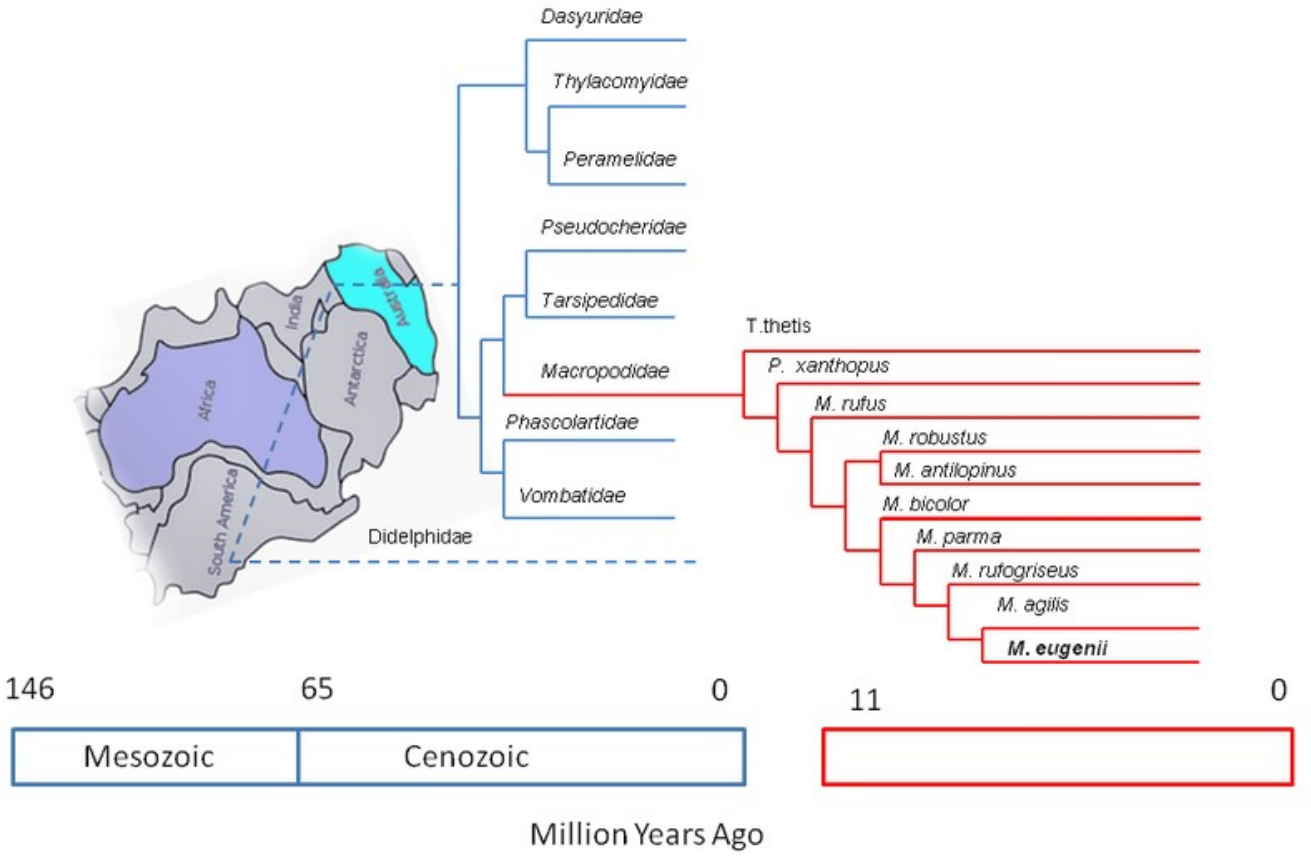
bif724 Assignment 2 winter 2016

Introduction

As you should have learned by now, phylogenetics is a branch of evolutionary biology that uses various means, currently mostly computational, to compare and study evolutionary relationships. One graphical result of this is a phylogenetic tree showing the degree of relatedness of different species, such as in the following;



(image by Graham Colm http://http://commons.wikimedia.org/wiki/File:Phylogeny_of_marsupials.png)

Phylogenetic software can draw trees representing a number of different relationships, including those between different species, or even between different orthologs and paralogs of a given protein. In order to draw a tree, you first need a multiple alignment of nucleotide or amino acid sequences suitable to your purpose. This multiple alignment, in a format such as CLUSTAL (http://web.mit.edu/meme_v4.9.0/doc/clustalw-format.html), can then be used by a tree drawing program. In order to get a multiple alignment you need to input a series of sequences to be aligned, in a format such as FASTA (in this case there are multiple records in FASTA format in the same file).

