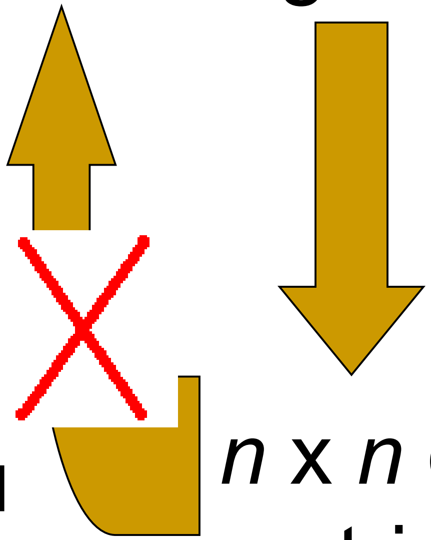
PHYLOGENETIC TREES

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 strings at internal nodes would best explain the character strings for the n observed species

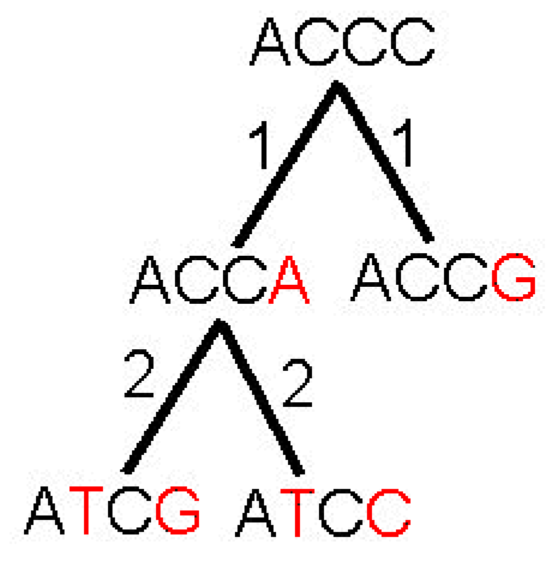
Character-Based Tree Reconstruction (cont'd)

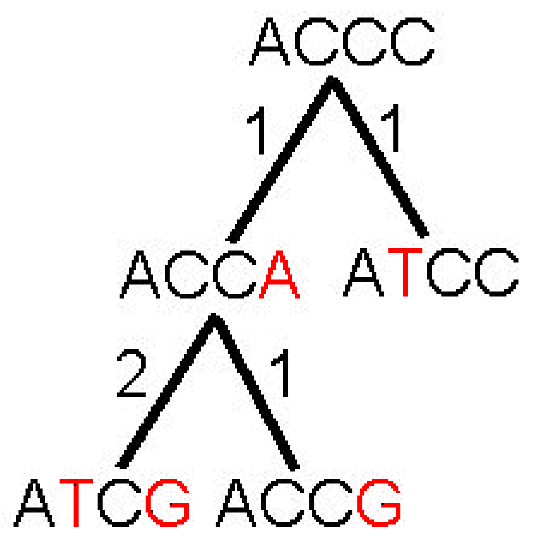
- Characters may be nucleotides, where A, G, C, T are **states** of this character.
- By setting the length of an edge in the tree to the Hamming distance, we may define the **parsimony score** of the tree as the sum of the lengths (weights) of the edges

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





Less Parsimonious

Score: 6

More Parsimonious

Score: 5

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.

- Because the characters in the string are independent, the Small Parsimony problem can be solved independently for each character. Therefore, to devise an algorithm, we can assume that every leaf is labeled by a single character rather than by a string of m characters.

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

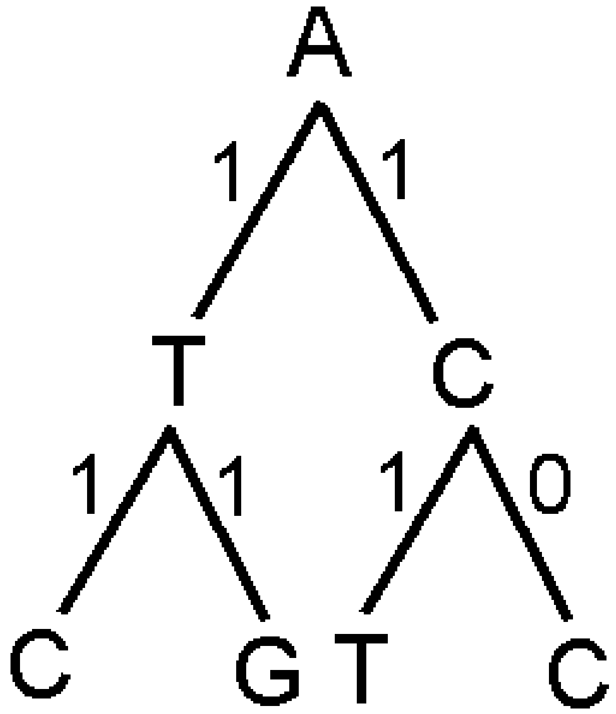
Small Parsimony Problem

	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

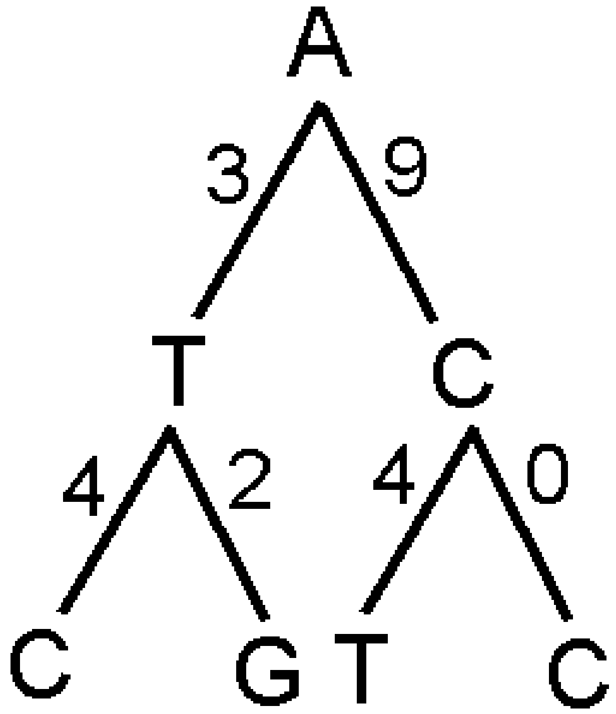


Small Parsimony Scoring Matrix:

	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})

