Molecular Evolution

Evolutionary Trees

How are these trees built from DNA sequences?

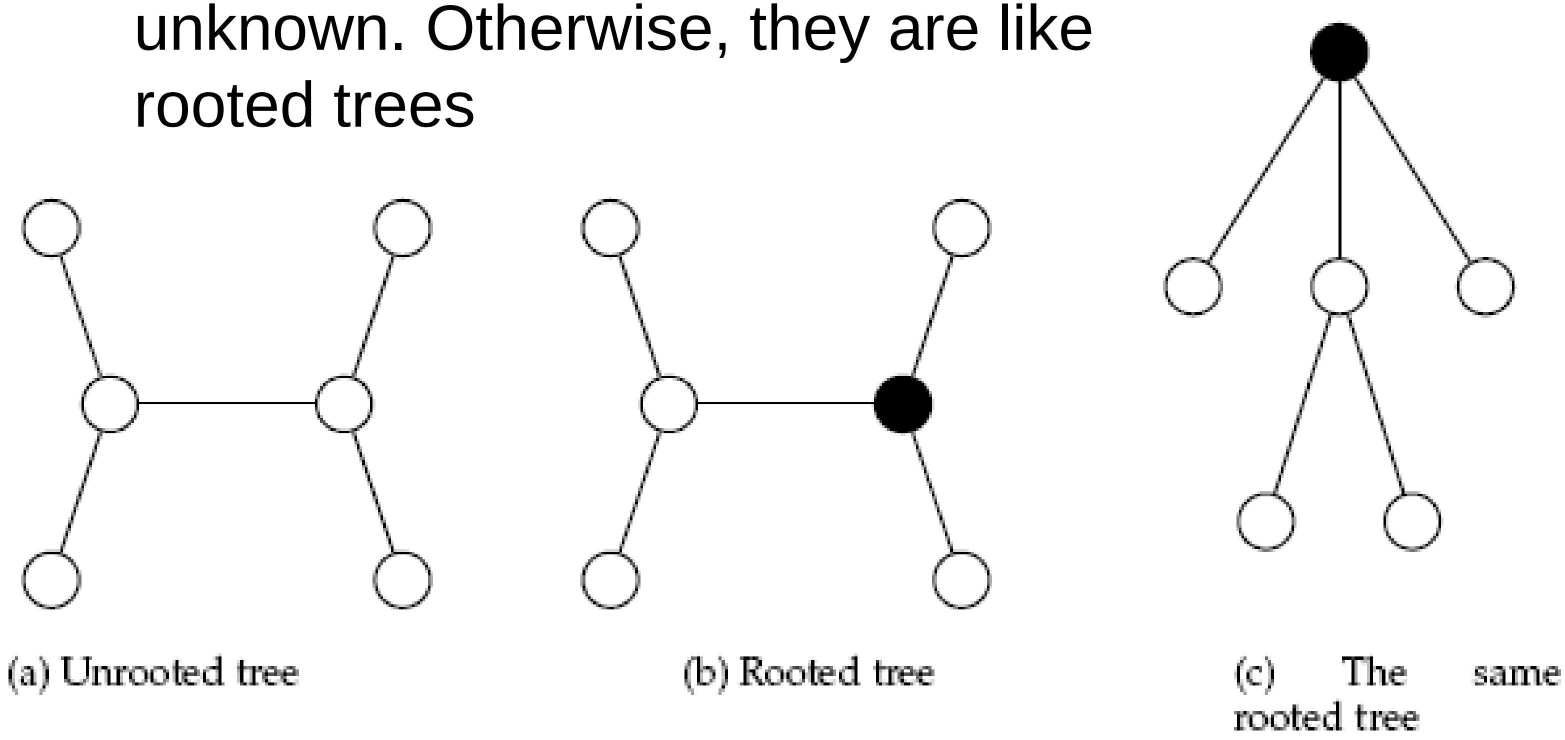
Evolutionary Trees

How are these trees built from DNA sequences?

- leaves represent existing species
- internal vertices represent ancestors
- root represents the oldest evolutionary ancestor

Rooted and Unrooted Trees

In the unrooted tree the position of the root (“oldest ancestor”) is unknown. Otherwise, they are like rooted trees



(a) Unrooted tree

(b) Rooted tree

(c) The same rooted tree

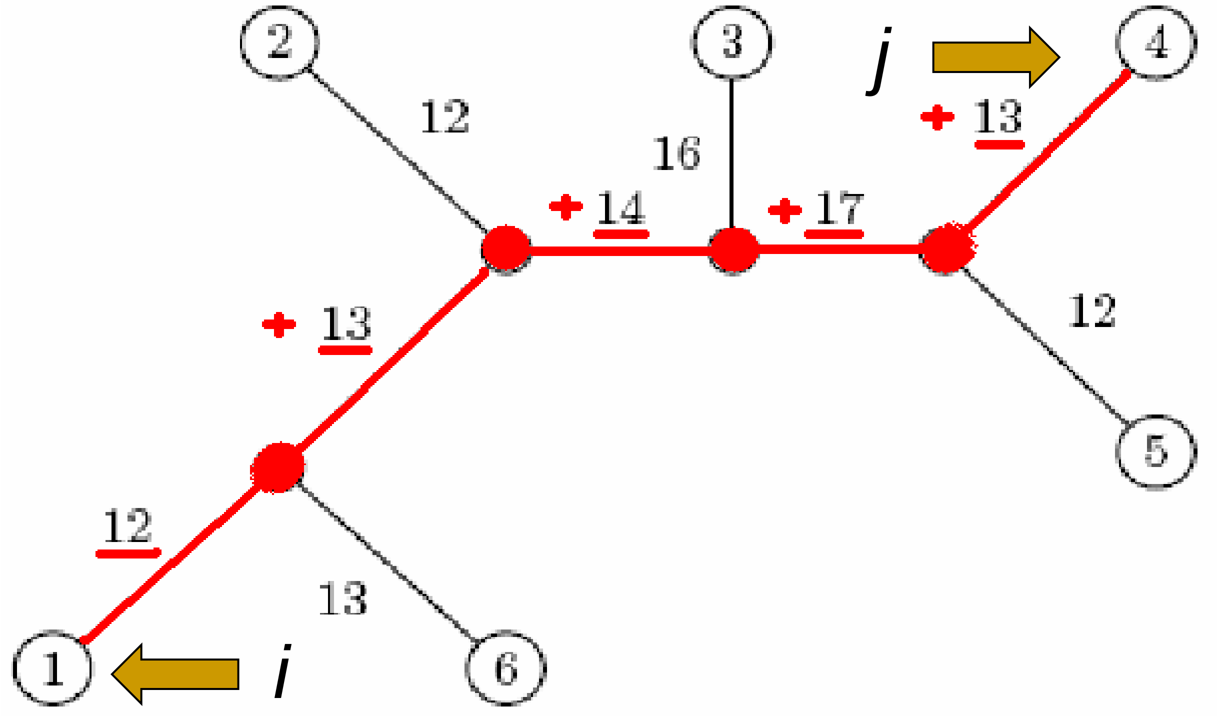
Distances in Trees

- Edges may have weights reflecting:
  - Number of mutations on evolutionary path from one species to another
  - Time estimate for evolution of one species into another
- In a tree *T*, we often compute

$d_{ij}(T)$ - the length of a path between leaves *i* and *j*

$d_{ij}(T)$ – ***tree distance between i and j***

Distance in Trees: an Exampe



$$d_{1,4} = 12 + 13 + 14 + 17 + 12 = 68$$

Distance Matrix

- Given n species, we can compute the n x n ***distance matrix*** D_{ij}
- D_{ij} may be defined as the edit distance between a gene in species i and species j, where the gene of interest is sequenced for all n species.

D_{ij} – ***edit distance between i and j***

Edit Distance vs. Tree Distance

- Given *n* species, we can compute the *n* x *n* ***distance matrix*** D_{ij}
- D_{ij} may be defined as the edit distance between a gene in species *i* and species *j*, where the gene of interest is sequenced for all *n* species.

  D_{ij} – ***edit distance between i and j***
- Note the difference with

  $d_{ij}(T)$ – ***tree distance between i and j***

Fitting Distance Matrix

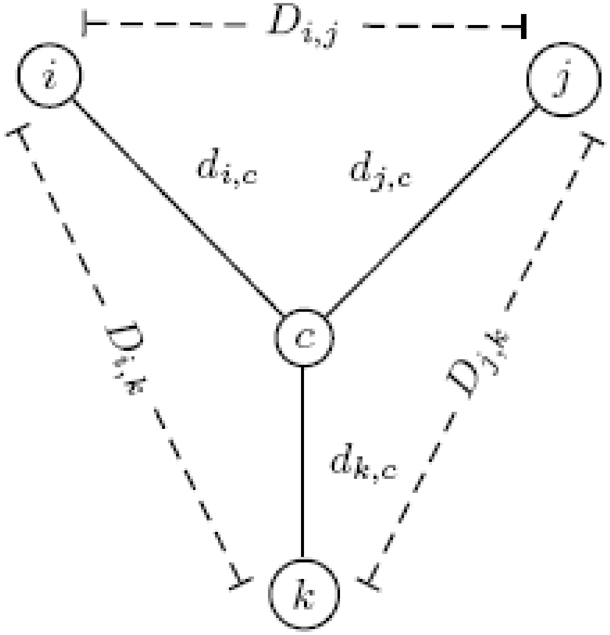
- Given *n* species, we can compute the *n* x *n* ***distance matrix*** D_{ij}
- Evolution of these genes is described by a tree that ***we don't know***.
- We need an algorithm to construct a tree that best ***fits*** the distance matrix D_{ij}

Fitting Distance Matrix

- Fitting means $\underbrace{D_{ij}}_{\text{Edit distance between species (}\textbf{\textit{known}}\text{)}} = \overbrace{d_{ij}(T)}^{\text{Lengths of path in an (}\textbf{\textit{unknown}}\text{) tree } T}$

Reconstructing a 3 Leaved Tree

- Tree reconstruction for any 3x3 matrix is straightforward
- We have 3 leaves *i*, *j*, *k* and a center vertex *c*



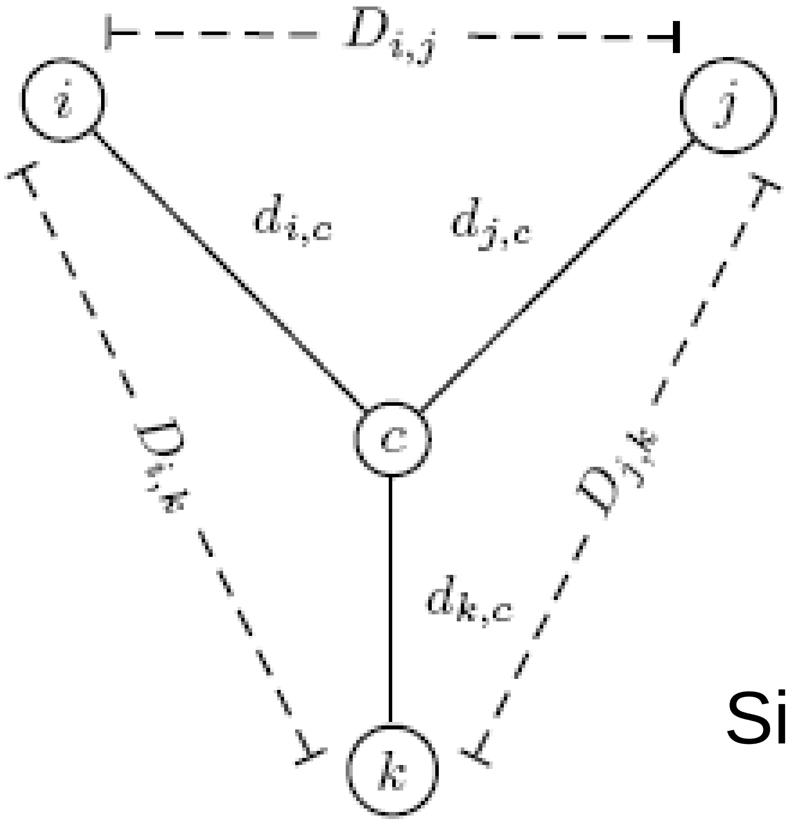
Observe:

$$d_{ic} + d_{jc} = D_{ij}$$

$$d_{ic} + d_{kc} = D_{ik}$$

$$d_{jc} + d_{kc} = D_{jk}$$

Reconstructing a 3 Leaved Tree (cont'd)



$$d_{ic} + d_{jc} = D_{ij}$$

$$\underline{+\ \ d_{ic} + d_{kc} = D_{ik}}$$

$$2d_{ic} + \underbrace{d_{jc} + d_{kc}} = D_{ij} + D_{ik}$$

$$2d_{ic} + \quad D_{jk} \quad = D_{ij} + D_{ik}$$

$$\mathbf{d_{ic} = (D_{ij} + D_{ik} - D_{jk})/2}$$

Similarly,

$$\mathbf{d_{jc} = (D_{ij} + D_{jk} - D_{ik})/2}$$

$$\mathbf{d_{kc} = (D_{ki} + D_{kj} - D_{ij})/2}$$

Trees with > 3 Leaves

- An tree with *n* leaves has *2n-3* edges

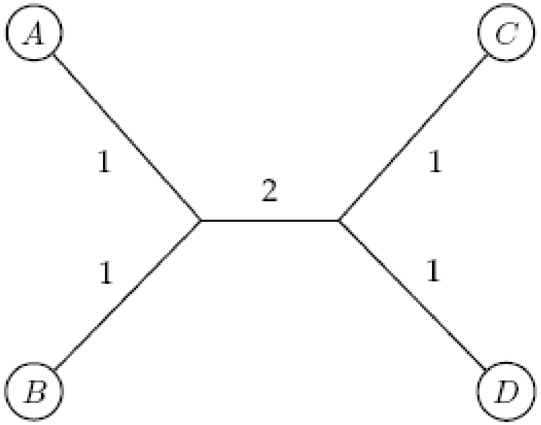
- This means fitting a given tree to a distance matrix *D* requires solving a system of “n choose 2” equations with *2n-3* variables

- This is not always possible to solve for *n* > 3

Additive Distance Matrices

Matrix *D* is ADDITIVE if there exists a tree *T* with $d_{ij}(T) = D_{ij}$

	A	B	C	D
A	0	2	4	4
B	2	0	4	4
C	4	4	0	2
D	4	4	2	0



NON-ADDITIVE otherwise

	A	B	C	D
A	0	2	2	2
B	2	0	3	2
C	2	3	0	2
D	2	2	2	0

?

