Evolution Reconstruction with Parallel DNA analysis in Julia

John N. Mofor

EECS MIT 2015

Table of Contents

Introduction

Evolution by itself is a very interesting topic. I fixated the tree of life for quite some time, and thought to myself: “Wouldn’t it be cool to programmatically derive such a tree?”

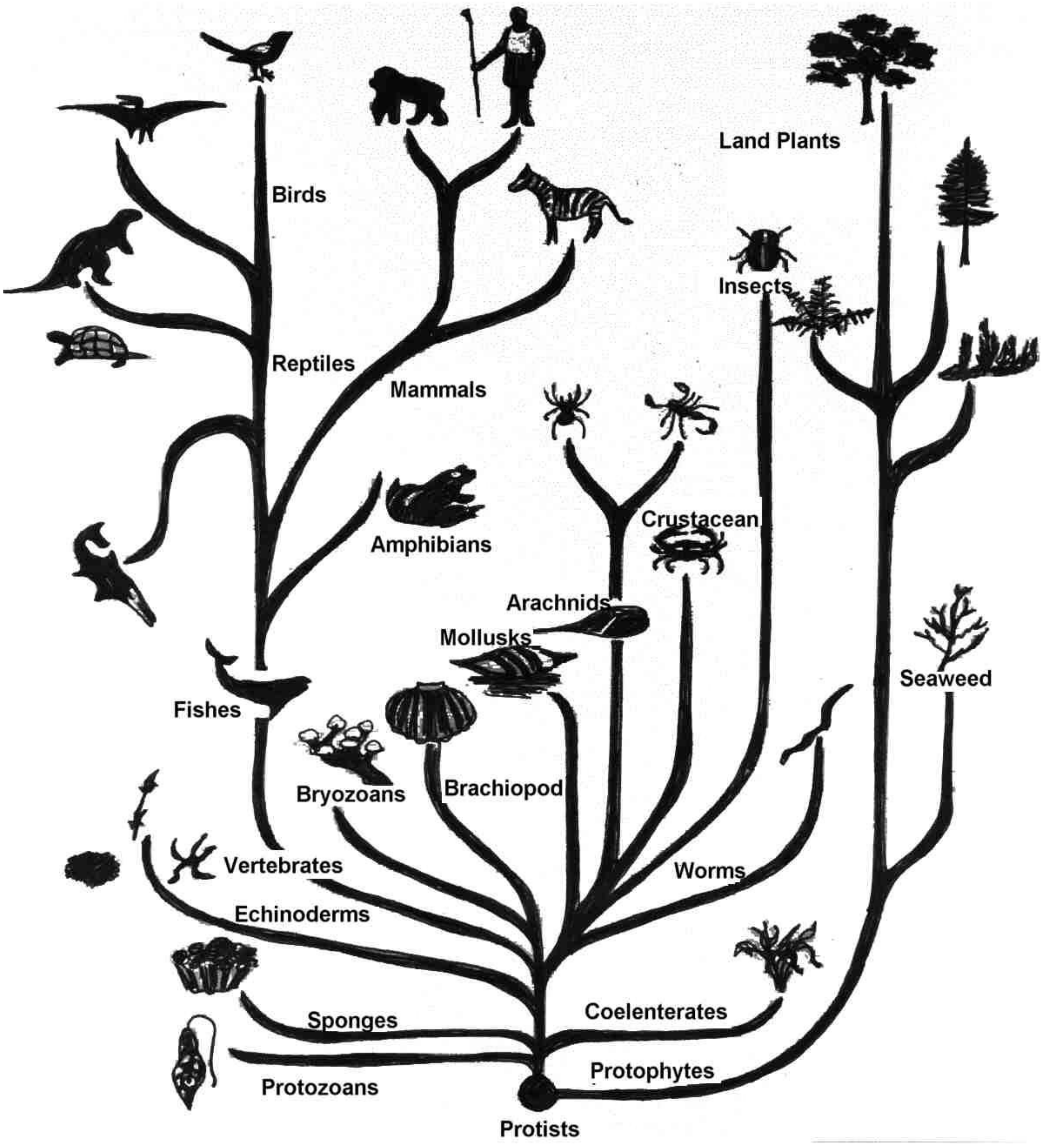


Fig 1: Tree of life

Before taking 18.337 (6.338), I always shied away from computation-heavy tasks because of the time they took. Coming up with this tree, I knew will require a lot of computational power, data, and some theory behind evolution.

