Format for Efficient Storage of Homology Relations

Week 8 Report: Efficient Implementation in C++

Kevin Gao

University of Toronto

July 21, 2022

Outlines

How to Query the Interval Index

Given two vertices and their corresponding labels, we can determine whether a vertex is an ancestor of another vertex via a range query of interval containment of their labels. A leaf label of value x can be thought as the interval $[x, x]$.

Given two leaves labeled x and y, we find the LCA by finding the shortest interval containing both x and y. This gives us the *lowest* common ancestor because the size of the interval is non-decreasing as we traverse from a leaf to root.

Given a gene represented as a leaf labeled x, in order to find all orthologs of the gene, we find all intervals containing x that are labels to a speciation node. We also remove the intervals (internal nodes) that are visited. After this, suppose we have non-overlapping intervals $S = \{[a, b],\ [c, d], \ldots\}$. Then, clearly, all the leaves with integer labels from a to b and so on are orthologs of the leaf labeled x.

Key Observations

Observation 1: Let x be a leaf in a tree. There is a unique path of length $depth(x)$ from x to the root.

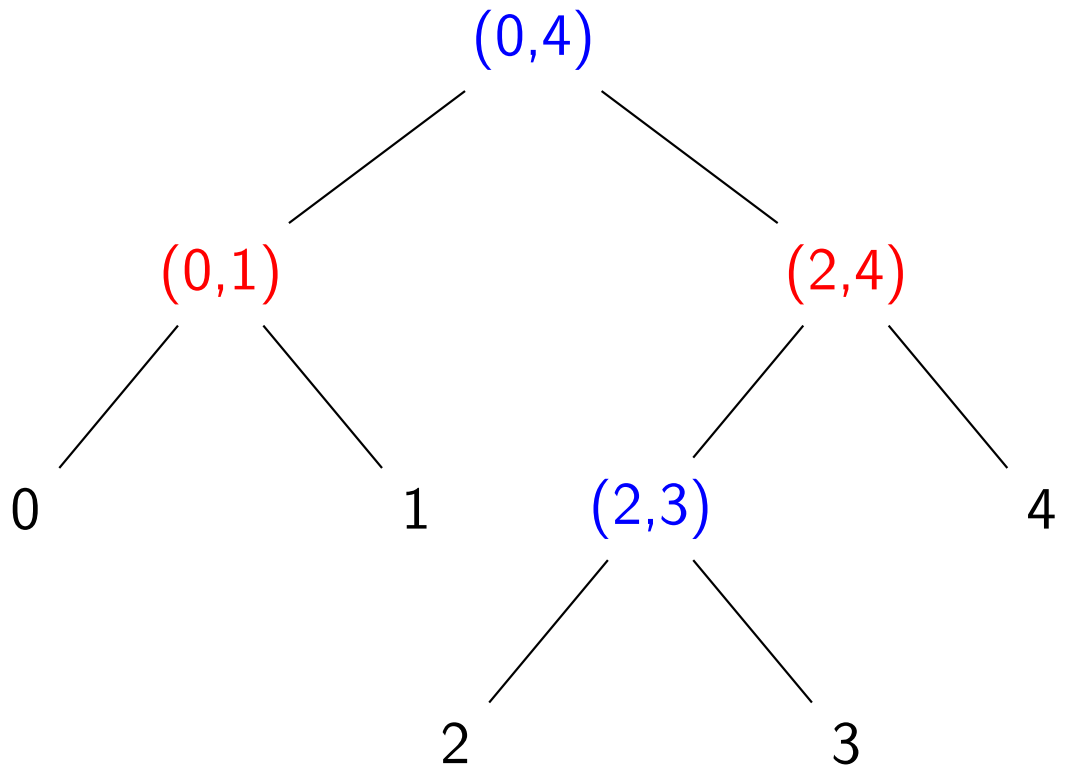
Observation 2: Let x be a leaf in a tree and let $P = x \to \cdots \to r$ be the unique path from x to the root r. Every internal node on this path P including r is labeled with an interval that contains x.

Observation 3: There are exactly $depth(x)$ internal nodes labeled with an interval that contains x.

Observation 4: Let a be an internal node, and let b be the immediate parent of a. Then, the interval represented by a is contained in the interval represented by b.

More on Query

Consider the example tree. Speciation nodes are colored blue, and duplication nodes are colored red.



More on Query

Suppose we want to find all orthologs of the gene labeled '2'. We consider all internal nodes on the unique leaf-to-root path originating from '2'. The intervals are: (2,3), (2,4), (0,4). We maintain two arrays: *orthologs* and *visited*.

orthologs stores the orthologs discovered. *visited* stores the intervals that are already visited on the path.

