Format for Efficient Storage of Homology Relations

Week 7 Report: Interval Labeling Index

Kevin Gao

University of Toronto

July 14, 2022

Outlines

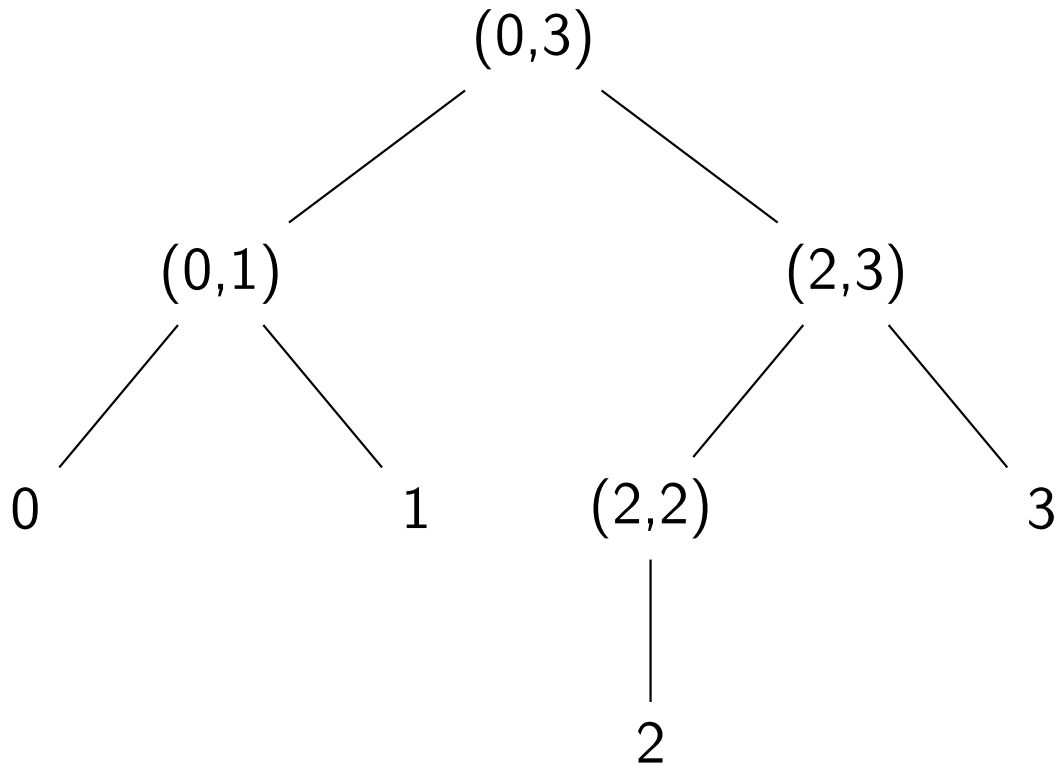
Idea of Interval-based Indexing

The interval index uses an interval-based labeling scheme. We begin by assigning every leaf x a unique label, denoted $label(x)$. This is maintained in two hash tables: one that maps leaf to label, and another one that maps label to leaf. A label to a leaf is called a **leaf label**.

Let v be an internal vertex, and suppose that v has a set of leaves in the subtree rooted at v, say L, with labels $L' = \{label(x) \mid x \in L\}$. Then, we assign v the label $\min\{L'\}\$\max\{L'\}$ where \$ is a delimiter. A label to an internal node is called an **internal label**. Given an internal label l, we use l_1 to denote the first part of the label (the portion before the delimiter), and use l_2 to denote the second part of the label (the portion after the delimiter). For clearity, we use (l_1, l_2) instead of $l_1\$l_2$ in future discussions.

An interval labeling can be constructed by first assigning leaf labels and iteratively updating the internal nodes on the leaf-to-root path for each leaf.

Example of Interval-based Index



Leaves are labeled sequentially from left to right. Internal nodes are labeled based on the minimum leaf label and maximum label in the subtree rooted at that internal node.

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.