I needed a way to circumvent the computational power barrier, given I did not have specialized computers for DNA analyses. During the course of this class, I was exposed to parallelism in a very fast novel language – Julia. My personal computer is a four core machine, and distributing the work to all the cores could spare me the need of a super computer, on a limited dataset.

Next, I needed data. I surveyed the internet in search of organisms' DNA. Fortunately, I could lay my hands on some DNA files, though in limited quantities. My dataset as of 12/14/2013 is hosted at http://web.mit.edu/moforj/Public/data. I could find only that of five mammals, but it was enough to, at the very least, build a working prototype.

With data and Julia in hand, I had enough tools to start off. Before jumping into programming however, I needed to learn some of the theory underlying the tree of life.

Unfortunately, that venture did not turned out so well, as the more I was digging in, the more extensive the reading was getting. Given my semester load, I had to abridge my research, after extracting a set of guidelines. With those guidelines, I could then apply some Artificial intelligence (AI) concepts to come up with an algorithm. I will be discussing these in greater details in subsequent sections.

Now with data, Julia, and an algorithm in hand, I could officially kick off the project.

Algorithm & Method

Background

In my research to understand the concept behind the tree of life, my take-home was pretty simple.

1. The DNA pattern of a node and that of its ancestor in the tree is very similar.
2. It is not the number of individual bases (Adenine-A, Cytosine-C, Thymine-T, and Guanine-G) found in their DNA, that determines their similarity. Instead, I should look for groups of them (contiguous sequence in DNA), which match. The idea is, groups of bases together are responsible for controlling certain features of the organism. So, we could track those features as an organism evolves, by tracking those groups, or similar groups.

In order to reduce the aforementioned two points into an algorithm, I needed an AI paradigm to simplify it all, and validate some assumptions, which could render my task computable.

Occam's razor is a principle of parsimony, which could be defined as thus:

In a situation which could be explained by several hypotheses, the most probable hypothesis is the one which requires the least number of assumptions.

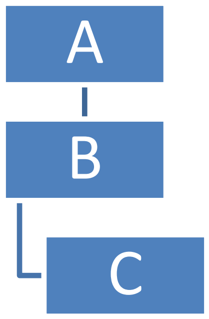
What this means in our context is:

The greater the similarity between two organisms' DNA, the more likely those organisms are adjacent in the tree of life. Note that adjacency here could be vertically (parent to child) or horizontal (between children).

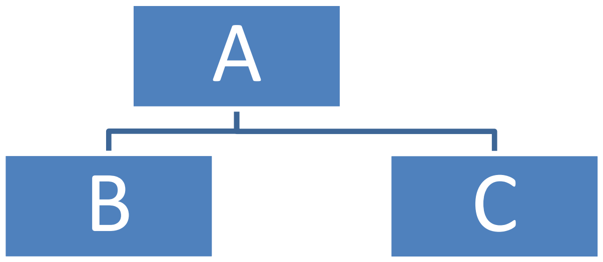
To illustrate this, consider this scenario:

- Let AB = distance from organism A to organism B,
- Let AC = distance from A to C

- If AB < AC, then assuming the tree is rooted at A, and that there are only these 3 organisms, B most probably evolved from A, and not from C. C then evolved from either A or B.
    - So if BC < AC, then C most probably evolved from B, and not from A. So the tree will look like:



    - On the other hand, if AC < BC, then C most probably evolved from A directly, but with more mutations than in B. Tree will look like:



Algorithm

The formal statement could sound like:

<u>An organism B, evolves from another organism A if organism A is the closest source to organism B.</u> Mathematically,

$$B \text{ evolves from } A \Rightarrow \forall X \in \text{organisms}, \overrightarrow{XB} \geq \overrightarrow{AB}.$$[1]

[1] Note that this is not an iff relationship(⇔) as we do not know for a fact that evolution/mutations obeys a pattern

The above equation reduces to a shortest path in a graph problem. Taking a step back, the shortest path approach makes sense because we are working on the bases of Occam's razor, which by itself could be thought of as a shortest path method of reasoning.