A number of software packages can prepare multiple alignments, and/or draw phylogenetic trees, including

- T-Rex http://www.trex.uqam.ca/
- MUSCLE http://www.ebi.ac.uk/Tools/msa/muscle/
- MAFFT http://mafft.cbrc.jp/alignment/server/
- CLUSTALW http://www.ebi.ac.uk/Tools/msa/clustalw2/

You can learn more about phylogenetics here (but it's not required for the assignment):

- http://www.ncbi.nlm.nih.gov/pubmed/12801728 (Phylogeny for the faint of heart: A tutorial)

Other resources include:

- http://www.scq.ubc.ca/introduction-to-phylogenetics/ (a good introduction)
- http://www.ucmp.berkeley.edu/exhibit/introphylo.html

Phylogenetics is a complicated topic, beyond the scope of this course, but as bioinformatics programmers we may need to write programs to assist in phylogenetic analysis. The first requirement of phylogenetic and multiple alignment software is a set of suitable sequences in an appropriate format.

In this assignment we will write a program that will obtain specified nucleotide and amino acid sequences from GenBank and then format and output these in FASTA format, suitable for input into a multiple alignment program such as MAFFT.

After aligning these sequences we will be able to see the results of our work by drawing a tree using the aligned sequences. Since the results of phylogenetic analysis depends on a number of factors, including the sequences used for comparison, this program you will write will let the user experiment with various sequences, by making it easy to obtain them and prepare them for alignment and phylogenetic analysis.

Mitochondrial DNA is a common source of sequences for studying evolutionary relationships. In fact DNA barcoding (the Barcode of Life project http://www.barcodeoflife.org/) uses a specific part of the gene for cytochrome c oxidase subunit I, named COX1 (aka COI). For the purposes of this assignment we will use the complete mitochondrion genome from a number of species that are separated by various evolutionary distances, but the same technique could be used to extract sequences from other sources as well.

See the end of the assignment for instructions on how to do draw phylogenetic trees with the data you generate.

Assignment

A common task for bioinformatics programmers is to extract specific information from one or more of many different databases. While this can be an awkward and time-consuming process when done by writing code to access the raw data, there are Perl code libraries to make data collection and formatting easy.

In this assignment you will write a Perl program that will use selected parts of BioPerl to extract and format specific GenBank sequences, as are needed to obtain a multiple alignment suitable for drawing a phylogenetic tree. Your program will output a fasta file containing a series of sequences for a selected mitochondrial gene from a range of species.

Specifications

1. The program will work as follows:
    1. The program will take two command line arguments. Check for two arguments first, and end the program if two are not provided. Make sure you give the user an appropriate message in this case.
        1. The first argument is the name of the file containing the list of GenBank ids to be used. The file will be a simple text file containing one id on each line, and **must be in the same directory as the executing program.** See the example file small_list. Assume that each of the ids refers to a valid GenBank entry.

