TreeScaper Manual

Version 1.0

June 30, 2022

Jeremy Ash[2,4],
Jeremy M. Brown[2],
David Morris[2],
Guifang Zhou[2],
Wen Huang[1],
Melissa Marchand[3],
Paul Van Dooren[1],
Kyle A. Gallivan[3],
and Jim Wilgenbusch[5]

[1]Department of Mathematical Engineering, ICTEAM,
Université catholique de Louvain, Belgium,
wen.huang@uclouvain.be

[2] Department of Biological Sciences and Museum of Natural Science,
Louisiana State University, Baton Rouge, LA, USA

[3] Department of Mathematics,
Florida State University, Tallahassee, FL, USA

[4] Current Address: Bioinformatics Research Center,
North Carolina State University, Raleigh, NC, USA

[5] Minnesota Supercomputing Institute,
University of Minnesota, Minneapolis, MN, USA

TABLE OF CONTENTS

This documents currently only collects implementation details and algorithmic concerns of certain functionalities in TreeScaper, in specific, some concerns noted in version 1.0.

1 Representation of tree

In TreeScaper, we consider the typical [(Is this typical?)]$_{\text{ZDcomm}}$ tree structure encountered in phylogenetic area, which is a *binary, unrooted, undirected* tree with only leaf nodes specified as well as an edge length/weight assigned to each edge. In real-world applications, we may find the following less general cases:

1. *Rooted case:* The tree is taken from a larger tree by trimming an internal edge. The node used to attached to that trimmed edge is then a root to the tree.

2. *Non-binary case(polytomies):* Binary in unrooted tree is equivlent to the statement that all internal nodes has degree of 3. Another equivalent way to interpret the "binary" features is that every internal node only has 2 "descendants". For example, pick any leaf node and find the other node that is connected to it. [The leaf node only has one node connected to it.]$_{\text{ZDcomm}}$ Such node connects to exactly 2 leaf node and 1 internal node [, or being a root in the rooted tree case]$_{\text{ZDcomm}}$.

   Note that such order of going from leaf to internal implies a direction/order in phylogenetic trees. Therefore, for non-binary phylogenetic tree, it means that at least one internal node has degrees of more than 3, i.e., it has more than 2 descendants. The other term used in literatures is "*polytomy*" for internal nodes with degrees more than 3.

3. Note that we generally do not consider directed tree structure.

Phylogenetic tree as a member of undirected weighted graph, a natural way to represent a phylogenetic tree is by adjacency matrix.

We further introduce the following notations that will be repeatedly used in later discussion.

1. Phylogenetic tree T.

2. The set of vertices $V(T), V$. The set of leaf nodes (usually labelled with taxon) $L(T), L$. The set of edges $E(T), E$.

3. The number of leaf nodes n.

1.1 Node specification and adjacency representation of tree.

Definition 1.1. Let the ordered set $v := \{v_1, \cdots, v_n\}$ be a given specification of nodes (including the unlabelled internal nodes), then the adjacency matrix

$$A_v := [A_{i,j}]_{n\times n}$$

completely represent the tree, where $A_{i,j}$ be the edge length between node v_i and v_j or 0 if no edge is found. The matrix A_v is referred as adjacency representation of the tree w.r.t. the specification v.

Note that only leaf nodes has natural labels, the specification is not necessarily unique. In other words, the adjacency representation A_v is not necessarily unique.

Remark 1.2. It is easy to see that for another ordered set $w = Pv$ permuted by the permutation matrix P, the adjacency representation is given by $A_w = PA_vP^T$.

Proposition 1.3. *Let A_v be an adjacency representation of a binary unrooted tree, then the i-th row has either*

1. *exactly* 1 *nonzero entry when v_i is a leaf node,*
2. *or exactly* 3 *nonzero entries when v_i is an internal node.*

In conclusion, the adjacency representation of a binary unrooted tree is not unique, over-parametrized and sparse. Therefore, we seek for more efficient and precise representation by imposing rules on specification of nodes, v, such that A_v has nonzero strcuture that can be easily exploited.

1.1.1 Node specification by reverse Breadth First Search (rBFS)

Recall that breadth-first search(BFS) of a graph is a search/traversal that starts from any given node and travel to all of the nodes that has distance 1 to the starting node and then to all of the nodes with distance 2 and so on. See more details with example in this wiki page. The starting node is usually chosen to be an internal node, or the root if it is defined.

However, for assigning a specification to nodes in phylogenetic trees, we have all the leafs specified already and no internal nodes specified. Therefore, we introduce a reverse breadth-first search(rBFS) that starts from the leaf nodes and travel in the order measured by how far the node is away from the entire leaf set.

In order to better describe the rBFS, we introduce the following measurement on how far a node is away from any given set.

$$d(n, v) := \text{the number of neighbors of node } n \text{ that is not in the set } v. \tag{1}$$

Note that such measuremeant is sloppy and not a distance. But it gives a good idea on how to access nodes that are very close to the given set.

