(array(c(
NA, NA, 0,
NA, NA, NA,
0,  NA, NA),
dim=c(3,3))))
```

Note that in R, arrays are constructed by reading in values from column 1 first then from column 2, etc. However, it is more intuitive to construct migration matrices by rows (i.e., first listing values for row 1, then row 2, etc.). Thus, here, arrays are typed in by rows and then transposed (using `t`). Also, spacing and hard returns can be used to visualize these values in the form of matrices.

So, let's say you want to add one more model to our 48-model `migrationArray`. Assume that this desired model is the same as model #1 (i.e., `migrationArray[[1]]`, where ((pop1, pop2) pop3))), but where asymmetrical migration rates between sister populations are established at the tips: one rate for pop1 –> pop2 and another rate for pop2 –> pop1. This `migrationList` would be given as

```
migration_1<-t(array(c(
NA, 1, 0,
2, NA, 0,
0, 0, NA),
dim=c(3,3)))

migration_2<-t(array(c(
NA, NA, 0,
NA, NA, NA,
0,  NA, NA),
dim=c(3,3)))

migrationList<-list(migration_1,migration_2)
```

So, to produce our desired model, we use the function `GenerateMigrationIndividualsOneAtATime`:

```
migrationIndividual<-GenerateMigrationIndividualsOneAtATime(collapseList=collapseList,
    n0multiplierList=n0multiplierList,growthList=growthList,migrationList=migrationList)
```

Then, we can add this model to the `migrationArray`:

```
migrationArray<-c(migrationArray,list(migrationIndividual))
```

and view it:

```
migrationArray[[49]]
```

Note that a `collapseList` must always be specified. However, the remaining three parameter matrices are optional, and if they are not specified, null matrices will be automatically produced in which all n0multipliers are set to one, and all growth and migration parameters are set to zero.

Thus, the same `migrationIndividual` as above can be produced by simply typing

```
migrationIndividual<-GenerateMigrationIndividualsOneAtATime(collapseList=collapseList,
    migrationList=migrationList)
```

4. Adding demographic events to a model that are not tied to a coalescence event

One can add additional complexity to generated models using the function `AddEventToMigrationArray`. This function can add additional demographic shifts over time that do not correspond to a splitting event. For example, if one would like to posit a divergence event that occurs when populations are in allopatry, followed by the resumption of migration at a latter time upon secondary contact, using `AddEventToMigrationArray`, one can add a new migration matrix to an existing model that is applied prior to the coalescence event. For details about how to add such demographic events that are not tied to a coalescence events to a model, see the R documentation (`?AddEventToMigrationArray`).

VI. Running a PHRAPL grid search

Once you have a model set in hand and the data have been subsampled and weights have been calculated, you are finally ready to calculate AIC values and parameter estimates for your dataset under the models using the `GridSearch` function.

This function is called `GridSearch` because it analyzes each model across a grid of parameter values as a way to estimate those values that yield the best fit to the data. Numerical optimization searches are also available within PHRAPL (type `?GridSearch` for more on these), but we don’t recommend using them due to their being more computationally onerous without imparting a compensatory improvement in performance. The parameter values contained in the grid can be specified by the user using the arguments `collapseStarts`, `n0Starts`, `growthStarts`, and `migrationStarts`, or default values can be used (to see these default values, type `?GridSearch`). When choosing grid values, keep in mind that the grid size, and thus number of simulations that PHRAPL must do, increases rapidly with each added parameter value, and thus there will be tradeoffs if you want to explore a large grid. Also, a model with many parameters will need to be analyzed using fewer specified values per parameter compared to a model with only one or two parameters in order to yield a parameter grid of the same size.

The main inputs for `GridSearch` are

1. `migrationArray` (the model set)
2. `observedTrees` (the set of subsampled trees)
3. `subsampleWeights.df` (the topology weights)
4. `popAssignments` (the number of tips subsampled per population)
5. `nTrees` (the number of trees to simulate for each point in a model parameter grid)
6. `subsamplesPerGene` (the number of subsample iterations)

A few things to note: `nTrees` should be set to 10,000 at a minimum and should be as high as logistically feasible (computational time increases linearly with increasing `nTrees`). Increasing `nTrees` can increase the accuracy and consistency of approximate lnLs.

If you only want to run a subset of the models within `migrationArray`, you can specify this using `modelRange` (e.g., `modelRange = 1:5` will run only the first five models). This is useful if you wish to break up analyses across a computer cluster. If a single model will take a long time to run, you can also set `checkpointFile = TRUE`, which will cause results to be printed to a checkpoint file throughout the analysis, such that, if the analysis needs to be terminated prior to completion, one can resume the analysis later from the most recently analyzed grid point.

If the dataset being analyzed contains loci that differ in their prevalence in a population (e.g., mix of diploid/haploid or organellar/nuclear loci), the relative scaling of effective population size should be inputted using `popScaling`. This argument takes a vector of scaling values, one value for each locus in the dataset (e.g., where a diploid nuclear locus = 1, X-linked locus = 0.75, mtDNA/chloroplast locus = 0.25, etc.).

To run a grid search on our toy dataset, let's just analyze the first three models using `nTrees = 1000` (so it runs quickly):

****