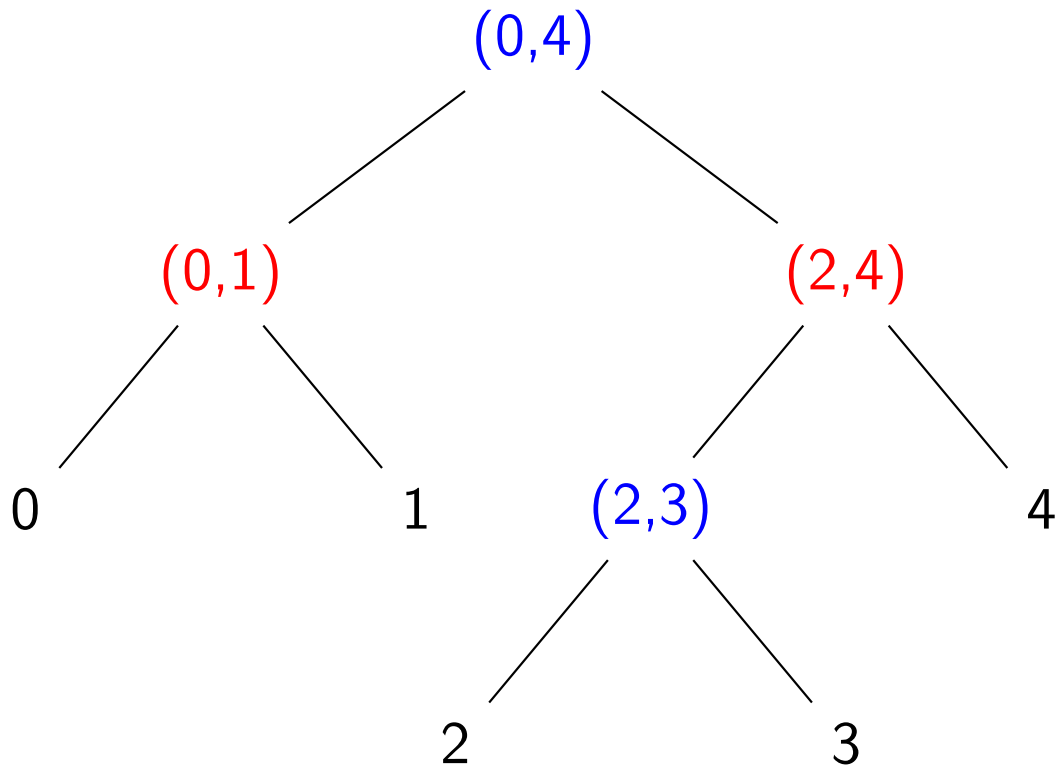
Observation 2: Let x be a leaf in a tree and let $P = x \to \cdots \to r$ be the unique path from x to the root r. Every internal node on this path P including r is labeled with an interval that contains x.

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

All Intervals Covering a Certain Point

One of the queries central to our algorithm is the problem of finding all intervals covering a point given a query point and a set of intervals to query from.

Based on Observation 3, with our labeling scheme, all the intervals containing a point x are labels of the internal nodes on the leaf-to-root path from the leaf labeled with x. We need to at least visit $depth(x)$ many nodes to collect all intervals containing x.

If we have the tree structure parsed from the input residing in the memory with efficient random access, we can traverse the tree starting from the node labeled with our query label x. This takes precisely $depth(x)$ node visits.

Otherwise, we can construct an interval tree.

Interval Tree

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.

Given a set of intervals S, the interval tree for S stores the median x_{mid}, along with the set of intervals overlapping with x_{mid} at the root. The left subtree of the root is an interval tree for all intervals whose right endpoint is less than x_{mid}. The right subtree is an interval tree for all intervals whose left endpoint is larger than x_{mid}.

Outlines

Data to Keep Track of

There are a few fields that we need to keep track of:

- Bijection between node and labels: use hashing?
- List of all speciation node labels
- List of all duplication node labels

With our current implementation, they are stored directly within the memory. But for our data format, we will encode these fields into a separate binary index file. Storing the index in a separate file would increase interoperability.

Outlines

Benchmark Result

Benchmarks are done on trees of various sizes. Compared to non-indexed methods, we can perform on average 2 to 3 times more operations per second.

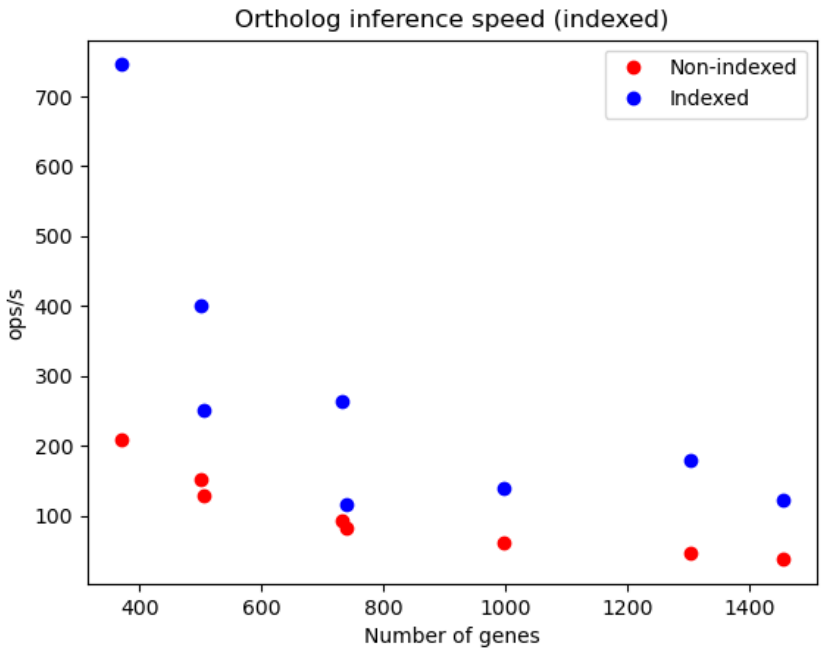


Figure: Operations per second on different trees.