Definition 1.4. For a tree T with the set of nodes $V(T)$ and the set of leaf nodes $l := \{l_1, \cdots, l_k\} \subset V(T)$. Reverse breadth-first search(rBFS) produces the running record of vertices X in the following order:

1. (Initialization) Let the running record be
$$X = V^0 = L.$$
2. Given the running record $X = \bigcup_{j=0}^{i} V^j$,
$$V^{i+1} := \left\{ n \in V(T) \setminus \bigcup_{j=0}^{i} V^j : d(n, v) \leq 1 \right\}.$$
3. Add V^{i+1} to the running record $X = \bigcup_{j=0}^{i+1} V^j$ and continue until $V^k = \emptyset$.

The final ordered set X is a node specification of $V(T)$. Note that such v is not unique as the order within each level V^i is arbitrary.

The following proposition guarantees that rBFS defined above is indeed a traversal on tree T by showing that every step in rBFS creates a subtree in T.

Proposition 1.5. *Let T be any tree with leaf nodes L. Let $\{T_i\}_{i=1}^k$ be a set of distinct subtrees in T, such that their leaf nodes exhaust L but $\bigcup_{i=1}^k V(T_i) \subsetneqq V(T)$. Then there always $v \in V(T) \setminus \bigcup_{i=1}^k V(T_i)$, such that*

$$d\left(v, \bigcup_{i=1}^{k} V(T_i)\right) \leq 1.$$

Proof. We proof by contradictory. Suppose $\forall v \in V(T) \setminus \bigcup_{i=1}^k V(T_i)$,

$$d\left(v, \bigcup_{i=1}^{k} V(T_i)\right) \geq 2.$$

To see it, notice that by definition of a subtree, each T_i has its root connected to only 1 node that is not in T_i, denote that node as y_i. By assumption,

$$d\left(y_j, \bigcup_{i=1}^{k} V(T_i)\right) \geq 2, j = 1, \cdots, k.$$

Remove all T_i from T, the remaining graph $\tilde{T}$ is still a tree with leaf nodes $\tilde{L} \subset \{y_1, \cdots, y_k\}$. However, by definition, any y_j is connected to at least 2 nodes in $V(T) \setminus \bigcup_{i=1}^k V(T_i)$, i.e., any y_j has at least degree of 2 in the new graph $\tilde{T}$.

Since a leaf node has degree exactly 1, the new tree $\tilde{T}$ is consist of no leaf node, which is impossible.

Therefore, there is at least one $d\left(v, \bigcup_{i=1}^k V(T_i)\right) \leq 1$, such that

1. when $d\left(v, \bigcup_{i=1}^k V(T_i)\right) = 1$, v is a leaf node of the new tree $\tilde{T}$,
2. or when $d\left(v, \bigcup_{i=1}^k V(T_i)\right) = 0$, i.e., v connects to all subtrees, which make the new tree $\tilde{T}$ a trivial graph with single node $\{v\}$.

□

This proposition guarantees step 2 in rBFS is always possible. Then at least on new node will be accessed in step 2, which makes rBFS a tree traversal.

Remark 1.6. Note that the rBFS is applicable to any tree structure. Also note that the rBFS is equivalent to trimming all leafs attached to V^{i+1} at i-th step and making V^{i+1} new leafs to the remaining tree.

Corollary 1.7. *rBFS creates a sequence of sets of subtrees that always exhaust the set of leaf nodes L, starting form trivial n subtrees*

$$\{T_1^{(1)}, \cdots, T_n^{(1)}\} = \{\{l_1\}, \cdots, \{l_n\}\}$$

, and at any intermediate j-th iteration, to,

$$\left\{\tilde{T}^{(j)}, T_k^{(j-1)}, \cdots, T_s^{(j-1)}\right\},$$

where $\left\{T_1^{(j-1)}, \cdots, T_s^{(j-1)}\right\}$ is from last iteration and at this iteration, the accessed v has neighbors in $T_1^{(j-1)}, \cdots T_{k-1}^{(j-1)}$ and forms the new tree $\tilde{T}^{(j)}$ by connecting $T_1^{(j-1)}, \cdots T_{k-1}^{(j-1)}$.

Remark 1.8. Note that the sets of subtrees are not ordered set. Implicit permutation has been made so that always $(T_1^{(j-1)}, \cdots, T_s^{(j-1)}$ are grouped together for simplier description.

Note that the post-order traversal also maintained similar structure, in which a node is never accessed until all subtrees under itself has been accessed. This will be an important features that affects the design of identificaition of bipartitions in later discussion.

Intuitively, a traversal of tree completely represents/characterized the tree itself. Later proposition gives a proof on how rBFS completely represents a tree from the aspect of adjacency matrix.

