2010

SearchTree: Mining robust phylogenetic trees

Akshay Deepak
Iowa State University

SearchTree: Mining robust phylogenetic trees

by

Akshay Deepak

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
David Fernández-Baca, Major Professor
Oliver Eulenstein
Xiaoqiu Huang

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I take this opportunity to thank all those who have made this research successful through their help and support, both directly and indirectly. First and foremost I want to thank my advisor Dr.David Fernández-Baca for being a very understanding guide with a patient ear and an excellent mentor. Not only he has provided immeasurable support towards my research but has always been available for discussions and brain storming sessions in spite of his busy schedule. He taught me how to do research and continues to do so. I feel very fortunate to work with such a professional and a balanced personality like him. I also thank Dr. Ovlier Eulenstien, with whom I have been working for the past couple of semesters and I can't help getting infected by his enthusiasm for research.

I also thank Dr. Mike Sanderson for providing the motivation for this work and also being the collaborator. Working with him has been a very enriching experience. Being a biologist, his knowledge about computer science and his technical skills are commendable. This makes communicating with him a lot more meaningful and efficient. Deployment of **SearchTree** at Redwood cluster (an advanced hardware configuration at his lab) improved the performance by almost 100%. I also thank him for allowing his literature to be extensively quoted in sections 1.1 on "*Phylogenetic Trees for Comparative Biology*" and 1.2 on "*Tree Pre-Computation*".

I also thank Dr. Xiaoqiu Huang for agreeing to be on my POS committee, despite his busy schedule, and for his inputs on my research work. Special thanks to Linda Dutton for always being there and making sure that all the paper work was completed in time so that this defense could go smoothly. I thank National Science foundation (grant: DEB-0829674) and the Computer Science department at Iowa State University for providing the financial support for this research. There are a lot of other people who have helped me settle down at ISU and continue to provide me support in different ways during my stay here. I will always remain indebted towards them.

ABSTRACT

Phylogenetic trees providing high quality information and at the same time covering large number of species are essential for comparative biology. It is a widely accepted fact that with the currently available resources we are far from assembling one completely sampled phylogenetic tree for all life (or one based on a very large subset of species), hence a need for an interim solution arises. Here we describe **SearchTree**, a software tool that allows users to query efficiently on an arbitrary user taxon list and returns high scoring matches from approximately one billion phylogenetic trees being constructed from molecular sequence data in GenBank. The core of **SearchTree** has two parts. The first is a pre-computed collection of phylogenetic species trees from GenBank sequence data consisting of approximately 10,000,000 data sets with 100 bootstrap trees for each set for a total of around 1 billion trees. The goal here is to ensure high 'coverage' (i.e., each taxon occurring in many trees). The second part is the search-retrieval process. The goal is to quickly retrieve the clusters and the subsequent trees from the large data set described above, maximizing the scoring function for the resultant set of trees and all the while keeping computational resources within a limit. Both parts were dealt separately due to their complexity; here we focus on the second part.

The complete pre-computed data set of phylogenetic trees will be around 500 GB. Fast response times are achieved by **SearchTree** through a combination of techniques from information retrieval, notably inverted indexing, and from computational phylogenetics, especially for constructing consensus trees. The use of Redwood cluster, an advanced hardware configuration specifically tuned for this kind of work, has further improved the query times by 100%.

CHAPTER 1. INTRODUCTION

The Tree of life depicts evolutionary relationship between all forms of life- that exist currently or ever existed on this Earth. Though construction of such a tree in its entirety is still not possible with the currently available information and computational resources, our understanding about it has advanced rapidly in the past decade thanks to exponential progress in fields of Genomics and Information Technology. Better knowledge of phylogenetic relationships amongst life on earth will contribute in improving human health, provide impetus to comparative developmental biology, help in saving agriculture and forestry from invasive species and pests and aid in efficient management of natural resources. These are just a few of numerous benefits we expect from phylogenetic trees (AToL. [1]). The current work is an effort in this direction to provide better computational tools for assembling robust phylogenies for meaningful subsets of the tree of life. In the following, we first motivate the problem, and review the approach proposed by Sanderson and McMahon (personal communication, 2008) to this problem. We then describe our contribution.

1.1 Phylogenetic Trees for Comparative Biology

Phylogenetic trees built largely from molecular sequence data have dramatically altered the toolkit available to comparative biologists. Reconstruction of ancestral traits (Blanchette et al. [15]), estimation of divergence times (Renner. [16]), co-diversification (McKenna and Farrell. [17]), and the reconstruction of contact histories in epidemics (Gilbert et al. [18]) are just a few examples of inferences that have been improved measurably by an infusion of phylogenetic information.

The demand for phylogenetic trees has been so high that comparative biologists themselves have turned to heuristic or even non-algorithmic methods for assembling trees comprehensive enough to contain the taxa in which they are interested. For example, the Phylomatic Project (Webb and Donoghue. [19]; http://www.phylodiversity.net) effectively provides a low level phylogeny of seed plants by combining a community-consensus view of ordinal level relationships (APWeb: Stevens. [20]) grafted to numerous lower level trees. Another example was the recent publication of a nearly complete supertree of 4500 mammal species, which challenged the conventional picture of rapid mammalian diversification associated with the K-T mass extinctions (Bininda-Emonds et al. [21]). It was also constructed by manually combining a deep phylogeny with numerous shallow ones. In this case, many of the components were built algorithmically with supertree methods, as was the deep phylogeny, but the authors felt that a global supertree at this scale was "computationally infeasible" and the final comprehensive tree was assembled manually. These and similar examples may reflect a growing disconnect between users' demands and phylogeneticists' ability to provide answers at a level of rigor each finds acceptable. If true, one measure of success of the "Assembling the Tree of Life" [1] enterprise might well prove to be how much it fosters rigorous comparative biology by workers who are not themselves the primary architects of these large-scale phylogenies.

To meet the accelerating demands of comparative biology, one fundamental obstacle must be overcome:

> *Mismatches between the taxa present in available phylogenetic trees and the list of taxa for which comparative data are available limit the size of data sets and the statistical power of comparative analyses based on them.*

Taxon mismatch is a consequence of many things. One is incompleteness of trees at the species level. This arises not only from an overall shortage of data across the tree of life - after all, only 10% of the 1.7 million described species have even a single sequence in GenBank -but also because even in the most intensive analyses of particular taxonomic groups of non-trivial diversity, the logistics of biodiversity sampling usually preclude inclusion of *all* species (Bininda-Emonds et al. [21]'s mammal phylogeny being a rare exception).

This thesis is an attempt to confront this basic mismatch problem by developing algorithms and software to facilitate rapid dissemination of robust phylogenetic trees tailored to specific needs of users in comparative biology. The tool we develop, **SearchTree**, offers an intermediate solution to the global needs of phylogenetic comparative biology. We say *intermediate*, because the ultimate solution, a well resolved and supported tree of enough species to satisfy most potential users, is arguably not yet feasible using existing data. Previous empirical work has explored pushing the limits on assembly (e.g.,Driskell et al. [22]), dealt with data fragmentation (e.g.,Sanderson et al. [23]), and the complexities of gene duplication (e.g.,Chang et al. [26]). These and similar studies at this scale (e.g., Moncalvo et al. [27]) stretch the current technological limits to accurate large tree construction in several respects, leading many workers to impose prior constraints on the analysis to make headway (e.g., Hibbett et al. [28]) or use sequential algorithms that add taxa to already constructed large trees (e.g., Ley et al. [29]). Additional work at this scale will be needed to make headway at larger scales. This suggests to use a compartmentalized approach to broad scale tree construction, in which trees based on GenBank sequence data are constructed within large phylogenetic neighborhoods, and queries are then restricted to those neighborhoods.

1.2 Tree Pre-Computation and Bootstrapping

In this section we describe the approach used by Sanderson and McMahon to construct the phylogenetic tree database used by **SearchTree**. The use of pronoun '*they*' and '*their*' in this section refers to 'Sanderson and McMahon' and 'Sanderson's and McMahon's ' respectively.

Selection of major clades

Construction of a robust complete phylogenetic tree with all 180,000 taxa in GenBank is not currently feasible. Even scaling phylogenetics to several thousand taxa runs into hard computational and methodological problems. Hence, analysis was conducted by compartmentalizing the eukaryotic tree into clades. The NCBI taxonomy (Sayers et al. [31]; Benson et al. [32]) was used as the basic reference for partitioning into clades of manageable size.

However, the NCBI tree itself is perhaps not the ideal estimate of the phylogeny of eukaryotes. They therefore also undertook a partition of eukaryotes by combining information from the TOL web project tree (Maddison et al. [2]) and the NCBI tree. One is more complete; the other more highly curated by a broader phylogenetics community.

Sequence clustering

The standard strategy, to assemble sets of homologous sequences (clusters) from a database (or set of genomes, for that matter) of all-against-all BLAST searches was used. PhyLOTA (Sanderson et al. [10]) uses this for clustering sequences, however in the PhyLoTA Browser clusters were constructed in the context of the NCBI taxonomy tree for convenience of display (thus child clusters were contained within parent clusters, following the NCBI hierarchy). Here they used true agglomerative hierarchical clustering (AHC: Day and Edelsbrunner. [33]) based on the BLAST estimates of sequence dissimilarity rather than the NCBI tree.

