Practical Computing for Bioinformatics - HW3

Adnan Kaan Ekiz(r0776549) - Olympia Gennadi(r0829391)

December 2020

Contents

1 Question 1 - Representation of the Tree

Given a rooted phylogenetic tree annotated with branch distances, such as:

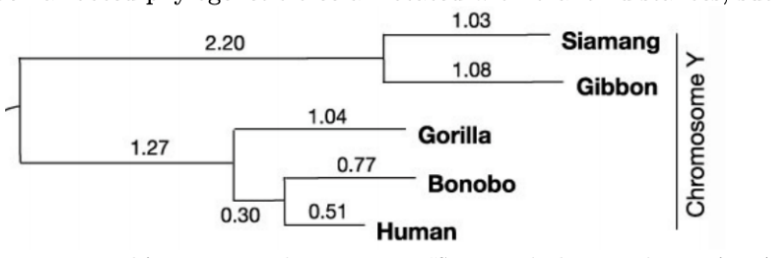

Figure 1: Given tree structure

To represent the data given above in the phylogenetic tree, we have generated a binary tree like structure in Prolog and this representation allowed us to show the tree with at most 2 leaves per node assuming the node have maximum number of children. The reason to use this structure was to make it easier for us to traverse the whole graph in an efficient manner. The structure that we have used to represent the tree can be seen below:

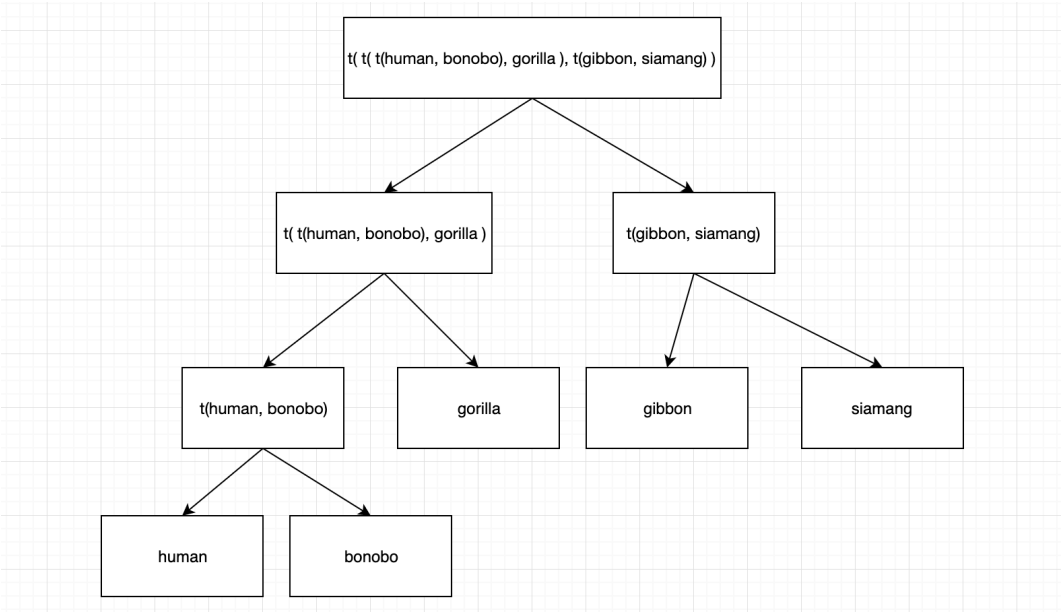

Figure 2: Representation of the whole tree

In this tree, left of the comma is representing the left child and right side of the comma is representing the right child given that child of a certain node can have more children or can also be a certain atom. Based on the structure of the tree that is given in here, explanation and implementation of the other predicates are produced to solve the problems. These predicates can be seen in other chapters.