To find the shortest path, we will use the famous **Dijkstra's** shortest path algorithm, whose implementation I will discuss subsequently.

Lastly, we need a distance metric. Because we need to consider bases (A, T, C & G) in groups, we need to use a *window sliding* technique, in which every window represents a group of continuous bases. The most efficient algorithm I know for this task is **Rolling Hash** by Robin-Karp.

I did not dig deep enough to find the ideal window size, so, for this experiment, I will make window size a variable, so we can vary it later when we have more information on the setup.

Method

The first task we need to perform is to actually read the DNA files, in a manner which will make distance computation easy. It is important to note that, we did not store all the DNA file into memory, as that would have been very space consuming, especially as we really do not need the DNA sequence itself. So, I needed a way to read the file, and just extract the useful information (compressed) out of it.

In that light, I applied Rolling hash right on the input stream, and stored the generated table. Found below is my Julia implementation of rolling hash.

The codes purpose is to generate a table (map) with:

- keys = hash of a DNA sequence of length window_size
- value = number of time that DNA sequence occurs in the strand.

Rolling Hash Implementation.

```
## Uses Robin-Karp's Rolling Hash.
## Complexity O(N), N = number of chars in DNA file.
@everywhere function generate_lookup_table_faster(location::String, window_size::Int64 =
10)
    # mapping of k-sequence hash to number of occurrences.
    lookup_table = Dict{Uint64,Int64}()

    # IOStream of reference .fa DNA file.
    file = open(location,"r");

    # Vector of size window_size, used to slide across DNA strand.
    container = Char[]

    # Char's to exclude.
    exclude = ['\n' 'N'];

    ## Defining some constants
    PRIME_BASE::Uint64 = unsigned(5);
    PRIME_MOD::Uint64 = unsigned(1000000007);
    current_hash = 0;

    # While we are not at the end of the file.
    while !eof(file)
        c = read(file,Char);

        if !in(c,exclude)

            # if the container is NOT full, add the next char in it.
            if (length(container) < window_size)
                push!(container,c);
                current_hash = (current_hash*PRIME_BASE + int(c)) % PRIME_MOD;
            end

            # if the container is FULL, compute hash, and remove first element.
            if (length(container) == window_size)
                if haskey(lookup_table,current_hash)
                     # Augment tally if it existed already.
                     lookup_table[current_hash]+=1;
                 else
                     # Initialize tally at 1.
                     lookup_table[current_hash] = 1;
                 end

                # Update hash and remove first element (Slide the window).
                current_hash = (current_hash -
int(container[1])*(PRIME_BASE^(window_size-1)) ) % PRIME_MOD;
                container = container[2:end];
            end
        end
    end
    return lookup_table
end
```

Rolling Hash Analysis

As indicated, the algorithm has

- O(N) runtime complexity, where N = number of bases in the DNA file.
- O($min(2 * 4^k, N)$) space complexity, where k = window_size.
    - 4^k because, there are 4 bases, and given a window size of k, there will be a maximum of 4^k possible keys in the table.
    - 2*4^k because, for every key, we will need an associated value.
    - min(2*4^k, N) because, N might be smaller than 2*4^k, in which case, we have a much tighter upper bound on the space, as we will not use more than N.

At this point, we have successfully efficiently extracted the relevant information about the DNA strand, into a very fast data structure, which should enable the rest of the program to run pretty quickly.

The next step, after building a DNA table lookup, is to compute a distance metric, given any two such tables.