AHC techniques were widely used in numerical taxonomy to construct phenograms, and have resurfaced in bioinformatics as a tool for assembling sequence clusters to identify putatively homologous sequences for subsequent alignment. Although this procedure also generates a collection of parent and child clusters (in a fashion less influenced by the NCBI tree), it does not by itself generate the two+ orders of magnitude increase in the number of clusters they are looking for. To do that, they take advantage of the sensitivity of cluster set construction (usually seen as a disadvantage), and build multiple cluster sets using different combinations of clustering algorithms (e.g. simple vs. complete linkage) and BLAST search parameter values. Every group in every dendrogram is then a cluster of sequences, together forming a very large pool of potential data sets to build trees of different sizes, properties, and taxon membership . This protocol avoids customary ad hoc decisions about parameter values, and lets the resulting phylogenetic trees simply speak for themselves.

Alignment and gene tree construction

Two multiple sequence alignments based on different algorithms were constructed for each sequence cluster. They are using Clustal W (Thompson et al. [34]) and Muscle (Edgar, R. C.

[35]) which are two workhorse programs that they believe can handle the diversity of cluster sizes and compositions, yet use different algorithms. For estimation of alignment quality, they use the two alignments with standard metrics (Thompson et al. [36]). Gene trees for each alignment are constructed using maximum parsimony approach.

Species tree assembly

Orthology detection: At this point in the informatics pipeline, a large collection of partially overlapping gene trees have been generated within each of the major eukaryotic clades. However, comparative biologists are interested in species trees, and many of these gene trees will contain gene duplications in protein families, or patterns that mimic duplications in single-copy loci such as lineage sorting, introgression, and (at a nontrivial rate) mistaken taxon identification. Although construction of species trees based on these kinds of non-orthologous or conflicting gene trees is an area of active research in phylogenetics the most conservative and widely adopted strategy is still to test and exclude gene trees if they contain paralogous sequences. They use a modification of a phylogenetic test for orthology (Sanderson et al. [24]) applicable to data sets in which there are multiple sequences per species. This method tests whether a tree constraining all sequences from the same species to be a clade is statistically worse than the unconstrained tree (signaling duplication, deep coalescence, or mistaken taxon name annotation). Experience suggests (Sanderson et al. [23]; Driskell et al. [22]; McMahon and Sanderson. [25]) the test is too conservative, sometimes rejecting perfectly fine large organellar data sets, for example, because of a single incorrectly identified sequence. To preclude this they modify their test along the lines of McMahon and Sanderson. [25] to pinpoint which taxa generate discordance and assay whether it is most likely to be a “mistake” or a true gene duplication confined to one or two taxa.

Species trees from individual gene trees

After exclusion of data sets with apparent duplication, species trees are inferred directly from the constrained-search gene trees described above. Species trees are reported without multiple accessions per terminal taxon. If NCBI recognizes subspecific taxa, these are the terminals reported on the tree.

Species trees constructed from supermatrices

To leverage both the amount of sequence data and its taxonomic breadth, and to generate a large pool of novel species trees, they construct a large collection of supermatrices from clusters that pass their orthology tests, adding the species trees derived from them to the repository, along with those species trees constructed from single clusters. Again, the general design principle is to provide a very large pool of trees with many combinations of taxa and loci, and let search and retrieval algorithms return the most useful of these to the user. It only needs to ensured that any single tree returned to the user must not have been built using redundant sequence data.

The confidence sets of trees

A “data set” consists of either an alignment for a cluster that passes the orthology test, or a concatenated supermatrix alignment of several such clusters assembled algorithmically as described above. For each data set, 100 trees in its confidence set are constructed using bootstrap estimates (Felsenstein, J. [37]) and are stored in the repository.

Our contribution

Having generated the trees, the next step is search and retrieval of trees with a desired characteristic of high overlap with the query taxa and a high value of scoring function of the resultant trees. All this being done within an acceptable time. This is where **SearchTree** comes into the picture and from here onwards our contribution in the project is discussed.

1.3 Search and Retrieval

Query processing is a two step process. In the first step, the user supplies with taxon ids as input. These map to sequence ids, which in turn point to the clusters which contain them. Clusters have trees based with sequence ids as their leaf set. Once the clusters -having significant overlap with the query taxon id set, are determined, the next step begins. This step involves, for each of the clusters from the first step, finding a *majority rule tree* (MRT) (Margush and McMorris. [8]) as a consensus tree representing the overlap of the cluster with the query taxon set.

Inverted file indexing [5] was used in the first step to fetch the clusters which had one or more number of taxa from the query set. There are around 3.5 million taxa occurring in 10 million clusters. Each taxa on an overage occurring in around 200 clusters. Initial attempt at keeping this large mapping between sequences and clusters on hard disk caused the first step consuming un-acceptable amount of time. Partially responsible for this was also the fact that both taxon ids and sequence ids needed to be searched in the index, which required a minimum of $O(\log n)$ time. Considering the number of sequence ids, this was indeed a lot of time for the first step itself. Hence, we decided to use in-memory inversion [5] technique for index construction and used direct addressing (3.2) which resulted in a "Look-up table" approach to fetch the sequence ids. This resulted in the minimum size of the index, allowing to bring it in memory during query processing, and the direct addressing allowed a sequence id to be searched in constant time removing the $O(\log n)$ factor. Though index construction time was increased and it left us with rebuilding the index from scratch every time the index was updated. Considering the fact that the index is expected to be updated only once in six months, with the release of a new version of GenBank, this change was acceptable. Chapter 4 "Journey from 45 seconds to under .45 seconds" presents an interesting account for this and also explains usage of direct addressing for clusters too.

The next step is retrieval and processing of the clusters and summarizing the final result as a MRT. The greatest challenge was dealing with the huge data set arising from one billion trees. In terms of sheer space, this meant 500 GB of trees. With current hardware, it is not possible to keep more than a very small percentage of this data in memory. This meant that trees had to be swapped in and out of memory from hard disk as required. Though tree compression was used to some extent, this remains in the final implementation as the most time consuming part, consuming more than 90% of the query process time. The other aspect was processing the clusters for the corresponding subtrees and finding the MRT quickly. The aim was to keep the whole process linear as even for moderate sized queries, around hundred thousand trees are processed. Again direct addressing was used to keep the algorithm for finding the subtree linear. To compute majority-rule consensus trees, a modified form of the linear time random algorithm given by Amenta et al. [3] was used.

1.4 Organization of the Thesis

From here onwards, the thesis is organized as follows: next is Chapter 2 which discusses the details of the query process. Chapter 3 goes through definitions and preliminaries helpful in understanding the rest of the thesis. Chapter 4 deals with the first part of the query process -accumulating the cluster ids having a minimum overlap with the query sequence set. Chapter 5 discusses the second part of the query process -generation of MRT. Going further, Chapter 6 has the experimental results and implementation details. Finally Chapter 7 addresses future work.

Appendix A describes the command line options for the tool while Appendix B lists out the limitations the current release of **SearchTree** imposes on the user query.

CHAPTER 2. PROBLEM DEFINITION AND THE QUERY PROCESS

Our objective is to develop algorithms, software, and a web accessible tree repository modeled on PhyLOTA Browser (Sanderson et al. [10]) that will quickly return a set of high quality phylogenetic trees constructed from sequence data in GenBank, having maximal match with the list of taxon names provided by the user. Currently **SearchTree** supports all the central functionalities. However the web accessible interface remains to be added. Step wise, this can be seen as:

1. Pre-computation of a very large collection of phylogenetic trees from Genbank sequence data with the aim of having high 'coverage' (i.e. each taxon occurring in many trees). Trees will be in the order of 1 billion ($\sim 10,000,000$ data sets $*100$ bootstrap trees). Currently we are quarter way through in reaching this goal of 1 billion. Data sets are constructed from individual sequence clusters and supermatrices assembled using algorithms designed to control missing data.

2. Develop algorithms and software tools to search and retrieve the subtrees across this collection that maximizes a score that reflects both tree quality and taxon matching with the query. This is where **SearchTree** comes into the picture. The scoring function capturing these two entities and the motivation behind the choice of the scoring function is discussed in the next section.

3. Deliver results in a robust web accessible repository building on schema in the PhyLOTA Browser (Sanderson et al. [10]). Currently **SearchTree** functions as a standalone tool and would be interfaced to the online community through a web service interface once the complete collection of trees has been generated. Though the computational resources

required to run **SearchTree** in terms of processor speed and RAM requirements are well within the limits of a desktop machine, however the space needed to store the phylogenetic trees on a hard drive limit its use as a standalone. It is here that its interface as a web service becomes very useful.

2.1 Problem Definition

The goal of **SearchTree** can be stated as following:

Input: A list of taxonomic names of species Q (the query set), and a user selected parameter α, which indicates the user's desired weighting of the relative importance of tree quality vs. taxon overlap.