Distance Based Phylogeny Problem

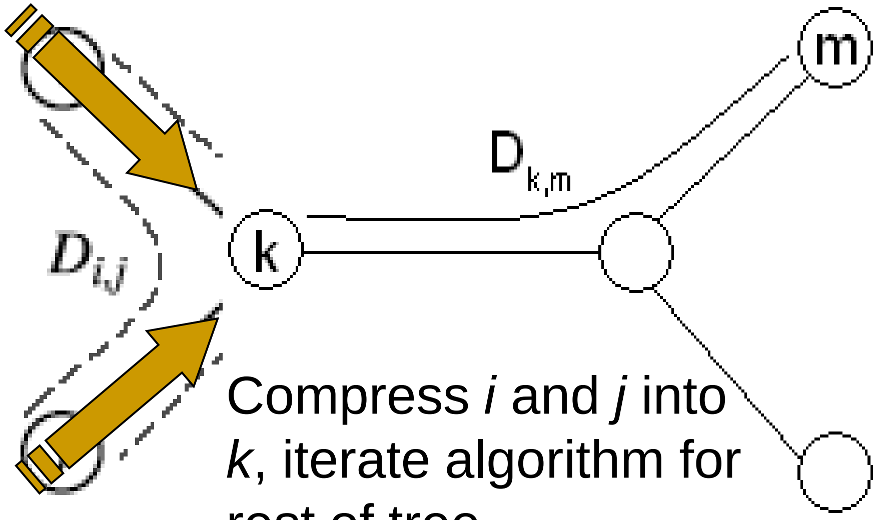
- Goal: Reconstruct an evolutionary tree from a distance matrix
- Input: *n* x *n* distance matrix D_{ij}
- Output: weighted tree *T* with *n* leaves fitting *D*

- If *D* is additive, this problem has a solution and there is a simple algorithm to solve it

Using Neighboring Leaves to Construct the Tree

- Find ***neighboring leaves*** *i* and *j* with parent *k*
- Remove the rows and columns of *i* and j
- Add a new row and column corresponding to *k*, where the distance from *k* to any other leaf *m* can be computed as:

$$D_{km} = (D_{im} + D_{jm} - D_{ij})/2$$



Compress *i* and *j* into *k*, iterate algorithm for rest of tree

Finding Neighboring Leaves

- To find neighboring leaves we simply select a pair of closest leaves.

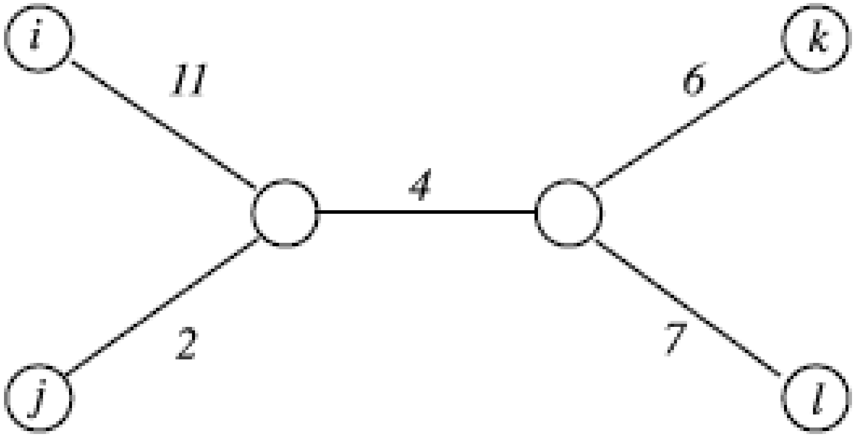
Finding Neighboring Leaves

- To find neighboring leaves we simply select a pair of closest leaves.

WRONG

Finding Neighboring Leaves

- Closest leaves aren’t necessarily neighbors
- *i* and *j* are neighbors, but (d_{ij} = 13) > (d_{jk} = 12)



- Finding a pair of neighboring leaves is a nontrivial problem!

Neighbor Joining Algorithm

- In 1987 Naruya Saitou and Masatoshi Nei developed a neighbor joining algorithm for phylogenetic tree reconstruction

- **Finds a pair of leaves that are close to each other but far from other leaves:** implicitly finds a pair of neighboring leaves

- Advantages: works well for additive and other non-additive matrices, it does not have the flawed molecular clock assumption

Degenerate Triples

- A degenerate triple is a set of three distinct elements $1 \leq i,j,k \leq n$ where $D_{ij} + D_{jk} = D_{ik}$

- Element *j* in a degenerate triple *i,j,k* lies on the evolutionary path from *i* to *k* (or is attached to this path by an edge of length 0).

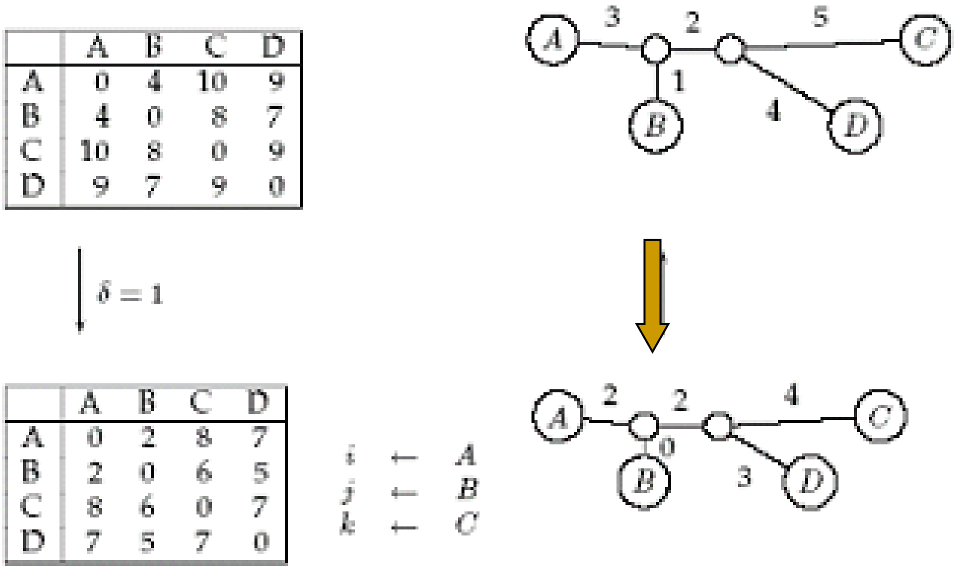
Looking for Degenerate Triples

- If distance matrix *D* **has** a degenerate triple *i,j,k* then *j* can be "removed" from *D* thus reducing the size of the problem.

- If distance matrix *D* **does not have** a degenerate triple *i,j,k, one can "create"* a degenerative triple in *D* by shortening all hanging edges (in the tree).

Shortening Hanging Edges to Produce Degenerate Triples

- Shorten all “hanging” edges (edges that connect leaves) until a degenerate triple is found



	A	B	C	D
A	0	4	10	9
B	4	0	8	7
C	10	8	0	9
D	9	7	9	0

$\delta = 1$

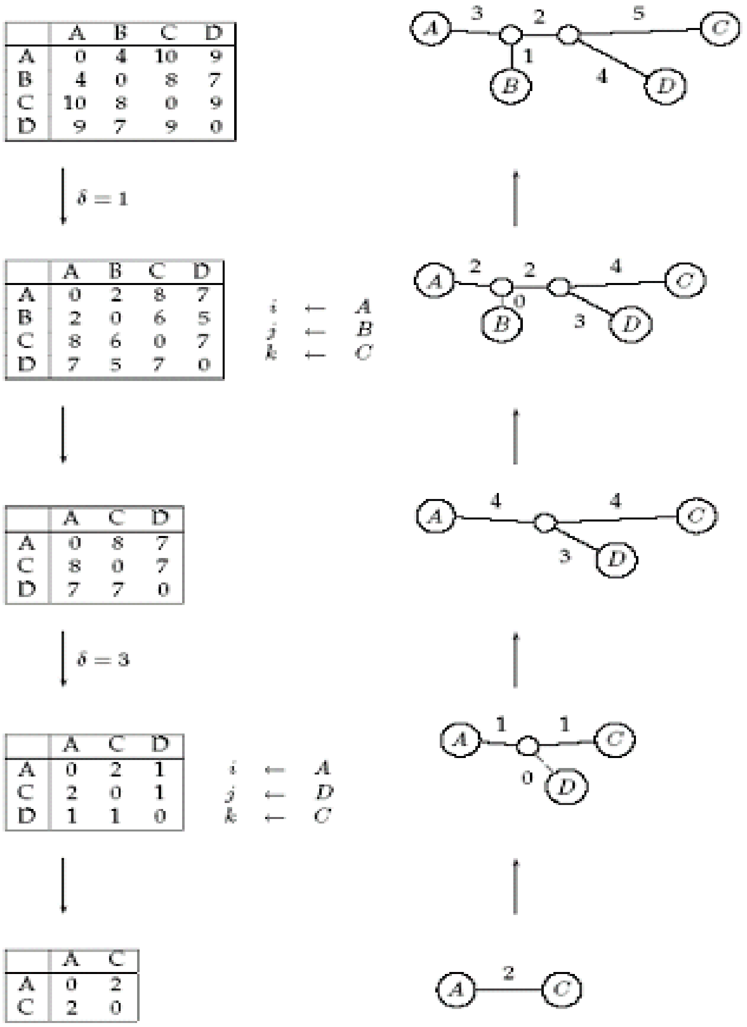
	A	B	C	D
A	0	2	8	7
B	2	0	6	5
C	8	6	0	7
D	7	5	7	0

$i \leftarrow A$
$j \leftarrow B$
$k \leftarrow C$

Finding Degenerate Triples

- If there is no degenerate triple, all hanging edges are reduced by the same amount δ, so that all pair-wise distances in the matrix are reduced by 2δ.
- Eventually this process collapses one of the leaves (when δ = length of shortest hanging edge), forming a degenerate triple *i,j,k* and reducing the size of the distance matrix *D.*
- The attachment point for *j* can be recovered in the reverse transformations by saving D_{ij} for each collapsed leaf.

Reconstructing Trees for Additive Distance Matrices



	A	B	C	D
A	0	4	10	9
B	4	0	8	7
C	10	8	0	9
D	9	7	9	0

$\delta = 1$

	A	B	C	D
A	0	2	8	7
B	2	0	6	5
C	8	6	0	7
D	7	5	7	0

$i \leftarrow A$
$j \leftarrow B$
$k \leftarrow C$

	A	C	D
A	0	8	7
C	8	0	7
D	7	7	0

$\delta = 3$

	A	C	D
A	0	2	1
C	2	0	1
D	1	1	0

$i \leftarrow A$
$j \leftarrow D$
$k \leftarrow C$

	A	C
A	0	2
C	2	0

AdditivePhylogeny Algorithm

```
AdditivePhylogeny($D$)
  if $D$ is a $2$ x $2$ matrix
    $T$ = tree of a single edge of length $D_{1,2}$
    return $T$
  if $D$ is non-degenerate
    $\delta$ = trimming parameter of matrix $D$
    for all $1 \leq i \neq j \leq n$
      $D_{ij} = D_{ij} - 2\delta$
  else
    $\delta = 0$
```

AdditivePhylogeny (cont'd)

```
Find a triple $i, j, k$ in $D$ such that $D_{ij} + D_{jk} = D_{ik}$
$x = D_{ij}$
Remove $j^{th}$ row and $j^{th}$ column from $D$
$T$ = AdditivePhylogeny($D$)
Add a new vertex $v$ to $T$ at distance $x$ from $i$ to $k$
Add $j$ back to $T$ by creating an edge ($v$,$j$) of length 0
for every leaf $l$ in $T$
  if distance from $l$ to $v$ in the tree $\neq D_{l,j}$
    output "matrix is not additive"
    return
Extend all "hanging" edges by length $\delta$
return $T$
```

The Four Point Condition

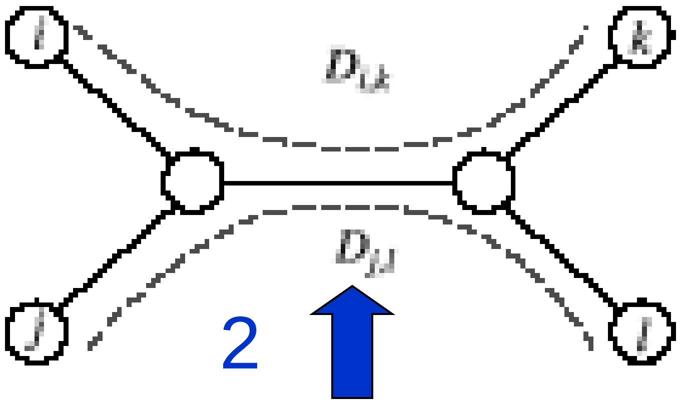
- AdditivePhylogeny provides a way to check if distance matrix *D* is additive

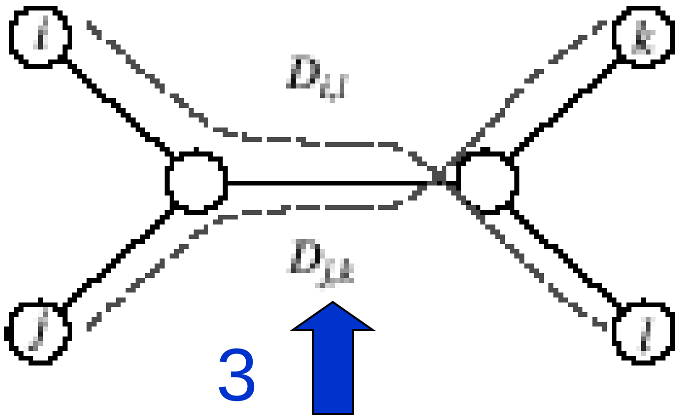
- **An even more efficient additivity check is the "four-point condition"**

- Let $1 \leq i,j,k,l \leq n$ be four distinct leaves in a tree

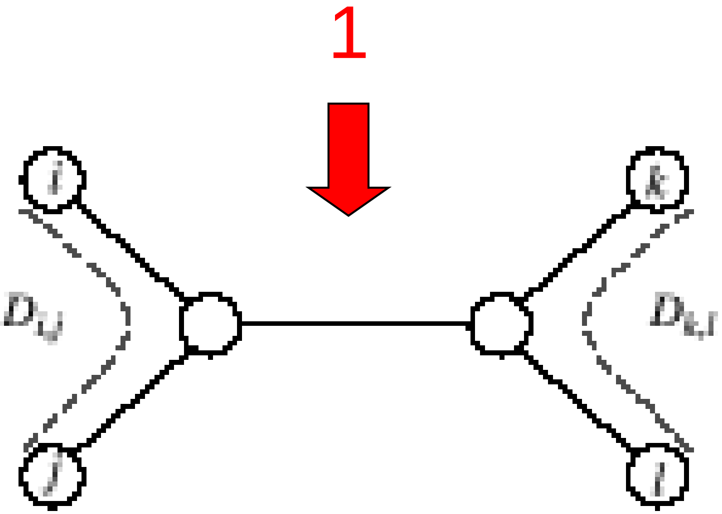
The Four Point Condition (cont'd)

Compute: 1. $D_{ij} + D_{kl}$, 2. $D_{ik} + D_{jl}$, 3. $D_{il} + D_{jk}$





2 and **3** represent the **same number**: **the length of all edges + the middle edge (it is counted twice)**



1 represents a **smaller number**: **the length of all edges – the middle edge**

The Four Point Condition: Theorem

- The four point condition for the quartet *i,j,k,l* is satisfied if two of these sums are the same, with the third sum smaller than these first two

- ***Theorem*** : An *n* x *n* matrix *D* is additive if and only if the four point condition holds for ***every*** quartet $1 \leq i,j,k,l \leq n$

Least Squares Distance Phylogeny Problem

- If the distance matrix *D* is NOT additive, then we look for a tree *T* that approximates *D* the best:

Squared Error : $\sum_{i,j} (d_{ij}(T) - D_{ij})^2$

- Squared Error is a measure of the quality of the fit between distance matrix and the tree: we want to minimize it.