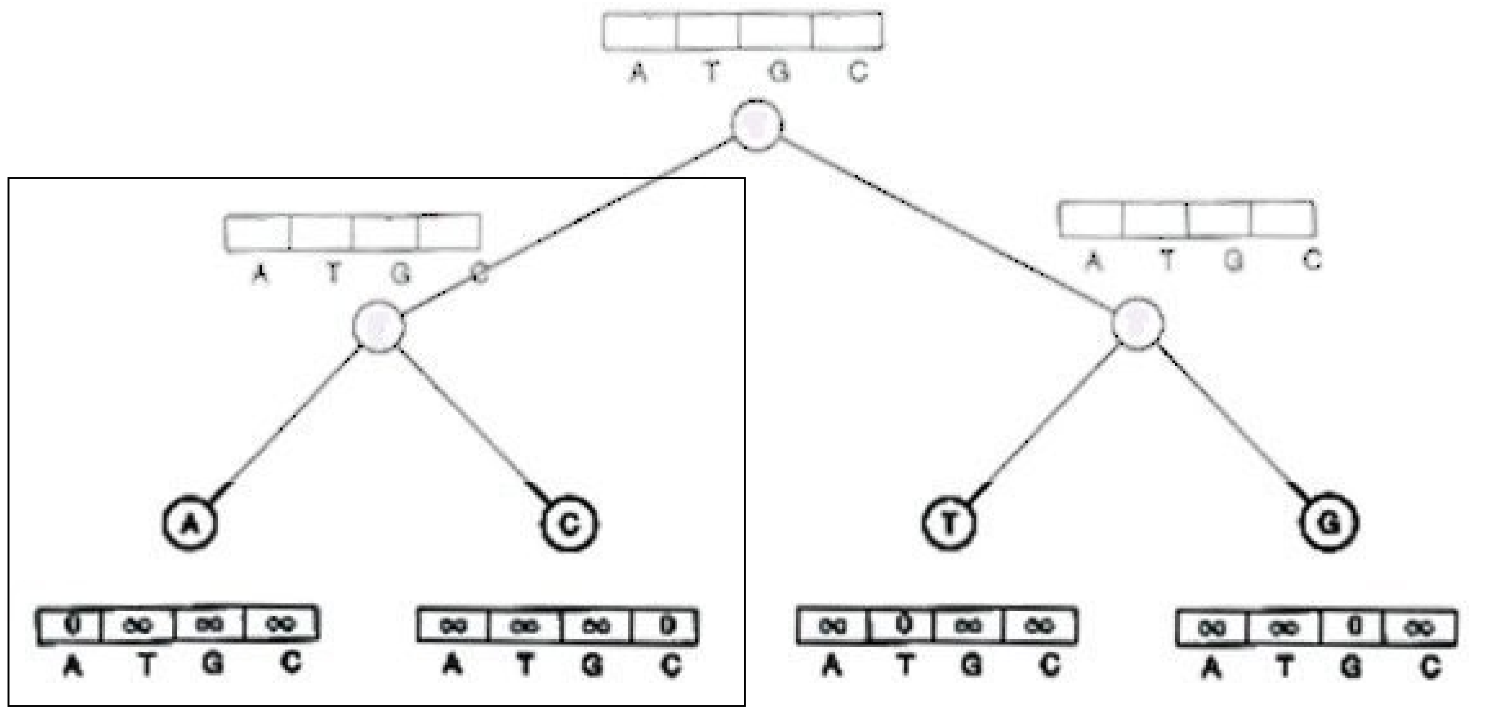
- Output: Labeling of internal vertices of the tree *T* minimizing the weighted parsimony score

Sankoff Algorithm: Dynamic Programming

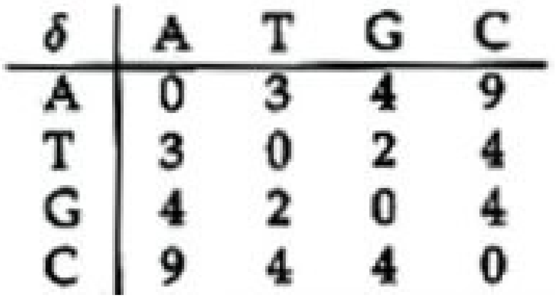
- Calculate and keep track of a score for every possible label at each vertex
  - $s_t(v)$ = minimum parsimony score of the **subtree** rooted at vertex *v* if *v* has character *t*
- The score at each vertex is based on scores of its children:
  - $s_t(\textbf{\textit{parent}}) = \min_i \{s_i(\textbf{\textit{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

Sankoff Algorithm (cont.)

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



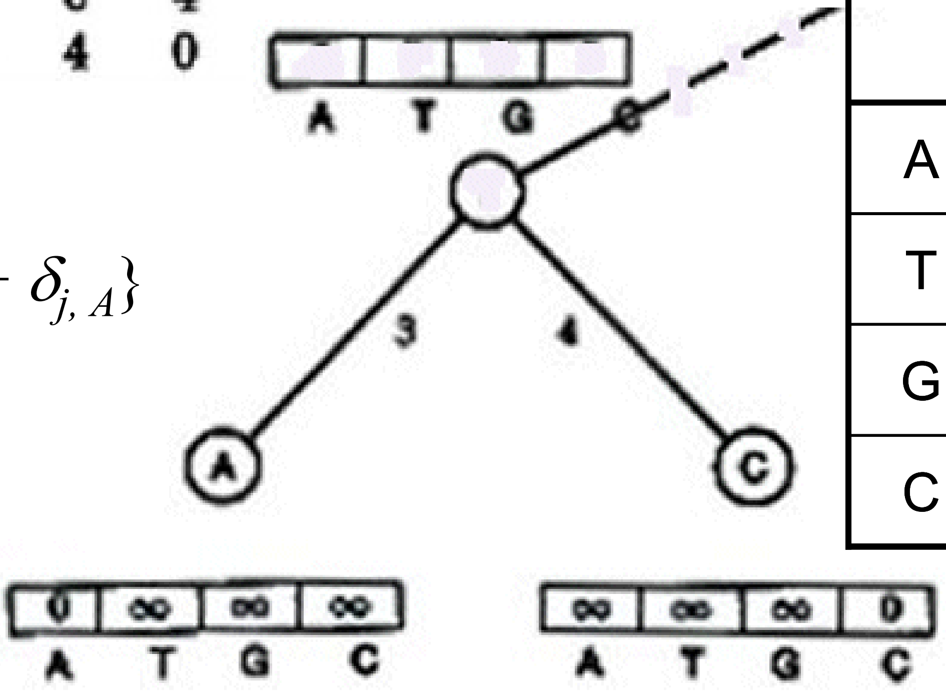
Sankoff Algorithm (cont.)



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

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

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



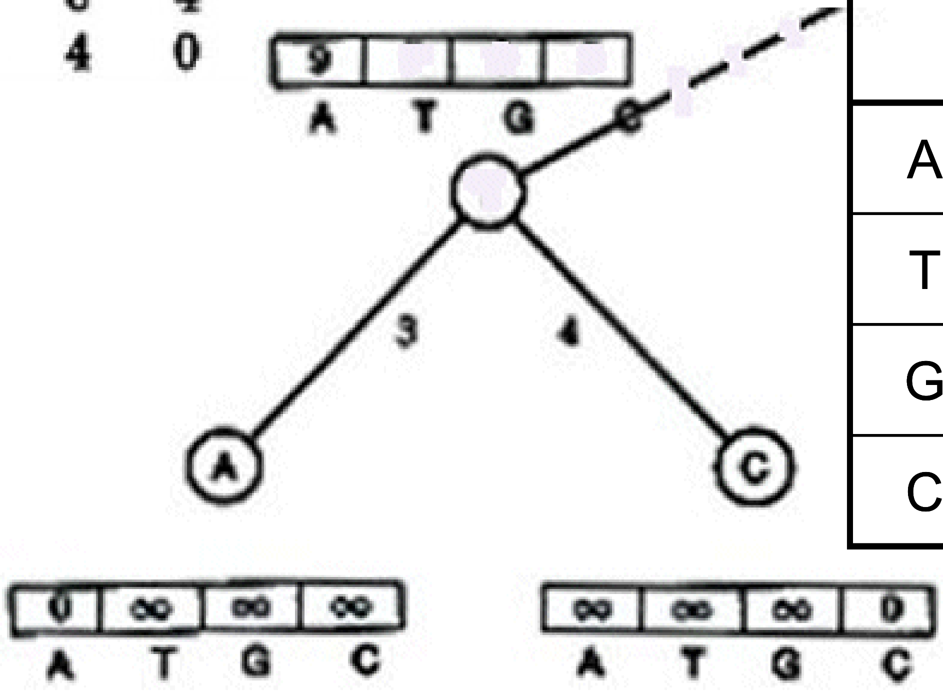
	$s_i(u)$	$\delta_{i,\, A}$	sum
A	0	0	**0**
T	∞	3	∞
G	∞	4	∞
C	∞	9	∞

Sankoff Algorithm (cont.)