Output: A ranked set of high scoring phylogenetic trees, from the collection D (the data base consisting of large collection of pre-computed phylogenetic trees) having an overlap with the query set Q, and *summarized* so that the resultant trees are only based on taxon from Q. Let X be the set of taxa shared between a tree T in D and the query set Q. The score of T with respect to Q is the weighted sum of quality of the subtree of T induced by X, $c(T, X)$ and the amount of overlap of T with Q i.e. $|X|$, the number of taxa shared in common between T and Q,

$$S(Q, A) = \alpha * c(T, X) + |X|$$

The trees are summarized per cluster. Each cluster has trees based on the same set of sequence data. The summarized tree is a majority rule tree for the set of subtrees based on the overlap X of the query set Q and the cluster. The quality of the summarized tree T induced by X, is the support number of T. Detailed discussion of how a representative tree is generated for the given overlap X and calculation of the support number is dealt in Chapter 5. The motivation behind the scoring function should be obvious by now; which is to balance between the two quality measures of the query. The amount of overlap with the query set Q i.e. how many taxa a tree is able to relate with the query and quality of the tree denoted by the support number which quantifies the confidence one can have in the relation depicted by the tree based on the support it got from the constituent trees in the cluster. And finally, the balance between the two quality measures is in the hands of the user which specifies the value α. Currently **SearchTree** reports both the measures separately and the task of incorporating α has not been added to its current release. The symbols involved in the problem are given a more formal treatment later when the query process is described.

Considering a practical solution to this problem, the following two limitations must be realized:

- It is not yet feasible to build a complete phylogeny for all 180,000 taxa in GenBank.
- It is not yet computationally feasible to build trees on demand for the user, based on an arbitrary input list.

Both limitations stem from the computational complexity or NP-completeness of nearly all the problems involved in assembling the aligning sequences, and constructing phylogenetic trees. To address the first limitation, we restricted our attention to the problems within clades from a set of approximately 1000 major clades of eukaryotes. To address the second limitation, we pre-compute a very large collection of alignments, phylogenetic trees, and tree confidence sets ($\sim 10^7$ alignments sets $*100$ trees in a confidence set $\simeq 10^9$ trees) based on multiple partitions of the sequence data and data combination into super matrices. The task is then to retrieve good trees representing good matches to the user's input list. This s done by **SearchTree**.

2.2 The Query Process

We shall discuss the query process in terms of notations which in turn represent the entities involved in the query process. These will also be used in the further chapters while going into details of different parts of the query process.

Taxon Id [$TaxonId$]: Represents a species. User queries are based on taxon ids. the taxon id set has a one-to-many mapping to the sequence id set (defined next). Currently there are around 250,000 taxa in the database.

Query Taxon set [QT]: Represents the set of query taxa as input by the user for a query.

Sequence Id [$SeqId$]: Represents a DNA sequence in GenBank. Each taxon id is mapped to one or more sequence ids. Sequences in turn form the leaf set on which clusters are built. This is the link between a taxon id and a cluster for determining the query overlap. The trees are grouped as clusters (discussed next). Currently there are around 4 million sequence ids.

Query Sequence set [QS]: Contains all the sequence ids to which any of the taxa in the query taxon set are mapped to.

Cluster Id [$ClusId$] and Tree Id [$TreeId\_ClusId$]: A cluster represents a sequence data set form GenBank. Each data set is bootstrapped to form a confidence set of 100 (or fewer) phylogenetic species trees. This group of bootstrapped trees forms a cluster. When complete, there will be around 10 million clusters. In the query process, phylogenetic trees are referred only through the corresponding clusters. Hence a tree id has two components— the cluster to which it belongs to and its position in the cluster amongst the trees in the cluster.

Query Overlap with a Cluster [QO_{ClusId}]: Represents the overlap of the query taxon set with the cluster $ClusId$; i.e all those taxa in the QT, that are mapped to at least one sequence in the leaf set of the cluster $ClusId$.

Subtree Cluster [Sub_{ClusId}]: For each tree in the cluster $ClusId$, a subtree is generated on the leaf set corresponding to the query overlap QO_ClusId. Coming from the same cluster, these subtrees form the subtree cluster.

Majority Rule Tree for the subtree cluster [MRT_ClusId]: Each Subtree Cluster is summarized by the corresponding majority rule tree for all the generated subtrees. It is this MRT which is reported to the user as the final result.

Having defined above the query process can be summarized in Figure 2.1.

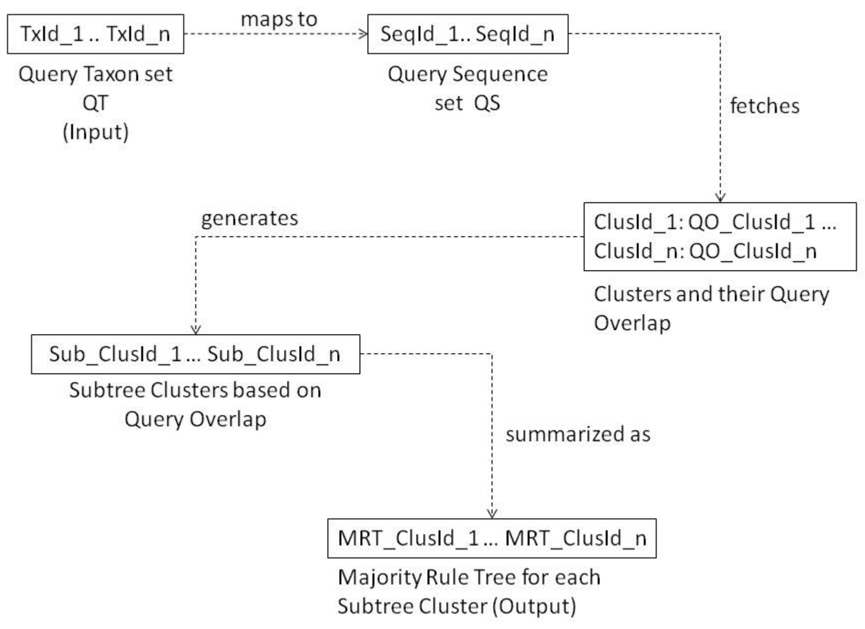


Figure 2.1 Query Process - A bird's eye view

CHAPTER 3. DEFINITIONS AND PRELIMINARIES

Here we introduce some concepts and techniques used in Chapters 4 and 5. Sections 3.1 - 3.3 describe the main ideas used for search and retrieval, while section 3.4 explains majority-rule consensus trees (MRTs).

3.1 Hash Table

A Hash table is a data structure that uses hash functions to efficiently map an input value to an associated numeric key. This key is used to index into an array (or a similar container) where the input value is stored. Hash functions are needed because the input value is not suitable, as it is, to be used as an index into the array. A hash function should be efficiently computable to find the index, of an input value, into the array. At times, it is possible that two different input values get assigned the same index or fall in the same bin in the array. This is called *collision*. One of the most commonly used way out of collision is *chaining* where in the same bin (the same slot in the array container), multiple values are stored in a chained fashion like linked lists. The design of the hash function guarantees that the probability of a collision is small, and hence allows indexing into the array container to be done in constant time. Universal hash functions [4] guarantee same performance for any random set of input values. These functions have been used to index taxon and sequence ids.

3.2 Direct Addressing

Direct addressing can be seen as a version of hashing where the input value itself is used to index into the associated array. Direct addressing is quite simple but it is used extensively in the current implementation to keep the query process linear in complexity and hence it is relevant to explain in the context of its current use. Given the varied form of the input data, directly enumerating it more than often requires the size of associated array container to be unacceptably large. To overcome this, in our implementation, the identifiers for the input data were replaced by sequentially generated numerical values. This ensured that the size of the array remains minimum and at the same time hashing is fastest. Such a replacement is only possible if the input is supplied in the sequentially generated form. The flow of the query process (Figure 2.1), allowed the input to be pre-processed in sequential form. However, given the sequentially generated value, it is easy to index back to the original value for reporting the output. This direct use of the sequentially generated values as input and then quickly mapping back to the original value for output has been exploited extensively to attain fast query processing. An alternative could have been some sort of hashing in 'constant' time, but considering the sheer numbers of input data, this 'constant' time was the difference between fraction of a second and few seconds which was significant when the total query processing time is considered. Figure 3.1 contrasts hashing and direct addressing.