1.1 Initialisation

While representing the data with a tree like structure, we have figured that it is also important to talk about how to initialise the tree and call the predicates. For each of the predicate described below,

initialisation process should be following:

- **Recent Common Ancestor:** initialize(X), recent_ancestor(Species, X, Y). In this statement, X is the tree and initialize(X) is initializing the tree structure, Species is the given input name and Y will be the result that we will want which is the most recent common ancestor
- **Ancestors:** initialize(X), ancestors(Species, X, Res). In this statement, X is the tree and initialize(X) is initializing the tree structure, Species is the given input name and Y will be the result that we will want which is a list of all the ancestor nodes that also includes the Species input
- **Total Nodes:** initialize(X), total_nodes(Res, X). In this statement, X is the tree and initialize(X) is initializing the tree structure whereas Res will be the total number of nodes in our tree structure

2 Question 2 - Most Recent Common Ancestor

In this part we want to figure out the most recent common ancestor between two species. To achieve this, we have designed the code in such a way that it will finally return the smallest subtree in which the two or more species are located and also has the described element in it. Therefore, we can get as a result the shortest common node of the two or more elements that we will meet if we begin to investigate the tree from the determined species to the root. The formation of the code includes the use of four base cases and four recursive terms.

2.1 Is in

Predicate is_in checks if one species exists in a specific subtree. This function is necessary since it will be used in many recursive cases for finding a specific species.

2.1.1 Base Cases

```
is_in(X, t(X,_)).
is_in(X, t(_,X)).
```

In the above lines we define two rules that are true if X is seen inside the tree. The base cases contain two parameters. The first parameter is the given element and the second describes the subtree that the element is placed in. By these predicates, we define that the specific species X belongs to the structures t(_,X) or t(X,_). That means, that in a binary subtree if the element X is the left or the right child, then it is contained in this. The character '_' indicates that we can have anything on the other side of the tree since the part that interest us the most is the investigation of species X.

2.1.2 Recursive Cases

```
is_in(X, t(L,_)):- is_in(X,L).
is_in(X, t(_,R)):- is_in(X,R).
```

In these lines we have recursive terms i.e.: terms that refer to itself. Specifically, the left command is true only when the right command occurs. The character ':-' distinguishes the commands between them. For instance, based on the first line, 'is_in(X,t(L,_))' is true only when 'is_in(X,L)' is true.

The recursions help us to find out if species X exists somewhere in the tree since it cannot be found directly but instead we need to look around the structure. We start searching the tree from the top and exploring the left and the right side of the initial tree. The parameters L and R indicate in which side of the tree the code is going to search. This gives us the possibility to travel through each side of the tree reaching every single node. By considering all the possible subtrees, if species X belongs in one of them and is either the left or the right child, then the Prolog recognizes one of the base cases and the program stops using the recursion. If X does not belong in the subtree, then the program will continue exploring the next nodes -left and rights- until finding the element.

2.2 Recent ancestor

The recent_ancestor predicate returns the smallest subtree in which two species belong. We have made two base cases and two recursive cases which are explained below.

2.2.1 Base Cases

```
recent_ancestor(X, t(X,Y), t(X,Y)).
recent_ancestor(X, t(Y,X), t(Y,X)).
```

In this part, we have created two additional rules. Recent_ancestor predicates have 3 parameters. The first parameter X is the species while the second parameter is the input tree. The third parameter is the smallest subtree in which the species X belongs joined by species Y and it will finally be returned. When species X is the right or the left child of a subtree with other species Y, we consider as a fact that their most recent common ancestor is their parent, the root consists of the specific subtree.

2.2.2 Recursive Cases

In order to answer to this question, we have to create two recursions essential for determining the ancestor between the two species.

```
recent_ancestor(X, t(L,_), Res):- is_in(X,L), recent_ancestor(X,L,Res).
recent_ancestor(X, t(_,R), Res):- is_in(X,R), recent_ancestor(X,R,Res).
```

Using the recursions of recent_ancestor we achieve the deep search of the tree. Initially, we pass through the left nodes and then through the right nodes until examining all the possible cases of constructed subtrees. The first step is to see if the species X exists in each of the possible subtrees by calling the predicate is_in. If the species appears in the subtree then we call the predicate recent_ancestor itself to examine the base cases and return the result which in this case is the third parameter Res. The result is the path of the smallest subtree containing the elements X and Y indicating their parent node which is actually their most recent common ancestor.

3 Question 3 - List of the Ancestor Nodes of a Given Tree

For the question 3, we have implemented a predicate that is returning the list of all the ancestor nodes of a given leaf of the tree up to the root node. To accomplish this task we have generated 2 helper predicates. We will first mention these predicates and then explain the predicates for generating the list of ancestors.