- **Least Squares Distance Phylogeny Problem**: finding the best approximation tree *T* for a non-additive matrix *D* (NP-hard).

UPGMA: Unweighted Pair Group Method with Arithmetic Mean

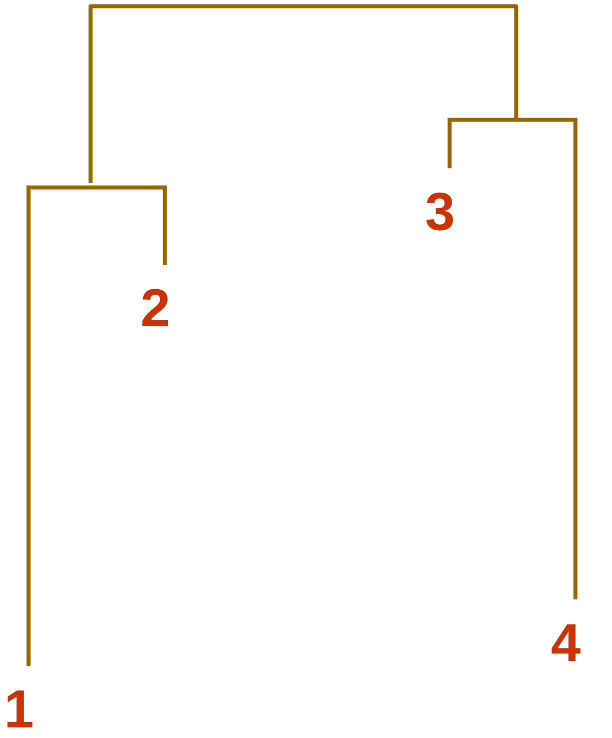
- UPGMA is a clustering algorithm that:
  - computes the distance between clusters using average pairwise distance
  - assigns a *height* to every vertex in the tree, effectively assuming the presence of a molecular clock and dating every vertex

UPGMA’s Weakness

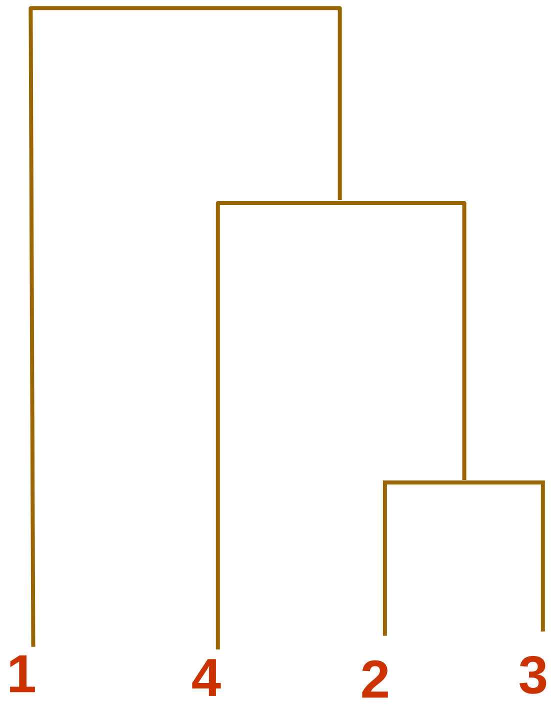
- The algorithm produces an **ultrametric** tree : the distance from the root to any leaf is the same
  - UPGMA assumes a constant molecular clock: all species represented by the leaves in the tree are assumed to accumulate mutations (and thus evolve) at the same rate. This is a major pitfalls of UPGMA.

UPGMA’s Weakness: Example

Correct tree



UPGMA



Clustering in UPGMA

Given two disjoint clusters C_i, C_j of sequences,

$$d_{ij} = \frac{1}{|C_i| \times |C_j|} \Sigma_{\{p \in Ci,\, q \in Cj\}} d_{pq}$$

Note that if $C_k = C_i \cup C_j$, then distance to another cluster C_l is:

$$d_{kl} = \frac{d_{il}\,|C_i| + d_{jl}\,|C_j|}{|C_i| + |C_j|}$$

UPGMA Algorithm

Initialization:

Assign each x_i to its own cluster C_i

Define one leaf per sequence, each at height 0

Iteration:

Find two clusters C_i *and* C_j such that d_{ij} is min

Let $C_k = C_i \cup C_j$

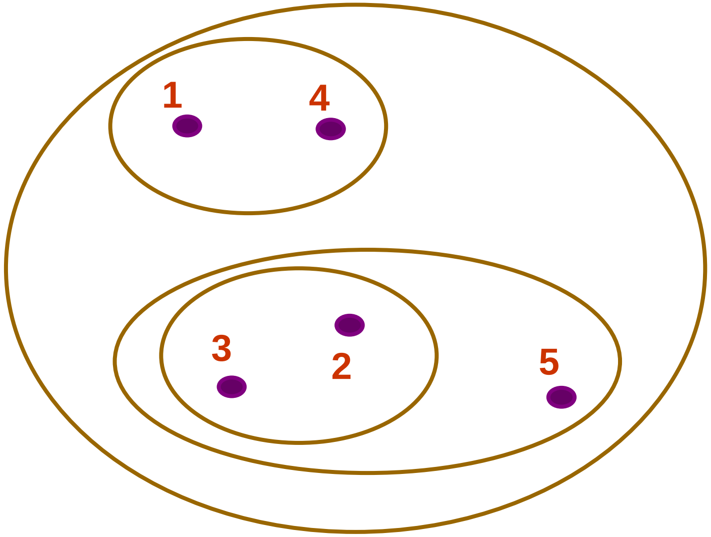
Add a vertex connecting C_i, C_j *and* place it at height $d_{ij}/2$

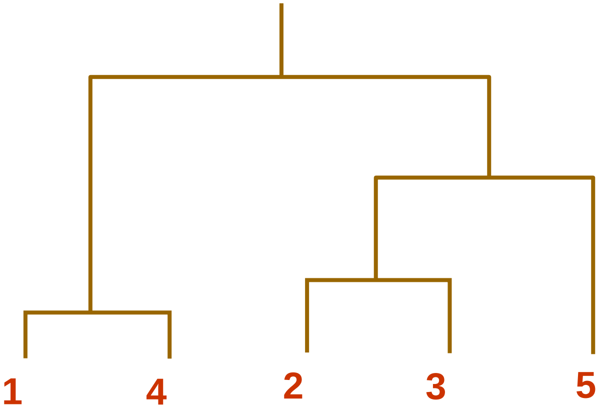
Delete C_i *and* C_j

Termination:

When a single cluster remains

UPGMA Algorithm (cont’d)

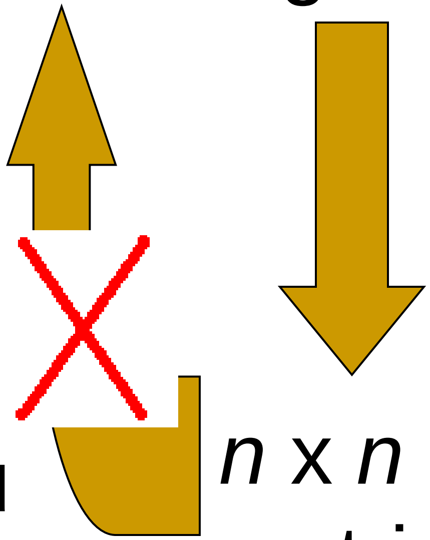




Alignment Matrix vs. Distance Matrix

Sequence a gene of length *m* nucleotides in *n* species to generate an…

n x *m* alignment matrix



CANNOT be transformed back into alignment matrix because information was lost on the forward transformation

Transform into…

n x *n* distance matrix

Character-Based Tree Reconstruction

- **Better technique**:
  - Character-based reconstruction algorithms use the *n* x *m* alignment matrix

    (*n* = # species, *m* = #characters)

    directly instead of using distance matrix.
  - **GOAL**: determine what character strings at internal nodes would best explain the character strings for the *n* observed species

Character-Based Tree Reconstruction (cont'd)

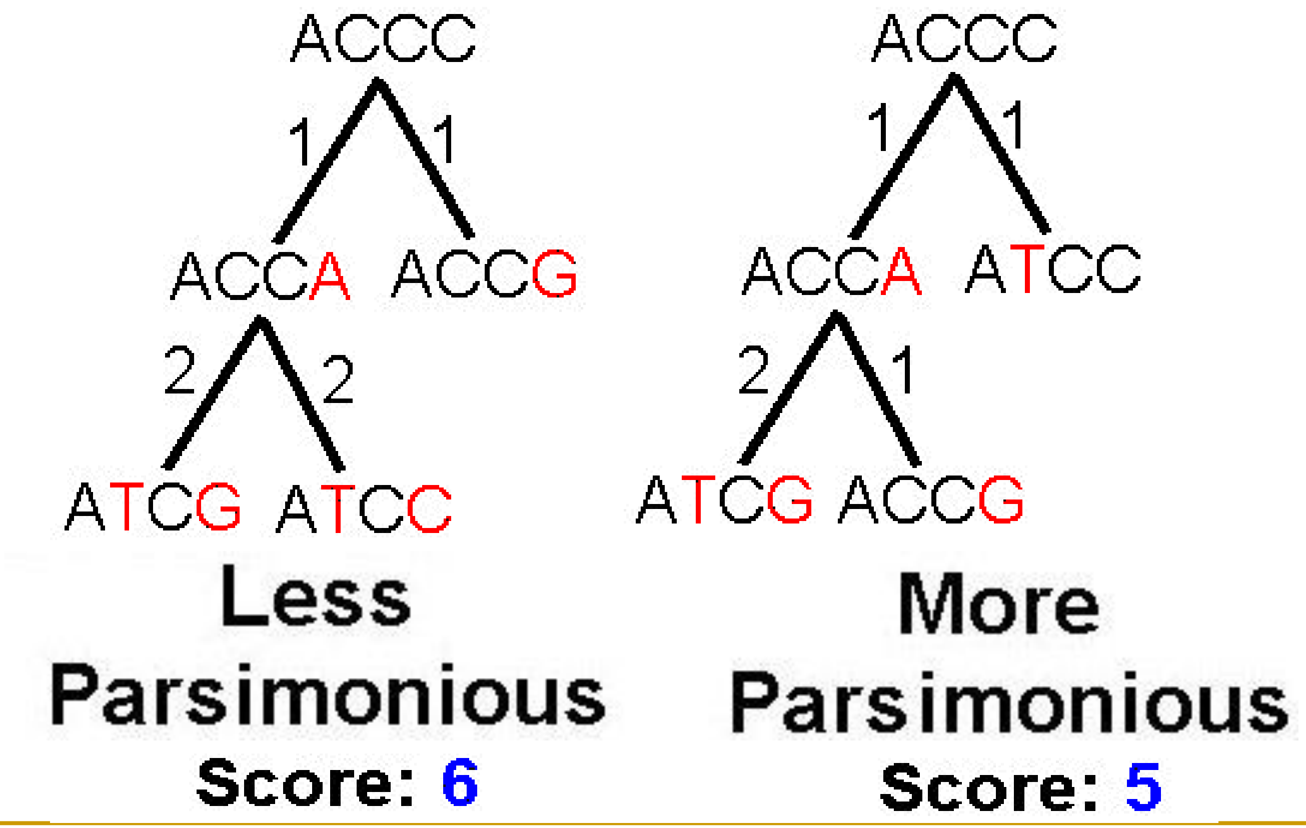
- Characters may be nucleotides, where A, G, C, T are **states** of this character. Other characters may be the # of eyes or legs or the shape of a beak or a fin.

- By setting the length of an edge in the tree to the Hamming distance, we may define the **parsimony score** of the tree as the sum of the lengths (weights) of the edges

Parsimony Approach to Evolutionary Tree Reconstruction

- Applies Occam’s razor principle to identify the simplest explanation for the data
- Assumes observed character differences resulted from the fewest possible mutations
- Seeks the tree that yields lowest possible **parsimony score -** sum of cost of all mutations found in the tree

Parsimony and Tree Reconstruction



Character-Based Tree Reconstruction (cont'd)

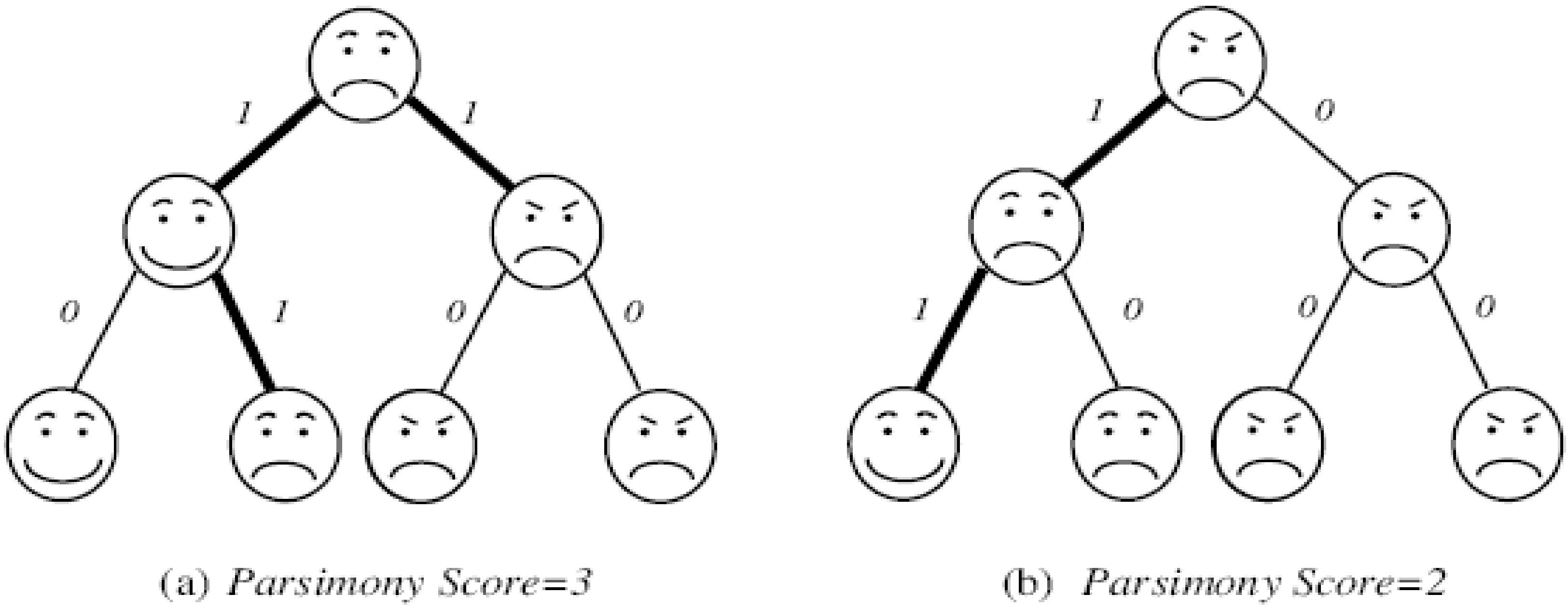


Figure 10.16 If we label a tree's leaves with characters (in this case, eyebrows and mouth, each with two states), and choose labels for each internal vertex, we implicitly create a *parsimony* score for the tree. By changing the labels in (a) we are able to create a tree with a better parsimony score in (b).