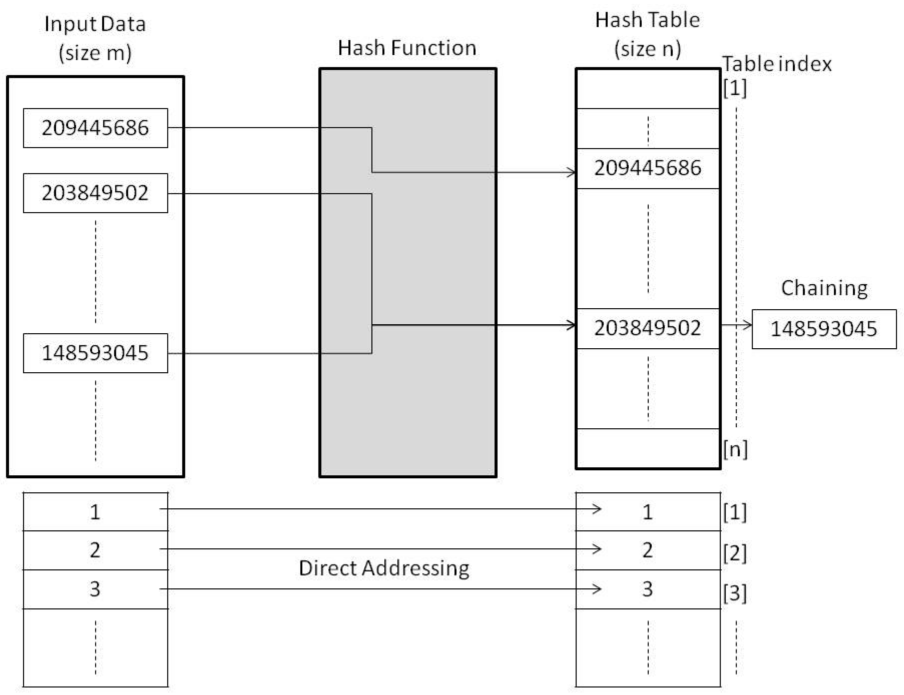


Figure 3.1 Hashing and Direct Addressing

3.3 Inverted Index

Indexing is used whenever there is a need to go through a large amount of data for sections containing certain specific terms called keywords. In the context of simple text search, there is a collection of documents, each having certain set of words and the query is to find collection of documents containing one or more words from the query set. Hence, to retrieve these documents efficiently, they are *indexed* based on the expected set of query words or keywords, contained in any of the documents. An index is a data structure that maps keywords to documents that contain them. In our case there are 10 million clusters each based on a unique set of sequences. During query processing, we are looking for clusters containing certain sequences as specified in the query. A good survey of different types of indexing techniques along with their construction and update strategies can be found in Zobel et al. [5], for our implementation we have used an inverted index. An inverted index consists of following two components:

Vocabulary or the search structure

This files stores for each keyword t, a count f_t of documents containing word t and a pointer in the inverted list (to be discussed next) for this word. For current implementation, sequences corresponds to words and documents corresponds to clusters.

Inverted file or Inverted list

This is a list that has an entry for each of the keywords. At each entry it stores pointers or identifiers to all the documents that contain the word corresponding to this entry. For sequences this means that there is an entry for each sequence and that the corresponding entry has all the clusters that contain the sequence in their leaf set. Figure 3.2 shows indexing between sequences and the corresponding clusters.

In the figure, it can be seen that f_t, the frequency count value was shifted from the vocabulary to the inverted list as it made the vocabulary more manageable. For index construction the in-memory inversion ([5]) technique was used. Though its construction is the most time

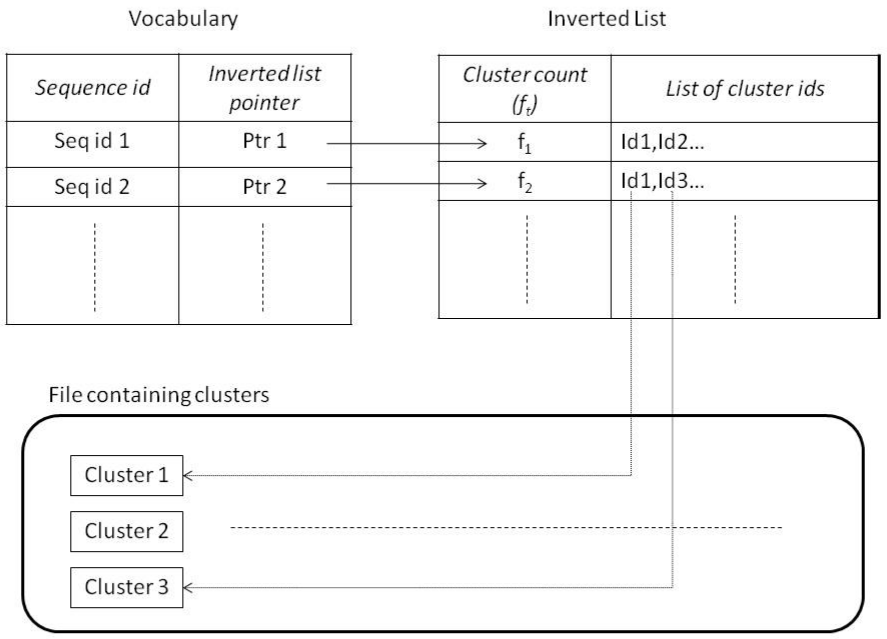


Figure 3.2 Inverted Indexing

consuming, it has the least memory footprint. The intention was to keep the index in memory instead of hard disk. This was done for the obvious speed advantage, but also because with the large number of clusters, keeping memory to minimum became most important. The algorithm makes two passes over the collection of documents (here clusters). In the first pass, the frequency value f_t is calculated for each of the keywords. Space is then allocated in the inverted list for each keyword based on the corresponding f_t value. This ensures that the size of inverted list remains small. In the second pass, the actual index is created. The finer details of the index building algorithm through this technique can be found in [5]. After having selected the index construction technique, there was very little choice regarding the index update strategy, except rebuilding it from scratch every time the index needs to be updated. This was because going for the minimum possible memory left no room for flexibility, i.e. to allow future addition of documents in the existing entry for a keyword in the inverted list. Again, this strategy is the most time consuming for obvious reasons, but acceptable as updates are expected to be rare. The sequence database would be updated only with every new release of

GenBank, which is once every six months.

3.4 Consensus Tree and Majority Rule Tree (MRT)

A *consensus tree* is a way of representing a group of input trees through one representative tree called consensus tree. These input trees can be viewed as 'rivals', because each tree in the group is competing with the rest to be represented in the consensus tree. The Adams consensus tree was the first approach towards constructing a consensus tree (E.N. Adams. [6]) and since then a lot of consensus tree methods have become available. Bryant. [7] has written a good survey cum classification of consensus methods of trees and also provides an interesting insight into the relation among them.

Here, we define majority rule tree for the case of rooted trees. This definition can be extended to un-rooted trees by converting un-rooted trees to rooted trees through use of an *outgroup* which is nothing but adding a distinguished taxon to the un-rooted tree to make it rooted . Consensus trees are defined only for the case where the input trees share a common leaf set. Let $L = l_1, l_2, ...l_n$ be the leaf set. Let $T = \{T_1, T_2, ...T_t\}$ be the set of input trees for which a representative consensus tree needs to be found. Each tree has n leaves labeled by leaves in L. Consider a node i in an input tree T_j. The set of all leaves in the subtree rooted at T_j causes a bipartition w.r.t. L. Hence a bipartition partitions the leaf set into two w.r.t. an intermediate node in the tree. Here we represent this bipartition only by the leaf set of the subtree rooted at T_j. Hence when we refer to some bipartition B, we are actually referring to some subset $\{l_1, l_2, l_3...\}$ of the leaf set L. Similarly, in the current context, the cardinality of bipartition B refers to cardinality of this subset. The MRT or the M_l tree is a tree which includes nodes for exactly those bipartitions which occur in more than half of the input trees; more generally in some fraction l of the input trees. Margush and McMorris [8] showed that a M_l tree exists for any $1/2 < l \leq 1$; elsewhere (McMorris et al. [9]) these trees have been referred to as M_l trees. For our problem definition we generate MRT for $l = 1/2$. For $l = 1$, this reduces to strict consensus trees. Figure 3.3 is a simple example for finding MRT $(l = 1/2)$ for an input set of 3 trees.

We define the *support number* of a MRT to be the average percentage of occurrence of its constituent bipartitions, in all the input trees. This support number is also taken as a 'quality measure' of the MRT.

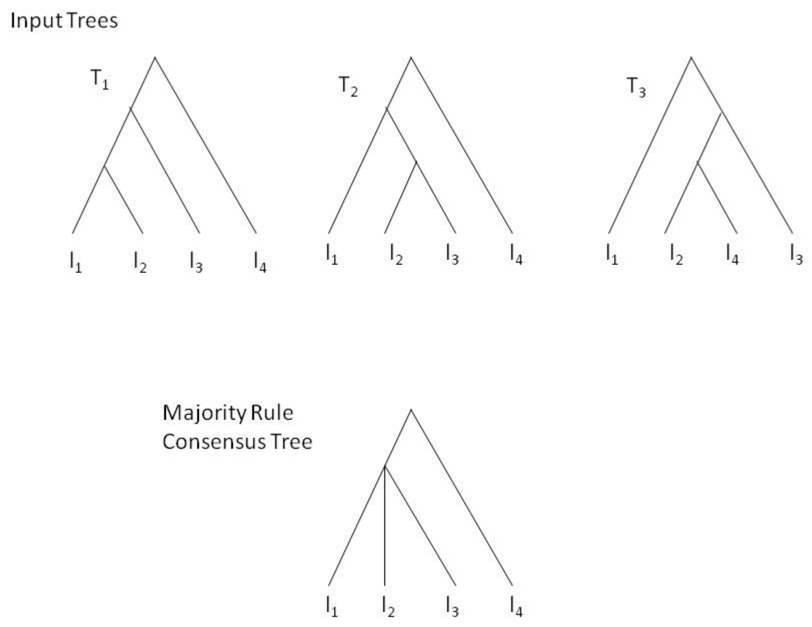


Figure 3.3 Example showing Majority Rule Tree determination

CHAPTER 4. IDENTIFYING THE OVERLAPPING CLUSTERS- 'A JOURNEY FROM 45 SECONDS TO .45 SECONDS'

This chapter deals with the first part of the query processing; i.e. fetching the clusters ids which have a significant overlap with the query taxon set. The sub problem is that given the set of query taxon ids, output the list of cluster ids which have a minimum overlap with the query set. Hashing, direct addressing and inverted indexing were used to significantly speed up the process. To appreciate this, let us first analyze a simple approach to the above problem, the one which will qualify as the first and obvious attempt.