The logic used here was a straight-forward difference in frequency, accumulation. That is,

<u>The distance between 2 look-up tables, is the sum of the differences in values for every key from both tables. If a key is not in the other table, then we assume a zero for that key.</u>
Mathematically,

$$D(A,B) = \sum_{key}^{key\ in\ \cup(A,B)} |\ A[key] - B[key]\ |$$

$$A[key] = 0\ if\ key \notin A, else\ A[key].$$
$$B[key] = 0\ if\ key \notin B, else\ B[key].$$

The Julia implementation I have takes:

- O(size(A) + size(B)) runtime complexity, given you will have to visit every key in both maps, but
- O(1) space complexity.

Now that we have a distance metric, we need Dijkstra's shortest path algorithm. The following is my implementation of Dijkstra.

Dijkstra's Shortest Path Implementation

```
## Returns the shortest path between any 2 nodes, or {} if target doesn't exist.
function dijkstra_shortest_path(graph::Dict{},start::String,target::String)
    # Sanity checks
    if !haskey(graph,start) || !haskey(graph,target)
            println("Invalid request");
            return ()
    end
    # Required data structures
    pq = PriorityQueue();
    parent = Dict();
    dist = Dict();
    visited = Set();
    generic_start_node = "Cell";
    current = start;
    dist[start] = 0;
    parent[start] = generic_start_node;

    # Main loop.
    while current!=target
        # Loop only over instances with out-edges.
        if haskey(graph,current)

            # Avoid inifinite loop as this may be undirected graph.
            push!(visited,current);
            # Filter for only nodes not visited already.
            for node in filter(n->!in(n,visited),[x for x in keys(graph[current])])
                distance = dist[current] + graph[current][node];
                # Condition for relaxation,
                # Relax iff we have found a shorter path.
                # Consider only 1 such paths if 2 'shortest' paths
                # have the same distance.
                if !(haskey(dist,node) && distance >= dist[node])
                      dist[node] = distance;
                      parent[node] = current;
                  end
                # Update the priority queue.
                pq[node] = dist[node];
            end
       end
       # Move to the next shortest path so far!
       current = dequeue!(pq);
    end
    # Reconstructing the path
    path = {};
    if current==target
        path_distances = {};
        # Backtrack, until the generic_start_node
        while current!=generic_start_node;
            push!(path,current)
            push!(path_distances,dist[current])
            current = parent[current];
        end
        # Reverse the path, and return it.
        reverse!(path);
        reverse!(path_distances);
     end
    return path,path_distances;
```

Dijsktra Analysis

The above implementation has:

- O($|V|\log|V| + |E|$) runtime complexity (V = number of vertices, E = number of edges), and
- O(|V|) Space complexity (for pq).

At this point, we have all the tools we need for the computation.

We now need to put everything together. First, let's try a serialized approach and see the bottle neck:

```
function compute_edges_serial(dna_files_dir,win::Int64=10)

    # Getting absolute refernces.
    dna_files = readdir(dna_files_dir);
    fpath(file) = "$(dna_files_dir)\\$(file)"

    # Initializing graph
    graph = Dict();
    n = length(dna_files);
    for i in 1:n
        graph[dna_files[i]] = Dict();
    end

    # Main Loop.
    start_time = time();
    tables = Dict();
    for i in 1:n
        if !haskey(tables,dna_files[i])

tables[dna_files[i]]=generate_lookup_table_fast(fpath(dna_files[i]),wi
n);
        end
        t1 = tables[dna_files[i]];
        for j in i+1:n
                if !haskey(tables,dna_files[j])

tables[dna_files[j]]=generate_lookup_table_fast(fpath(dna_files[j]),wi
n);
                end
                t2 = tables[dna_files[j]];
                graph[dna_files[i]][dna_files[j]] =
graph[dna_files[j]][dna_files[i]] = get_distance(t1,t2);
        end
    end
    println("Total time: ",time() - start_time);
    return graph
end
```

The bottle neck in the above piece of code is the for-loop which contains generate_lookup_table_faster. Even though that function takes O(N) which is the best complexity, we are still executing all that work in serial, while other cores are just idling.