Small Parsimony Problem

- Input: Tree *T* with each leaf labeled by an *m*-character string.

- Output: Labeling of internal vertices of the tree *T* minimizing the parsimony score.

- We can assume that every leaf is labeled by a single character, because the characters in the string are independent.

Weighted Small Parsimony Problem

- A more general version of Small Parsimony Problem
- Input includes a *k* * *k* scoring matrix describing the cost of transformation of each of *k* states into another one
- For Small Parsimony problem, the scoring matrix is based on Hamming distance

  $d_H(v, w) = 0$ if $v=w$

  $d_H(v, w) = 1$ otherwise

Scoring Matrices

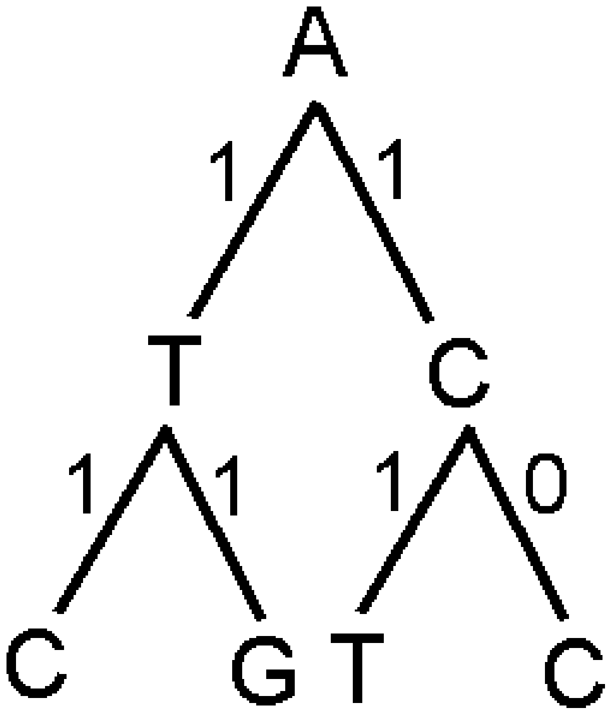
Small Parsimony Problem

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

Weighted Parsimony Problem

	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

Unweighted vs. Weighted

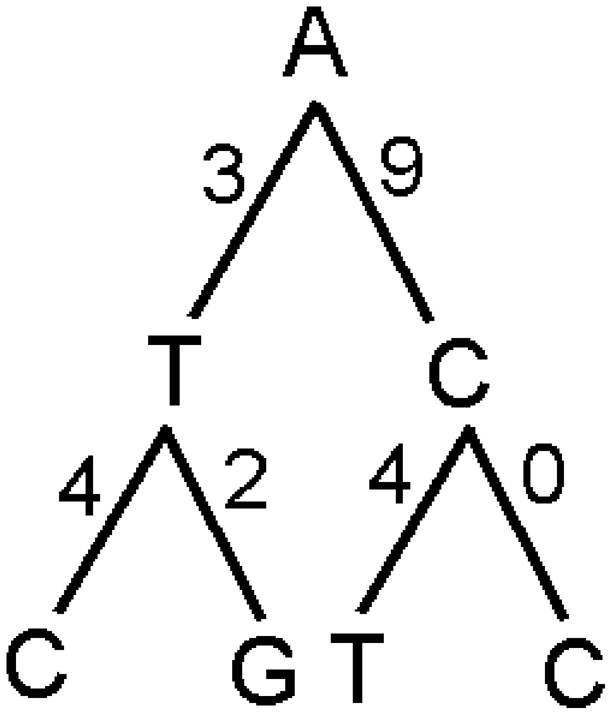


Small Parsimony Scoring Matrix:

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

Small Parsimony Score: 5

Unweighted vs. Weighted



Weighted Parsimony Scoring Matrix:

	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

Weighted Parsimony Score: 22

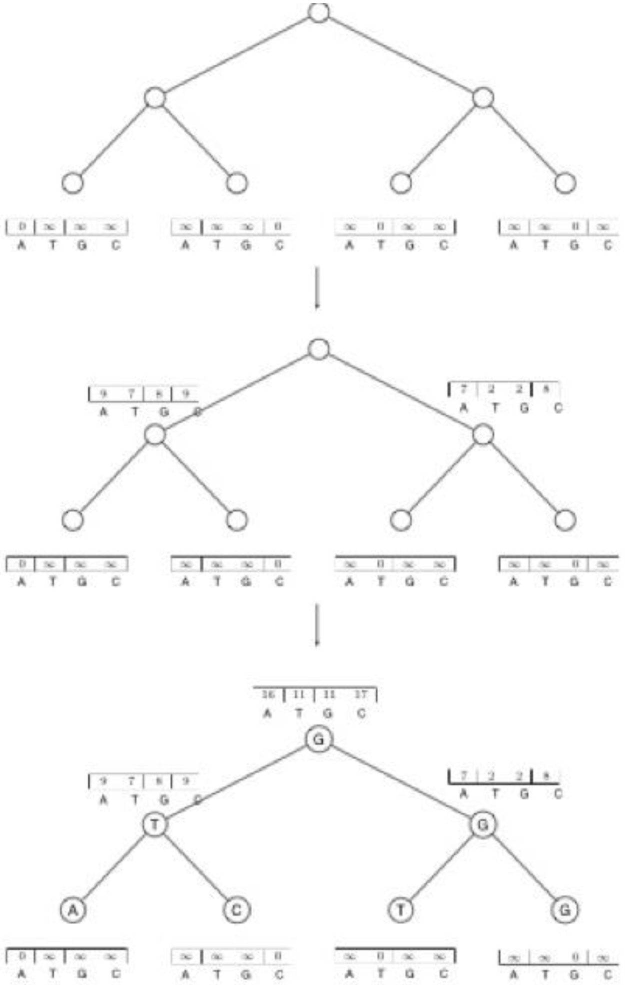
Weighted Small Parsimony Problem: Formulation

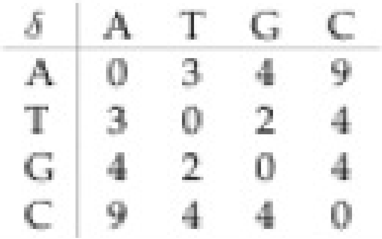
- Input: Tree *T* with each leaf labeled by elements of a *k*-letter alphabet and a *k* x *k* scoring matrix (δ_{ij})

- Output: Labeling of internal vertices of the tree *T* minimizing the weighted parsimony score

Sankoff’s Algorithm

- Check children’s every vertex and determine the minimum between them
- An example





δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

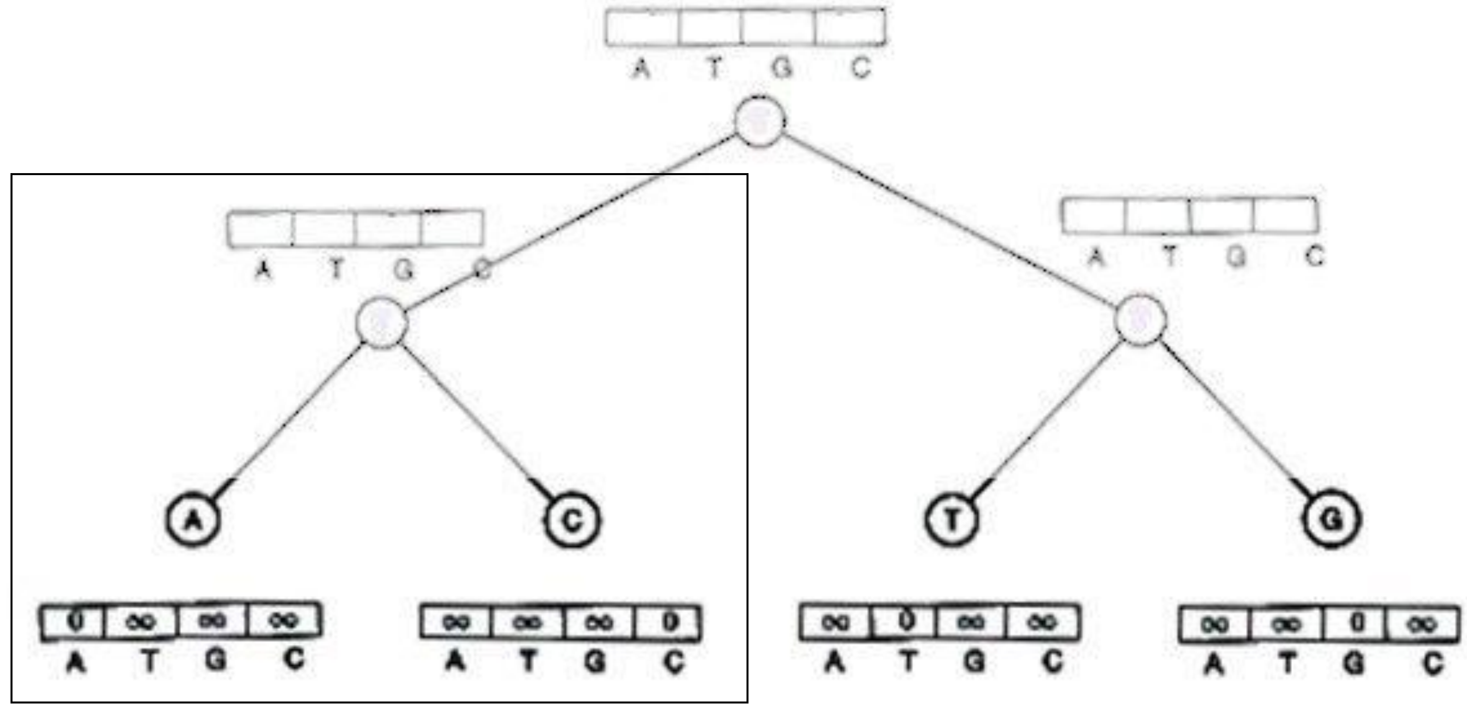
Sankoff Algorithm: Dynamic Programming

- Calculate and keep track of a score for every possible label at each vertex
  - $s_t(v)$ = minimum parsimony score of the **subtree** rooted at vertex *v* if *v* has character *t*
- The score at each vertex is based on scores of its children:
  - $s_t(\textbf{\textit{parent}}) = \min_i \{s_i(\textbf{\textit{ left child }}) + \delta_{i,\,t}\} +$

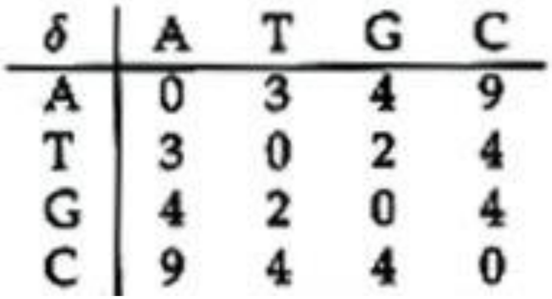
    $\min_j \{s_j(\textbf{\textit{ right child }}) + \delta_{j,\,t}\}$

Sankoff Algorithm (cont.)

- Begin at leaves:
  - If leaf has the character in question, score is 0
  - Else, score is ∞

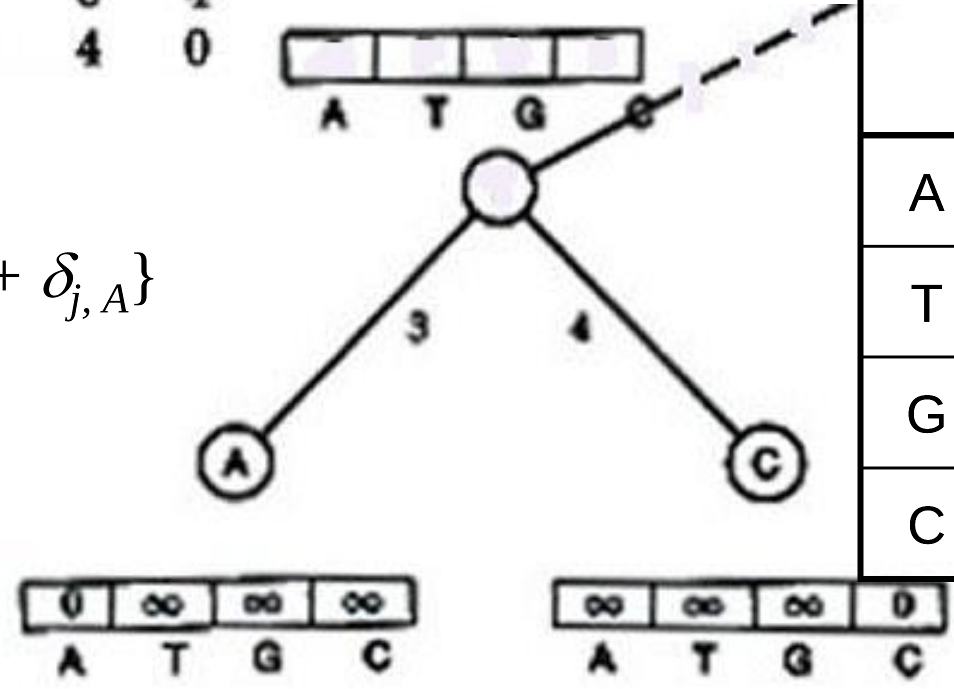


Sankoff Algorithm (cont.)



δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

$$s_t(v) = \min_i \{s_i(u) + \delta_{i,\,t}\} + \min_j\{s_j(w) + \delta_{j,\,t}\}$$

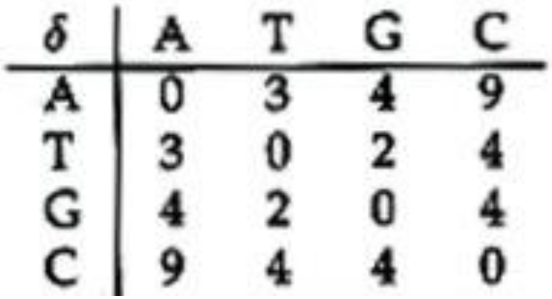


$$s_A(v) = 0 + \min_j\{s_j(w) + \delta_{j,\,A}\}$$

	$s_i(u)$	$\delta_{i,\,A}$	sum
A	0	0	**0**
T	∞	3	∞
G	∞	4	∞
C	∞	9	∞

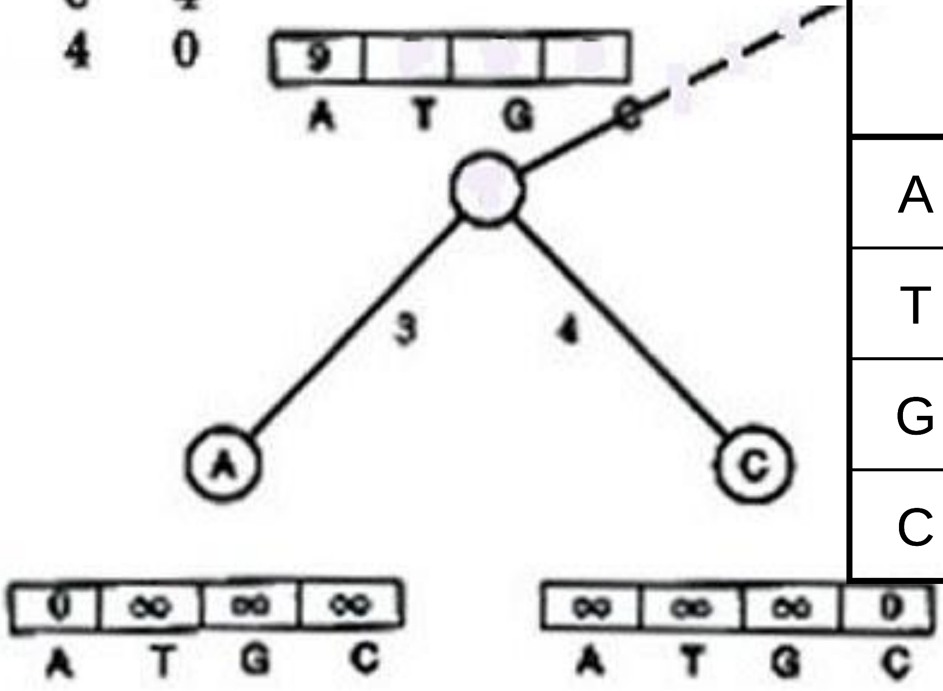
Sankoff Algorithm (cont.)

$$s_t(v) = \min_i \{s_i(u) + \delta_{i,\,t}\} + \min_j\{s_j(w) + \delta_{j,\,t}\}$$



δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

$s_A(v) = 0 + 9 = \mathbf{9}$



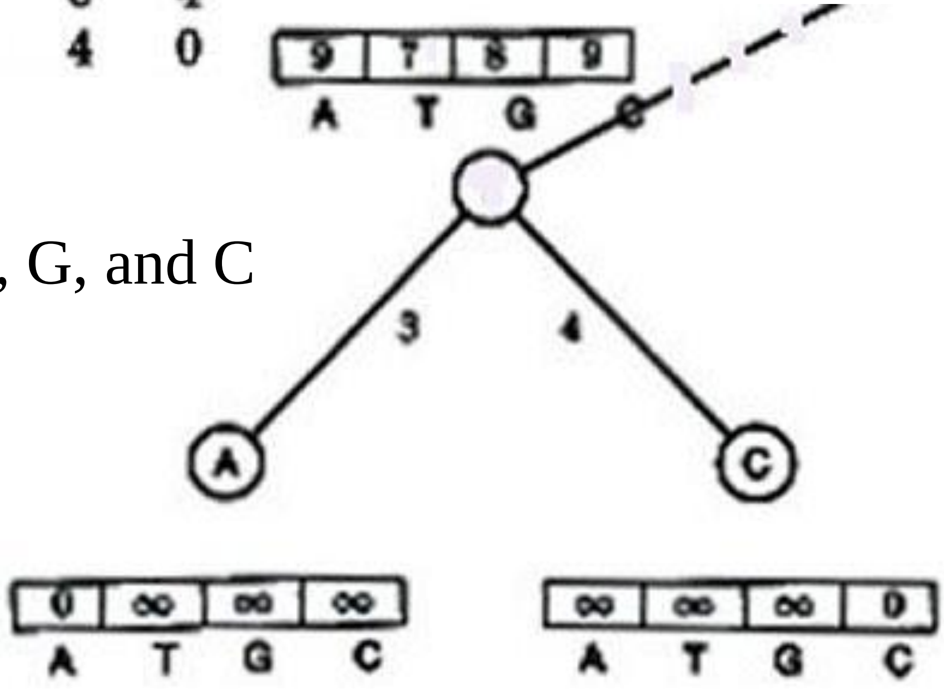
	$s_j(u)$	$\delta_{j,\,A}$	sum
A	∞	0	∞
T	∞	3	∞
G	∞	4	∞
C	0	9	**9**

Sankoff Algorithm (cont.)

δ	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

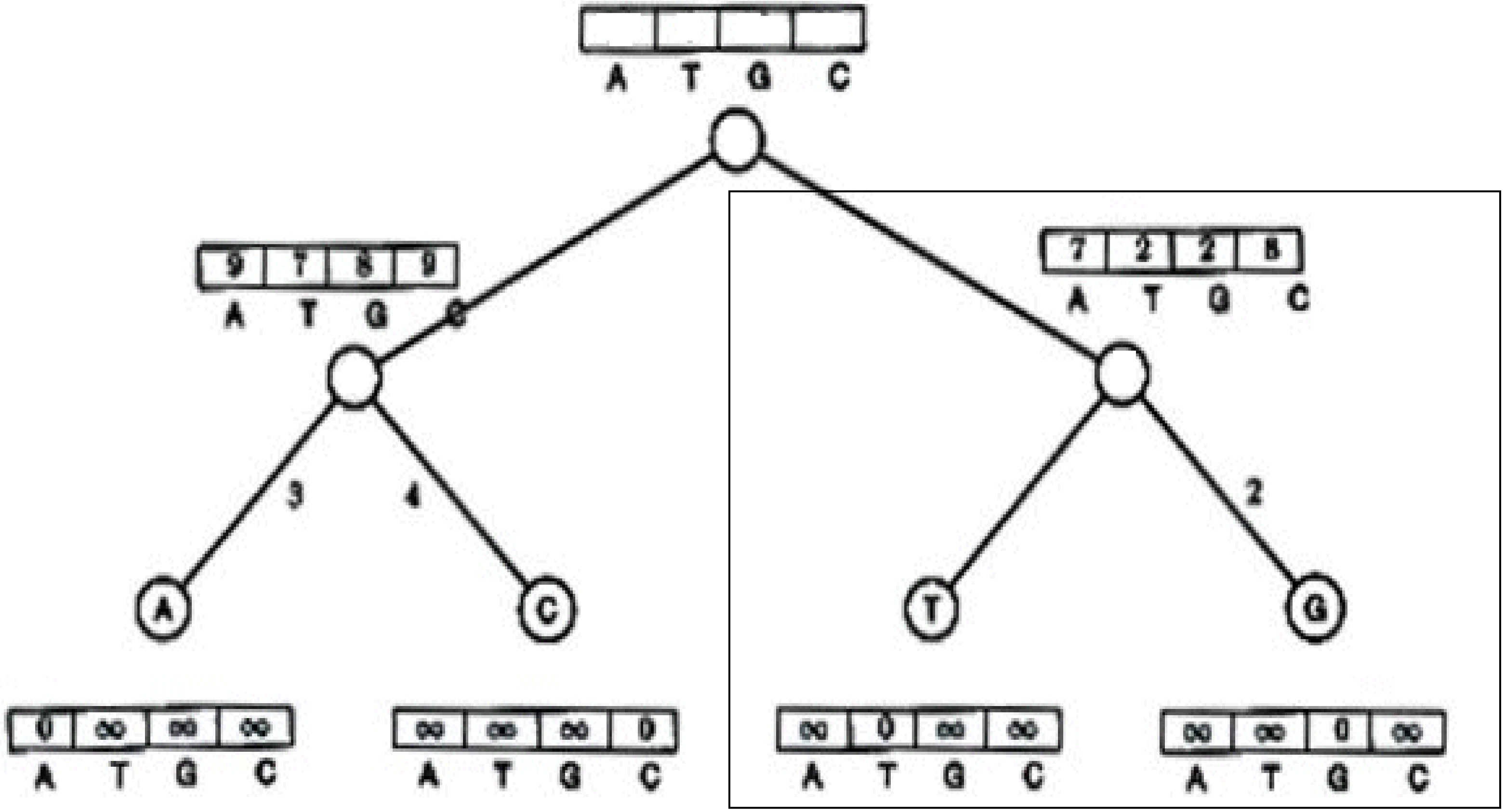
$$s_t(v) = \min_i \{s_i(u) + \delta_{i,\,t}\} + \min_j\{s_j(w) + \delta_{j,\,t}\}$$

Repeat for T, G, and C



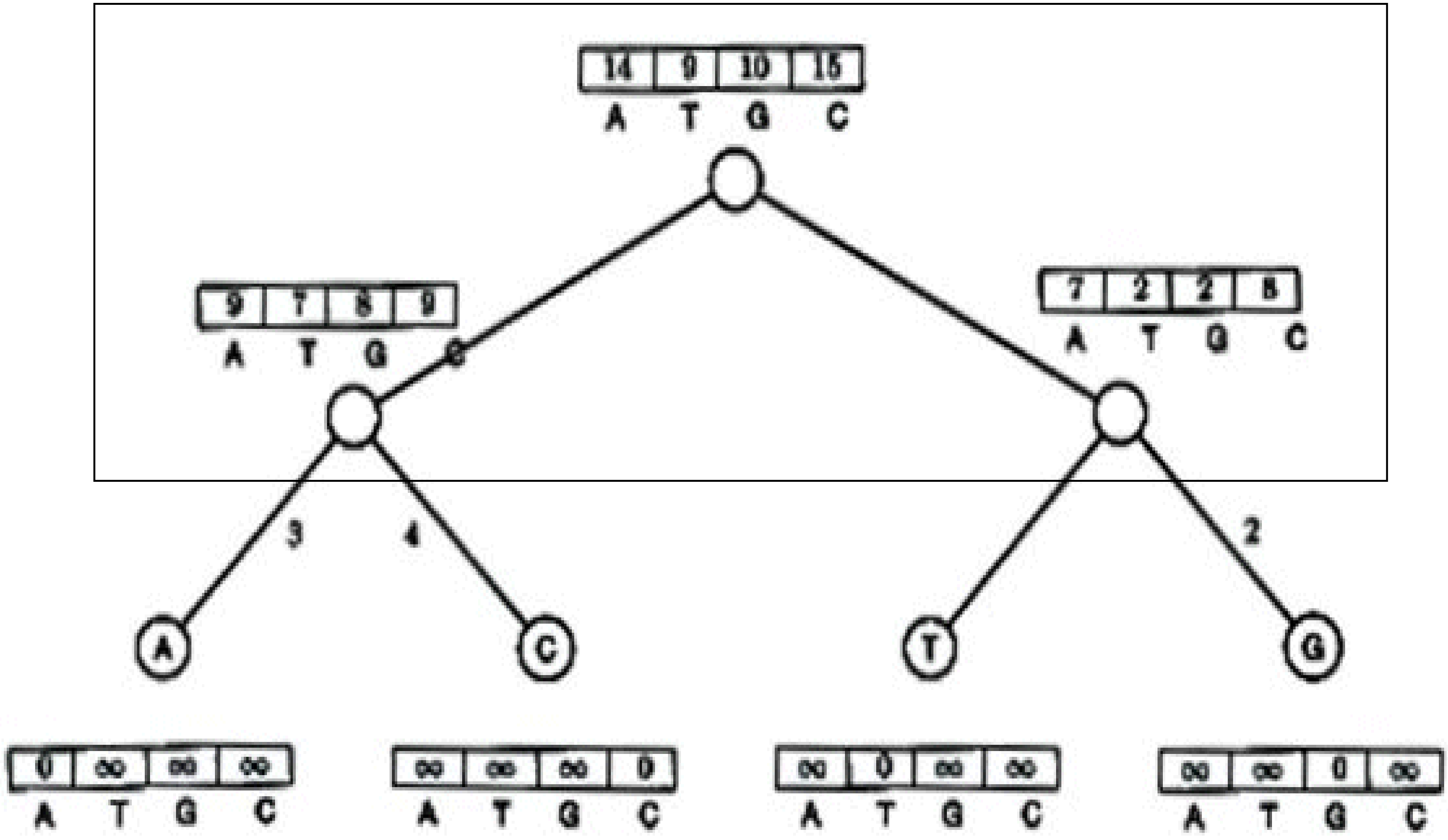
Sankoff Algorithm (cont.)