4.1 First shot at the problem

Algorithm is given as:

Algorithm 1 First shot at the problem

```
1: Input: $QT$
2: Output: Clusters having significant overlap $CS$
3: For each $TaxonId$ in the $QT$, list out all the sequence ids $TaxonId$ maps to. This
   gives $QS$.
4: For each $SeqId$ in $QS$, list out all the cluster ids which have $SeqId$ in their leaf set.
   Simply combine the Cluster sets for each of the $SeqId$ and let it be called $CombCS$.
   This combined set might have multiple entries for a cluster if the cluster has more
   than one sequences from the set $QS$.
5: Process $CombCS$, considering each $ClusId$ only once and storing the overlap count
   for each cluster i.e. the number of query taxa it overlaps with. Recall that a cluster
   overlaps with a Taxon if the cluster has at least one Sequence to which the Taxon
   maps to. Let this Cluster set be denoted by $CS$.
6: From the $CS$, remove clusters which have overlap count of less than the minimum
   limit.
7: return $CS$.
```

Let us analyze the above algorithm for its complexity. Let the total number of taxon ids and sequence ids in data base be $|TaxonIds|$ and $|SequenceIds|$ respectively. For step 2, since each sequence is reverse mapped to only one taxon, storing the taxon-sequence mapping in a simple grouped like fashion makes sense where all the sequences belonging to a taxon are stored in one place in a file as a group and then we move onto the next taxon. Then we index each taxon with a pointer to its group in the file. We store this index in a list and retrieve the pointer for a taxon by doing a binary search on the list in $\log(|TaxonIds|)$ time. Thus step 2 takes $O(|QT| \log(|TaxonIds|) + |QS|)$ time. Step 3 is executed in a similar fashion, that is indexing the clusters based on the sequence ids. Hence step 3, takes $O(|QS| \log(|SequenceIds|) + |CombCS|))$. For step 4, one can sort $CombCS$ based on $ClusId$s then linearly go over the sorted list looking for unique $TaxonId$s for each partition based on $ClusId$. This will also require reverse mapping each $ClusId$ to the corresponding $TaxonId$ in the query set which can be done in linear time as part of the earlier steps. Also some sort of a flag would be required to be attached to the $TaxonId$ so that it can be considered only once for each partition of $ClusId$s. All done, this step will take $O((|CombCS|) \log(|CombCS|))$ time mainly because of the sorting time. If we store the $ClusId$s for each $SeqId$ in a sorted form, then the sorting complexity can further reduce to $O((|CombCS|) \log(|QS|))$. Generating $ResCS$ in step 5 looks easy and can be done in linear time $O(|CS|)$. It would not be fair to draw up the overall complexity of the algorithm, because in practice each step has a significant share in the overall time taken and numbers never get so large that one step clearly outweighs the rest in terms of the processing time. Moreover, it will also help in calibrating the speed ups gained as they are applied independently to the steps.

This is pretty much what the first implementation looked like and it took around 45s just to process a large sample query. This sample query had:

- 38 query taxon ids
- 2763 sequence ids
- 314724 cluster ids (size of the combined list of cluster ids from step 3)
- 107900 unique cluster ids ($|CS|$ in step 4)

- 500 valid cluster ids satisfying the minimum overlap criteria ($|ResCS|$ in step 5)

This does not represent what a biologist might query but, for all purposes, comfortably beats the limits that any large query by a biologist might push to in terms of numbers and the resulting complexity. We also used this query to bench mark performance throughout the implementation phase. However, in the results section we used typical queries supplied by biologists which performed better as expected. We know look into the speed ups.

4.2 Speed up - Hashing

In step 2, hashing is used to index a taxon id to the pointer in the mapping file where the corresponding group of sequence ids is stored. Universal hash functions guaranteed constant time performance for any random set of input values. This reduced the complexity from $O(|QT|\log(|TaxonIds|) + |QS|) \rightarrow O(|QT| + |QS|)$. Also, the output -i.e. the sequence ids -were in the form of sequentially generated numerical ids (like $0 \ldots n$) and not the original sequence ids. This replacement of the original ids with the sequential numerical ids was done in the pre-processing stage and is a pre-requisite for direct addressing. The mapping file used here is quite small and hence was kept in memory.

4.3 Inverted Index in memory

Before we move on, it is necessary to discuss how sequences are mapped to the clusters which contained them. A sequence can occur in more than one clusters and a cluster has multiple sequences in its leaf set. This many to many mapping prompted us to use inverted indexing to map sequences to clusters. Considering the large size of the cluster data (500 GB) and aiming at an efficient index maintenance, the initial attempt was to use merged-based inversion for index construction and distribute it using a document-distributed architecture [5]. Extra buffer space was provided for each sequence so that new clusters could be added during maintainence. Though this served the purpose, accessing the index file from hard disk took quite some time and proved to be a bottleneck. So it was thought to bring the index into memory. It was a pleasant surprise that keeping the memory footprint to minimum, it was possible to bring the index for the whole cluster data into memory. Hence we switched to in-memory inversion for index construction, as it guaranteed minimum memory requirements and kept the whole index in one file. Index construction and maintenance time increased but this was not an issue as it would be built only with a new release of GenBank as already discussed before.

4.4 Speed up - Direct Addressing

Once the index is in memory, a $SeqId$ still needs to be indexed into the vocabulary to fetch the pointer in the inverted list. For this we used direct addressing. The pre-processed sequential numerical ids of sequences allowed them to be indexed directly into the vocabulary, built as an array, hence taking time of the order $O(1)$. This reduced the complexity of step 3 from $O(|QS|\log(|SequenceIds|) + |CombCS|)) \to O(|QS| + |CombCS|))$.

In step 4, instead of sorting $CombCS$ and then doing a linear pass, $ClusId$s were grouped per $TaxonId$. Along with it there was a flag array which said whether the corresponding $ClusId$ has already appeared in the list or not and a count array which kept track of the overlap count of this $ClusId$. Direct addressing of $ClusId$ again allowed indexing into this flag array in $O(1)$ time. Also the size of the flag array that equals total number of clusters in the databse $\sim$ 10 million, was well within limits of computational resources. This brought the processing time of step 4 from $O((|CombCS|)\log(|QS|)) \to O((|CombCS|))$. Algorithm 2 describes the speed up for step 4 using direct addressing. In the algorithm, flow enters step 7 only if a $ClusId$ is encountered for the first time for a given $TaxonId$. Flow enters step 10 only if a $ClusId$ is encountered for the first time for any $TaxonId$. the loop in line 13 resets $flagArr$ for the next partition. The loop in line 4 resets $ClusArr$ for the next query.

4.5 Summary

Efficient memory management in the implementation also helped quite a bit in speeding up the implementation. Implementation related details are mentioned in Chapter 6. The speed ups came into existence in an incremental fashion as the implementation matured and it was

Algorithm 2 Speed up using Direct Addressing

```
1:  Input: $CombCS$ partitioned per taxon from the query set,done as part of step 2
    and step 3 in optimal linear time. The pre-initialized cluster count array $countArr$
    and the flag array $flagArr$
2:  Output: Clusters having significant overlap $CS$ ($CS \leftarrow empty$, at the start)
3:  For each cluster partition in $CombCS$, do following
4:    init $list \leftarrow empty$
5:    for each $ClusId$ in this partition, do following
6:      if $flagArr[ClusID]$ is not set
7:        set $flagArr[ClusID]$
8:        increment $countArr[ClusID]$
9:        if $countArr[ClusID] = 1$
10:         add $ClusID$ to $CS$
          end if
11:       add $ClusId$ to $list$
        end if
      end for
12:   for each $ClusId$ in $list$, unset $flagArr[ClusID]$
      end for
    end for
13: for every entry of $ClusId$ in $CS$
14:   attach $countArr[ClusID]$ to the entry
15:   reset $countArr[ClusID] = 0$
    end for
16: return $CS$
```

thrilling to see the final processing time for this first part of query processing to come under .45 seconds. This was very encouraging and important too as it was known that reading clusters from hard disk in the second part of query processing is going to be an inevitable bottleneck and hence it was important to do the rest as fast as possible. Exploration of direct addressing and hashing was also very useful in the second part as we will see in the next chapter.

Figure 4.1 depicts the just described approach in identifying overlapping clusters using hashing and direct addressing as speed ups. Note that each step is linear in complexity which is also optimal.