```
  modelRange<-1:3
  nTrees<-1000

  result<-
GridSearch(migrationArray=migrationArray,modelRange=modelRange,popAssignments=popAssignments,
    observedTrees=observedTrees,subsampleWeights.df=subsampleWeights.df,
    subsamplesPerGene=subsamplesPerGene,nTrees=nTrees)
```

****

`GridSearch` outputs a list containing

1. Parameter grids (with AICs) for each model (itself a list)

2. A results table that gives AICs, lnLs, **K**s (`params.K`), parameters (`params.vector`), and parameter estimates for each model

Grid points with an `AIC = NA` signal parameter combinations that did not yield enough matches between observed and expected trees to calculate a lnL. Parameter estimates equal to `NA` occur when those parameters are not included in the model. Parameter estimates are obtained by model averaging across the grid. Parameters for **t**, **n**, **g**, and **M** are given for each time slice in the model, starting at the tips and moving toward the root (`t1` = coalescence time at time 1, `m2` = ancestral migration rates at time 2, etc.). For **t**, the coalescing populations follow the underscore and are separated by a period (e.g., `t2_1-2.3` gives the coalescence time for ancestral populations 1-2 and 3 at time 2); for **m**, migrants move from the population before the period to the population after the period (e.g., `m1_2.1` gives the rate of migration at the tips, from population 2 into population 1).

VII. Saving, compiling, summarizing, and visualizing results

The `ConcatenateResults` function can compile results from across separate runs into a single table. If you are analyzing models in separate runs, you will want to save `GridSearch` results in a file to access later, e.g.:

****

```
save(list="result",file="phraplOutput_models1-3.rda")
```

****

IMPORTANT NOTE: The results saved from a GridSearch should literally be named “result”, for that is the object name that the next function discussed (ConcatenateResults) looks for when summarizing GridSearch results from across different runs.

When concatenating results, you can either specify the names of the files containing the results to be concatenated (by giving a vector of names to the argument `rdaFiles`) or the directory in which these files are stored (using `rdaFilesPath`). So, for our toy dataset, type

****

```
totalResults<-ConcatenateResults(rdaFiles="phraplOutput_models1-3.rda",
    migrationArray=migrationArray)
```

****

This table can be outputted to a file by specifying an output filename using the `outFile` argument.

You can calculate model averaged parameter values across all models using the `modelAverages` function, e.g.:

****

```
modelAverages<-CalculateModelAverages(totalResults)
```

****

The default is to model average parameters across all models (assuming that the parameter = 0 when it is not specified in a model). Alternatively, one can model average across only those models in which a given parameter is explicitly incorporated by setting `averageAcrossAllModels = FALSE`.

Upon inspecting the results, you may feel inclined to create a 3-D image of the best model, e.g.:

```
PlotModel(migrationIndividual=migrationArray[[1]],taxonNames=c("A","B","C"))
```

A statuette of this model can then be printed using 14 carat gold, which could be useful as a project souvenir/retirement nest egg, e.g.:

```
http://www.shapeways.com/product/BHPZB3WUC/phrapl-four-populations?optionId=40165805
```

VIII. Testing species delimitation hypotheses

Finally, if the question of interest pertains to the delimitation of putative species that are specified in the dataset, then for a given hypothesized tree, you will want to compare the fit of a “full tree” model for which all coalescence times are estimated (i.e., are non-zero) to “collapsed” models for which one or more coalescence times are set to be zero. To do this, rather than adding additional models to a `migrationArray`, these collapsed models should be specified when running a `GridSearch`. Thus, for a given set of models in a `migrationArray` that specify fully resolved trees, the collapsing of specific nodes can be specified using `GridSearch`s `setCollapseZero` argument. This argument inputs a vector that defines which coalescence time parameters in a model one would like to be set to zero. So, for our toy dataset, setting `setCollapseZero = 1` when fitting an ((A,B),C) isolation-only model will effectively collapse populations A and B into a single population; setting `setCollapseZero = 1:2` will simulate a single panmictic population.

Below is an example for how to test the existance of 1, 2, or 3 species using the first three models run above using the toy dataset. I have increased the number of nTrees to 10000 such there are enough simulated trees to calculate the log-likelihood for all the models. These runs will take a few minutes.

First, set relevant parameters:

```
modelRange<-1:3
nTrees<-10000
```

Then run and save analyses for three species models:

```
result<-GridSearch(migrationArray=migrationArray,modelRange=modelRange,
    popAssignments=popAssignments,observedTrees=observedTrees,
    subsampleWeights.df=subsampleWeights.df,subsamplesPerGene=subsamplesPerGene,
    nTrees=nTrees)
save(list="result",file="phraplOutput_models1-3_3species.rda")
```

Run and save analyses for two species models:

```
result<-GridSearch(migrationArray=migrationArray,modelRange=modelRange,
    popAssignments=popAssignments,observedTrees=observedTrees,
    subsampleWeights.df=subsampleWeights.df,subsamplesPerGene=subsamplesPerGene,
    nTrees=nTrees,setCollapseZero=1)
save(list="result",file="phraplOutput_models1-3_2species.rda")
```

And then run and save analyses for single species models:

```
result<-GridSearch(migrationArray=migrationArray,modelRange=modelRange,
    popAssignments=popAssignments,observedTrees=observedTrees,
    subsampleWeights.df=subsampleWeights.df,subsamplesPerGene=subsamplesPerGene,
    nTrees=nTrees,setCollapseZero=1:2)
save(list="result",file="phraplOutput_models1-3_1species.rda")
```

Concatenate results:

```
totalResults<-ConcatenateResults(migrationArray=migrationArray)
```

And model average parameters

```
modelAverages<-CalculateModelAverages(totalResults)
```

Finally, you can calculate the genealogical divergence index (gdi) between populations A and B using the model averaged parameter values of migration rate and divergence time. This index is a composite metric that estimates overall divergence (between 0 and 1) from the combined effects of genetic drift and gene flow (0 = panmixia; 1 = strong divergence).

```
gdi<-CalculateGdi(modelAverages$t1_1.2,modelAverages$m1_1.2)
```