Proposition 1.9. *The adjacency representation of a tree w.r.t. the node specification v generated from rBFS can be partited into block matrix such that the diagonal blocks are always zeros*

$$A_v = \begin{bmatrix} \mathbf{0}_{n\times n} & & B_1 & \\ & \begin{bmatrix} \mathbf{0}_{|V^1|\times|V^1|} & B_2 & \\ & \ddots & \vdots \\ B_2^T & \cdots & \mathbf{0} \end{bmatrix} & \\ B_1^T & & \end{bmatrix}$$

or

$$A_v = \begin{bmatrix} \mathbf{0}_{n\times n} & & B_1 & \\ & \begin{bmatrix} \mathbf{0}_{|V^1|\times|V^1|} & B_2 & \\ & \ddots & \vdots \\ B_2^T & \cdots & \begin{bmatrix} 0 & * \\ * & 0 \end{bmatrix} \end{bmatrix} & \\ B_1^T & & \end{bmatrix}$$

where each row in $B_i \in \mathbb{R}^{|V^i|\times(|V|-\sum_{j=1}^{i}|V^j|)}$ has only 1 nonzero entry and $*$ represents unknown real number.

Proof. For the first one, we have all leaf nodes $V^0 = L$. Other than the trivial two-node tree that give $A_l = \begin{bmatrix} 0 & * \\ * & 0 \end{bmatrix}$, no leaf node will be connected to another leaf node, which makes the first block diagonal of size $d_0 \times d_0$ being $\mathbf{0}$.

Since all leaf nodes are connected to one node, which means all leaf nodes are connected to a node in $v \setminus V^0 = \{v_{|V^0|+1}, v_{|V^0|+2}, \cdots\}$. Since nonzero in a row of adjacency matrix represents an edge, B_1 to the right of $\mathbf{0}_{|V^0|\times|V^0|}$ must have exactly 1 nonzero at each row.

For the second one and the rest, they are

$$V^{i+1} := \left\{ n \in V(T) \setminus \bigcup_{j=0}^{i} V^j : d(n, v) = 1 \right\}.$$

Notice that $\forall n \in V^{i+1}$, it only has one edge connected to node in $V(T) \setminus \bigcup_{j=0}^{i} V^j$. Further notice that the $\bigcup_{j=0}^{i+1} V^j$ can only connect to $\bigcup_{j=i+1}^{k} V^j$ via V^{i+1}.

If any of two $n_1, n_2 \in V^{i+1}$ are connected with an edge, they run out of edge budget and no other node in V^{i+1} can be attached to the connected component includes n_1, n_2. However, a tree is a connected graph, which implies $b^{i+1} = \{n_1, n_2\}$. In this case, we arrive at the lowest-right-most block in A_v of the form $\begin{bmatrix} 0 & * \\ * & 0 \end{bmatrix}$ and no block to its right exist in this case.

Otherwise, no nodes from V^{i+1} are connected to each other with one edge, which makes the i-th $|V^i| \times |V^I i$ diagonal being $\mathbf{0}$. Then every node in V^{i+1} still has 1 edge connected to a node in $V \setminus \bigcup_{j=0}^{i+1} V^j$, which corresponds to exactly 1 nonzero in each row of B_i to the right of $\mathbf{0}_{d_i\times d_i}$. □

Corollary 1.10. *During the rBFS of a tree, if the exact edge and its edge length between a node in V^i and $v \setminus \bigcup_{j=1}^{i} V^j$ can be identified, the tree can be fully represented by v and those edges.*

Proof. This comes from the symmetry in A_v as well as the fact that all blocks to the right of the **0** diagonal blocks has only 1 nonzero on each row. Being able to fill those blocks in the right completely represents the symmetric A_v, and therefore completely represents the tree. □

1.2 Bipartition representation of tree.

Bipartitions is an important objects of the analysis in phylogenetic trees. It involves with the distance computation [and some examples]$_{\text{ZDcomm}}$.

The idea of bipartition is a direct result of the fact that no loop exists in a tree. Removing any edge e from a tree will break the tree into two connected components. Consider the set of leaf nodes L, the two connected components then gives a bipartition on $L = L_1(e) \cup L_2(e)$, where $L_i(e)$ are leaf nodes in a same connected component.

Definition 1.11. For a given tree T, the set of bipartitions $\mathcal{B}(T)$ in this tree is the collection of bipartitions on leaf set induced by removing 1 edge of the tree cross the edge weights.

$$\mathcal{B}(T) = \{(L_1(e), L_2(e)) : e \in E(V)\}.$$

Note that $(L_1(e), L_2(e))$ is not an ordered pair. Usually we assume $L_1(e)$ to be the smaller set.

Note that a tree with $|E(T)|$ edges defines $|E(T)|$ bipartitions but not arbitrary collections of $|E(T)|$ bipartitions can be found in a tree structure. In particular, the bipartitions that can be found in a tree strcture obeys the following proposition.

Proposition 1.12. *For a tree T and for arbitrary $b \in \mathcal{B}(T)$, b is either trivial bipartition of the form*

$$b = (\{l\}, L \setminus \{l\}), l \in L$$

or there exists a subset of bipartition $\{b_i\}_{i=1}^{k} = (L_{i,1}, L_{i,2}) \in \mathcal{B}(T)$, such that

$$b = (\cup_{i=1}^{k} L_{i,1}, L \setminus \cup_{i=1}^{k} L_{i,1}).$$

For the latter case, we refer it as b can be constructed by $\{b_i\}_{i=1}^{k}$. [Note that $L_{i,1}$ does not necessarily be the smaller set in here.]$_{\text{ZDcomm}}$

Proof. This simply comes from the no-loop properties in trees. The trivial case is simple.