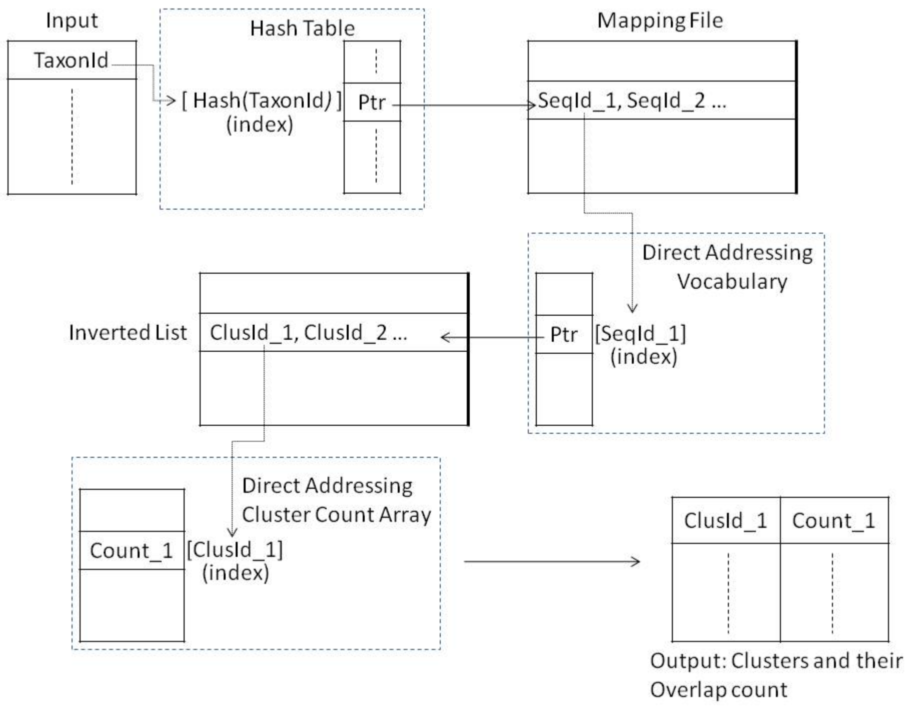


Figure 4.1 Identifying overlapping clusters using hashing and direct addressing

CHAPTER 5. SUMMARIZING CLUSTERS AS MRT

5.1 Introduction

Here we deal with the second part of query processing which is also more time consuming in terms of sheer processing time. This was also the more challenging part and having already developed the infrastructure for hashing and direct addressing, eased things a bit. Input to this part is the cluster set $ResCS$ from the part one of the query process described in previous chapter, and this part involves reading the clusters from hard disk, generating subtrees with respect to the query set QS and finally summarizing the subtree cluster as MRT which is the final output. As mentioned before, the total size of all the clusters ($\sim$ 1 billion) is around 500 GB on hard disk and might only increase, if later it is decided to add more sequences, and hence more clusters in the database. This is very likely considering that Genbank is only bound to get bigger. The time consuming part is reading these clusters from hard disk during the query process. The possibility of keeping this cluster database in memory either whole or in part is ruled out due to current level of hardware advancement and the cost factor. A desktop machine with a latest hardware configuration can be stretched up to 8 GB of RAM (Random Access Memory), -limiting memory available to a process. A hardware configuration for a server can have around 32G GB. Either of these form only a negligible portion of the required 500 GB and hence will not cause any improvement in the performance. Hence there is no alternative to reading the clusters form hard disk. Having fetched the clusters in memory as specified by $ResCS$ from the first part, the next step is to generate subtrees for the trees in each of the cluster based on overlap with the query set QS. We use a simple linear time algorithm to do this which makes use of direct addressing on sequence ids which constitute the leaf set for any tree in a cluster. Each cluster now consists of subtrees with leaf set being a subset of the query set QS. Each cluster is summarized as a consensus tree, which in the current case is the MRT, over the generated subtrees of the cluster. The support number, representing the quality of the MRT, is also calculated in this step. We use a linear time algorithm for generating the MRT. Details of the algorithm are given in subsequent sections.

5.2 Linear time Subtree generation

We use a linear time algorithm to generate subtrees. The algorithm is linear in terms of number of nodes the original tree has. The algorithm builds the subtree from leaves to the root by traversing the original tree in a post-order fashion (children first). An intermediate node in the original tree can have either of the following consequences in the subtree:

1. None of its children exist in the subtree: In this case the intermediate node gets passed as $NULL$ or empty to its parent in the subtree generation process.

2. Only one of its children exist in the subtree: In this case the intermediate node exists as degree two vertex in the tree, which is not a valid phylogenetic tree structure, hence it passes its only child to its parent. This effectively means a contraction of the node in the final subtree [12].

3. At least two children exist in the subtree for this node: In this case this node exists in the subtree and it passes itself to its parent in the subtree generation process.

For the case, when the node is a leaf node, a similar argument holds:

1. The $SeqId$ corresponding to the leaf node exists in the query set QS: this leaf exists in the subtree and hence the leaf passes itself to its parent.

2. The $SeqId$ corresponding to the leaf node does not exist in the QS: this leaf does not exist in the subtree and hence passes itself as NULL to its parent.

The algorithm looks linear as of now except the case for the leaf node, where we need to check if the $SeqId$ corresponding to the leaf exists in the query set QS. To check this, we used direct addressing on sequence ids and the check was carried out in constant time providing the algorithm its linearity. There is a Boolean array *flag* of the size $|SequenceIds|$ which is pre-initialized as

- $flag[SeqId] = true$ if $SeqId$ exists in QS
- $flag[SeqID] = false$ otherwise

This array is initialized in the time $O(|QS|)$ for each query. After a query is processed, it is reset for the use in next query in the same $O(|QS|)$ time. This is done similarly as was shown in section 4.4 while discussing speed up using direct addressing.

An interesting point to bring up here might be to question why did we not think of using some algorithm whose complexity is in terms of size of the overlap with the query set QS rather than the size of the leaf set of the original tree. The point is that anything using the overlap set directly, would require some sort of preprocessing of the original tree from its present bare minimum. At best this pre-processing can be done in linear time w.r.t number of nodes in the tree, for instance by preparing the tree for LCA(Least Common Ancestor) query in linear time [30]. This pre-processing can be done in two ways:

1) After the tree is fetched into the memory: Pre-processing itself would effectively mean same complexity as that of current algorithm to generate the subtree.

2)Store the pre-processed tree in hard disk. This would add linear time complexity in reading the pre-processed part for each tree. Not to forget that even linear time in reading from hard disk is significantly more than any corresponding memory operation.

In Chapter 7, while describing future work, we do propose something like this which is useful if the read trees are to be used again. This is very likely to happen if the user is querying within a set of clades that can be seen as a biological domain for all the queries done by the user. This would mean, to start with, the response time would be slow but as more and more queries would come in, the system would only get faster and faster and if the number of queries are large, then this initial *setup* time can be considered as a constant factor.

5.3 Linear time MRT generation

We use the linear time majority rule tree algorithm given by Amenta et al. [3]. It is a randomized algorithm with expected running time $O(nt)$, where $t =$ number of trees and $n =$ size of the leaf set of each tree. This running time is optimal, since just the reading a set of t trees on n taxa requires $O(nt)$ time. The expectation of the running time is linear over random choices made during the course of the algorithm, independent of the input and thus, on any input, the linearity holds with high probability. We modified this algorithm slightly to remove the uncertainty in the final output. This does increase the time complexity to $O(nt * (k/w))$, where k is the overlap size and w is the size of the machine word. However as we will discuss later that in practice the modified algorithm turns out to be slightly faster for our case.

We will discuss the original algorithm along with the modification. Though the algorithm works for both rooted and un-rooted trees, we discuss the algorithm for rooted trees as all the phylogenetic trees in the cluster database are rooted. We will use the notations for input trees, leaf set and bipartitions as introduced for MRT in Chapter 3. The algorithm runs in two passes. In the first pass we find the majority bipartitions and in the second pass we construct the MRT using these bi-partitions. It ends by checking the output tree for errors due to (very unlikely) bad random choices. We shall not discuss the end part as we have avoided this possibility of error due to the modification.

5.3.1 Finding Majority Bipartitions

In the first pass, each input tree is traversed in post-order(children first), at each node processing the bipartition represented by the subtree at that node. Processing means, counting the number of times a bipartition occurs. The counts are stored in a hash table, which also allows to update the count. The entry in table also contains the cardinality of the bipartition along with the count which is needed later. Storing the bipartitions as bit string representations using direct addressing will require exponential space and will pose a limitation on the required space.

Universal hash functions are used for hashing as they guarantee same performance for any random set of input values. Let the function be called h. In the initialization phase of the algorithm, a prime number m is picked, which is the size of the hash table and a list $a = (a_1, a_2, \ldots, a_n)$ of n random integers in $(0, \ldots, m-1)$. Value of n refers to the the size of the leaf set.

Let $B = (b_1, b_2, \ldots, b_n)$ be the bit-string representation of a bipartition. The universal hash function h, is defined as:

$$h(B) = \sum_{i}^{n} b_i a_i \bmod m$$

Clearly $h(B)$ is always a number in $0, \ldots, m-1$.