- (2, 3): speciation node, $\mathit{visited} = []$, $\mathit{orthologs} = []$
- (2, 4): duplication node, **skip**, $\mathit{visited} = [(2, 3)]$, $\mathit{orthologs} = [2, 3]$
- (0, 4): speciation node, $[(0, 1)]$ after excluding the visited intervals, $\mathit{visited} = [(2, 3), (2, 4)]$, $\mathit{orthologs} = [2, 3]$
- Result: $\mathit{visited} = [(2, 3), (2, 4), (0, 1)]$, $\mathit{orthologs} = [2, 3, 0, 1]$

Outlines

Binary Index Format

We store the index in a binary format. The whole index is divided into three blocks:

- Block 1: Leaf labels
- Block 2: Internal labels
- Block 3: Duplication nodes

Now, we discuss the content of each block in more details.

Block 1: Leaf Labels

Block 1 records the leaves and their labels. The first 4 bytes indicates the number n of leaves stored in this block, followed by n entries of variable length.

Each entry contains the following data:

- node type (4 bytes)
- integer label (4 bytes)
- gene name length (4 bytes)
- gene name (variable length, as specified by the previous field)
- hash (4 bytes): generated based on the absolute position of the node in the original XML document

Block 2: Internal Node Labels

Block 2 records the internal nodes and its labels. The first 4 bytes indicates the number m of internal nodes stored in this block, followed by m entries of 16 bytes.

Each entry contains the following data:

- node type (4 bytes): duplication/speciation/ambiguous
- integer label, min (4 bytes)
- integer label, max (4 bytes)
- hash (4 bytes): generated based on the absolute position of the node in the original XML document

Block 3: Duplication Nodes

Block 3 records a subset of internal nodes that are duplication nodes. The first 4 bytes indicates the number d of internal nodes stored in this block, followed by d entries of 16 bytes.

Each entry contains the following data:

- node type (4 bytes): duplication/speciation/ambiguous
- integer label, min (4 bytes)
- integer label, max (4 bytes)
- hash (4 bytes): generated based on the absolute position of the node in the original XML document

Outlines

Three Classes

There are three main classes:

- GeneTreeNode: Result of parsing the original XML file; stores structural information of a gene tree node; contains `children`, `parent`, and `get_ancestors`.
- IndexedGeneTreeNode: Result of parsing index file or manual creation when generating the index; no structural information; contains name of the node, type of the node, label, and the hash value of its original position in the XML file.
- GeneTree: Handles the I/Os with the gene tree and all underlying data structures; contains methods and fields like `get_all_orthologs`, `root`, `write_index`, etc.

Four Hashtables

Three hash tables are constructed during parsing: `leaves_map`, `leaves`, leaf_labels, internal_nodes.

- `leaves_map`: hash (int) $\rightarrow$ GeneTreeNode object
- `leaves`: gene name (string) $\rightarrow$ IndexedGeneTreeNode object
- `leaf_labels`: leaf label (int) $\rightarrow$ IndexedGeneTreeNode object
- `internal_nodes`: hash (int) $\rightarrow$ IndexedGeneTreeNode object

Queries

Given a string gene name, find the corresponding IndexedGeneTreeNode object using `leaves`.

Extract the leaf label (int) and hash value from the IndexedGeneTreeNode object. Use the hash value to find the corresponding GeneTreeNode object through `leaves_map`.

Once we have the GeneTreeNode object, we can find all the ancestors (GeneTreeNode object). Hash the ancestors, and use `internal_nodes` to find the corresponding IndexedGeneTreeNode object containing internal labels. The labels are already sorted in increasing order by the range (because they are converted from a leaf-to-root path, recall Observation 4).

Queries cont'd

Perform the algorithm described earlier. After this, we will have a list of integer leaf labels. Use either `leaves_map` or `leaves` to convert the labels back to string gene names.

One Interval Tree

An interval tree is constructed when parsing the index file. It stores the labels of all duplication nodes as intervals.

Interval tree (centered interval tree) is a tree data structure for storing intervals and efficient interval queries. As we will discuss later, interval tree takes $O(n \log n)$ time to construct, and query (point query or interval query) takes $O(\log n)$ using an algorithm resembling binary search. n is the number of intervals, which in the context of our problem is the number of internal nodes in the original tree.

We need this data structure to determine whether a pair of orthologs is one-to-one, one-to-many, or many-to-many.

Interval Tree for Many-to-X Relationship

Let x be a leaf that we are querying for all the orthologs. Suppose we are currently considering an ancestor y of x which is a **speciation** node. Further, suppose that we have not encountered any duplication nodes on the path from x to root. Then, all descendants of y that are not visited are orthologs of x.

We can query the interval tree of duplication nodes and which descendants of y are duplication nodes. Let D be the set of labels of descendants of of y that are duplication nodes. Then, for each $d = (d_1, d_2) \in D$, the unvisited leaves with label $d_1, \ldots, d_2$ are one-to-many orthologs of x.

Next Week!

- Benchmark!
- Fix minor bugs and improve performance and stability
- Derive a bound on time and space complexity of this implementation (might take longer)

Outlines

Zhang, J., Lovette, K.: XimpleWare W3C Position Paper. In: W3C Workshop on Binary Interchange of XML Information Item Sets (2003)

Haim Kaplan, Tova Milo, and Ronen Shabo. 2002. A comparison of labeling schemes for ancestor queries. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '02). Society for Industrial and Applied Mathematics, USA, 954–963.