Supplementary Pseudocode

Figure S1 demonstrates the algorithm for the naïve all-against-all LCA parser, with pseudocode given in Algorithm S1. Algorithm S2, Algorithm S3, and Algorithm S4 spell out the pseudocode for the Hashing-Intersection algorithm demonstrated in main text Figure 4 of our paper.

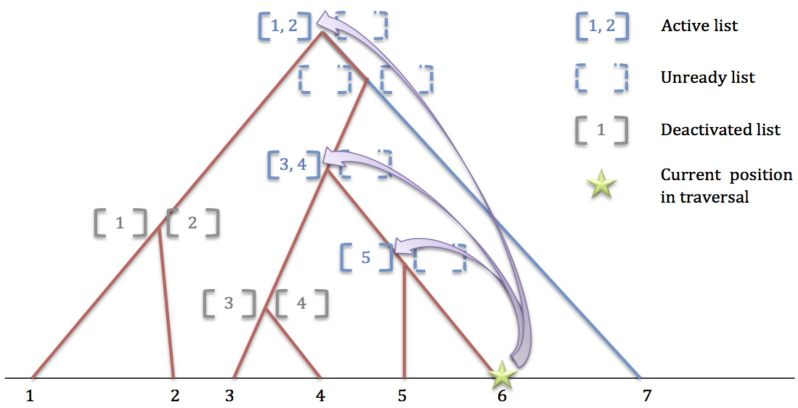

Fig. S1. The algorithm used for parsing a tree to get the all-pair tMRCAs. We perform an in-order traversal of the tree using a stack data structure to maintain the lineage leading from the root to the currently traversed node, whose elements are lists representing all internal nodes along the lineage, containing either all the leaves from the left sub-tree of the corresponding internal node (when we are currently in its right sub-tree), or nothing (when we are currently in its left sub-tree), and the height of the internal nodes. When we get to a new leaf, we can report the tMRCA of this leaf with all previously traversed leaves according to the records in the stack. The traversal only needs linear time, but as we need to report a tMRCA for each pair of leaves, the complexity of the output is the overall time complexity, $O(n^2)$.

Algorithm S1 In-Order Traversal All-Pair-tMRCA Extraction

```
ParseLCA(T)
A = {}
Tokens = tokenize T into [“(” “)” “,” numbers “;”]*
Leaves[] = [ ] // list for leaves
TMRCA[] = [ ] // list for tmrca’s
for token in Tokens
    if token == “,”|“;”
        pass
    if token == “(”// new internal node
        Leaves.push([ ])// prepared for storing leaves under this internal node
        TMRCA.push(None)// not yet known the height of this internal node
    if token == numbers
        parse the numbers into leaf and tmrca
        Leaves[−1].append(leaf)// Leaves[−1] is the last element of Leaves
        if TMRCA[−1] ==None
            TMRCA[−1] = tmrca
        for i in range(Leaves)
            for leaf_1 in Leaves[i]
                hash([leaf, leaf_1]) to TMRCA[i] in A
    if token == “)”
        if Leaves has reached it’s bottom
            pass
        else
            index = Leaves.last_index()// get the last element’s index of Leaves
            Leaves[index − 1].extend(Leaves[index])
            Leaves.pop()
            get tmrca from next numbers; move forward token’s pointer
            if TMRCA[index − 1] ==None
                TMRCA[index − 1] = TMRCA[index] + tmrca
                TMRCA.pop()
            else
                TMRCA.pop()
return A
```