Once a bipartition is hashed into a bin in the hash table, it needs to be uniquely identified within a bin in case of a collision. To identify a bipartition uniquely within a bin, in the original algorithm, the authors use a second hash function to hash bipartitions within a bin. They suggest a very large prime number for this hash function to minimize the probability of a collision within a bin. The prime number of the second hash function can be taken to be quite large as there is no hash table being constructed here. The possibility of a collision at the bin level is then inversely proportional to the product of the two prime numbers corresponding to the two hashing functions. This probability can be expected to be really low even for one collision for all bipartitions in all the trees. However, since the worst case possibility of a collision cannot be ruled out, hence they suggest a check for the collision detection, which is again linear. If a collision is detected, we restart the process with a different choice of prime number numbers and random integers a_is.

Instead of using another hash function within a bin, we use the bit representation of the bi-partition itself to store a bi-partition in the bin. This means that every time we want to compare two bi-partitions, we need $O(k)$ operations where k is the size of the overlap of the cluster with the QS. As a worst case possibility, this might add a $O(k)$ factor to the complexity, however in practice the following two things need to be considered:

1)Comparing two bipartitions actually takes $O(k/w)$, where w is size of the machine word. In practice, the overlap is around 100 for a large query and running it on a 64 bit processor (which we currently do), only takes a constant number of steps in the order of (100/64). Yes, it does add a factor of $O(k/w)$ to the linear complexity of the algorithm, as required in storing the bipartition at each node, but again as mentioned before that this is of the order (100/64), which is very small. Moreover, the average size of a cluster (and not the overlap) in the database is around 50. So stretching the overlap of a cluster to 100 is also like a worst case consideration.

2)The possibility of collision within a bin for k=100 with the size of hash table in the current implementation as one million is 1 in 100 per Cluster, which is negligible in practice. Hence, very rarely are we going to need to compare the bi-partitions as required in resolving a collision in the bin.

The advantage of above is that the check to detect a collision at the bin level, as mentioned by authors originally, need not be carried out in our case. Not to mention it also saved implementation time. Also calculation of hash codes for hashing at bin level need not be carried out. Though all this does not change the complexity of the algorithm in terms of the existing linear polynomial order, it does save some time by a constant factor which is helpful in our goal of keeping the query time to a minimum. This makes even more sense in our case, since the check needs to be carried out irrespective of whether a collision occurred or not. Our aim is not to match or better the complexity of the original algorithm (as it is already optimal) but to adapt it to best suit our case.

One nice thing about the hash code in current implementation is that it can be calculated in linear time $O(nt)$. This is possible because the hash code actually needs to be computed only for the leaf nodes; i.e., once for the whole cluster, and then can be recursively calculated for rest of the nodes. This is possible because of following:

Fact 1 *Consider a node with the corresponding bipartition having bit string representation as B. For simplicity, let it have two children with bit string bipartitions as B_L and B_R. Then:*

$$h(B) = (h(B_L) + h(B_R)) \bmod m$$

Fact 1 is true because B_L and B_R represent the disjoint sets of sequences, so that

$$h(B) = (\sum_{B_L} b_i a_i \bmod m) + (\sum_{B_R} b_i a_i \bmod m)$$

where a_i is the prime number assigned to sequence i by the universal hash function.

Fact 1 is used to compute the hash code recursively during the post order traversal. The hash code for each node is stored at the node as it is calculated. The hash code for leaf node is just looked up using direct addressing. Since for any internal node, the hash code for its children has been computed, its hash code is computed in constant time. Even if the tree is not binary, the total time to compute hash codes of all internal nodes using Fact 1 remains linear in the number of nodes.

The cardinality of the bipartition is computed in constant time at each node using a recursion technique similar to the one described above. We now move onto the next step of constructing the majority rule tree.

5.3.2 Constructing the MRT

Once all the counts are present in the table, the next step is to compute the MRT. The counts allow to identify the majority bipartitions that appear in more than $t/2$ (or lt for a more generic approach) trees. Putting the bipartitions together correctly to form the majority rule tree is not totally straightforward. For this, authors use following three more facts.

Fact 2 *The parent of a majority bipartition B in the majority rule tree is the majority bipartition B' of smallest cardinality such that B is a subset of B'.*

Fact 3 *If the majority bipartition B' is an ancestor of a majority bipartition B in an input tree T_j, then B' is an ancestor of B in the majority rule tree.*

Fact 4 *For any majority bipartition B and its parent B' in the majority rule tree, B and B' both appear in some tree T_j in the input set. In T_j, B' is an ancestor of B, although it may not be B's parent.*

Fact 4 holds true because of pigeon-hole principle as both B and B' appear in majority of the trees ($> t/2$), so they must appear in some tree together. A pre-order traversal is done of each of the input trees. While traversing each tree, a pointer is kept to c, which is the last node corresponding to a majority bipartition that is an ancestor of the current node in the traversal. At the start of the traversal of a tree T, c is initialized to be the root, which always corresponds to a majority bipartition. Going further, let C be the bipartition corresponding to c.

At a node i, the stored hash codes are used to find the record for the bipartition B in the hash table. If B is not a majority node, it is ignored. If B is a majority node and a node for B does not yet exist in the output tree, a new node is created for the output tree and with its parent pointer pointing to C. On the other hand, if a node in the output tree does exist for B, its current parent P is considered in the output tree. If the cardinality of P (stored in the hash table entry for P) is greater than the cardinality of C, we switch the parent pointer of the node for B to point to the node for C. When the algorithm run is over, each node B in the output tree, interior or leaf, points to the node of smallest cardinality that was an ancestor in any one of the input trees. Considering, Facts 2, 3, and 4, this implies that the output tree is the correct majority rule consensus tree and we have our MRT ready which is the final output.

Algorithm 3 gives the pseudo-code for the majority tree algorithm by Amenta et al. [3].

Figure 5.1 and Figure 5.2 depict a run of the above algorithm for a simple example .

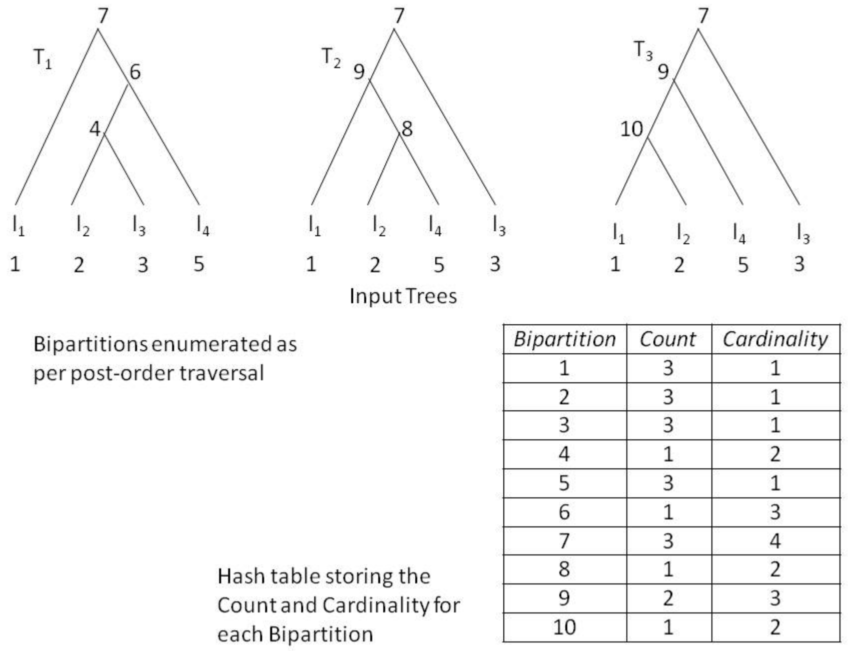
5.3.3 Summary

Summarizing clusters as MRT involves two steps. First the subtrees are generated for each cluster with respect to the query overlap. This is done in linear time using direct addressing. The flow of query process, allowed the data to be preprocessed to facilitate direct addressing. The complexity of subtree generation is linear in the number of nodes of the original tree. Next step was generating the MRT for each processed cluster. For this we used a randomized linear time algorithm by Amenta et al. [3] with a slight modification to remove the randomness at the expense of complexity. It was done because, in our case, the increase in complexity was outweighed by processing time gained on other fronts as already described. Hence the gain was in terms of processing time in practice. The majority rule consensus tree algorithm runs in $O(tn)$ time (ignoring the complexity change due to modification) where t= number of trees in the cluster and n= number of nodes. It does two traversals of the input set, and every time it

Algorithm 3 Pseudo-code for the majority tree algorithm by Amenta et al. [3]

```
1:  **Input:**A set of t trees, $T = T_1, T_2, ..., T_t$.
2:  **Output:**The majority tree, $M_l$, of $T$
3:  Pick prime number $m$ and random integers for the hash function $h$
4:  For each tree $T_i$ in $T$
5:    Traverse each node, $x$, in post order.
6:    Compute hash code $h(x)$ and attach to node $x$
7:    Insert the bipartition corresponding to node $x$ into the hash table
8:    Also compute the cardinality of the bipartition and insert it into the hash table
    end for
9:  For each tree $T_i$ in $T$
10:   Let $c$ point to the root of $T_i$
11:   Traverse each node, $x$, in pre order.
12:   If $x$ is a majority node,
13:     If not existing in $M_l$
14:       Add it to $M_l$, set its parent to $c$.
15:     Else $x$ already exists in $M_l$,
16:       Update its parent if required
        end if
17:     In recursive calls, pass $x$ as $c$.
      end if
    end for
```

visits a node it does a constant number of operations, each of which requires constant expected time. Calculations related to scoring functions were done while generating MRT without any overhead in complexity. Since we store the bipartitions, an alternative $O(m^2)$ algorithm (Felsenstein. [13]) ($m=$ number of majority bipartitions) for MRT construction could be used instead of current $O(nt)$ algorithm, once the majority bipartitions are known. However, to list out the majority bipartitions, one needs a pass of $O(nt)$ anyway and it did not make much sense to use this. Also, we did not want to deviate from the original algorithm much, as if in future we decide to use the original randomized version, it would be a lot easier with the current implementation.