2. The second argument is the name of the gene for which the sequences should be extracted.

2. Create two Bio::SeqIO objects with the appropriate filenames, and the -format option set to 'fasta'. There will be one for the protein sequences and one for the dna sequences. **The files must be created in the same directory as the executing program.** These files will contain your output.
    1. The dna file name **must** use the pattern dna_wwww_xxxx.fa where wwww is your first name, and xxxx is the name of the gene that was searched for, e.g. for me one of the files would be dna_john_COX1.fa
    2. The protein file name **must** use the pattern aa_wwww_xxxx.fa where wwww is your first name, and xxxx is the name of the gene that was searched for, e.g. for me one of the files would be aa_john_COX1.fa
3. For each of the ids in the file you will do all of the following:
    1. Use the id to get the file from the GenBank server, using the class Bio::DB::GenBank.
    2. Use the methods in the resultant Bio::Seq object to extract the following:
        1. The **full** binomial name (e.g. Homo sapiens neanderthalensis). Use the Bio::Seq object to get a Bio::Species object, and use the appropriate method of the Bio::Species class (see the api for details) to get the desired information. Once you have this, replace the spaces with underscores.
        2. The translation for the desired gene. To get this you will need to search through all the features of the Bio:Seq object, and look for each feature with the name CDS (use the method get_SeqFeatures() to get an array of Bio::SeqFeatureI objects, and use the primary_tag() method). If the feature is named CDS, use the method get_tag_values() with the argument 'gene' to look for the correct gene. Once you have the right gene, use get_tag_values() with the argument 'translation' to get the protein sequence.
        3. The DNA sequence for the desired gene. To get this, first get the start, end, and strand for the desired gene. Use the Bio::SeqFeatureI methods start(), end(), and strand(). Then use the start and end positions to get the appropriate subsequence from the sequence, using the Bio::Seq method subseq(). If the strand is -1 then you will have to get a reverse complement of the subsequence, using the Bio::Seq revcom() method.
    3. Once you have the desired parts, save the data as follows:
        4. Create a Bio::Seq object, with the -id being the binomial name as described above, and the -seq being the dna sequence. Use the Bio::SeqIO write_seq() method to add the sequence to the appropriate file.
        5. Repeat for the protein sequence.
4. Tell the user that the files have been created, where they are, and what their names are.

2. You can make the following assumptions:
    1. All of the GenBank ids in the file list are all existing entries in the database.
    2. The file list is in the same directory as your program.
    3. The command line arguments are given in the correct order.
    4. The gene name given on the command line is valid and found in all files in the file list (i.e. you don't have to check whether the gene is found in all files, assume that it is. This is to simplify the program, to focus on use of BioPerl.)

Important tips

- Start by figuring out how to get all the required information for just one file. Do this by writing a poc (proof of concept) program, which is a little program to do just this part. Just hard code the name of one GenBank record, and have your program get the file and extract the required information and print it to the screen. Manually compare the information you are getting with what you should be getting to make sure it is exactly correct. Only when your technique to extract the required information is working should you then write the rest of the code.
- Use the datasets provided in the provided tarball to test your code, **but remember that you must hardcode nothing specific to these files (file names, species names, gene names, etc.).** This means that although you are using the data provided for testing purposes, your code should work properly as specified in this document for any valid GenBank ids of the correct type.
- I will be testing your code on my computer, so make sure you don't hardcode anything specific to your environment, such as file paths.
- See the provided files for example output. Make sure your output for each exemplar gene matches the provided example output for that same gene **precisely**. This means that your output files should match the provided examples exactly, character for character, including the order of the sequences, newlines, spaces, etc. If your output matches for the example files it should match for any valid GenBank ids and gene combination, and I may test your code with data other than the examples provided.
- Remember the linux command 'diff', which will compare two files for differences – there should be none between your output and the appropriate examplar. Type 'man diff' on a linux system for details.
- Note that the consequence of any errors in your output may be a deduction of at least 50%, in keeping with the bioinformatics workplace, where incorrect output can have serious consequences, such as retracted papers, and wasted research time and money. Re-read the following post http://boscoh.com/protein/a-sign-a-flipped-structure-and-a-scientific-flameout-of-epic-proportions.html, which discusses the major results of a small error in a bioinformatics program.
- Remember that the purpose of assignments is to apply your knowledge to a practical problem in bioinformatics, but you are not expected to do so without help from me. You are encouraged to ask questions of me when needed, either to clarify the specifications, or when you get stuck. This is an important part of the learning process.

Rules

- This is an individual assignment, worth 15% of your final grade.
- You must use the appropriate parts of BioPerl wherever possible to complete the assignment.
- This assignment is due on Monday March 28th at 11:59pm and will be submitted by email, following these instructions;
  - Send me an email with the following **exact** subject line:
    bif724-a2 submission
  - Your email will have a single file attached, your finished source code, which **must** be named **exactly** according to the following formula;
    your lowercase first name + _ + a2 + .pl
    e.g. for me the file would be john_a2.pl
    Do not put this file in a zip file.
    Do not include a library file (see below).
- Extension: you should plan to get the assignment submitted by the above due date, but if you

need an extension, you are automatically granted an extension of up to 48 hours. After this time late penalties will apply. You do not need to request this extension, or notify me that you are using it, just submit your work when ready, as stated above, and if it is done before the final deadline, you will not receive a late penalty.