For any internal node $v \in V(E)$, there are $k \geq 3$ edges $e_1, \cdots, e_k$ attached to v, those edges defines k bipartitions on the leaf set L.

Consider removing all of those edges which yields k subtrees and we have a k-partition on leaf set L as

$$L = \bigcup_{i=1}^{k} L_i$$

where L_i is the leaf set of the i-th subtree.

Notice that $b_i = (L_i, L \setminus L_1) \in \mathcal{B}(T)$ is induced by removing edges e_i respectively. Further notice that

$$L \setminus L_1 = \bigcup_{i=2}^{k} L_i$$

which complete the proof. □

Recall that under the rBFS order, we only need to identify 1 extra edge at every node to represent the tree, which is also associated to 1 bipartitions. It is natural to ask if the set of bipartitions enough to represent the tree.

Proposition 1.13. *Let $\mathcal{B}(T)$ be a set of bipartitions induced by a tree T. A rBFS of T can be recovered by $\mathcal{B}(T)$ by the following pseudo algorithm. If the record of edge length of the edge that induces a bipartition, $w(b(e)) = w(e)$ is provided, the tree T can be completely reconstructed.*

1. *(Initialization) Let $i \leftarrow 0$. Construct $|L|$ nodes, denote them as $l_1, \cdots, l_n$ and assigned them with bipartitions $(\{l_1\}, L \setminus \{l_1\}), \cdots, (\{l_n\}, L \setminus \{l_n\})$ respectively. Denote the note set $V^0 = L$ and bipartitions set $\mathcal{B}(V^0) = (\{l_1\}, L \setminus \{l_1\}), \cdots, (\{l_n\}, L \setminus \{l_n\})$.*
2. *Given V^j and $\mathcal{B}(V^j)$ for $j = 0, \cdots, i$, find all $b \in \mathcal{B}(T) \setminus \bigcup_{j=0}^{i} \mathcal{B}(V^j)$, such that b can only be constructed by bipartitions in $\bigcup_{j=0}^{i} \mathcal{B}(V^j)$.*
3. *For every found b that can be constructed by $\{b_j\}_{j=1}^{k}$, construct a node v_b, appoint the new node to the set V^{i+1} and assign the node with b. Appoint b to $\mathcal{B}(V^{i+1})$.*
4. *Connect v_{b_i} and v_b, if weight record is provided, with edge length $w(b_i)$.*
5. *Terminate if $\mathcal{B}(T)$ is exhausted. Otherwise $i \leftarrow i + 1$ and repeat step $2 - 4$.*

Proof. We only need to notice that step 2 is equivalent to finding a node in the step 1 of rBFS.

For any node v satisfies $d\left(v, \bigcup_{j=0}^{i} V^j\right) = 1$, let $e_1, \cdots, e_k$ be its edges. By definition, only one of these edges has the other node not in the set $\bigcup_{j=0}^{i} V^j$, WLOG, let it be e_k. Then $b_1, \cdots b_{k-1} \in \bigcup_{j=0}^{i} \mathcal{B}(V^j)$. Notice that b_k can be constructed by $b_1, \cdots b_{k-1}$, which implies that the entire

$$\left\{ n \in V(T) \setminus \bigcup_{j=0}^{i} V^j : d\left(n, \bigcup_{j=0}^{i} V^j\right) = 1 \right\}$$

will be found.

On the other hand, for $v \notin \bigcup_{j=0}^{i} V^j$ and $d\left(v, \bigcup_{j=0}^{i} V^j\right) \neq 1$, we have

$$d\left(v, \bigcup_{j=0}^{i} V^j\right) \geq 2.$$

In other words, v is connected to at least 2 nodes that are not in $\bigcup_{j=0}^{i} V^j$. WLOG, let those edges be e_{k-1}, e_k, that connects to v_{k-1} and v_k. Let b_i be bipartitions induced by e_i. Since both v and v_{k-1} are not in $\bigcup_{j=0}^{i} V^j$, the bipartition b_{k-1} induced from e_{k-1} is not in $\bigcup_{j=0}^{i} \mathcal{B}(V^j)$ and same for b_k. However, b_k can be constructed by $b_1, \cdots, b_{k-1}$, therefore, no node other than those from

$$\left\{ n \in V(T) \setminus \bigcup_{j=0}^{i} V^j : d\left(n, \bigcup_{j=0}^{i} V^j\right) = 1 \right\}$$

will be found. □

Remark 1.14. The rBFS is an traversal to the tree and therefore being able to perform an rBFS is equivalent to being able to represent the tree. Therefore, the pseudo algorithm for performing rBFS from the set of bipartitions (and record of edge length/weights) is a constructive proof of saying that the set of bipartitions (and the record of edge lengths/weights) completely represent the tree. Also note that in TreeScaper we have not implemented the rBFS from the set of bipartition because we do not have an application scenario.