A way to fix that could be, by sending off that heavy computation to separate cores, and grabbing the result when we need it. This is what the following implementation achieves.

```
function compute_edges_parallel(dna_files_dir,win::Int64=10)
    # Getting absolute refernces.
    dna_files = readdir(dna_files_dir);
    fpath(file) = "$(dna_files_dir)\\$(file)"
    # Making sure we have enough processors
    n = length(dna_files);
    diff = nworkers() - n;
    if diff <= 0
            warn("Not enough workers!!");
    end
    # Initializing graph
    graph = Dict();
    for i in 1:n
            graph[dna_files[i]] = Dict();
    end
    # Main Loop.
    references = [(@spawn
(generate_lookup_table_faster(fpath(dna_files[i]),win))) for i=1:n]
    tables = Dict();
    for i in 1:n
        if !haskey(tables,dna_files[i])
            tables[dna_files[i]]=fetch(references[i]);
        end
        t1 = tables[dna_files[i]];
        for j in i+1:n
            if !haskey(tables,dna_files[j])
                tables[dna_files[j]]=fetch(references[j]);
            end
            t2 = tables[dna_files[j]];
            graph[dna_files[i]][dna_files[j]] =
graph[dna_files[j]][dna_files[i]] = get_distance(t1,t2);
        end
    end
    return graph
end
```

Serial Code analysis

Runtime*:*

$$O\left(\sum_{f}^{f \in dna\_files\_dir} size(f) \right)$$

Space:

$$O\left(\sum_{f}^{f \in dna\_files\_dir} \min(\,(2 * 4^{k}), size(f)) \right)$$

Parallel Code analysis

Runtime:

$$O(\max([size(f)\ \forall f \in\ dna\_files\_dir\ ])$$

Space:

2 * space complexity of serial version – this is because, when we sent over the work to separate cores, we needed to copy the table back to the main core to perform the distance computation. As a result, we end up with copies of each table, though we gain enormously on runtime.

At this point, we have all the functions we need.

Results

In my current data set, I have solely a group of mammals. True this venture would have been much more convincing if I had a lot more animals from different kingdoms, but here is what I have currently.

Evol Results

NB:

- "fx.fa" is the DNA_file of organism x.
- show_evol shows the evolutionary path from organism A to B.
- At the end of the evolutionary path, is found the distance between the organism A and organism B.

```
In [8]:
show_evol("fdog.fa","fchimp.fa")
```

```
fdog.fa -> fchimp.fa @ 14446647
```

```
In [9]:
show_evol("fmaternal.fa","fpaternal.fa")
```

```
fmaternal.fa -> fpaternal.fa @ 442718
```

```
In [10]:
show_evol("fmaternal.fa","fchimp.fa")
```

```
fmaternal.fa -> fchimp.fa @ 2911556
```

```
In [11]:
show_evol("fpaternal.fa","fchimp.fa")
```

```
fpaternal.fa -> fchimp.fa @ 2911214
```

The most noticeable point to notice here is, given only the data set at my disposal, these results make sense.

The difference between a female human (“fmaternal.fa”) and a male human (“fpaternal.fa”) is an order of magnitude less than the difference between a chimp and a human, which is itself an order of magnitude less than the difference between a chimp and a dog (which belongs to *Canivora* order, while chimp and humans are in the same order - *Primates*).

Serial vs Parallel

Serial	Parallel
`141.53399991989136s`	`72.4539999961853s`

It takes about 69s to fully process “fmouse.fa” which is the largest file - hence the 72.45 seconds of the parallel computation.

Conclusion

All in all, I wish I had more data, for different species in different kingdoms, to be able to produce that tree. Unfortunately, I could not find them. My hope is to continue this project in Julia sometime in the future, when I have much more data, and have mastered some more of the theories behind evolution.

Nonetheless, this project was a lot of fun, and I got hands-on experience with Julia, and parallelism.