3.1 Append

We generate append to combine 2 of the lists that are given into 1 big list. To generate this predicate, we have used 2 predicates that consists of 1 base and 1 recursive case.

```
append([],X,X).
append([X|L1],L2,[X|L3]):-
        append(L1,L2,L3).
```

3.1.1 Base Case

```
append([],X,X).
```

Base case condition consists of 3 parameters, Prolog considers this fact as true when the first list is empty. Third parameter is then returned as the same as second list. Which will be later combined with the recursive case so that elements in the first list will be added at the beginning of the third parameter (list).

3.1.2 Recursive Case

```
append([X|L1],L2,[X|L3]):-
        append(L1,L2,L3).
```

Whereas in the recursive case, in each call we are trying to add 1 element at the beginning of the result list until there are no more elements in our first parameter list. This process occurs by taking one element from the first list and calling the same function with the rest of the list each time.

3.2 Is in

Since the same predicate mentioned above, we are skipping this part in order to not duplicate the same thing that is written before.

3.3 Ancestors

Finally, we have created this predicate to generate the list of all the ancestor nodes in a recursive fashion as mentioned earlier. Structure of the code with recursive and base case as follows:

3.3.1 Base Case

```
ancestors(X, t(X,Y), [t(X,Y)]).
ancestors(X, t(Y,X), [t(Y,X)]).
```

When we are able to determine that the element that we are searching is inside of the tree structure, we are returning this structure inside of the list as the first element that is considered to be the ancestor of it. Therefore, when this list value is returned to the recursive body of the predicate we will be appending the other found values to it by using the "append" predicate that is described above.

3.3.2 Recursive Case

```
ancestors(X, t(L,R), Res):-
    is_in(X,L), ancestors(X, L, C), append([t(L,R)],C,Res).
ancestors(X, t(L,R), Res):-
    is_in(X,R), ancestors(X, R, C), append([t(L,R)],C,Res).
```

In the recursive case, since we know that the element that we have searched is not directly inside (need to traverse further) the structure that we have created, that means that we need to check further by recursively iterating through the structure. However, in order to avoid unnecessary work we first check if the given element is inside the subtree structure that we are planning to search by calling the "is_in" predicate.

If we know that element we are searching for is inside of our structure, we then call the predicate itself to deepen the search inside of our subtree. When the "ancestors" predicate returned a list we then use our "append" predicate to add our newly found ancestor into the list of ancestors and return the appended result list recursively.

4 Question 4 - Number of Nodes in a Tree

In the last chapter, we want to count the number of nodes existed in the tree. We again take advantage of the recursions that Prolog provides.

4.1 Is tree

We first create one predicate called is_tree. This rule decides which type of structure is actual tree and returns false if its an atom. The predicate is shown below.

```
is_tree(t(_,_)).
```

A binary tree is a structure composed of one root node and two branches with children who may expand with more children. Based on this and the structure of the tree that we defined in the first chapter, we set as a rule that any subtree that has this structure is a binary tree. In that way, we can avoid the misconception that a leaf node is a tree and thus consider the species located in the leaves as trees. For example, with an input t(human,bonobo) the program will return true, whereas with an input (human) the program will return false since human is not a tree by itself.

4.2 Total Nodes

Taking into consideration the previous base case is_tree, we constructed one recursive case which calculate the total amount of nodes of the initial tree consisted of parents and children. In a fast calculation with the eye, the number of nodes is 9. This can be proved with the following part of the code.

4.2.1 Base Case

```
total_nodes(Res,T):- not(is_tree(T)), Res is 1.
```

We have set two parameters for the predicate total_nodes; Res and T. Res is the number of nodes while T is the tree in which the program searches. The command 'not(is_tree(T))' checks if T is a tree based on the predicate we have defined earlier. When the predicate is called, we need to check first the structure of the given tree T. If T is a tree and not a leaf then the program terminates this case. But, if T is a leaf node, we move on to the next fact 'Res is 1' and Res takes the value 1. To sum up, what we actually do in this case is counting the number of leaves and setting the total node number as 1 if node is seen.

4.2.2 Recursive Case

```
total_nodes(Res,t(L,R)):- total_nodes(X,L), total_nodes(Y,R), Res is X+Y+1.
```

In this line we calculate the total nodes with the help of the previous case. We have also two parameters; the number of nodes that the program will finally return for the defined tree. When we call the command, the program will use the predicate itself to execute a binary searching.