Corollary 1.15. *Any (weighted) tree can be completely represented by the bipartitions it induces on the leaf set (with the record of edge lengths/weights).*

1.3 Implementation details of tree representation

The implementations in this part includes 3 main pieces:

1. Conversion from tree input format (Newick) to a classic tree format in address space (linked list).
2. Representation of bipartition in address space and the computation associated to it.
3. Identification (Fast look up) of the bipartitions.

1.3.1 Conversion from Newick to linked list

Newick format is essentially an post-order traversal that can be easily implemented by a stack of notes.

Definition 1.16. The post-order traversal given by Newick form string can be implemented as followed.

1. (Initialization) Linked node set $V = \emptyset$, `i = 0`, string `s`, Stack container `C`.
2. If `s[i] == '('`. (Internal node.)
   (a) Form a new node v, add it to V.
   (b) Draw edge from the top of `C` to v.
   (c) Push v to `C`.
3. If `s[i] == 'a' to 'Z'`, collect the label name and edge length/weight until `','`. (Leaf node.)
   (a) Form a new leaf node v and labelled it, add it to V.
   (b) Draw edge with the collected edge length/weight from the top of `C` to v.
   (c) Push v to `C`.
4. If `s[i] == ')'`, collect edge length/weight until `','`. (Finish the traversal under an internal node.)
   (a) Pop the top of `C`.
   (b) Assign the collected edge length/weight to the edge between the top of `C` and the node just got popped.
5. `i = i + 1`. Repeat step $2 - 4$.

TreeScaper V1.0. The linked list form of tree is the most straight forward format one can store a tree in address space and that is the strategy used in TreeScaper V1.0. In particular, the linked list form is implemented by intuitive (not necessarily simple) `linked-node-base` data stracture, which is a customized node object with pointer and contents.

The equivalence between the tree and bipartitions it generates was not exploited in TreeScaper V1.0 and therefore the linked node also carry bipartitions informations for later computations, which is redundant in terms of informations.

There is also another issue of redundant informations caused by labels. The labels creates from the list of taxa may be very long but TreeScaper V1.0 store a copy of the exact taxon names in every trees, which may be larger than the entire tree plus all bipartitions in address space. Not to mention that the taxa labels, which is essentially the leaf node can be easily specified by the biparititons.

Also note that for unknown reasons (maybe for reducing memory usage), the TreeScaper V1.0 use `singly-linked-list` which significantly limited the options of accessing a node. Thanks to the fact that no access to a single node is required, the `singly-linked-list` structure does not make significant impact on complexity.

TreeScaper V2.0. The linked list form of the tree is still used in TreeScaper V2.0 as intermediate representation for fast rBFS purpose. It is not clear if there is a faster way to directly perform rBFS from Newick string.

Since the linked list is only temporary and the tree node carries very simple information, we implemented an `array-base` linked list for efficient and easy-to-recycle purposes. Once the bipartition representation is computed, the linked list form is released and reused for the next tree. Also since rBFS requires access from child node to parent node, TreeScaper V2.0 implements a `array-base-doubly-linked-list` structure.

1.3.2 Bitstring representation and computation of bipartitions.

Bitstring representation.

Bipartition on a given set is essentially an assignment function from the set to a set with only 2 elements. Those that are assigned to the same values belongs to a same group. In address space, the smallest unit that can carry information of a set of 2 elements is `bit` that stores 0 or 1. Therefore, the assignment from leaf set L with n elements to $\{0,1\}$ can be stored as a bitstring of length n, where the i-th bit's values is the assignment of the i-th taxon.

Definition 1.17. Suppose leaf nodes are specified with natural number $1,\cdots,n$ as $l_1,\cdots,l_n$. Let $b=(L_1,L_2)$ be a bipartitions on L, then a bitstring representation of b is given by

$$s_1(b)=\{\delta_{l_i,L_1}\}_{i=1}^n$$

or

$$s_2(b)=\{\delta_{l_i,L_2}\}_{i=1}^n$$

where $\delta_{l_i,L_j}=1$ if $l_i\in L_j$ or 0 otherwise.

Note that $s_1(b)$ and $s_2(b)$ are complementary to each other, doing a `XOR` operation on each element results in all-1 sequence.

[Since the smallest unit we can operate on `C++` has 8 bits, the actual bits of the bitstring in address space is $\lceil n/8\rceil\times 8$.]$_{\text{ZDcomm}}$

Note that the notion of "i-th taxon" implies that there is a ordered specification on leaf set. We defer this discussion to the implementation details of the customized object `TaxonList`.

Also note that complementary bitstrings represent the same bipartition. Therefore, we propose the following regulation for unique representation of bipartition on address space. [Is it necessary?]ZDcomm

Definition 1.18. The *regular* bitstring of a bipartition $s(b)$ in address space is the one of $s_1(b), s_2(b)$ that has leading 0.

Bipartition computation.

Thanks to Prop. 1.12, we can construct a bipartition out of other bipartitions associate to a common node. As long as we can guarantee that bipartitions of all edges except one of an node are computed, the remaining one bipartition is computable.