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

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

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



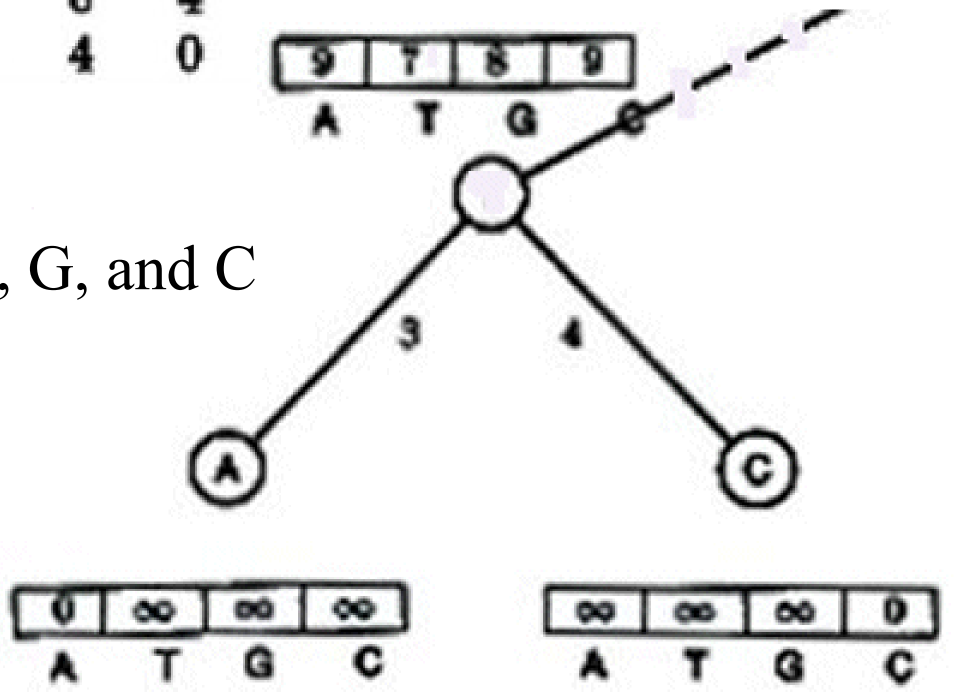
	$s_j(u)$	$\delta_{j,\,A}$	sum
A	∞	0	∞
T	∞	3	∞
G	∞	4	∞
C	0	9	**9**

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

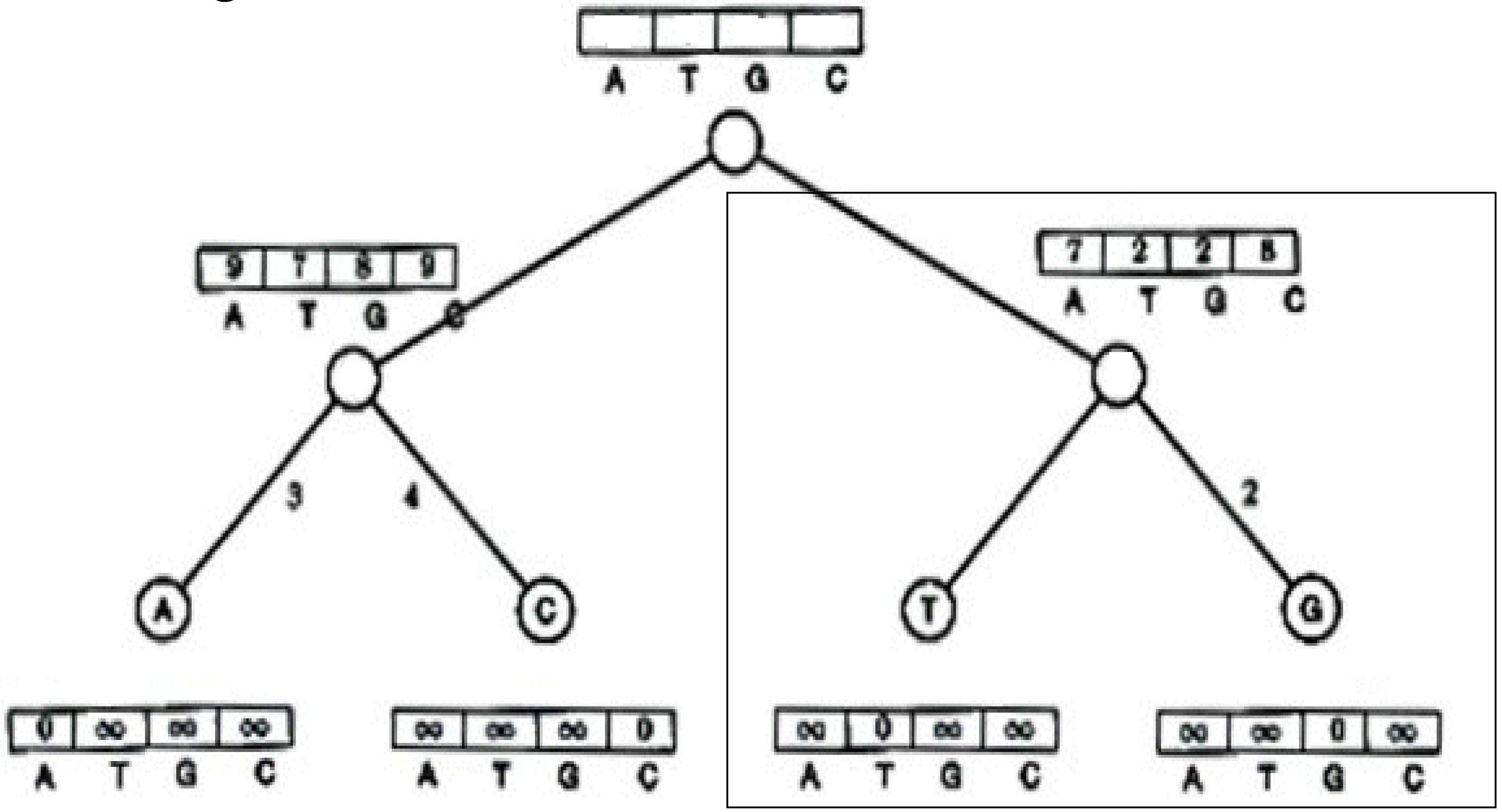
$$s_t(v) = \min_i \{s_i(u) + \delta_{i,\,t}\} + \min_j\{s_j(w) + \delta_{j,\,t}\}$$