Repeat for right subtree



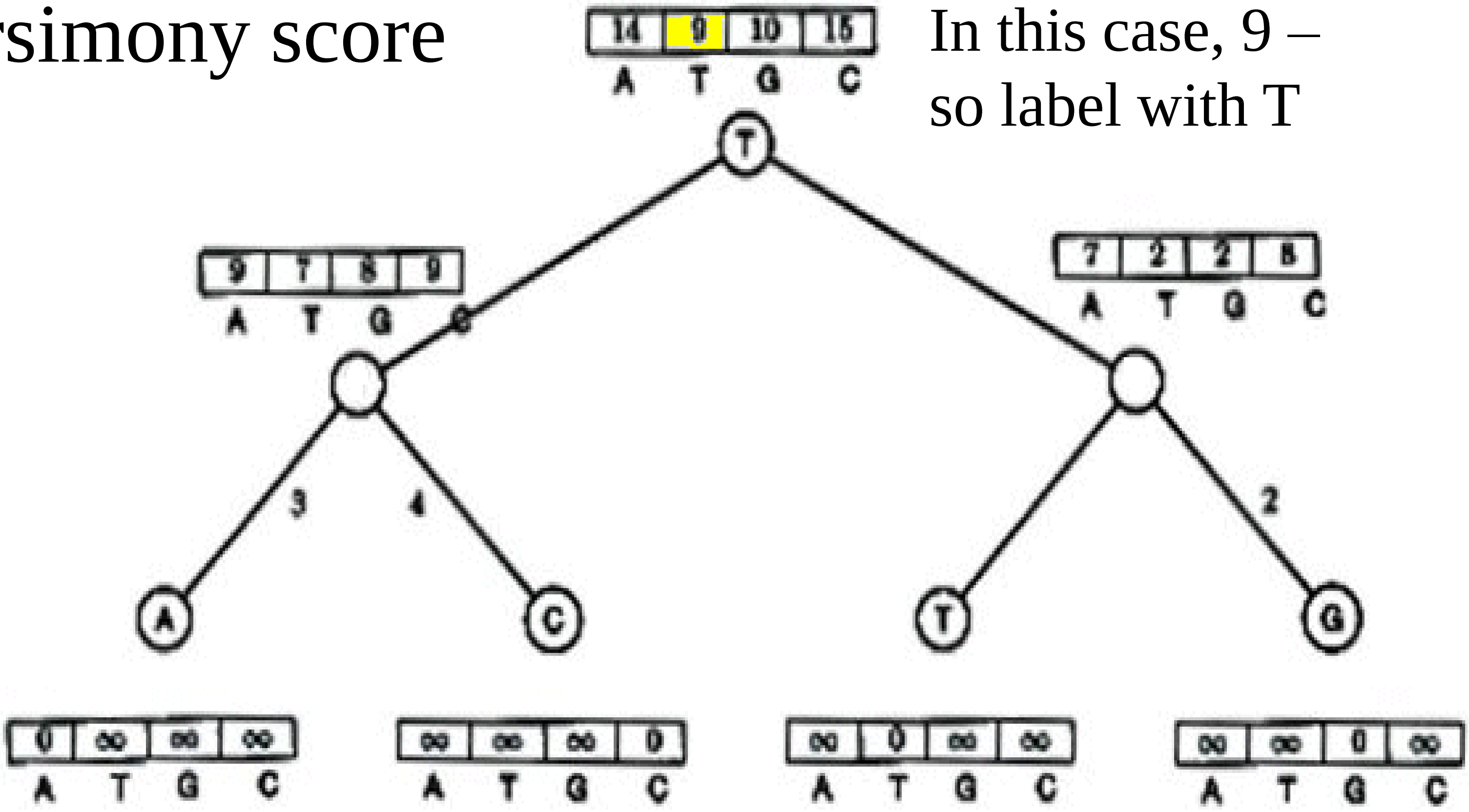
Sankoff Algorithm (cont.)

Repeat for root



Sankoff Algorithm (cont.)

Smallest score at root is minimum weighted parsimony score



In this case, 9 – so label with T

Sankoff Algorithm: Traveling down the Tree

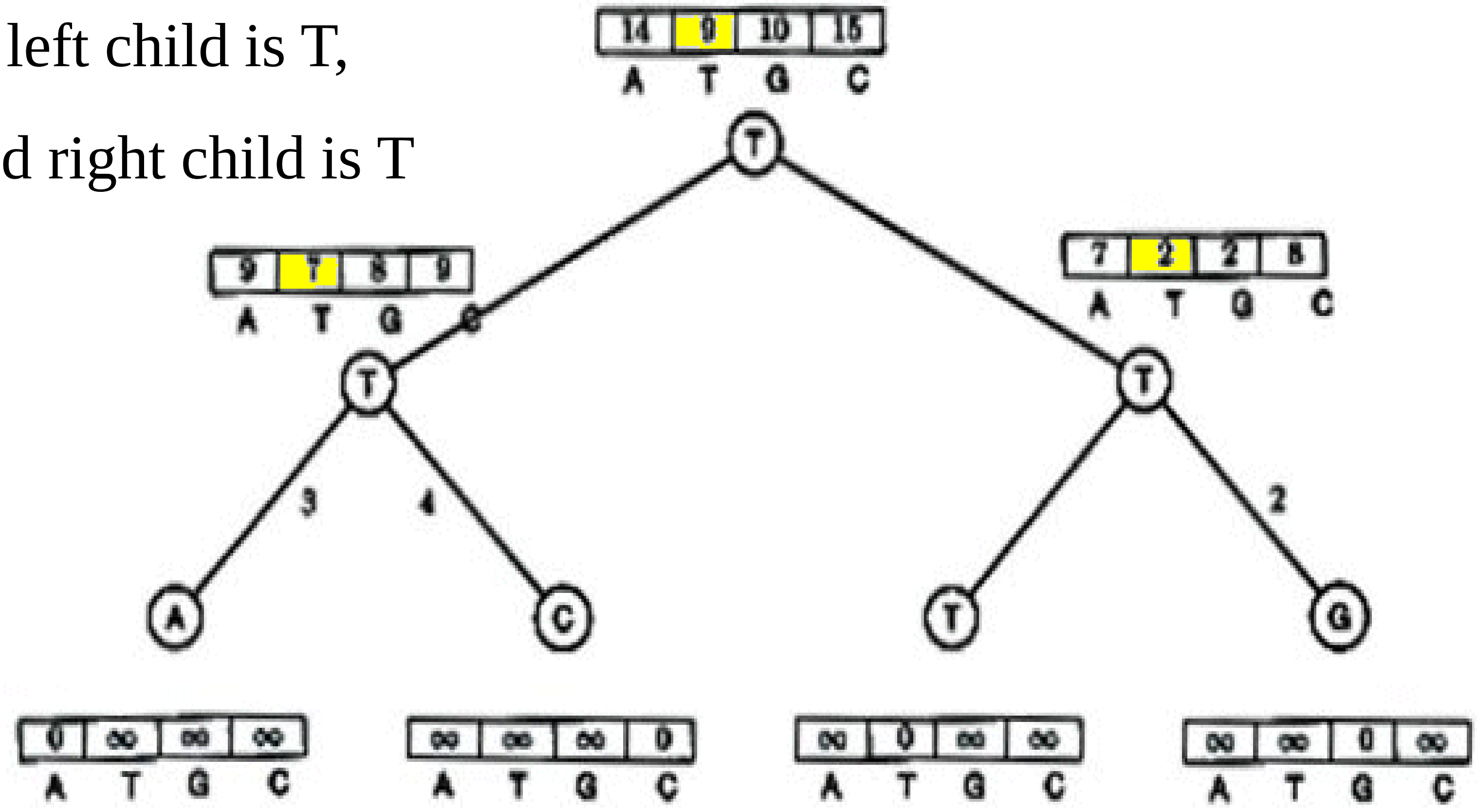
- The scores at the root vertex have been computed by going up the tree
- After the scores at root vertex are computed the Sankoff algorithm moves down the tree and assign each vertex with optimal character.

Sankoff Algorithm (cont.)

9 is derived from 7 + 2

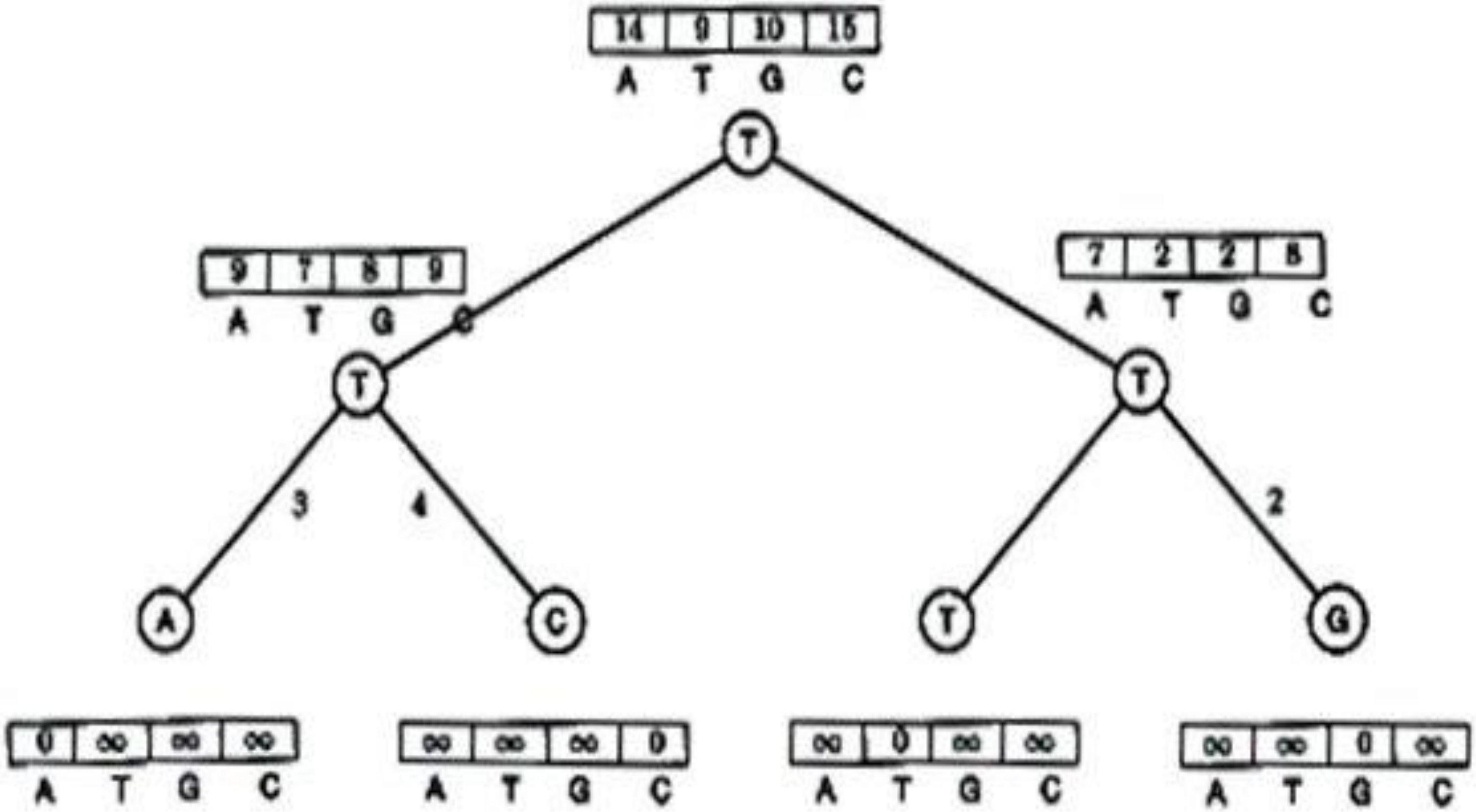
So left child is T,

And right child is T



Sankoff Algorithm (cont.)

And the tree is thus labeled…

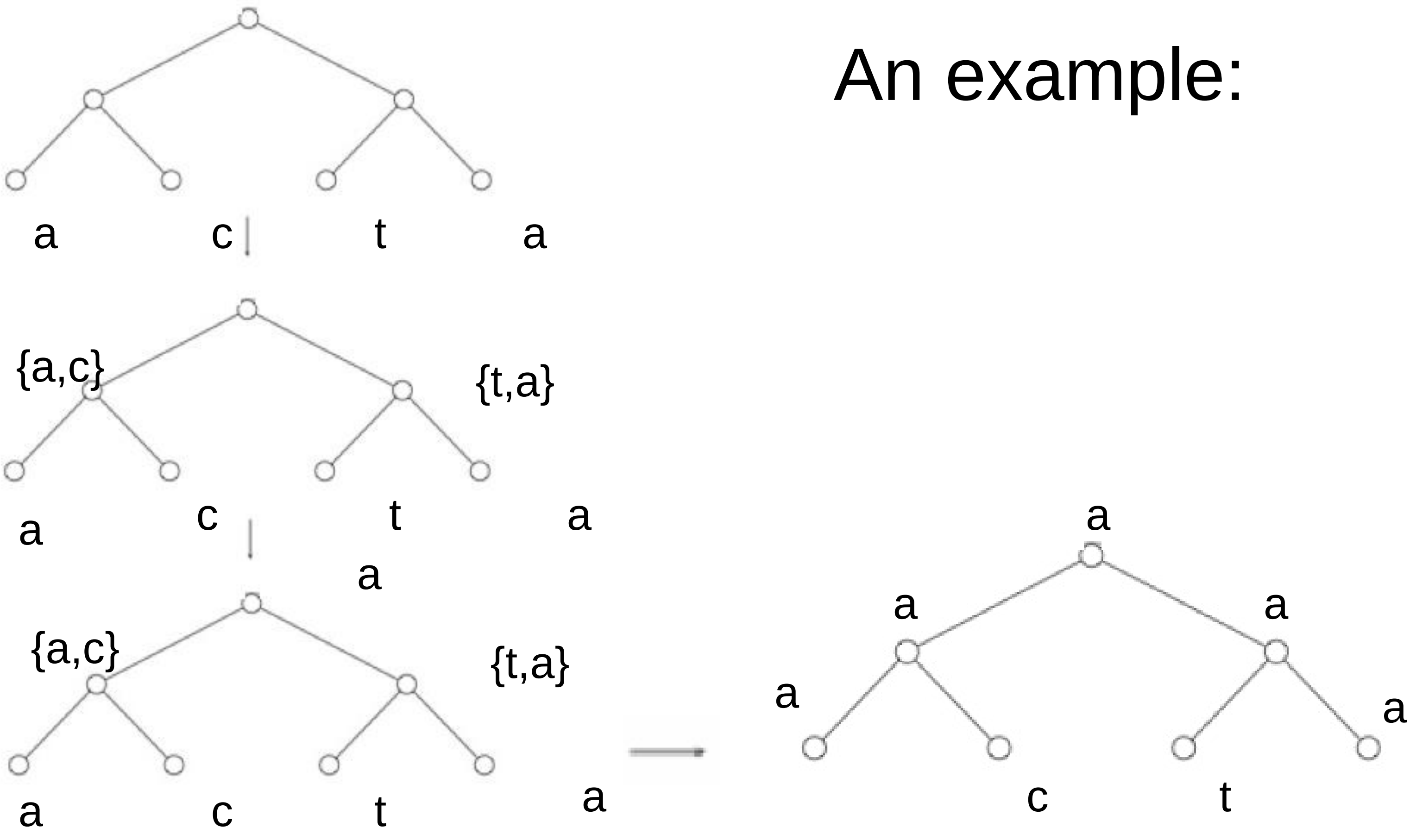


Fitch’s Algorithm

- Solves Small Parsimony problem
- Dynamic programming in essence
- Assigns a set of letter to every vertex in the tree.
- If the two children’s sets of character overlap, it’s the common set of them
- If not, it’s the combined set of them.

