Parsimony

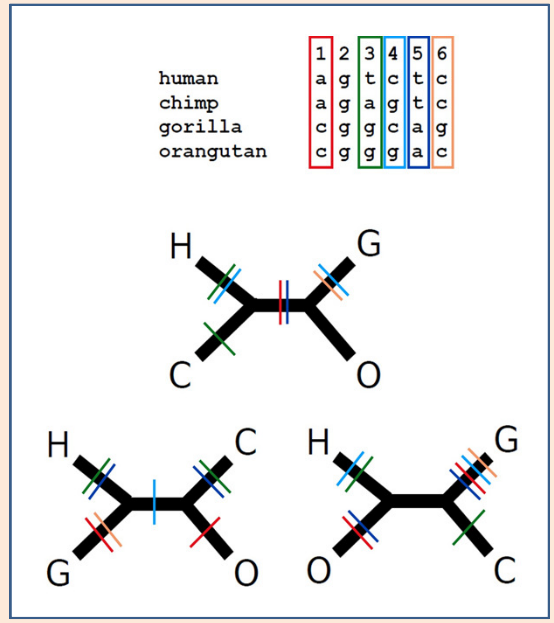
Small Parsimony and Search Algorithms

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein

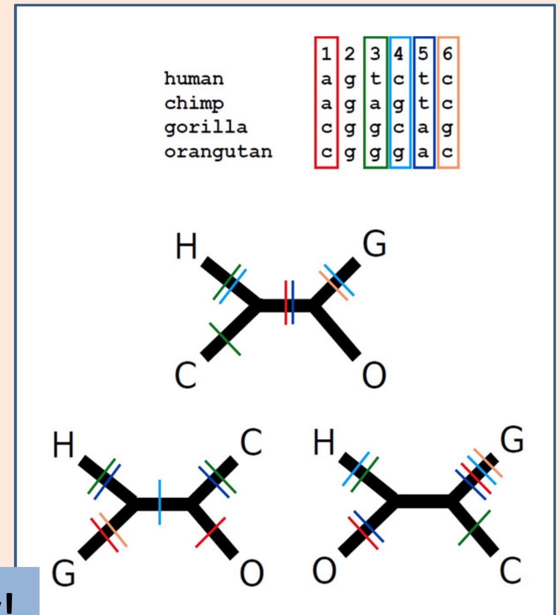
A quick review

- The parsimony principle:
  - Find the tree that requires the fewest evolutionary changes!
- A fundamentally different method:
  - Search rather than reconstruct
- Parsimony algorithm
  1. Construct all possible trees
  2. For each site in the alignment and for each tree count the minimal number of changes required
  3. Add sites to obtain the total number of changes required for each tree
  4. Pick the tree with the lowest score



A quick review

- The parsimony principle:
  - Find the tree that requires the fewest evolutionary changes!
- A fundamentally different method:
  - Search rather than reconstruct
- Parsimony algorithm
  1. Construct all possible trees — Too many!
  2. For each site in the alignment and for each tree count the minimal number of changes required — The small parsimony problem
  3. Add sites to obtain the total number of changes required for each tree
  4. Pick the tree with the lowest score



Large vs. Small Parsimony

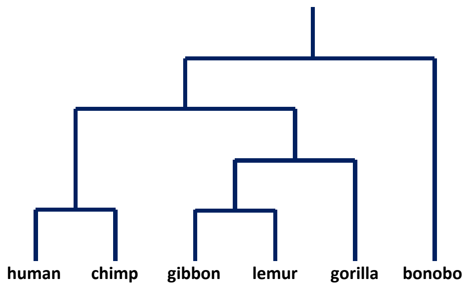
- We divided the problem of finding the most parsimonious tree into two sub-problems:
  - **Large parsimony:** Find the topology which gives best score
  - **Small parsimony**: Given a tree topology and the state in all the tips, find the minimal number of changes required

- Large parsimony is “NP-hard”

- Small parsimony can be solved quickly using Fitch’s algorithm

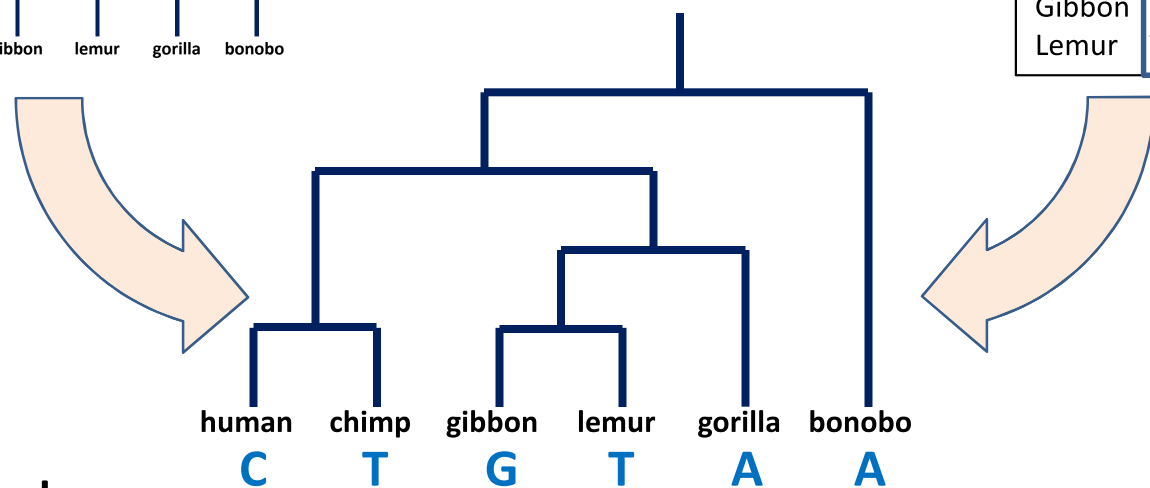
The Small Parsimony Problem

- Input:
  1. A tree topology:



  2. State assignments for all tips:

Human	C	A	C	T
Chimp	T	A	C	T
Bonobo	A	G	C	C
Gorilla	A	G	C	A
Gibbon	G	A	C	T
Lemur	T	A	G	T



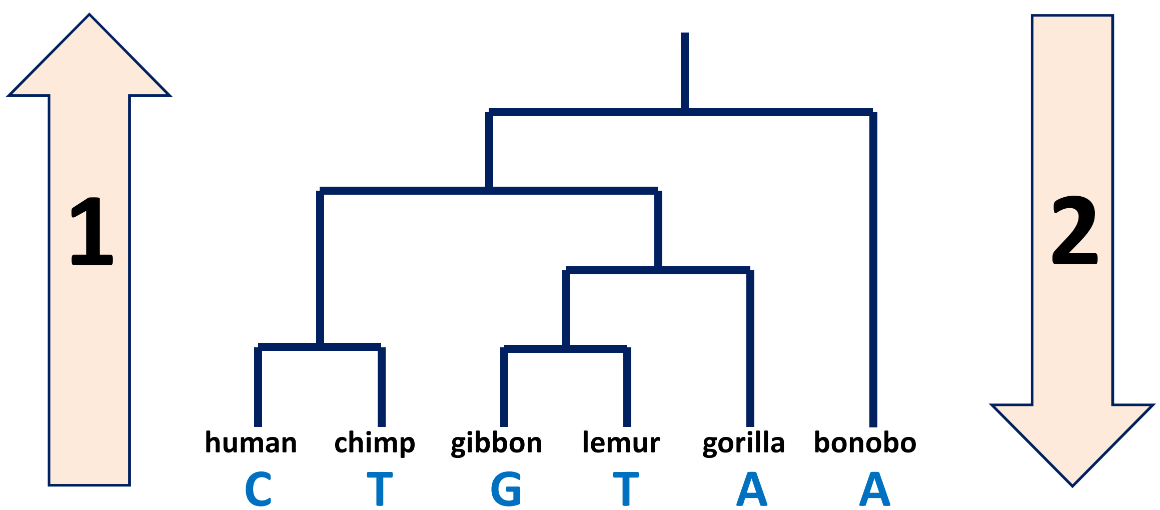
- Output:

The minimal number of changes required: ***parsimony score***

(*but in fact, we will also find the most parsimonious assignment for all internal nodes*)

Fitch's algorithm

- Execute independently for each character:
- Two phases:
  1. **Bottom-up phase**: Determine the set of possible states for each internal node
  2. **Top-down phase**: Pick a state for each internal node

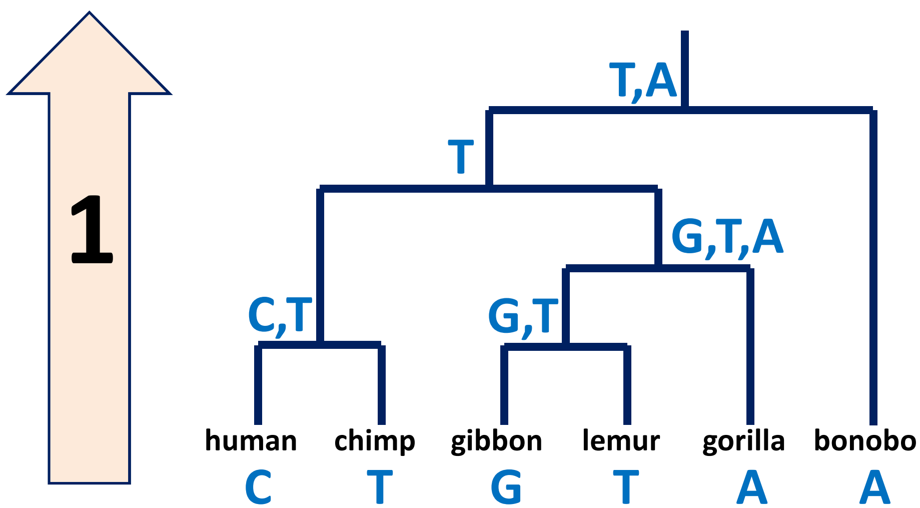


1. Fitch's algorithm: Bottom-up phase

(Determine the set of possible states for each internal node)

1. Initialization: $R_i = \{s_i\}$ for all tips
2. Traverse the tree from leaves to root ("post-order")
3. Determine R_i of internal node i with children j, k:

$$R_i = \begin{cases} if \;\; R_j \cap R_k \neq \phi \rightarrow R_j \cap R_k \\ otherwise \;\; \rightarrow R_j \cup R_k \end{cases}$$