Repeat for T, G, and C



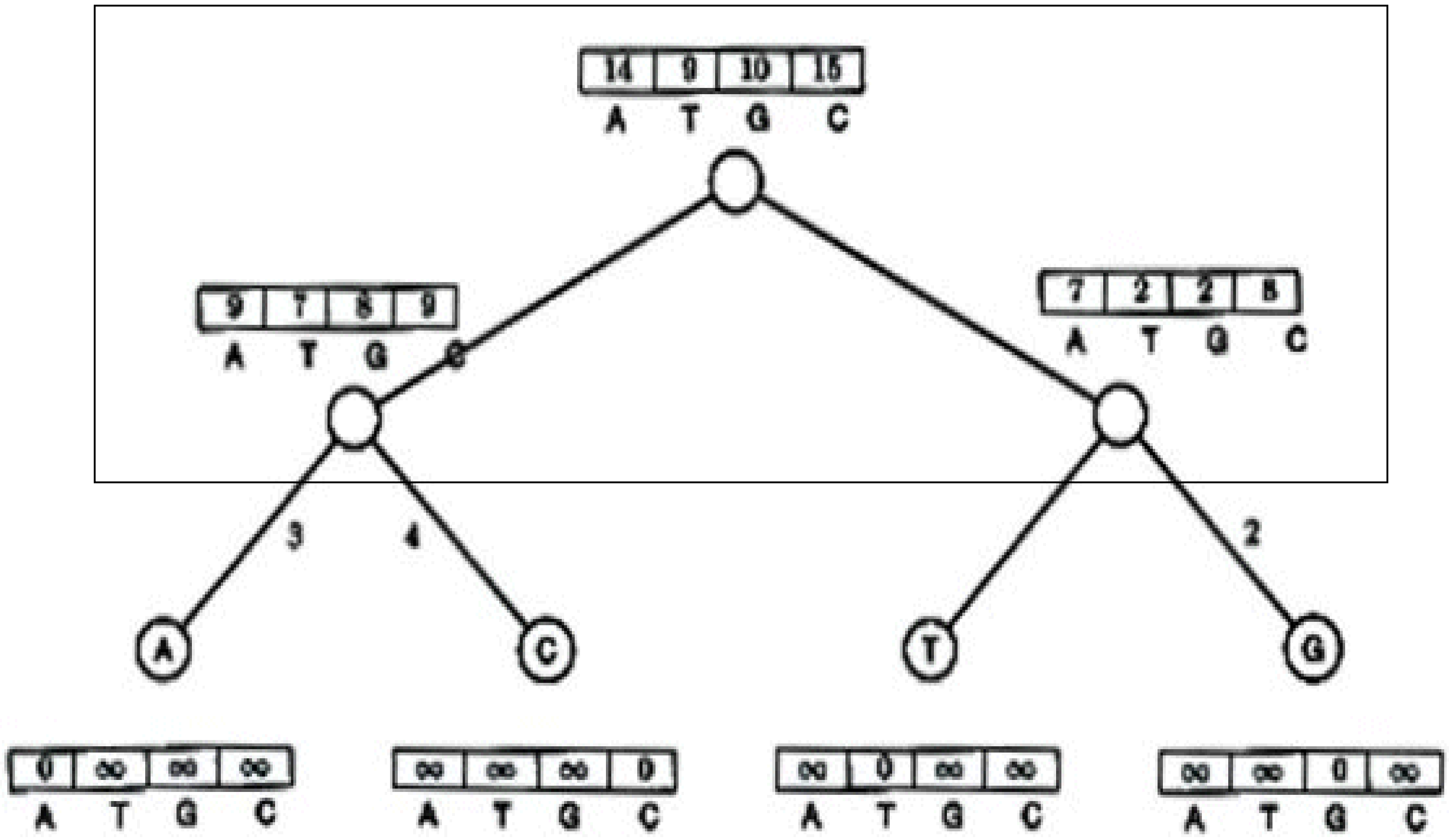
Sankoff Algorithm (cont.)

Repeat for right subtree



Sankoff Algorithm (cont.)