In terms of bitstring representation, if the k-partition of the given node induced from removing all k edges attached to the node is

$$L = \bigcup_{i=1}^{k} L_i$$

and if the bitstrings of bipartition $(L_i, L \setminus L_i)$ for $i = 1, \cdots, k-1$ are given by

$$j\text{-th bit} = \begin{cases} 1, l_j \in L_i \\ 0, l_j \notin L_i \end{cases},$$

then the bitstring of L_k can be easily found as the bit-wise-or of the bitstrings of $L_1, \cdots, L_{k-1}$.

Finally, notice that both post-order traversal and rBFS of a tree fit into the conditions we mention above and therefore we have the pseudo code for computing bipartition.

Definition 1.19. Given a post-order traversal or rBFS of a tree with leaf node L, all bipartitions of a tree is found as followed.

1. (Initialization.) For each leaf l_i, assigned bitstring

$$j\text{-th bit} = \delta_{i,j}$$

2. Perform the traversal. If the current node is an internal node, compute the bit-wise-or of all of its "child" nodes or all known bitstrings within its neighbor node. Assign the resulting bitstring to the current node.

Note that the post-order traversal starts from a node, which induce a direction on tree where every node only has one "parent" node. On the other hand, rBFS always found those nodes that only has one edge connected to unaccessed nodes, which both meet the first condition of having only one edge of a node not yet dealt with. The fact that we assign $[\delta_{i,j}]_{j=1}^{n}$ to leaf node and starting the traversal from leaf to internal meet the second condition.

Also note that the bitstring is not necessarily *regular*.

TreeScaper V1.0. If the post-order traversal starts with the root as the node attached to l_1, only $[1, 0, 0, 0, \cdots, 0]$ that has leading 1 is produced from the code and then TreeScaper 1.0 makes special adjustment to this bitstring. [More precisely, it drops all trivial bipartitions induced by a leaf node, therefore, this one is dropped.]ZDcomm

TreeScaper V2.0. Since the exact bipartition is no longer explicitly stored in a tree, instead, every unique bipartition encountered is stored in a dynamic array and identified by its index of this array, the output of this pseudo code are kept as it is but a regulated version of bitstring will be sent to the dynamic array.

rBFS in tree with access to any neighbors of any node.

Definition 1.20. Provided a tree with access to all neighbors of any node and the leaf nodes L, define a status of a node: `finished`. The rBFS is perform as followed.

1. (Initialization) Mark all leaf nodes with status `finished`. Create queue container `C`. Push all non-`finished` neighbors of leaf nodes to `C`.
2. Pop `C`. If the popped node has all but one of its neighbor nodes marked `finished`.
   (a) Mark the node `finished`.
   (b) Push the only non-`finished` neighbor node of this node to `C`.

   Otherwise, do nothing.
3. Repeat step 2 until `C` is empty.

Note that only TreeScaper V2.0 implement rBFS.

1.3.3 Identification of bipartitions

For leaf nodes L with size $|L| = n$, the largest possible number of edges is obtained in the case of binary tree with

$$|E| = 2n - 3.$$

On the other hand, the space of all possible bipartitions on the leaf set L, which is equivalent to the space of all *regular* bitstring, is 2^{n-1}. Note that $2^{n-1} \gg 2n - 3$ when n is large.

Since the later phylogenetic analysis includes repeatedly comparing bitstrings, it may be worthwhile to compute a simplier identification of bitstrings other than the bitstring itself stored in address space.

The argument behind this need of idenfication replies on the assumptions:

1. The comparison between bitstring is costly.
2. There is a unique cheap identification

   $$\mathcal{I} : M \to N$$

   of all encountered *regular* bitstrings/bipartitions, $b \in M$, where the comparsion on space of N is a lot cheaper than the comparsion in M, so that

   $$b_1 = b_2 \Leftrightarrow \mathcal{I}(b_1) = \mathcal{I}(b_2)$$

   and the bit-wise operations done in the idenfication plus the comparison in N is less than the bit-wise operation of the comparison in M.

Please note that this assumption does not always hold, especially when the bitsize of operated unit is taken into consideration. We will see an simple example in below.

A general solution for identifying small set of elements in a large space is the hash table in which $\mathcal{I}$ is defined as a map from object to a basic `type` *that behaved random enough*, see more details in this wiki page.

Recall that the computation of bitstring is done in a traversal of the tree recursively, it is ideal if $\mathcal{I}$ can be done in the same recursive way that reuses the already computed $\mathcal{I}(b)$ in the following way

1. (Initialization) For leaf node $l_i \in L$, compute $\mathcal{I}(l_i)$.

2. Perform post-order traversal or rBFS, for every node v arrived, use the known $\mathcal{I}(v_i), i = 1, \cdots, k-1$ to compute $\mathcal{I}(v)$ where $\{v_i\}_{i=1}^{k-1}$ are $k-1$ out of k neighbors of v that have been arrived before.