Every time it reaches a node it will use the previous case to see if it is a parent node or a leaf. We are going to explain the two possible options:

- If we are on a parent node the base case fails. The program returns back and it will continue further to check for the leaf (atom) node.
- If the node is a leaf, total number of nodes is determined as 1 and will be added to the total node value of the parent node.

The process continues by jumping on a different node and subtree each time. It counts the number of total nodes of each subtree. X corresponds to the number of nodes in left leaves, Y corresponds to the number of nodes in right leaves while the total amount increases every time by one, indicating that we have also the parent of those children.

5 Extra

5.1 Common Ancestors

In addition to the predicates described above, we have also generated a predicate that gives the closest species to a described species by checking the phylogenetic tree. This means that when we give a species as a parameter it should return the closest different species as a return. Code is shown below:

```
common_ancestor(X,Y,t(X,Y)):- not(is_tree(Y)).
common_ancestor(X,Y,t(Y,X)):- not(is_tree(Y)).
common_ancestor(X,Y,t(X,t(Y,_))).
common_ancestor(X,Y,t(X,t(_,Y))).
common_ancestor(X,Y,t(t(Y,_),X)).
common_ancestor(X,Y,t(t(_,Y),X)).
common_ancestor(X,Y,t(L,_)):- common_ancestor(X,Y,L).
common_ancestor(X,Y,t(_,R)):- common_ancestor(X,Y,R).
```

Since this is not what is asked in the assignment, we just wanted to present this as a side note. Reason why it is called "common ancestor" is because the given parameter element and the result will have the same recent common ancestor. For example:

- If common_ancestor(human, Y, t(t(t(human, bonobo), gorilla), t(gibbon, siamang))) is called, then Prolog returns Y=bonobo
- If common_ancestor(gorilla, Y, t(t(t(human, bonobo), gorilla), t(gibbon, siamang))) is called, then Prolog returns Y=bonobo and Y=human
- If common_ancestor(siamang, Y, t(t(t(human, bonobo), gorilla), t(gibbon, siamang))) is called, then Prolog returns Y=gibbon

6 Appendix - Full Code

```
initialize(X) :- X = t( t( t(human, bonobo), gorilla ), t(gibbon, siamang) ).

common_ancestor(X,Y,t(X,Y)):- not(is_tree(Y)).
common_ancestor(X,Y,t(Y,X)):- not(is_tree(Y)).
common_ancestor(X,Y,t(X,t(Y,_))).
common_ancestor(X,Y,t(X,t(_,Y))).
common_ancestor(X,Y,t(t(Y,_),X)).
common_ancestor(X,Y,t(t(_,Y),X)).
common_ancestor(X,Y,t(L,_)):- common_ancestor(X,Y,L).
common_ancestor(X,Y,t(_,R)):- common_ancestor(X,Y,R).

recent_ancestor(X, t(X,Y), t(X,Y)).
recent_ancestor(X, t(Y,X), t(Y,X)).
recent_ancestor(X, t(L,_), Res):- is_in(X,L), recent_ancestor(X,L,Res).
recent_ancestor(X, t(_,R), Res):- is_in(X,R), recent_ancestor(X,R,Res).

is_in(X, t(X,_)).
is_in(X, t(_,X)).
is_in(X, t(L,_)):- is_in(X,L).
is_in(X, t(_,R)):- is_in(X,R).

append([],X,X).
append([X|L1],L2,[X|L3]):-
        append(L1,L2,L3).

is_tree(t(_,_)).

total_nodes(Res,T):- not(is_tree(T)), Res is 1.
total_nodes(Res,t(L,R)):- total_nodes(X,L), total_nodes(Y,R), Res is X+Y+1.

ancestors(X, t(X,Y), [t(X,Y)]).
ancestors(X, t(Y,X), [t(Y,X)]).
ancestors(X, t(L,R), Res):-
    is_in(X,L), ancestors(X, L, C), append([t(L,R)],C,Res).
ancestors(X, t(L,R), Res):-
    is_in(X,R), ancestors(X, R, C), append([t(L,R)],C,Res).
```

```
% initialize(X) :- X = t( t( t(human, bonobo), gorilla ), t(gibbon, siamang) ).
% initialize(X) :- X = t(el, t(be, t(ce, t(de, t(fe,ge))))).
```