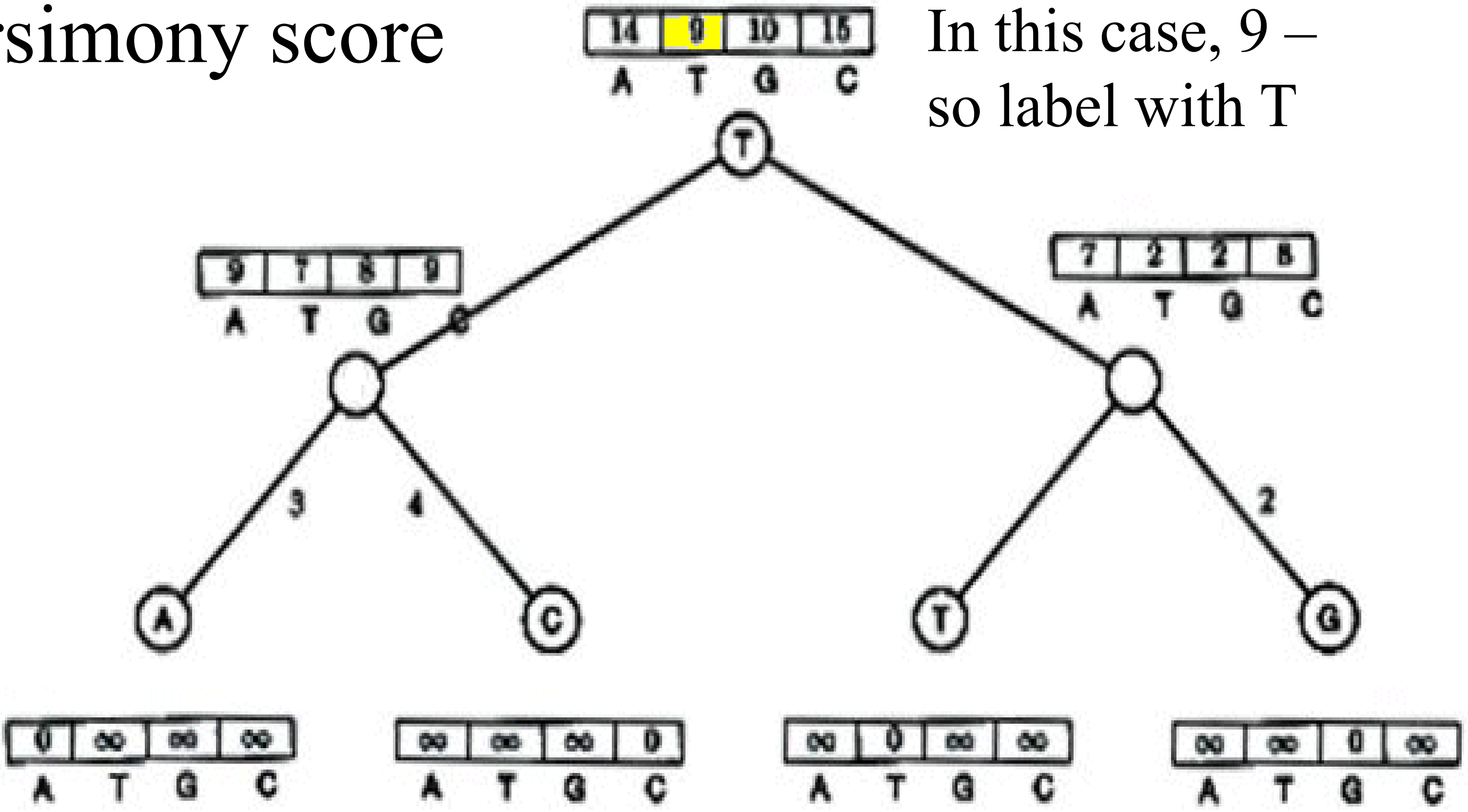
Repeat for root



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

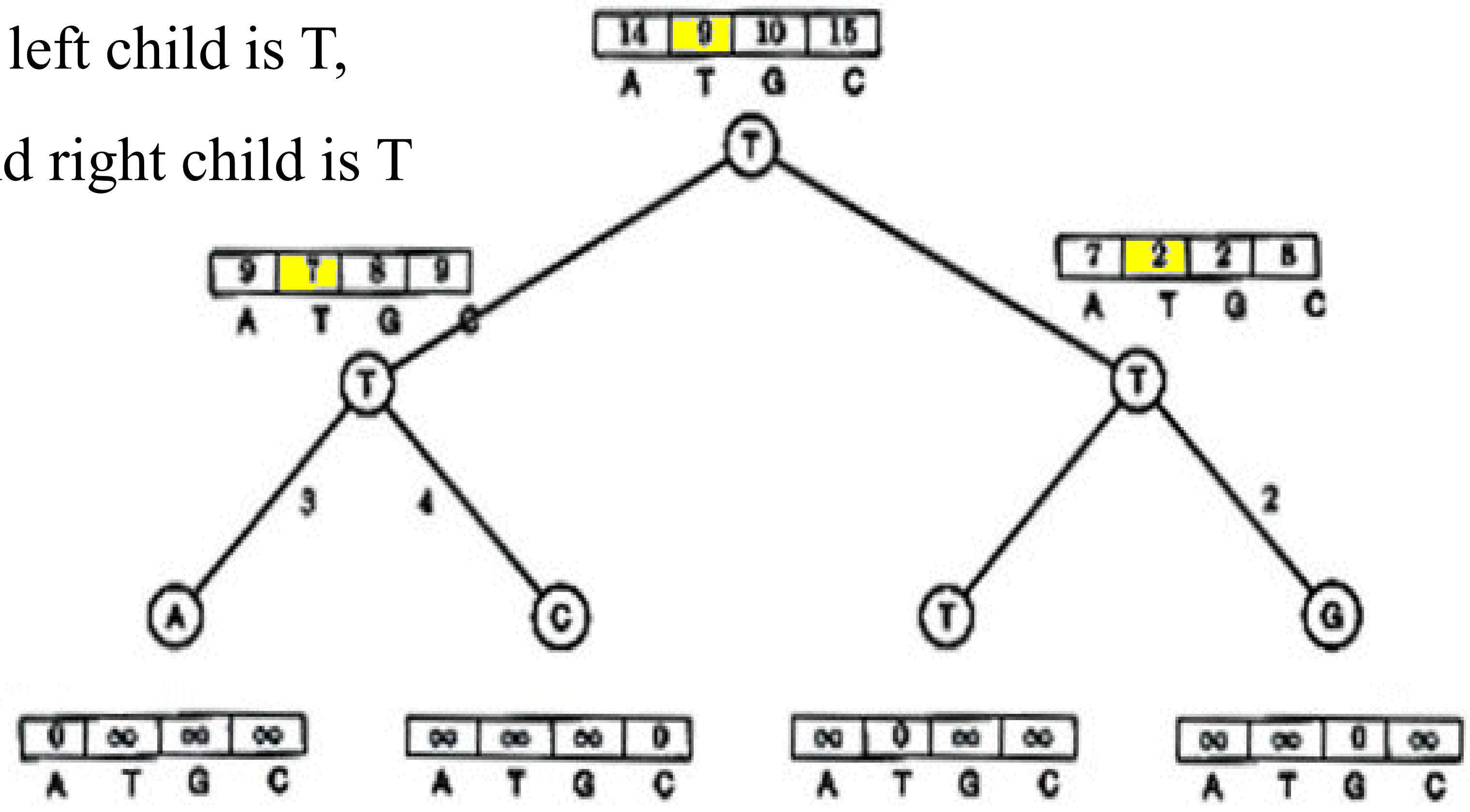
- The scores at the root vertex have been computed by going up the tree
- After the scores at root vertex are computed the Sankoff algorithm moves down the tree and assign each vertex with optimal character.

Sankoff Algorithm (cont.)