Algorithm S2 Hashing Based Block-Finding Algorithm (Part 1)

```
TreePreProcess(tree1, n)
Repo1[], Tempo[] = {}, {} // the final and temporary hashtable for tree1
List1[] = [ ] // in-order leaves
List2[] = [0]^n // order look-up table
Stack[] = [ ] // cache the unprocessed internal nodes
L[] = tokenize tree1 into ["(" ")" "," numbers ";"]*
for token in L
    if token == ","|";"
        pass
    if token == "(" // a new internal node
        Stack.push(None)
    if token == numbers
        parse the numbers into sample and tMRCA
        List1.append(sample)
        List2[sample − 1] = List1.index(sample) // sample begins from 1
        if Stack[−1] =None
            Stack[−1] = tMRCA
            list = [tMRCA, List2[sample − 1], List2[sample − 1],None]
            if tMRCA in Tempo // may be duplicated
                Tempo[tMRCA].append(list)
            else
                Tempo[tMRCA] = [list]
        else
            Tempo[Stack[−1]][−1][3] = List2[sample − 1]
    if token == ")"
        if Stack has reached it's bottom
            list = Tempo[Stack[−1]].pop()
            if Stack[−1] in Repo1
                Repo1[Stack[−1]].append(list)
            else
                Repo1[Stack[−1]] = [list]
        else
            get tMRCA from next numbers; move forward token's pointer
            start, end = Tempo[Stack[−1]][−1][1], ...[3]
            list = Tempo[Stack[−1]].pop()
            if Stack[−1] in Repo1
                Repo1[Stack[−1]].append(list)
            else
                Repo1[Stack[−1]] = [list]
            tMRCA+ = Stack.pop()
            if Stack[−1] ==None
                Stack[−1] = tMRCA
                list = [tMRCA, start, end,None]
                if tMRCA in Tempo
                    Tempo[tMRCA].append(list)
                else
                    Tempo[tMRCA] = [list]
            else
                Tempo[Stack[−1]][−1][3] = end
return Repo1, List1, List2
```

Algorithm S3 Hashing Based Block-Finding Algorithm (Part 2)

```
TreePostProcess(tree_2, Repo_1, List_1, List_2, n)
Tempo[] = {}// the temporary hashtable for tree2
Result[] = {}// the final result hashtable
List_3[] = [ ]// in-order leaves and their indices in List_1
Stack[] = [ ]// cache the unprocessed internal nodes
L[] = tokenize tree_2 into ["(" ")" "," numbers ";"]*
for token in L
    if token == ","|";"
        pass
    if token == "("
        Stack.push(None)
    if token == numbers
        parse the numbers into sample and tMRCA
        List_3.append(List_2[sample - 1])// sample begins from 1
        if Stack[-1] ==None// come from left branch
            Stack[-1] = tMRCA
            start = List_3.index(List_2[sample - 1])
            list = [tMRCA, start, start,None]
            if tMRCA in Tempo// may be duplicated
                Tempo[tMRCA].append(list)
            else
                Tempo[tMRCA] = [list]
        else // come from right branch
            Tempo[Stack[-1]][-1][3] = List_3.index(List_2[sample - 1])
    if token == ")"
        tMRCA = Stack[-1]
        if tMRCA in Repo_1
            start, middle, end = Tempo[tMRCA][-1][1], ...[2], ...[3]
            list_left = List_3[start : middle]
            list_right = List_3[middle + 1 : end]
            Intersection(tMRCA, list_left, list_right, Repo_1, List_1, Result)
        if Stack has reached it's bottom
            Tempo[Stack[-1]].pop()
        else
            get tMRCA from next numbers; move forward token's pointer
            start, end = Tempo[Stack[-1]][-1][1], ...[3]
            tMRCA+ = Stack.pop()
            if Stack[-1] ==None
                Stack[-1] = tMRCA
                list = [tMRCA, start, end,None]
                if tMRCA in Tempo
                    Tempo[tMRCA].append(list)
                else
                    Tempo[tMRCA] = [list]
            else
                Tempo[Stack[-1]][-1][3] = end
return Result
```

Algorithm S4 Hashing Based Block-Finding Algorithm (Part 3)

INTERSECTION($tMRCA, list_{left}, list_{right}, Repo_1, List_1, Result$)

$list_{candidate} = Repo_1[tMRCA]$
for $list$ in $list_{candidate}$
// we should greedily search evidence for unchanged pairs
    for $can_{left}, can_{right} = list_{left}, list_{right}$
        and $can_{left}, can_{right} = list_{right}, list_{left}$
    // we should perform both intersection and reverse intersection
        $left, right = [\ ], [\ ]$
        for $leaf$ in can_{left}
            if $leaf$ in range $[list[1], list[2]]$
                $left$.append($List_1[leaf]$)
        for $leaf$ in can_{right}
            if $leaf$ in range $(list[2], list[3]]$
                $right$.append($List_1[leaf]$)
        if $left$ and $right$ not empty
            Hash ($left \times right$) to $tMRCA$ in $Result$
            // $\times$ is cartesian product, and each pair will be hashed