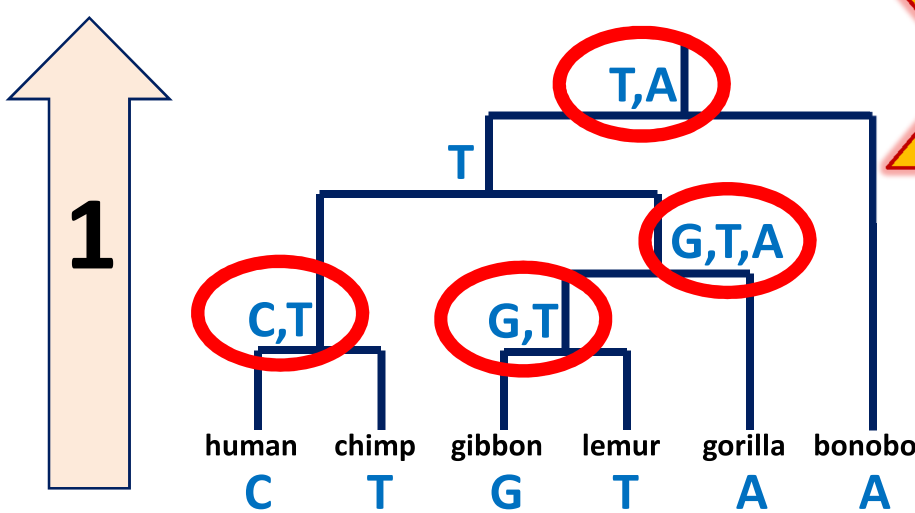
Let s_i denote the state of node i and R_i the set of possible states of node i

1. Fitch's algorithm: Bottom-up phase

(Determine the set of possible states for each internal node)

1. Initialization: $R_i = \{s_i\}$
2. Traverse the tree from leaves to root ("post-order")
3. Determine R_i of internal node i with children j, k:

$$R_i = \begin{cases} if \;\; R_j \cap R_k \neq \phi \rightarrow R_j \cap R_k \\ otherwise \;\; \rightarrow R_j \cup R_k \end{cases}$$



Parsimony-score = # union operations

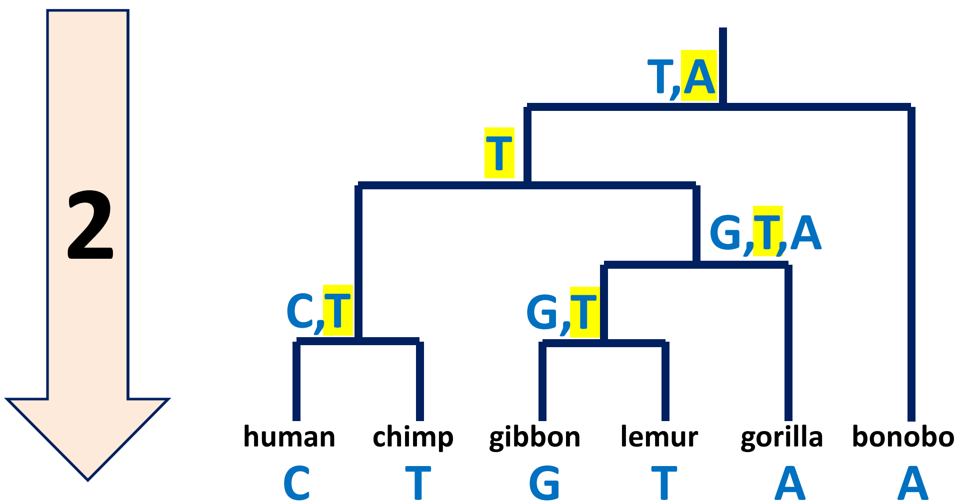
Parsimony-score = 4

2. Fitch's algorithm: Top-down phase

(Pick a state for each internal node)

1. Pick arbitrary state in R_{root} to be the state of the root ,s_{root}
2. Traverse the tree from root to leaves ("pre-order")
3. Determine s_i of internal node i with parent j:

$$s_i = \begin{cases} if \quad s_j \in R_i \rightarrow s_j \\ otherwise \rightarrow arbitrary \quad state \in R_i \end{cases}$$



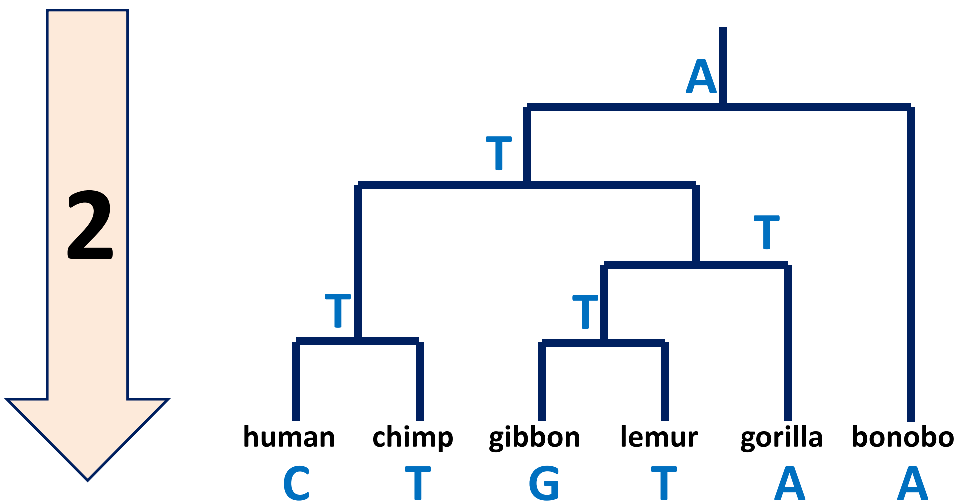
Parsimony-score = 4

2. Fitch's algorithm: Top-down phase

(Pick a state for each internal node)

1. Pick arbitrary state in R_{root} to be the state of the root ,s_{root}
2. Traverse the tree from root to leaves ("pre-order")
3. Determine s_i of internal node *i* with parent *j*:

$$s_i = \begin{cases} if \quad s_j \in R_i \rightarrow s_j \\ otherwise \rightarrow arbitrary \quad state \in R_i \end{cases}$$



Parsimony-score = 4

And now
back to the “big” parsimony problem

…

*How do we find the most parsimonious tree amongst the **many** possible trees?*

Searching tree space

- **Exhaustive search**:

  Up to 8-10 leaves (10k-2m unrooted trees, 135k-34m rooted)

  Guaranteed results

- **Branch-and-bound*:**

  Up to 10-20 leaves

  Guaranteed results!!!