9 is derived from 7 + 2

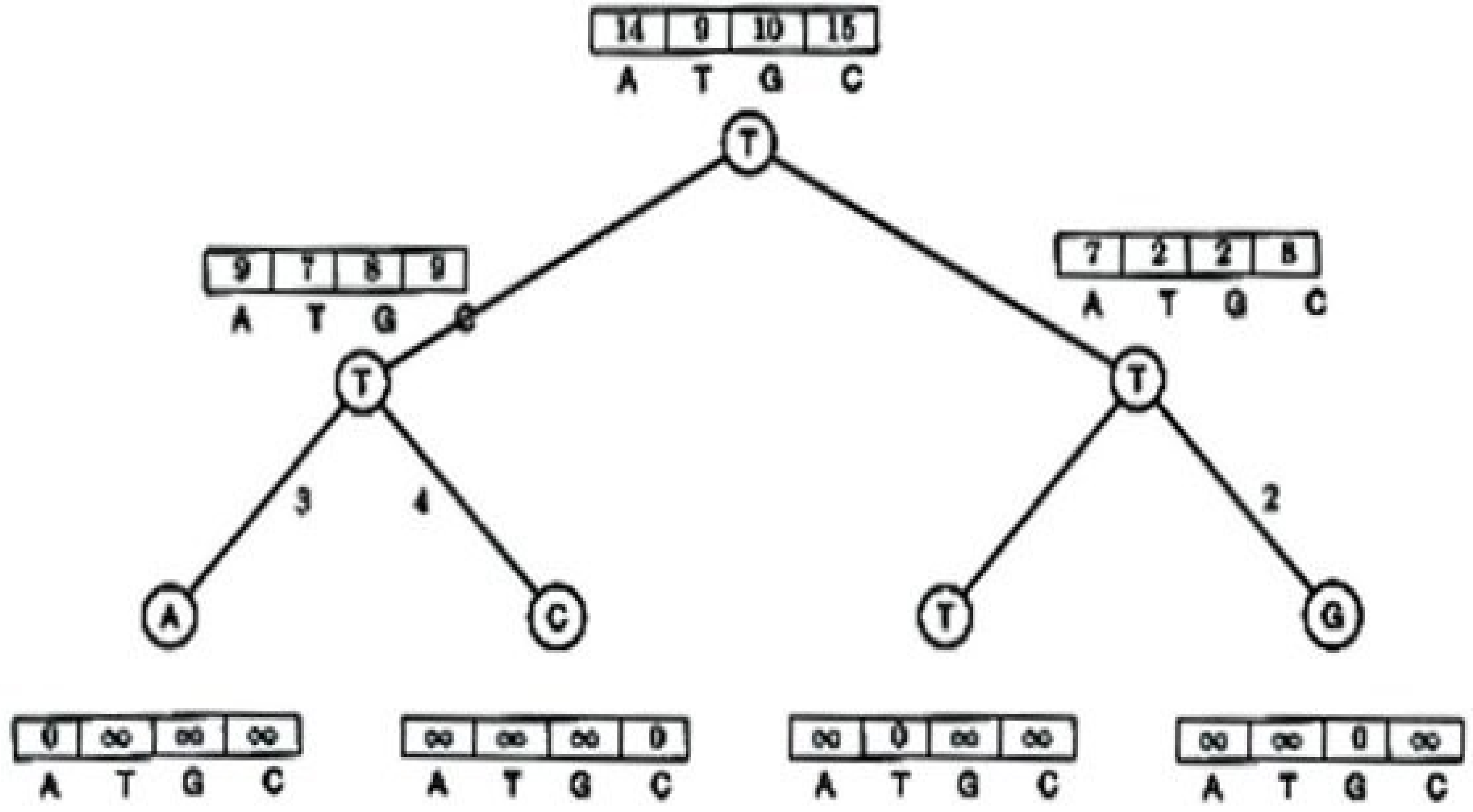
So left child is T,

And right child is T



Sankoff Algorithm (cont.)

And the tree is thus labeled…



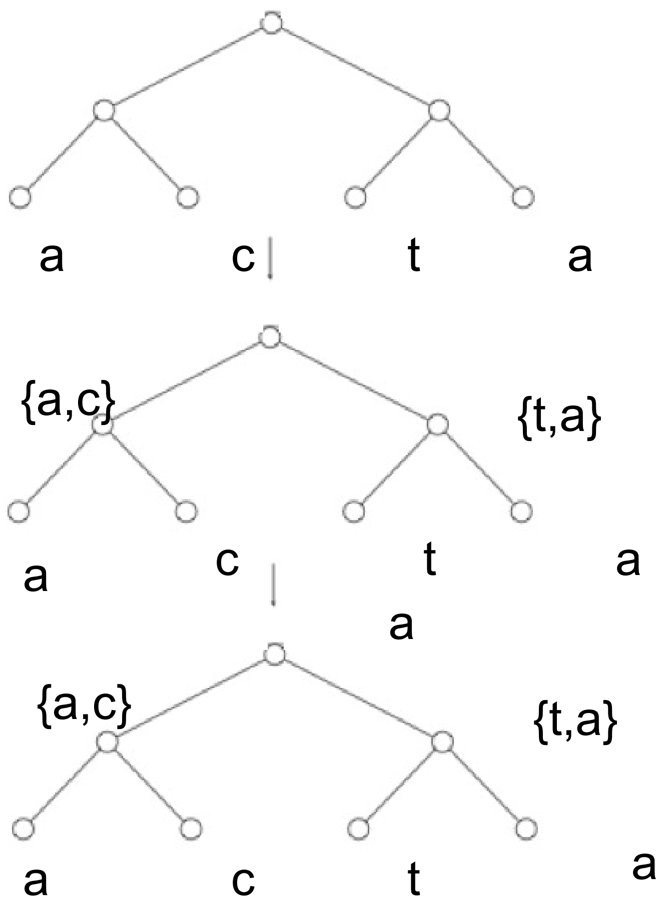
FITCH’S ALGORITHM

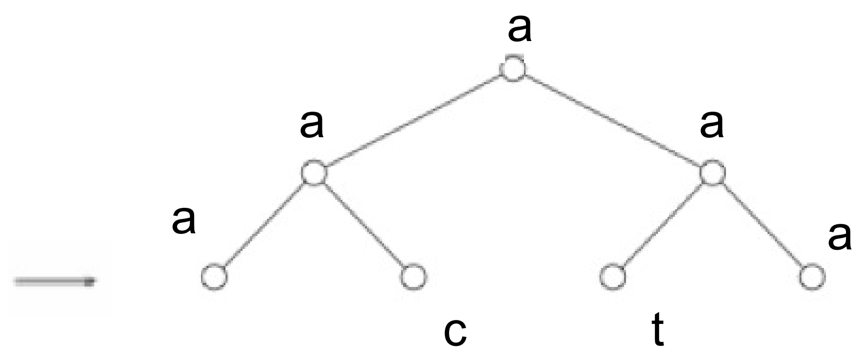
Fitch's Algorithm

- Solves Small Parsimony problem
- Dynamic programming in essence
- Assigns a set of letters 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

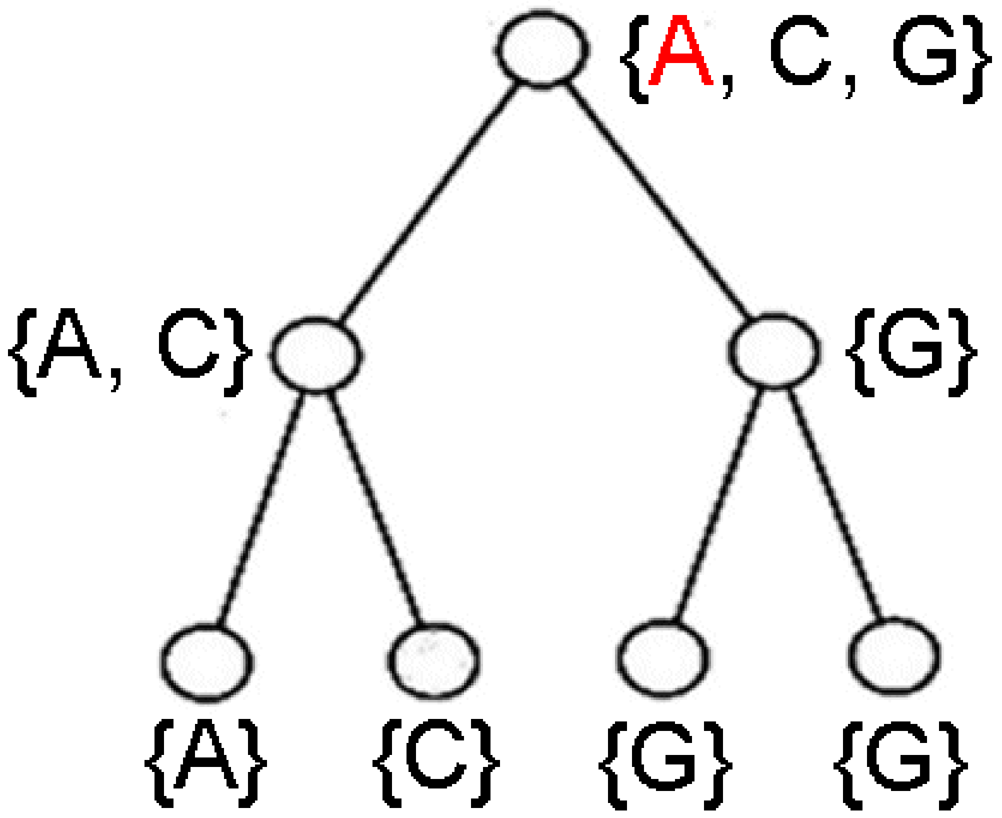
- Each node's set is the combination of its children's sets (leaves contain their label)
  - E.g. if the node we are looking at has a left child labeled {A, C} and a right child labeled {A, T}, the node will be given the set {A, C, T}

Fitch Algorithm (cont.)