Discussion of uniqueness

Note that we have not discussed the uniqueness of $\mathcal{I}$. Since we have little control/knowledge of the set of bipartitions encountered in the phylogenetic analysis w.r.t. a set of trees that share same leaf set L beforehand, there will be no guarantee on uniqueness. The case where $b_1 \neq b_2$ but $\mathcal{I}(b_1) = \mathcal{I}(b_2)$ is usually referred as *collision* in hash table. The "*behaving randomly enough*" requirement on $\mathcal{I}$ aims on avoiding collison as mush as possible.

Another way to handle collision is by setting a collision beam of those $b_1 \neq b_2$ but $\mathcal{I}(b_1) = \mathcal{I}(b_2)$, which is another record of b that share the same value under $\mathcal{I}$. The hash table partically implement the collision beam by defining multi-level hash-value with multi-$\mathcal{I}_j$ as follow:

$$\mathcal{I} : M \to N_1 \times \cdots \times N_k, b \mapsto (\mathcal{I}_1(b), \cdots, \mathcal{I}_k(b))$$

where $\mathcal{I}_{j+1}$ is in more random than $\mathcal{I}_j$(and therefore usually more costly) so that for $b_1 \neq b_2$ with $\mathcal{I}_j(b_1) = \mathcal{I}_j(b_2)$, it is more likely to have $\mathcal{I}_{j+1}(b_1) \neq \mathcal{I}_{j+1}(b_2)$. In terms of complexity, $\mathcal{I}_{j+1}(b_2)$ is never computed unless there is a $b_1 \neq b_2$ found in previous records that

$$\begin{cases} \mathcal{I}_1(b_1) = \mathcal{I}_1(b_2) \\ \quad \vdots \quad . \\ \mathcal{I}_j(b_1) = \mathcal{I}_j(b_2) \end{cases}$$

In other words, when a collision upto j-th hash values happened, compute the $j+1$-th hash values and the collection

$$\{(\mathcal{I}_1(b_1), \cdots, \mathcal{I}_j(b_1), \mathcal{I}_{j+1}(b_1)), (\mathcal{I}_1(b_2), \cdots, \mathcal{I}_j(b_2), \mathcal{I}_{j+1}(b_2))\}$$

is essentially the collusion beam at

$$(\mathcal{I}_1(b_1), \cdots, \mathcal{I}_j(b_1)).$$

Note that there is still no guarantee of uniqueness and in practise there is no knowledge beforehand on how many $\mathcal{I}_j$ is enough to obtained uniqueness of a given tree set.

TreeScaper V1.0. In this early version, not much large tree set is tested and therefore a 2-level hash table is chosen for $\mathcal{I}$ as followed.

Definition 1.21. The identification mapping of *regular* bistrings in TreeScaper V1.0 is defined as followed.

1. (Initialization)

   (a) Generate random number $0 < B_1 \ll B_2$ in type `unsigned long long` where B_1 is a reasonable large enough number. They define $N_1 = \{0, \cdots, B_1 - 1\}$ and $N_2 = \{0, \cdots B_2 - 1\}$.

   (b) Generate random number $r_1, \cdots, r_n$ uniformly on N_2.

(c) Assign them to

$$\mathcal{I}_2(l_j) = \mathcal{I}_2(b_j) = r_j$$

and then assign

$$\mathcal{I}_2(l_j) = \mathcal{I}_1(b_j) = r_j \ \texttt{mod} \ B_1$$

for $j = 1, \cdots, n$

2. Perform post-order traversal, for every node v arrived, let $\{v_i\}_{i=1}^{k-1}$ be $k-1$ out of k neighbors of v that have been arrived before, compute

$$\mathcal{I}_1(v) = \left(\sum_{i=1}^{k-1} \mathcal{I}_1(v_i) \right) \ \texttt{mod} \ B_1$$

and

$$\mathcal{I}_2(v) = \left(\sum_{i=1}^{k-1} \mathcal{I}_2(v_i) \right) \ \texttt{mod} \ B_2$$

Note that both $\mathcal{I}_1, \mathcal{I}_2$ are mandatory in TreeScaper V1.0 because it does not implement a sphochastic hash table.

Recall that both arithmetic addition and modulo requires 2 bitwise operations and TreeScaper V1.0 operates on the `unsigned long long` unit of size 64 bits.

Let us leave the random number generator in initialization alone, for a binary tree with n leaves, TreeScaper V1.0 requires $64\lceil n/64 \rceil \times 8$ bit operations to compute $\mathcal{I}$ at nontrivial b. The comparison on N_1 then require 64 bit operations. On the other hand, comparing *regular* bitstring directly requires $64\lceil n/64 \rceil$.

Suppose there are n_t binary trees that contains $n_t(n-3)$ nontrivial bipartitions (possibly with repeated bipartitions), every *regular* bitstring needs to be compared $(n_t-1)(n-3)-1$ times in later phylogenetic analysis. Suppose there are n_c collisions at the first level, then extra n_c comparison on N_2 is required, which makes the total bit opeartions as

1. $64((n_t-1)(n-3))(n-3)\lceil n/64 \rceil \div 2$ for direct comparsion on *regular* bitstrings.