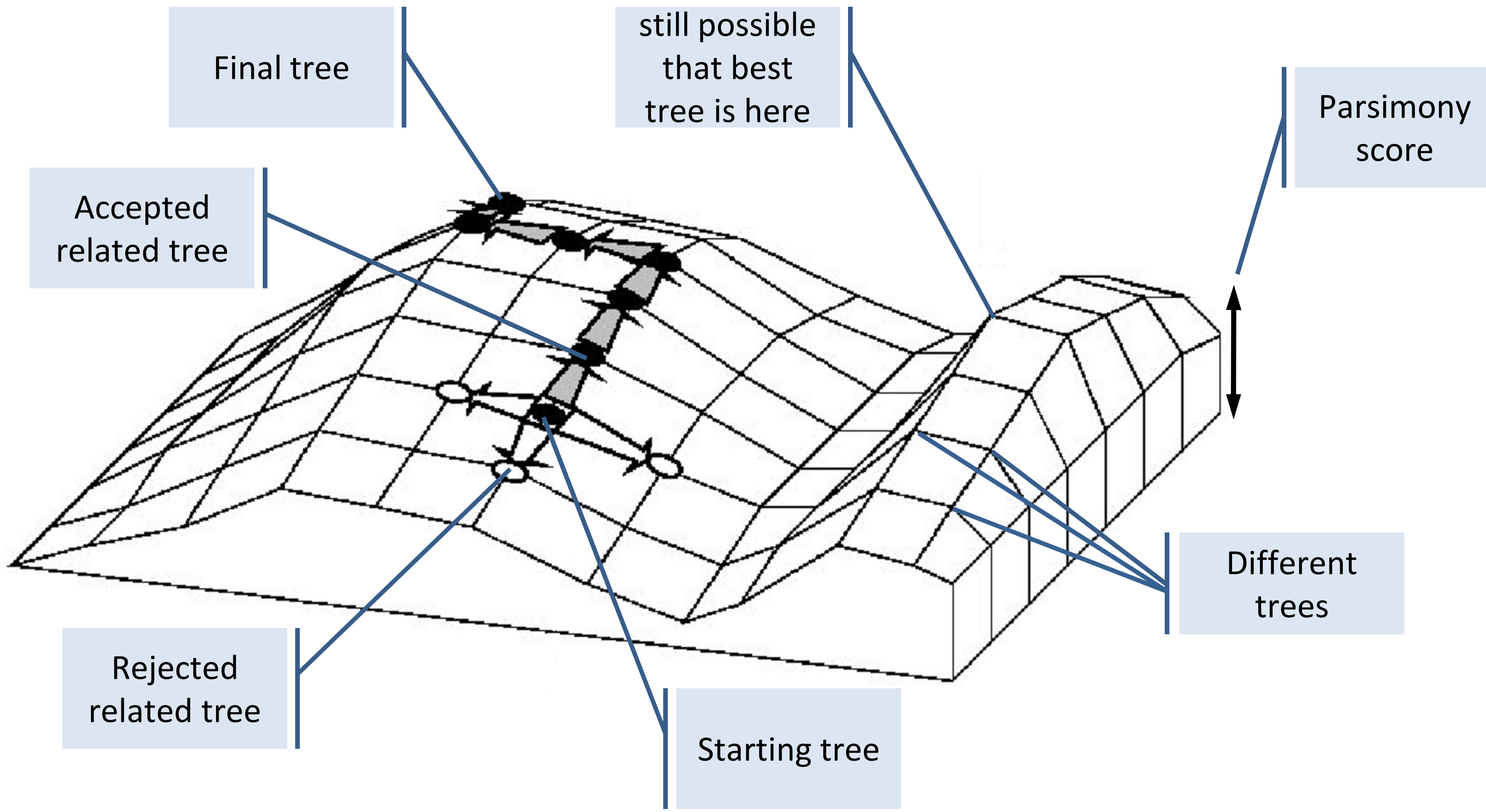
  * Branch-and-bound is a clever way of ruling out most trees as they are built, so you can evaluate more trees by exhaustive search.

- **Heuristic search** (e.g. hill-climb)**:**

  20+ leaves

  May not find correct solution.