- Late assignments will be deducted 25% per day or part day late.
- Submitting an assignment that does not compile and/or run, or does not meet major parts of the requirements, or produces output that differs vastly from what is required, will result in a mark of 0. This is to avoid students intentionally submitting unfinished work to avoid late penalties. As per the rubric, "All assignments must be satisfactorily completed (i.e. they must work as specified in the rubric) in order to pass the course." This means that you will still have to get an assignment finished and working, even if you don't get a mark for it, due to late penalties or due to the submission of unfinished work. The best way to avoid late penalties is to start assignments early, follow the tips and suggested steps, and ask for help as soon as possible.
- Structure your program so that it ends logically in all cases, not by use of exit.
- Use subroutines where appropriate if you wish (not required), but for marking purposes include them in your assignment file, not in a separate library file. Remember that by convention when subroutines are in a perl file they are found at the bottom of the file, after all the code of the 'main' program. If you do use subroutines, remember that a subroutine does one thing only, and shouldn't print anything itself. Even though it may be in the same file as the main, it should be self contained, and not refer to variables outside the subroutine. Any input to the subroutine must be passed via @_.
- Make sure that you neatly and consistently indent all code in blocks, as shown in class.
- Do not double space the entire program (i.e. don't put a blank line after each line of code), but use blank lines thoughtfully to separate sections of code and make the program more readable.
- Make sure that you do not get any warnings when you run your program. If you do you must eliminate them because they indicate potential problems.
- Documentation is important. You must add comments near the beginning of your program (remember nothing ever goes before the shebang line) stating your name, course number, assignment number, and the purpose of the assignment. Add additional comments throughout the program to explain key points, non obvious code, etc.
- Immediately after the part containing your name etc. you must also include the following declaration adding your full official name, as it appears in Seneca's records, and your id # where indicated. Your assignment will not be marked without this part present. You may copy and paste this text into your program, since it is a college requirement, and not part of the course material:

  ```
  # I declare that the attached assignment is wholly my own work in accordance with
  # Seneca Academic Policy. No part of this assignment has been copied manually or
  # electronically from any other source (including web sites) or distributed to other students.
  # Name:
  # ID:
  ```

Plagiarism

- This assignment is to be 100% individual effort.
- Plagiarism includes the submission of ANY work that is not completely your own. You are not allowed to use any code segments or anything else written by others, in any form, including

class examples or any material that I have provided.
Plagiarism includes but is not limited to:

- Using any part of another student's assignment code, verbatim or paraphrased.
- The changing of comments or variable names in someone else's work and submitting the new version.
- Using the exact text of these assignment instructions in any part of your assignment including program documentation or user instructions.
- The use of code, text, images or anything else from text books, Internet sites, class examples or any other source. It does not matter if the author of the material offers it for use freely. You are not allowed to use anything at all that you didn't create yourself in any part of an assignment.
- Having other students' code in your account, whether you use it or not. Remember that if another student is found with your code in their account, you will both be held responsible, so never share any of your code with anybody, and set permissions to protect your zenit account (see addendum for details).

After your program is working

- The following steps are not part of the assignment and you will not be marked on them, but they are fun and interesting and will show you the practical use and importance of the data your program is outputting.
- The following steps will show you why it is crucial that your programs output the correct data, since you or others may be basing conclusions on the data your program produces.
- Once you have the assignment working properly, use it with the file in the example tarball called small_list to produce the FASTA files for COX2. Then use the aa sequences as input to MAFFT (link above) to get a multiple alignment, and examine the results:
  - Examine the multiple alignment in Jalview. Look for areas that are not conserved across all species. Do you see any conservation between similar species?
  - Examine the tree produced from the multiple alignment (use UPGMA for Method). Does this tree make sense given the species involved? What about if you use the NJ method? Does that tree represent known relationships as well?
- Try the same steps with the DNA sequences and compare your results. Does the tree drawn using DNA data look more accurate?
- Try the two above steps using COX1 instead, and compare your results.
- Repeat the above steps with the file in the example tarball called big_list, which has a wider range of species separated by greater genetic distances.
- Note that multiple alignments and phylogenetic analysis are complicated topics, and selection of optimal settings and methods for the alignment and tree-drawing programs requires a more detailed understanding of how the processes work than there is time to cover in this course.