Other Considerations

Skewed trees and especially trees with a lot of speciation nodes but relatively fews duplication nodes (and vice versa) has visible performance drop compared to other trees.

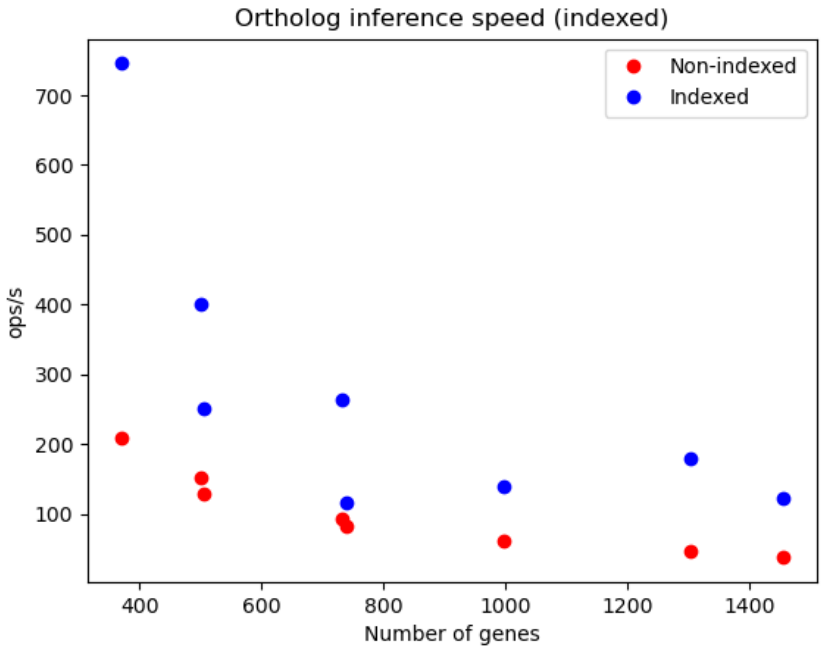
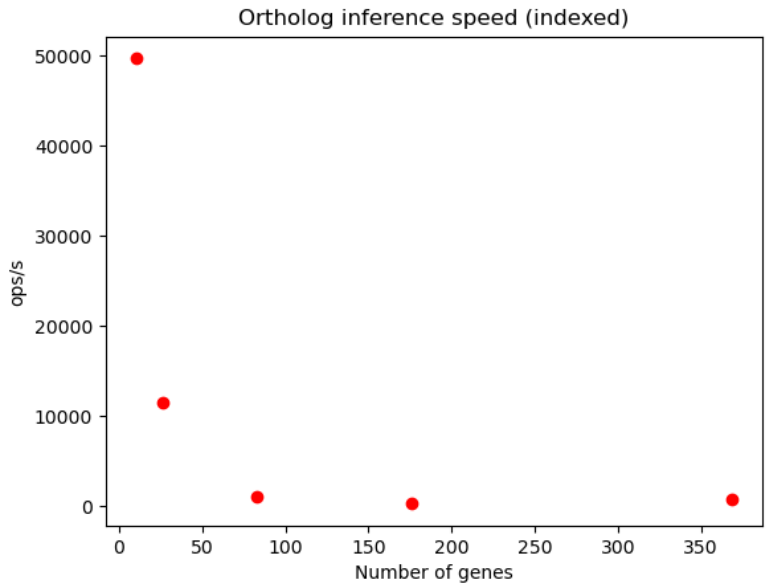
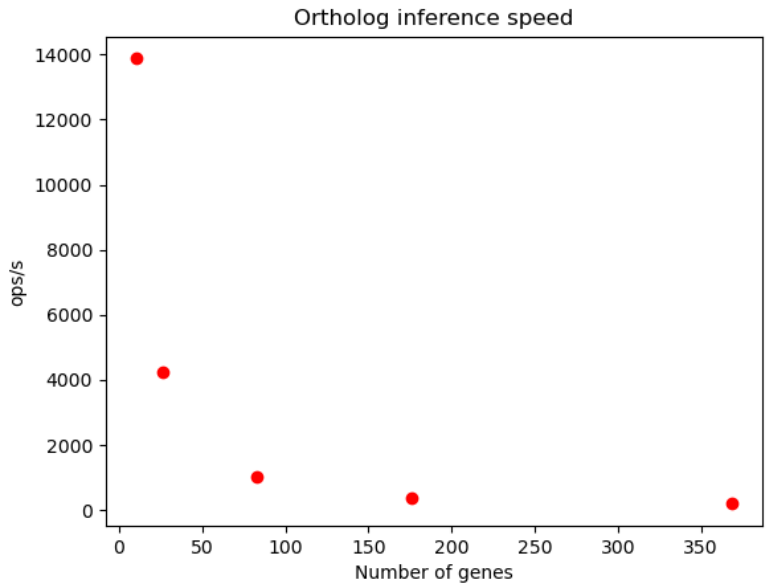


Figure: Two trees of size 500 but very different number of speciation nodes shows performance fluctuation depending on number of speciation/duplication.

The Whole Picture





Outlines

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