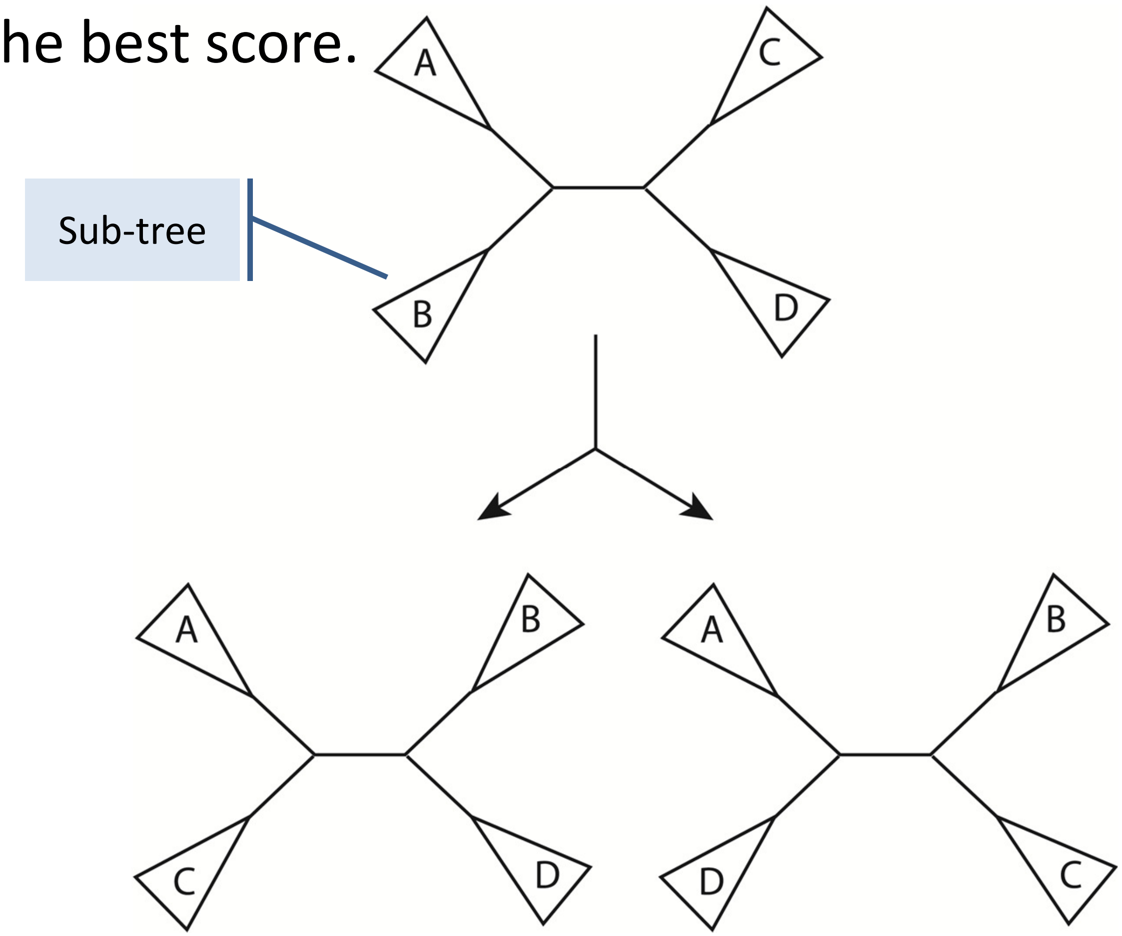
Hill-climbing

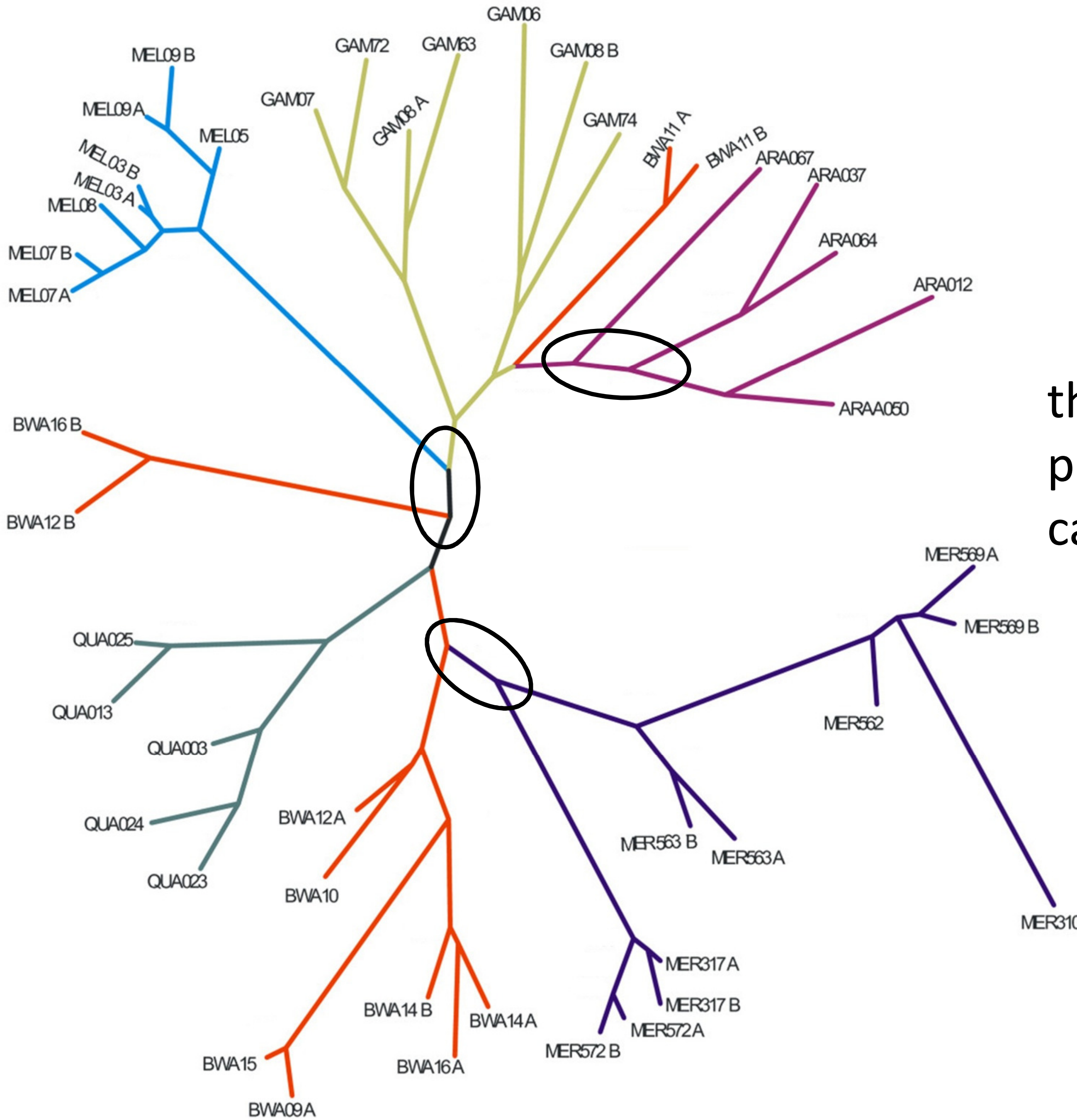


A “greedy” algorithm

Nearest-Neighbor Interchange (NNI)

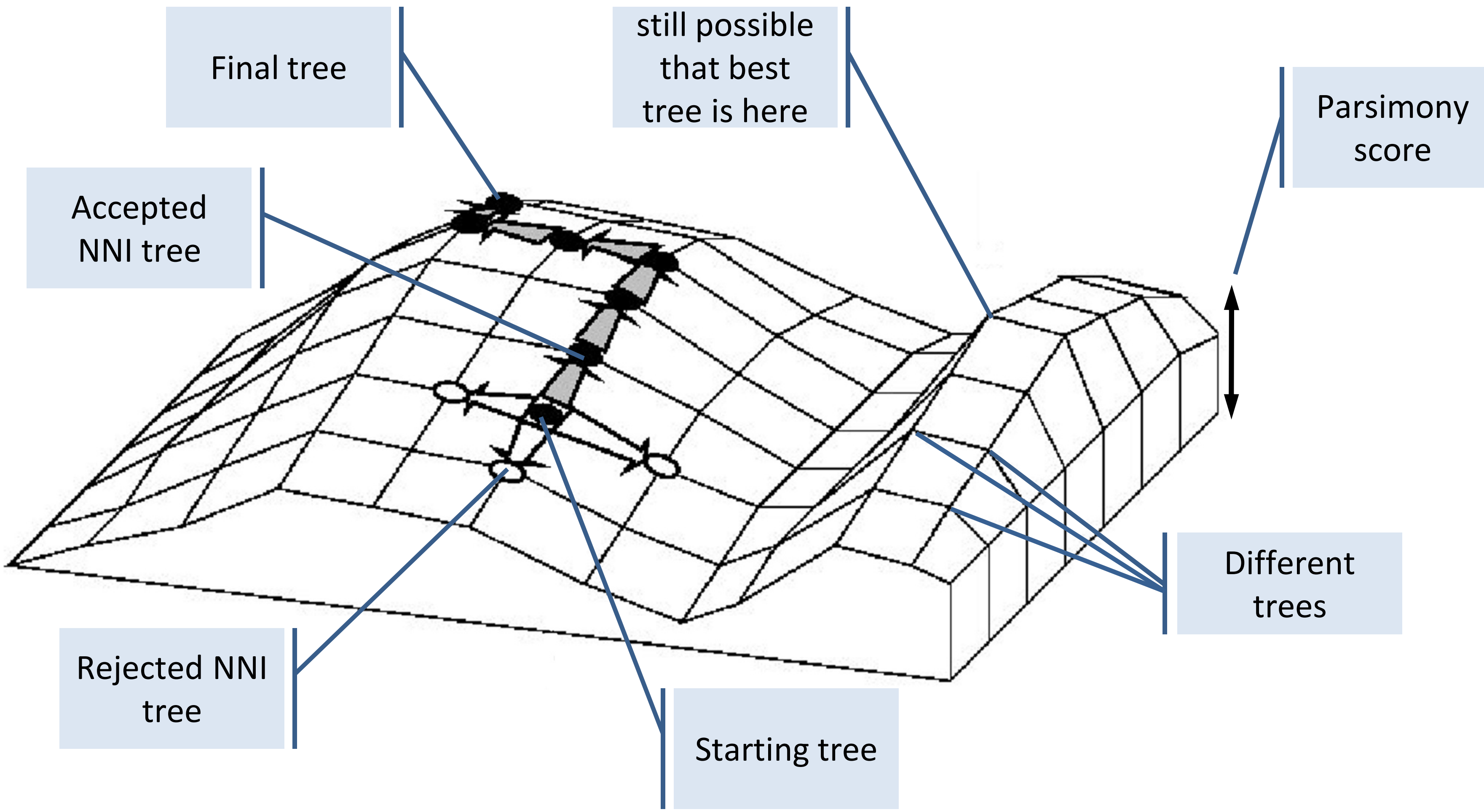
1. Find a tree with some score.
2. At **each internal branch** consider the two alternative arrangements of the 4 sub-trees.
3. Keep the tree that has the best score.
4. Repeat.





three (of many) places where NNI can be considered

Hill-climbing with NNI



A "greedy" algorithm

How can we improve this algorithm and increase our chances of finding the optimal tree?