Fitch's Algorithm (cont'd)

An example:



Fitch Algorithm

1) Assign a **set of possible letters** to every vertex, traversing the tree from leaves to root

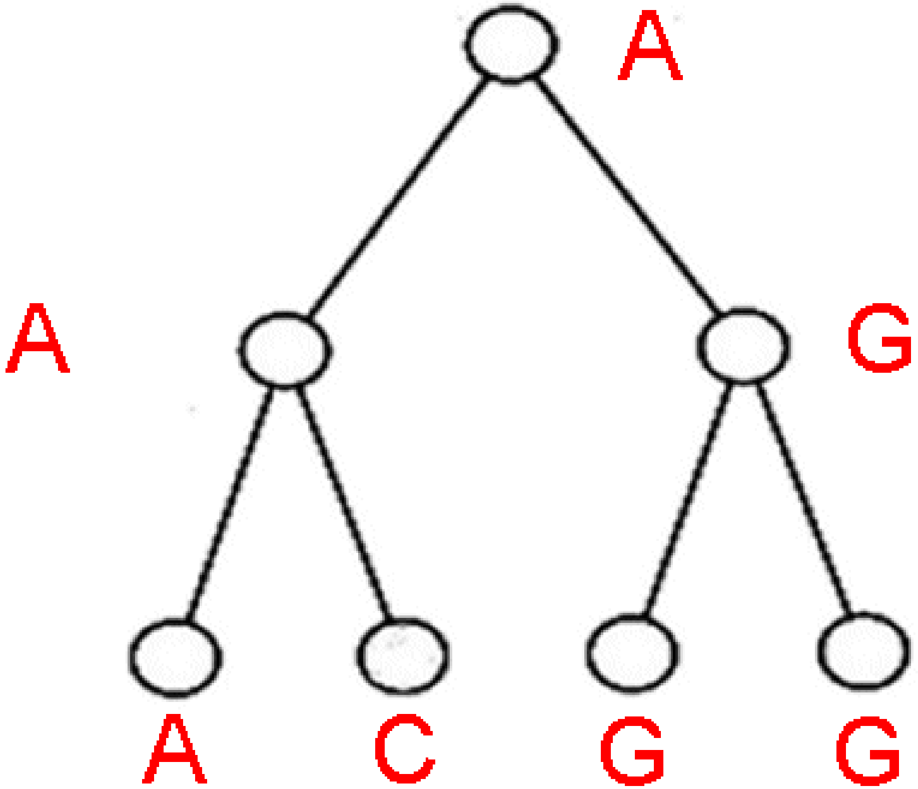
- Each node's set is the combination of its children's sets (leaves contain their label)
    - E.g. if the node we are looking at has a left child labeled {A, C} and a right child labeled {A, T}, the node will be given the set {A, C, T}

Fitch Algorithm (cont.)

2) Assign **labels** to each vertex, traversing the tree from root to leaves

- Assign root arbitrarily from its set of letters
- For all other vertices, if its parent's label is in its set of letters, assign it its parent's label
- Else, choose an arbitrary letter from its set as its label

Fitch Algorithm (cont.)

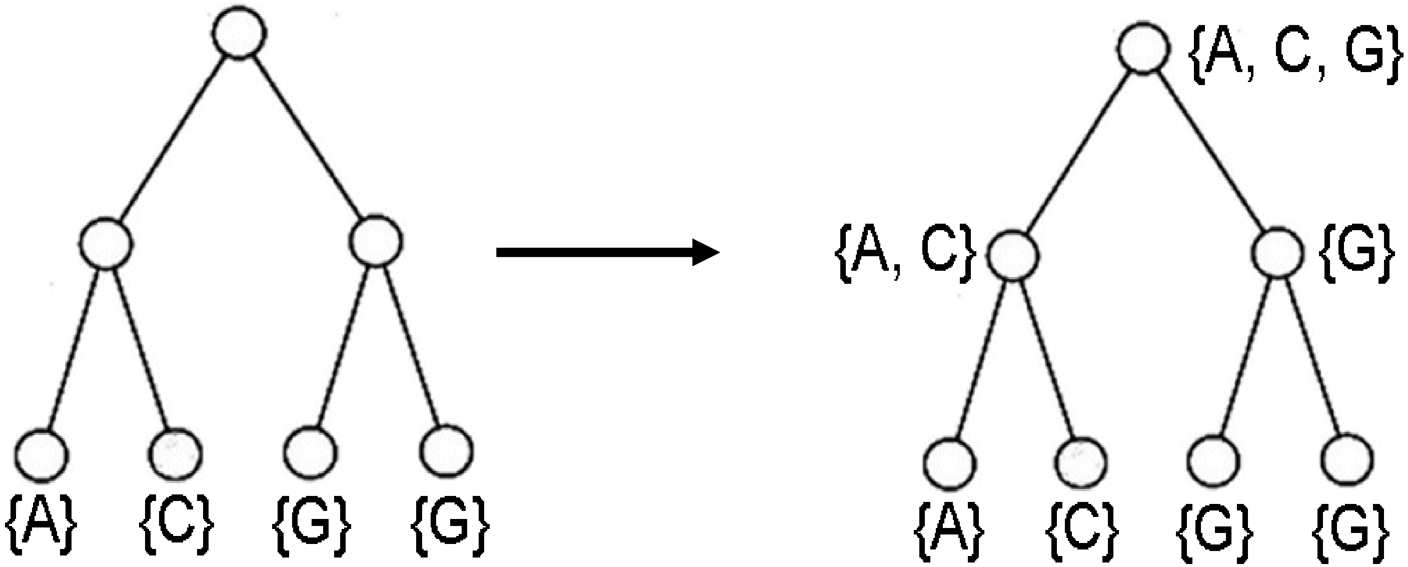


Fitch vs. Sankoff

- Both have an O(*nk*) runtime
- Are they actually different?
- Let’s compare …

Fitch

As seen previously:



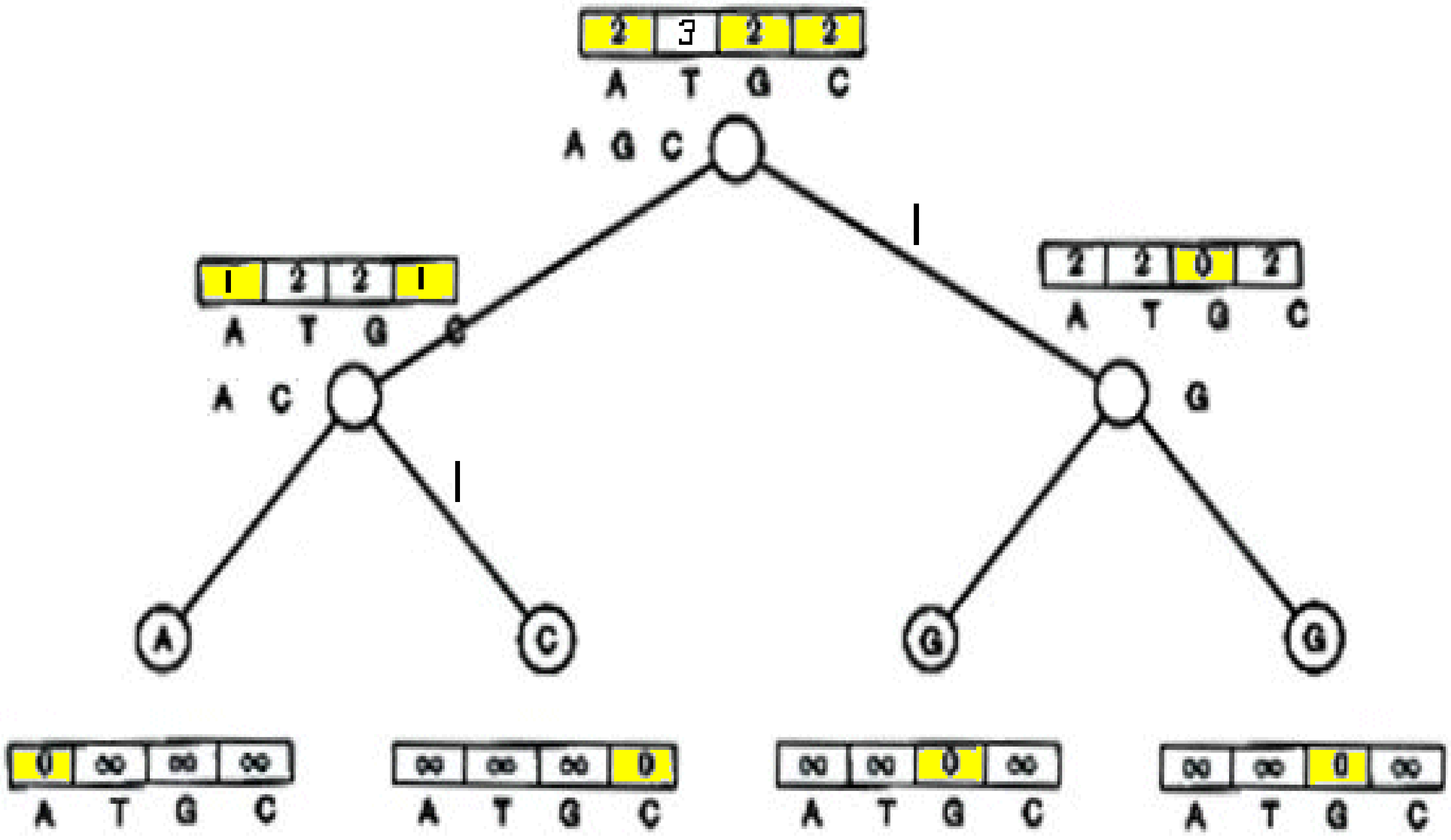
Comparison of Fitch and Sankoff

- As seen earlier, the scoring matrix for the Fitch algorithm is merely:

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

- So let’s do the same problem using Sankoff algorithm and this scoring matrix

Sankoff


2 3 2 2
A T G C
A G C
1 2 2 1
A T G C
A C
2 2 0 2
A T G C
G
1
1
A
C
G
G
0 ∞ ∞ ∞
A T G C
∞ ∞ ∞ 0
A T G C
∞ ∞ 0 ∞
A T G C
∞ ∞ 0 ∞
A T G C

Sankoff vs. Fitch

- The Sankoff algorithm gives the **same** set of **optimal** labels as the Fitch algorithm
- For Sankoff algorithm, character *t* is *optimal* for vertex *v* if $s_t(v) = \min_{1 \leq i \leq k} s_i(v)$
    - Denote the set of optimal letters at vertex *v* as *S*(*v*)
        - If *S*(*left child*) and *S*(*right child*) overlap, *S*(*parent*) is the intersection
        - Else it's the union of *S*(*left child*) and *S*(*right child*)
        - This is also the Fitch recurrence
- The two algorithms are **identical**

Large Parsimony Problem

- Input: An n x m matrix M describing n species, each represented by an m-character string
- Output: A tree T with n leaves labeled by the n rows of matrix M, and a labeling of the internal vertices such that the parsimony score is minimized over all possible trees and all possible labelings of internal vertices

Large Parsimony Problem (cont.)

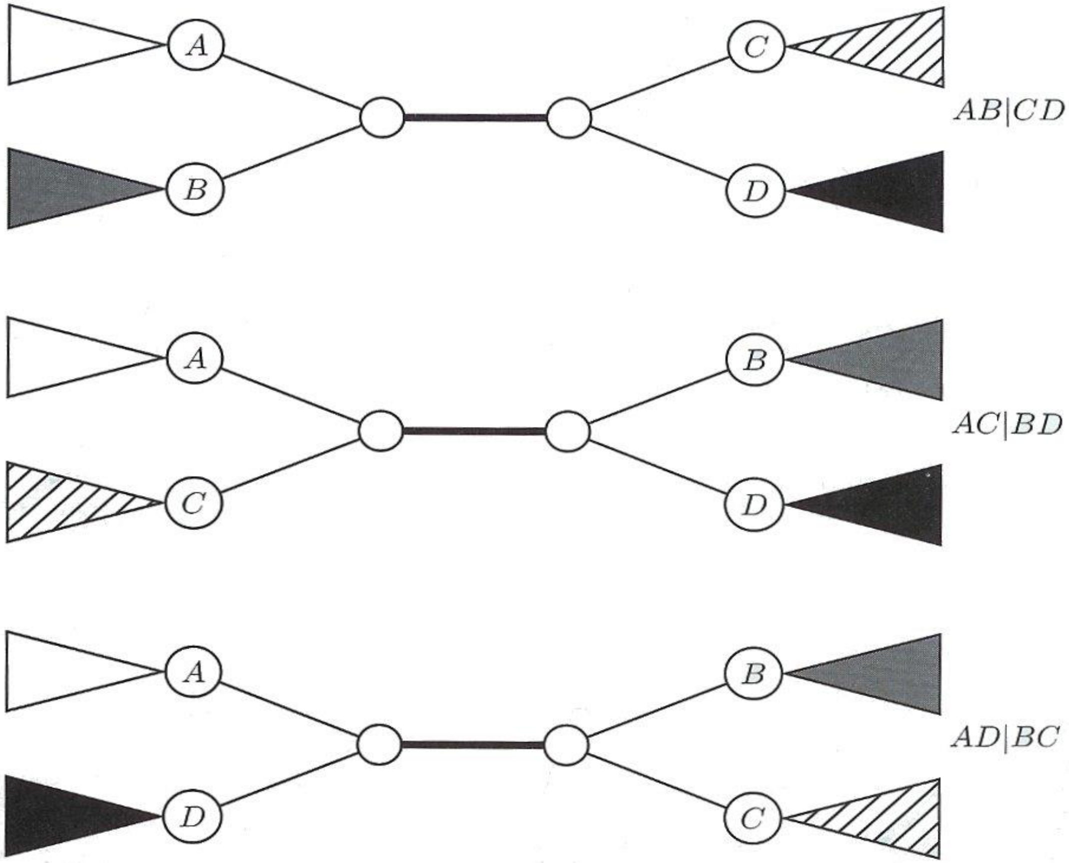
- Possible search space is huge, especially as *n* increases
  - $(2n - 3)!!$ possible rooted trees
  - $(2n - 5)!!$ possible unrooted trees
- Problem is NP-complete
  - Exhaustive search only possible w/ small *n*(< 10)
- Hence, branch and bound or heuristics used

Nearest Neighbor Interchange

A Greedy Algorithm

- A Branch Swapping algorithm
- Only evaluates a subset of all possible trees
- Defines a *neighbor* of a tree as one reachable by a *nearest neighbor interchange*
    - A rearrangement of the four subtrees defined by one internal edge
    - Only three different rearrangements per edge