   As for every one in $n-3$ nontrivial bipartitions in a tree, it needs to be compared with other $(n_t-1)(n-3)$ nontrivial bipartitions.

   Because comparison is symmetric, the halve of those operations are not required.

2. $512n_t(n-3)\lceil n/64 \rceil + (64((n_t-1)(n-3))(n-3) + 64n_c) \div 2$ for 2-level hash table.

   As hash values of $n_t(n-3)$ nontrivial biparititons and $n_t n$ trivial bipartitions need to be computed. Then the comparison happens on the type `unsigned long long`.

   Note that those hash values of trivial bipartitions does not have to be recomputed but it was recomputed anyway in TreeScaper V1.0.

Assuming the no collision case $n_c = 0$, we have

$$n_t \left[(n-19)\lceil n/64 \rceil - (n-3) \right] \geq (n-3)(\lceil n/64 \rceil - 1)$$

Simple algebra shows that when $n \leq 64$, this inequality never holds, which implies that the hash table approach slows down the later analysis. When $n > 64$, $n_t > 3$ guarantees the inequality. For

actually analysis, we are safe to assume that $n_t \gg 0$. Therefore, TreeScaper V1.0 is not optimal for trees with leaf set smaller than 64.

Also note that in large set of trees with $n = 100$ and over $20,000$ distinct *regular* bitstrings, collision on the 2-level hash table is observed in TreeScaper V1.0, which violates the uniqueness assumption of $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2)$. The analysis is then not reliable for those trees containing collided bipartitions.

TreeScaper V2.0. Since the issue of collision is observed, TreeScaper V2.0 drops the 2-level hash table and implements a single identificaition mapping with a dynamic-array-based collision beam.

Recall that TreeScaper V2.0 does not guarantee the bitstring being *regular* in rBFS, we also want the identification mapping satisfing the condition

$$\mathcal{I}(s(b)) = \mathcal{I}(\overline{s(b)})$$

where $s(b)$ is a bitstring and $\overline{s(b)}$ is the complementary bitstring.

Such identificaiton is surprisingly simple as any fixed modulo of the the binary number represented by the bitstring, i.e.,

$$\mathcal{I}(b_s) = \left(\sum_{i=1}^{n} b_s[i] * 2^{i-1} \right) \ \texttt{mod} \ N$$

for any given N.

Definition 1.22. The identification mapping of *regular* bistrings in TreeScaper V2.0 is defined as followed.

1. (Initialization)
   (a) Compute the number of maximal number of containers N of some type such that
   $$\mathcal{I}(s_1(b)) = \mathcal{I}(s_2(b)), \forall b.$$
   (b) Compute
   $$\mathcal{I}(s(l_i)) = 2^{i-1} \ \texttt{mod} \ N.$$
   (c) Construct dynamic array `B` that collects distinct *regular* bitstrings and send the *regular* bitstring $s(l_i), i = 1, \cdots, n$ to `B`.
   (d) Construct a dynamic array of dynamic arries of `int`, `M`, for every computed $\mathcal{I}(s(l_i))$, construct a new dynamic array of `int` with 1 element, the indices of *regulated* $s(l_i)$ in `B`. Push the dynamic array to `M` with label $\mathcal{I}(s(l_i))$.
2. Perform rBFS, for every node v arrived, let $\{v_i\}_{i=1}^{k-1}$ be $k-1$ out of k neighbors of v that have been arrived before, compute
   $$\mathcal{I}(v) = \left(\sum_{i=1}^{k-1} \mathcal{I}_1(v_i) \right) \ \texttt{mod} \ N.$$
   (a) If dynamic array labelled with $\mathcal{I}(v)$ exists in `M`. Perform bitstring comparision directly between the computed bitstring and the records in the collision beam.
      i. If no matched bitstring is found, collision happened. Push the *regulated* new bitstring to `B` and push the new reference to the new bitstring to the collision beam.

ii. Otherwise, do nothing.

(b) Otherwise, push the new bitstring to `B`. Create a new dynamic array of reference labelled with $\mathcal{I}(v)$ with the reference of the new bistring. Push the new dynamic array to `M`.

Note that during the computation, there is also records of bitstring references and bitstring idenficaitions attached to the tree being maintained. The reference to all bipartitions on `B` and their identifications $\mathcal{I}(b)$ are recorded.

According to the discussion we had before, the record of bipartition (references) completely represent the tree.

Note that the complexity of computing $\mathcal{I}$ drops by a havle comparing to TreeScaper V1.0. More importantly, N can operate on $16, 32, 64$ bits space, which makes the identificaition benificial for the case $n > 16$ when not much distinct bipartitions are expected comparing to the case TreeSacper V1.0, which is never benificial until $n > 64$.

Another improvement of TreeScaper is the dynamic-array-base collision beam in which comparision is done by direct bitstring comparision. Although this may be slower than the comparision on the second level of the hash table, it is guaranteed to avoid collision completely. The issue of slow comparision within collision beam can be improved by choosing N large enough so that every collision beam is not too large.

References