The parsimony algorithm

1) *Construct all possible trees* ***or search the space of possible trees using NNI hill-climb***

2) *For each site in the alignment and for each tree count the minimal number of changes required* ***using Fitch's algorithm***

3) *Add all sites up to obtain the total number of changes for each tree*

4) *Pick the tree with the lowest score* ***or search until no better tree can be found***

Phylogenetic trees: Summary

Parsimony Trees:

1) *Construct all possible trees* ***or search the space of possible trees***
2) *For each site in the alignment and for each tree count the minimal number of changes required* ***using Fitch's algorithm***
3) *Add all sites up to obtain the total number of changes for each tree*
4) *Pick the tree with the lowest score*

Distance Trees:

1) *Compute pairwise corrected distances.*
2) *Build tree by sequential clustering algorithm (UPGMA or Neighbor-Joining).*
3) *These algorithms don't consider all tree topologies, so they are very fast, even for large trees.*

Maximum-Likelihood Trees:

1) *Tree evaluated for likelihood of data given tree.*
2) *Uses a specific model for evolutionary rates (such as Jukes-Cantor).*
3) *Like parsimony, must search tree space.*
4) *Usually most accurate method but slow.*

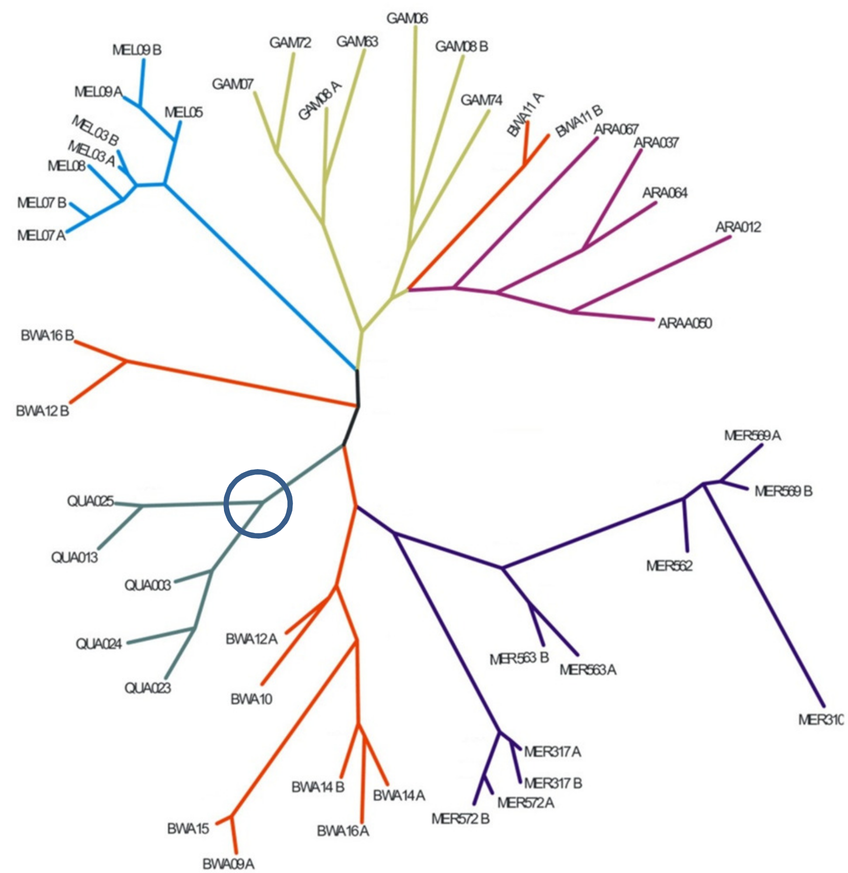
Branch confidence

How certain are we that this is the correct tree?

Can be reduced to many simpler questions - how certain are we that each **branch point** is correct?

For example, at the circled branch point, how certain are we that the three subtrees have the correct content:

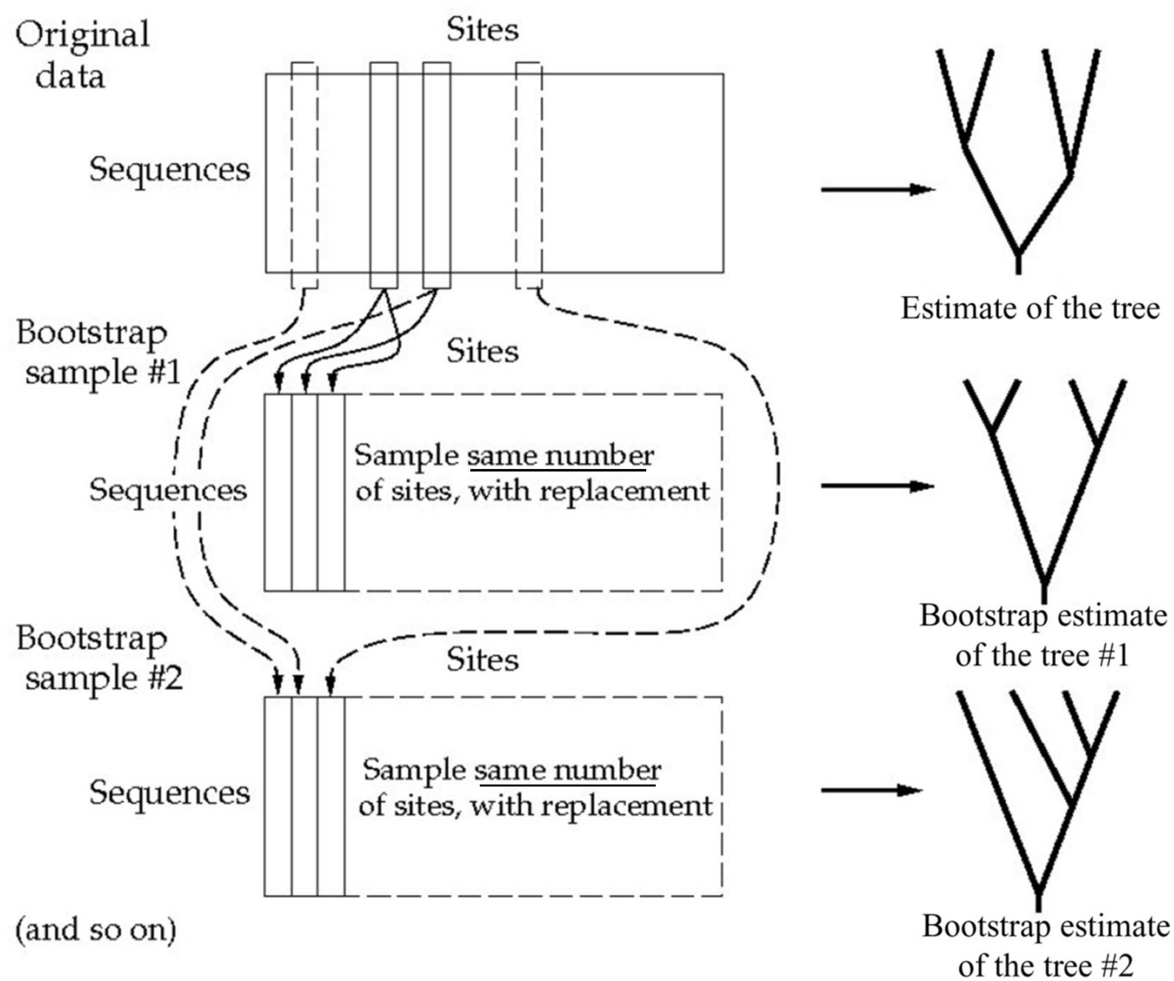
subtree1 - QUA025, QUA013
subtree2 - QUA003, QUA024, QUA023
subtree3 - everything else



Bootstrap support

Most commonly used branch support test:

1. *Randomly sample alignment sites.*
2. *Use sample to estimate the tree.*
3. *Repeat many times.*



(sample with replacement means that a sampled site remains in the source data after each sampling, so that some sites will be sampled more than once)

Bootstrap support

For each branch point on the computed tree, count what fraction of the bootstrap trees have the same subtree partitions (regardless of topology within the subtrees).

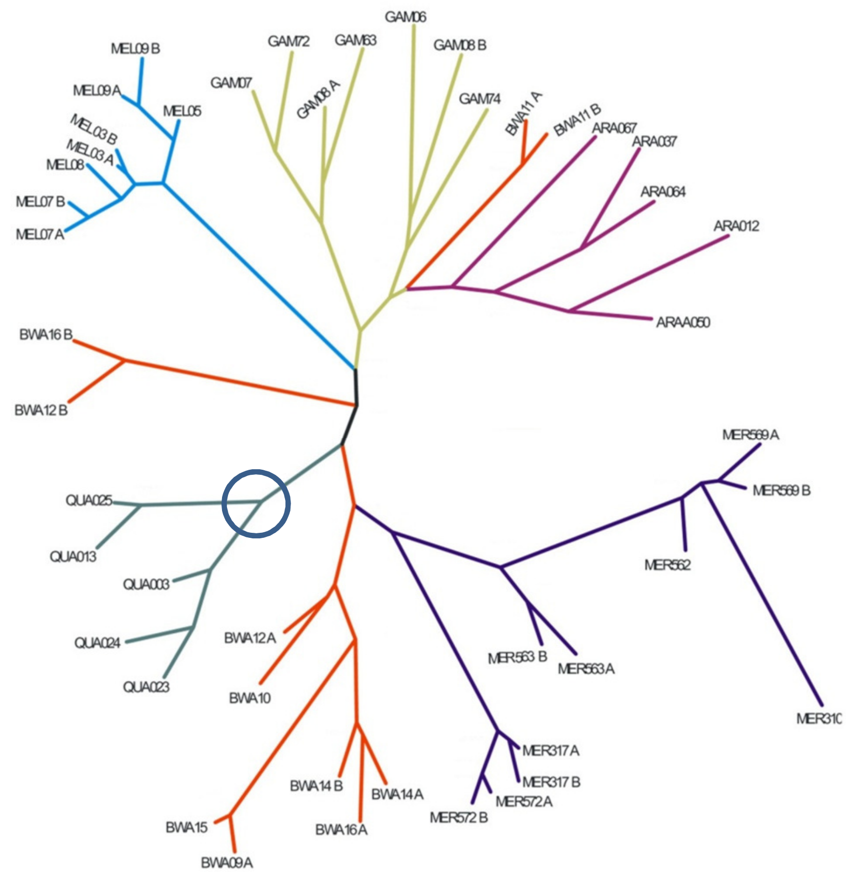
For example at the circled branch point, what fraction of the bootstrap trees have a branch point where the three subtrees include:

subtree1 - QUA025, QUA013

subtree2 - QUA003, QUA024, QUA023

subtree3 - everything else

This fraction is the **bootstrap support** for that branch.



Original tree figure with branch supports
(here as fractions, also common to give % support)

low-confidence branches
are marked