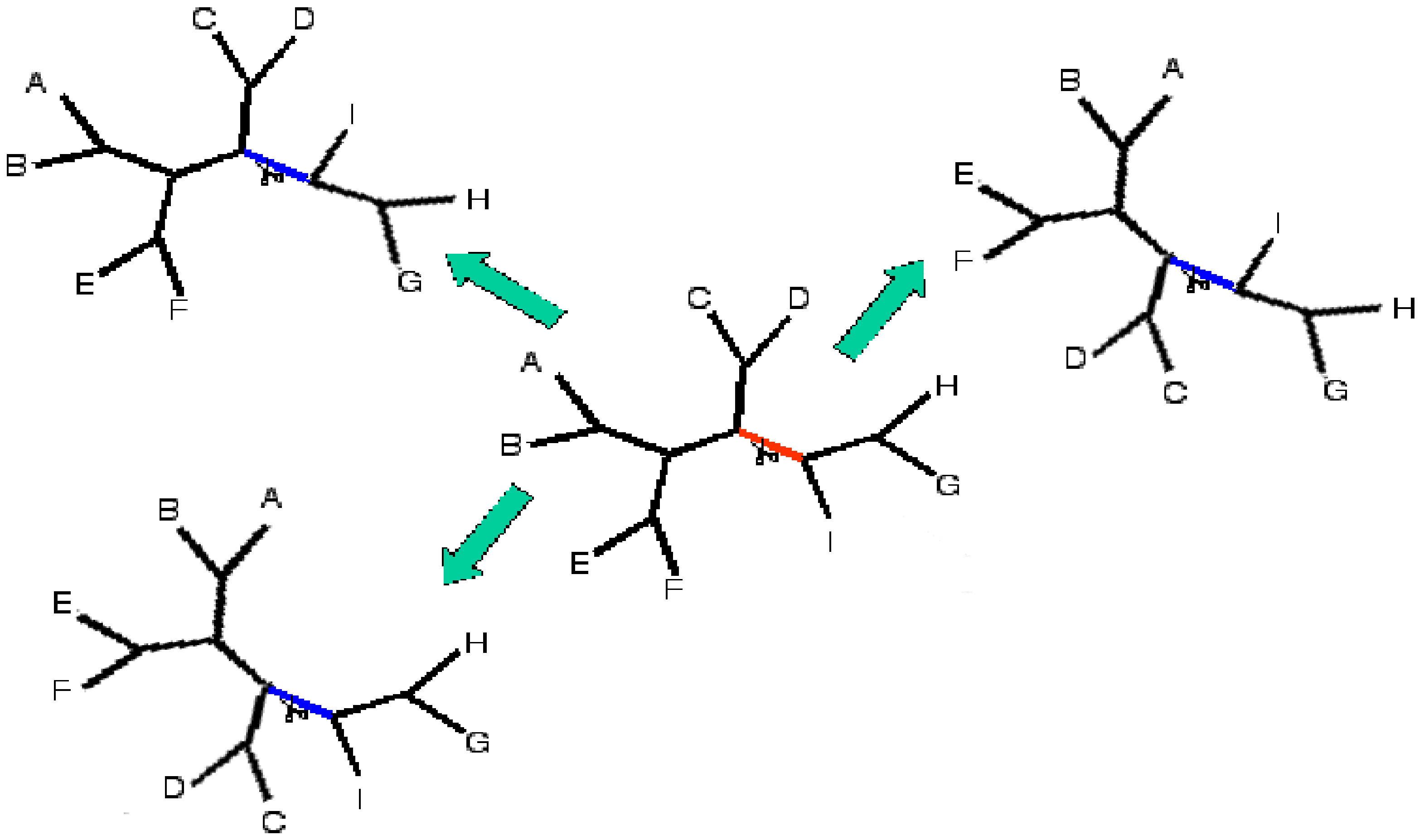
Nearest Neighbor Interchange (cont.)



Nearest Neighbor Interchange (cont.)

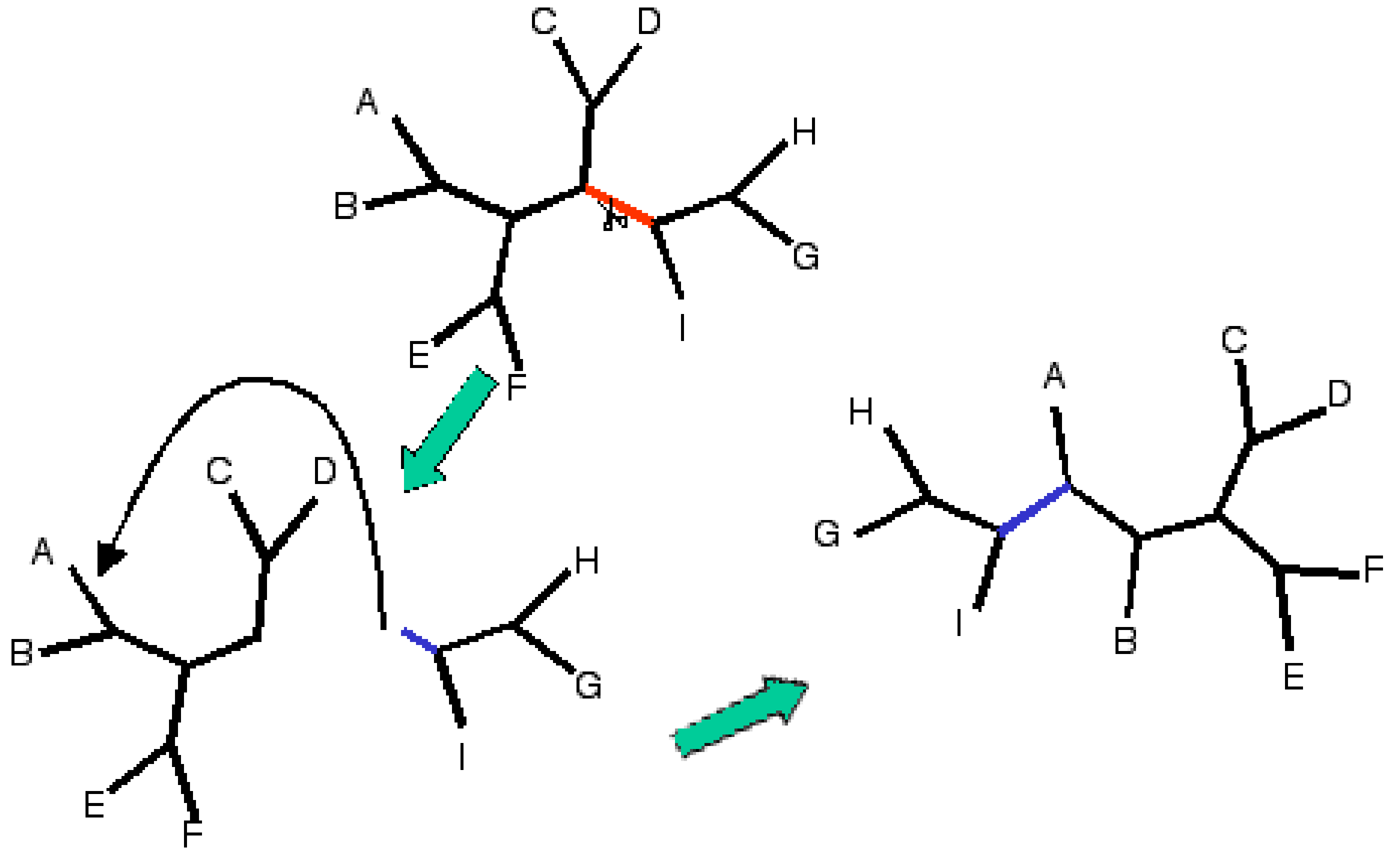
- Start with an arbitrary tree and check its neighbors
- Move to a neighbor if it provides the best improvement in parsimony score
- No way of knowing if the result is the **most** parsimonious tree
- Could be stuck in local optimum

Nearest Neighbor Interchange



Subtree Pruning and Regrafting

Another Branch Swapping Algorithm

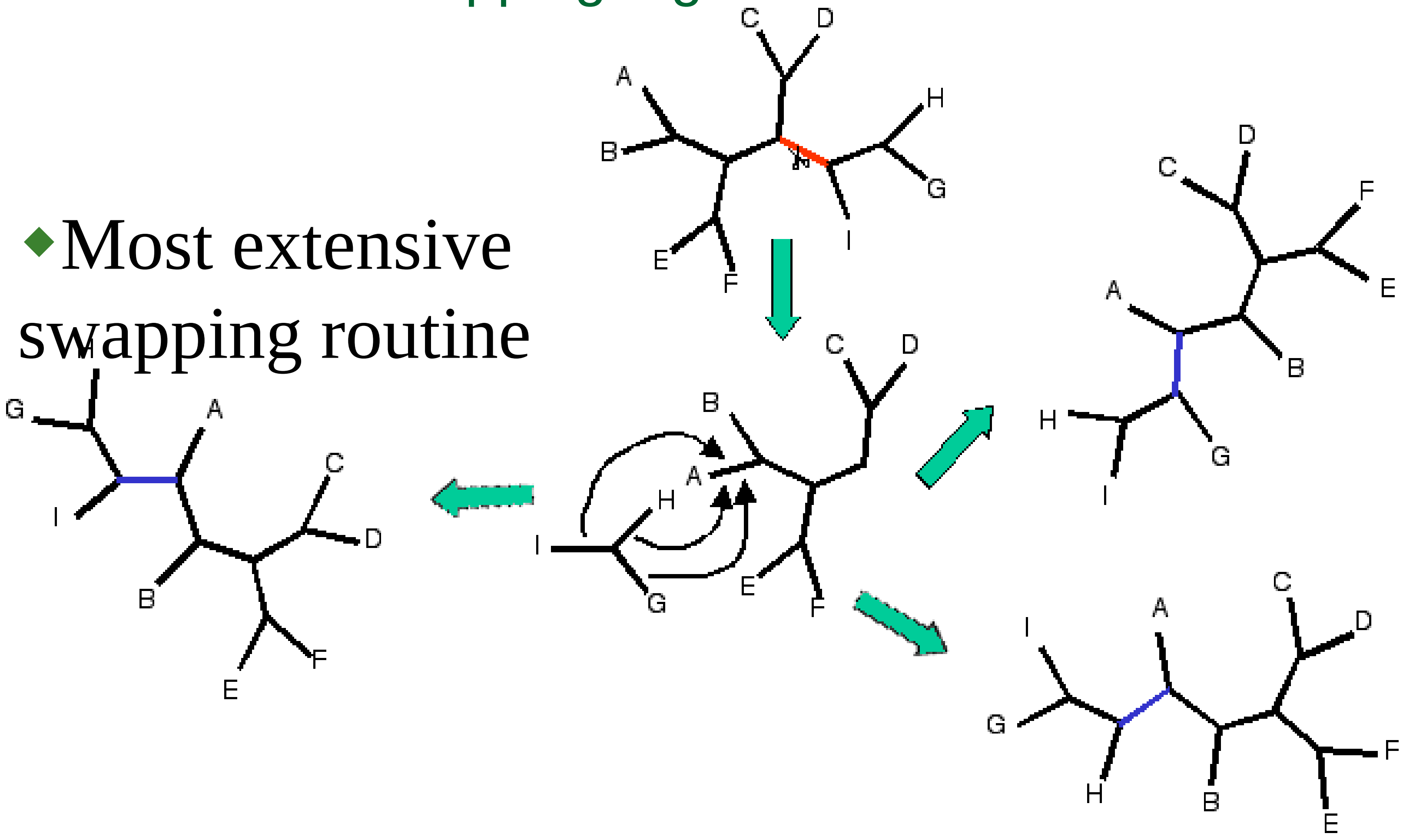


http://artedi.ebc.uu.se/course/BioInfo-10p-2001/Phylogeny/Phylogeny-TreeSearch/SPR.gif

Tree Bisection and Reconnection

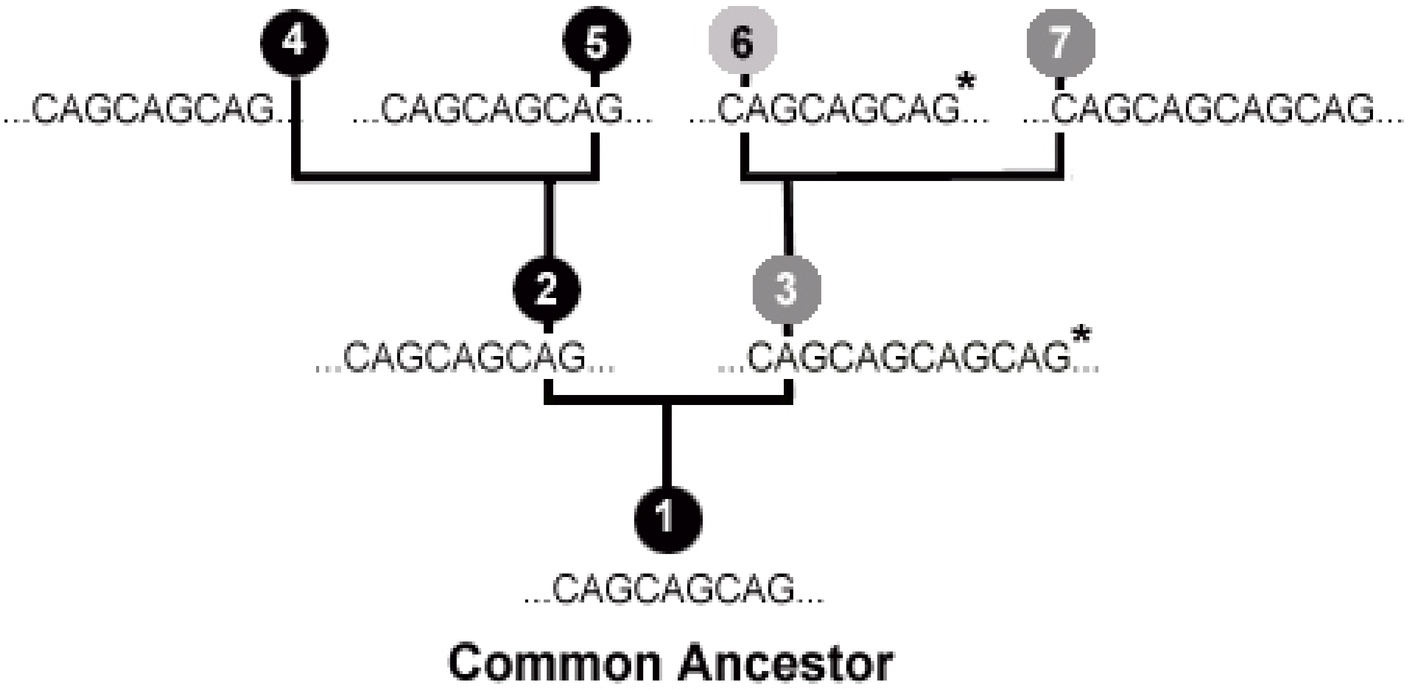
Another Branch Swapping Algorithm

- Most extensive swapping routine



Homoplasy

- But what if this was the real tree?



Sources

- http://www.math.tau.ac.il/~rshamir/ge/02/scribes/lec01.pdf
- http://bioinformatics.oupjournals.org/cgi/screenpdf/20/3/340.pdf
- http://www.absoluteastronomy.com/encyclopedia/M/Mi/Minimum_spanning_tree.htm
- Serafim Batzoglou (UPGMA slides) http://www.stanford.edu/class/cs262/Slides
- Watkins, W.S., Rogers A.R., Ostler C.T., Wooding, S., Bamshad M. J., Brassington A.E., Carroll M.L., Nguyen S.V., Walker J.A., Prasas, R., Reddy P.G., Das P.K., Batzer M.A., Jorde, L.B.: Genetic Variation Among World Populations: **Inferences From 100 *Alu* Insertion Polymorphisms**