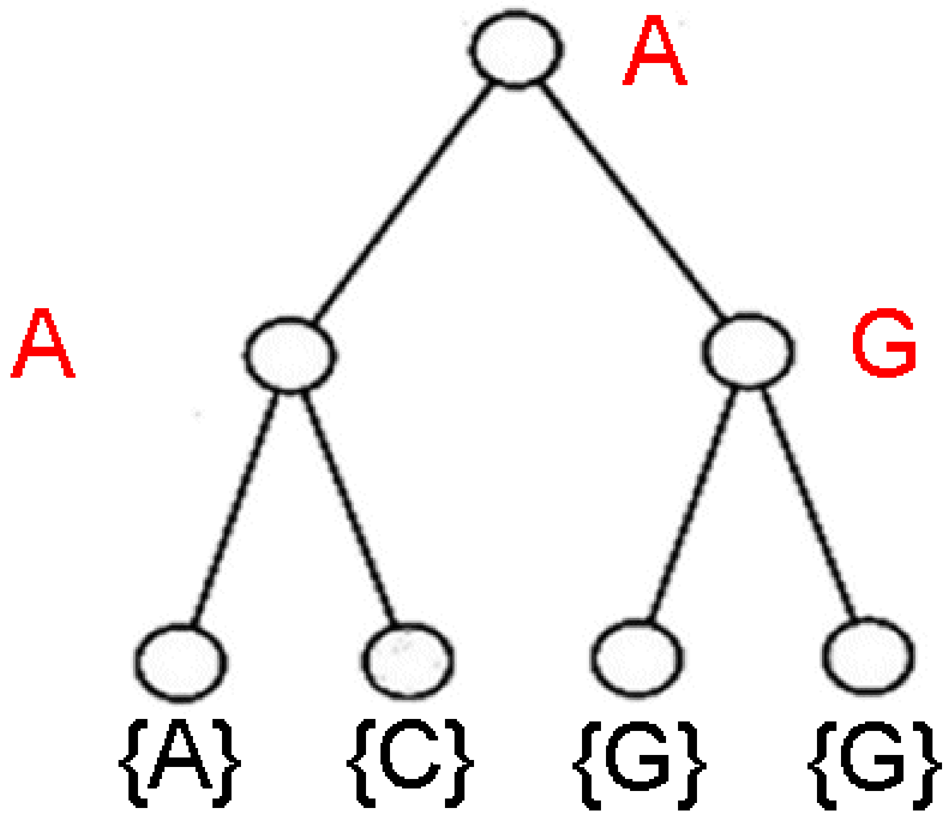
2) Assign **labels** to each vertex, traversing the tree from root to leaves

- 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

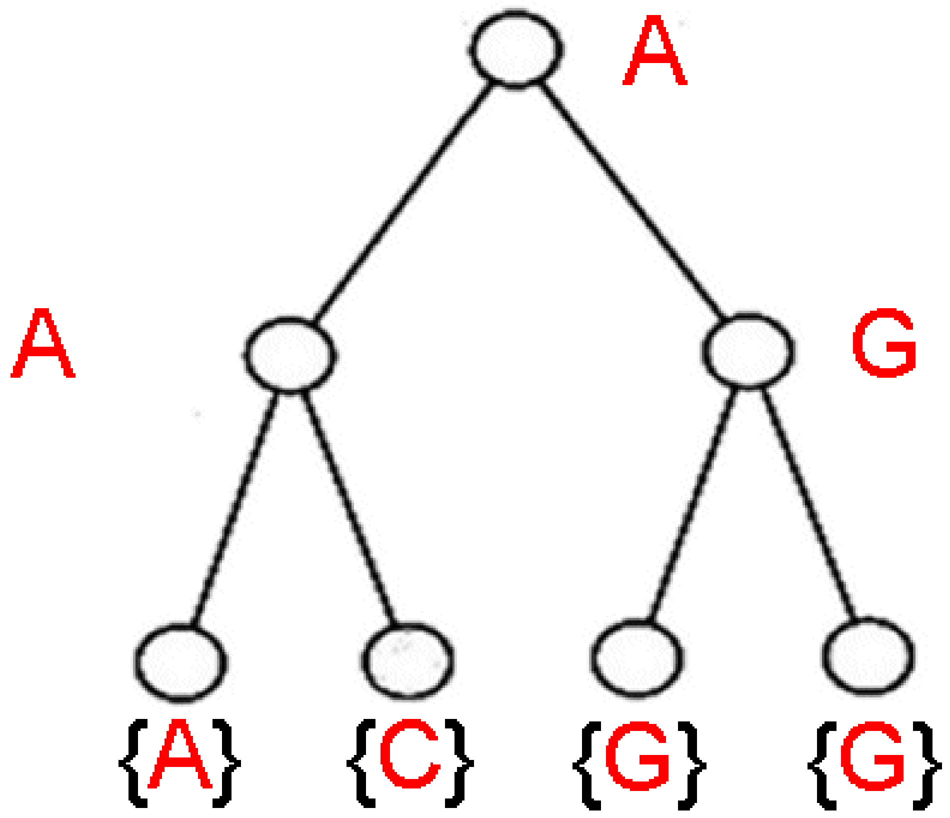
Fitch Algorithm (cont.)



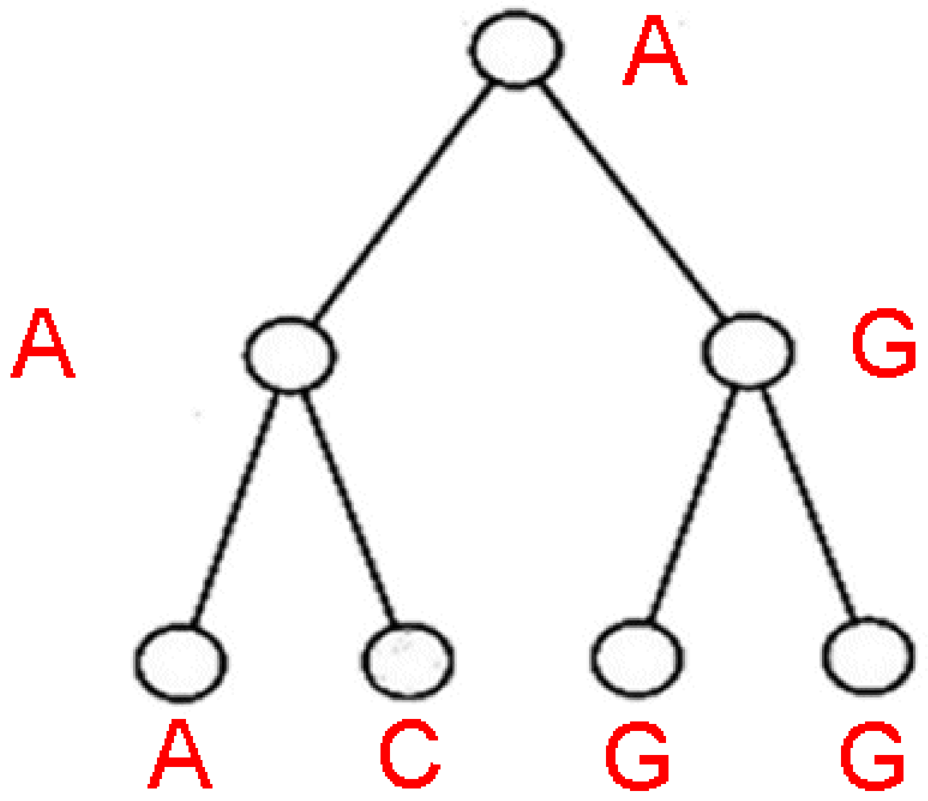
Fitch Algorithm (cont.)



Fitch Algorithm (cont.)



Fitch Algorithm (cont.)

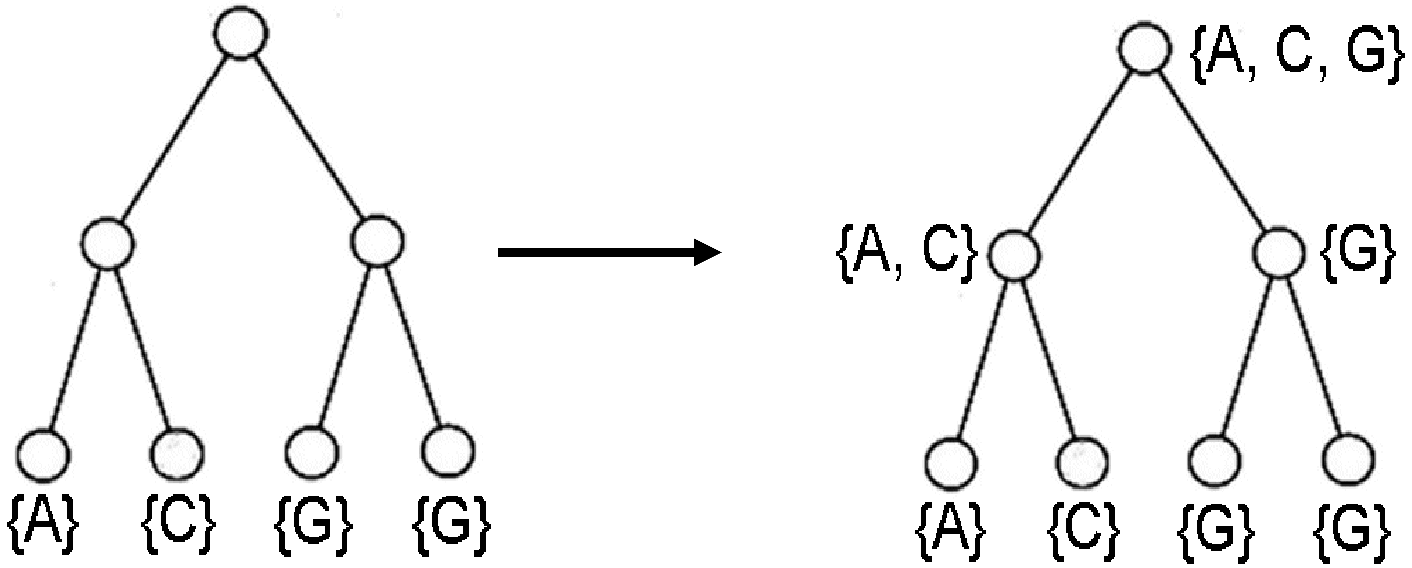


Fitch vs. Sankoff

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

Fitch

As seen previously:



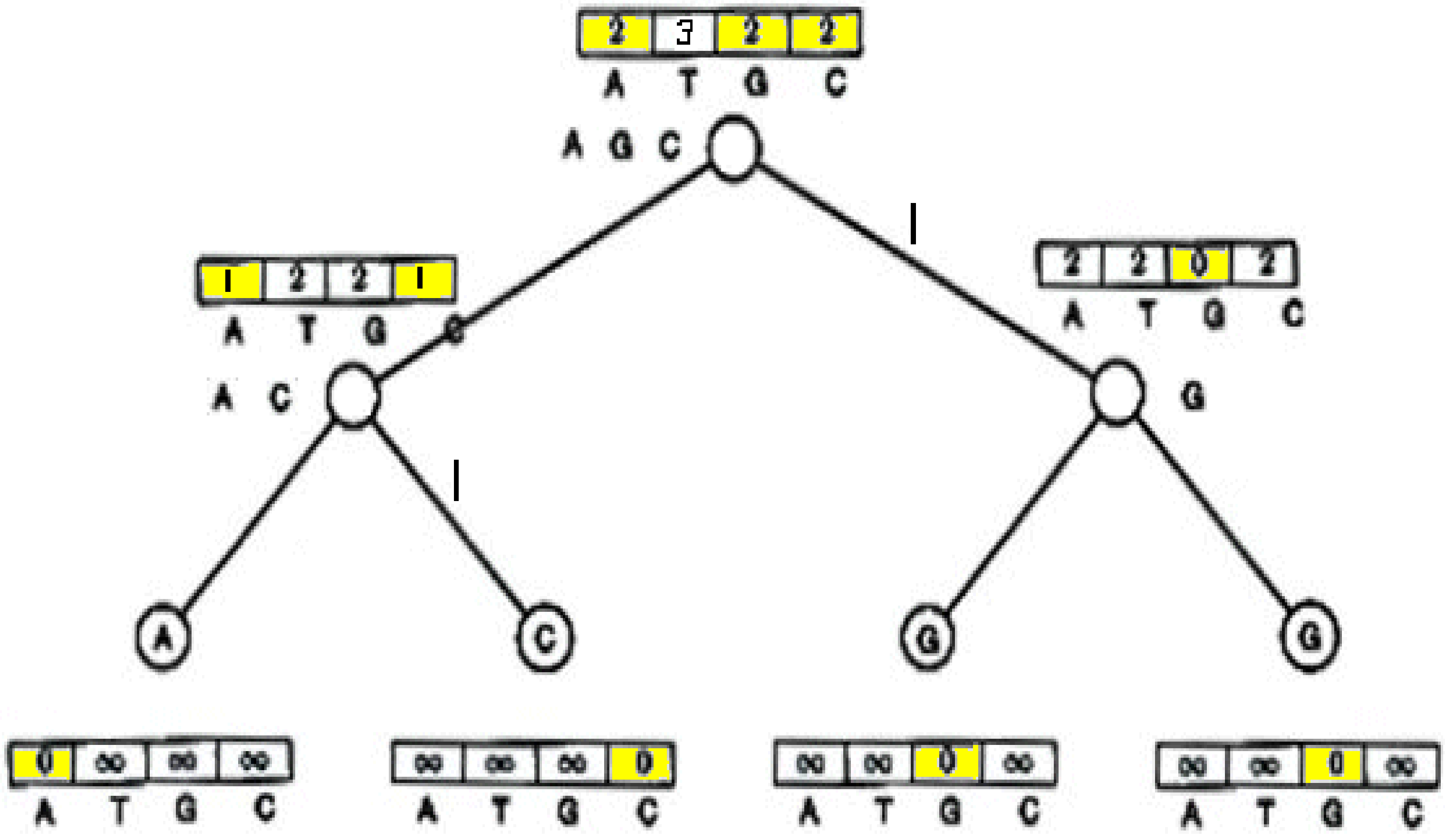
Comparison of Fitch and Sankoff

- 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