T1
T2
T3
Input Trees
Bipartitions enumerated as per post-order traversal
Hash table storing the Count and Cardinality for each Bipartition

Bipartition	*Count*	*Cardinality*
1	3	1
2	3	1
3	3	1
4	1	2
5	3	1
6	1	3
7	3	4
8	1	2
9	2	3
10	1	2

Figure 5.1 Counting the bipartitions in the input trees

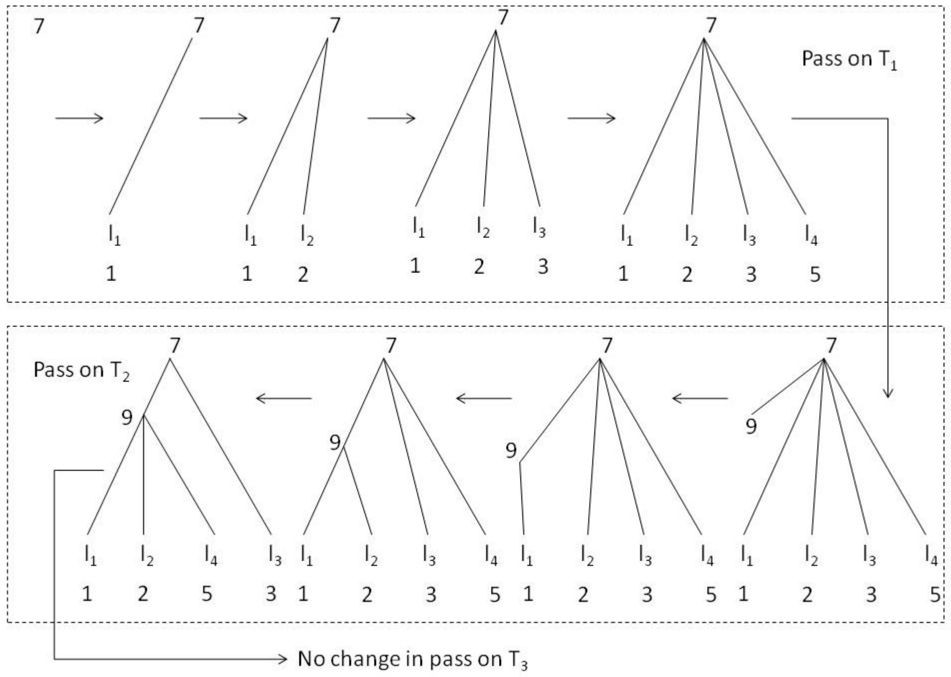


Figure 5.2 Constructing MRT from the majority bipartitions

CHAPTER 6. IMPLEMENTATION DETAILS AND RESULTS

6.1 Implementation

SearchTree was implemented in C++ with a client-server architecture. One reason for this design choice was that it seemed the best way to allow future extensions of our system to allow parallel query processing and session based query processing for each user.

C++ was chosen (over more the portable option JAVA), because speed was our top priority throughout the implementation. To make the implementation fast, we avoided use of container classes, recursion and frequent memory allocation. Though this made parts of the implementation lengthy and complicated, it was worth the effort considering the savings in query processing time. Dynamic memory allocation is a costly operation in any language. Irrespective of the amount of memory allocated, there is a fixed cost every time memory is allocated dynamically [14]. For this reason, we restricted all memory allocation operations to initialization phase.

Currently **SearchTree** runs as a standalone, however the final goal is to make it available to online community through a web service interface. It can run on any recent configuration desktop machine with a hard drive capacity of over 500 GB. As a standalone, the current implementation runs on Solaris. Appendix B explains the command line usage for **SearchTree**.

6.2 Results

We evaluated **SearchTree** for different sizes of query resulting in different number of clusters in the result set. Any consistent analysis w.r.t to the query size or the number of clusters can not be drawn because of the variation in size of the clusters. This in turn affects the query processing times because more than 90% of the time is spent in reading the clusters from the hard disk. The reading time of the clusters in turn depends upon how large the clusters are. However, on an average the system takes around 1.5-2 s for processing every 100 clusters with maximum possible detailed ouput. Apart from this, the in-memory processing takes around a second for every 1000 clusters. This is quite fast considering that every cluster in turn involves processing of 100 trees. Apart from this there is a constant factor of another second to flush the statistics to a file when maximum possible details are enabled. We did test **SearchTree** on query sets provided by biologists, results for two of which we present here. However due to insufficient number of such biologically relevant query sets, a thorough performance analysis has not been done yet. Next we discuss two boilogically relevant query sets.

Query set 1

This set contains legumes of the White mountains in California. Details of the query set are:

- Number of taxa: 31
- Number of sequences: 142
- Number of valid clusters: 62

The above query took 2.15 s averaged over 10 runs.

Query set 2

This set contains legumes of the Huachuca mountains in Arizona. Details of the query set are:

- Number of taxa: 50
- Number of sequences: 232
- Number of valid clusters: 35

the above query took 2.31 s averaged over 10 runs.

Combining above two query sets gives us an estimate of the "species pool" for the broader region of the Western US from which these local floras were sampled. Details of the combined query set are:

- Number of taxa: 81
- Number of sequences: 374
- Number of valid clusters: 98

The above query took 2.58 s averaged over 10 runs.

CHAPTER 7. FUTURE WORK

Once available online, **SearchTree** can prove to be a valuable resource for rapid dissemination of phylogenetic trees across the comparative biological community. It is designed to scale, and can support more trees and clusters in the database with the growing Genbank without change in performance in terms of time. The query processing times have been impressive and this would further encourage its use for the online community.

We have developed a robust infrastructure upon which further expansion can take place for better serving the demand for phylogenetic trees. We would like to make it generic and portable. This would mean that anybody can download **SearchTree** and based on their specific data set, get phylogenetic trees specific to their area of interest. Making it compatible with un-rooted trees might also be useful here. Another aspect also includes to have a central repository of data set, from where anyone can download and upload clusters specific to their own interest.

One major aspect on which we want to work next is reporting the branch length in the generated MRT. These branch lengths would depict the evolutionary distance between species represented by the leaf set of the MRT. This would of great value addition to a biologist mining trees from **SearchTree**. Also something of great value would be incorporating use of taxonomic intelligence tools to handle homonyms and synonyms between GenBank and user queries.

Another improvement would be session based query. It is very likely that a particular user would query for taxa within a domain specific to his interest. This means that there would be certain portion of cluster set that would be frequently required. This cluster set can be cached to avoid reading it again from the hard disk. This can bring drastic change in processing time.

Also, once a cluster is in memory, it can be processed for faster generation of sub trees like LCA based subtree generation. This will have complexity in terms of query overlap size rather than the size of original subtree.

In practice we observed that a lot of clusters fetched provide very little or no information about the overlapping query set. This typically happens when the generated MRT has a star-like structure(s) very close to the root or at the root itself. Unfortunately this only comes to light, after the MRT has been generated. Some sort of scoring scheme for clusters, which indicates such behavior for the query overlap, can avoid the need for reading the whole cluster. Of course, this scoring scheme should be efficient in encoding, so that it does not lead to a comparative overhead in terms of reading clusters from hard disk.

APPENDIX A. COMMAND LINE OPTIONS

We present the command line options for **SearchTree** like a man page found in a UNIX based operating system. I thank Mike Sanderson for drafting this section. This has an additional option for querying based on sequence ids instead of taxon id. We had this option for testing phase only but later thought it might be useful to add it formally.

NAME

SearchTree – query the tree database for overlaps

SYNOPSIS

SearchTree [-f taxonfile] [-L ID1 ID2 ... IDN] [-k number] [-q searchtype] [-t treefile] [-z labelformat] [-s tablefile]

DESCRIPTION

Performs a search of our database of trees based on a query set of taxon IDs. The database consists of blocks of confidence sets of trees (e.g.,100 bootstrap trees per block), where each block is built from an alignment of a sequence cluster (a set of homologous sequences). The program finds the clusters that overlap with this query, determines the reduced majority rule consensus tree induced by the overlapping taxa for each cluster, and reports some statistics on the reduced trees. If no filename is provided and no taxon list is provided, the program accepts input from STDIN in the same format as described for the -f option below.The options are as follows

-f IDfile

Take input list of taxon/sequence IDs from a file. Each row should contain one ID.

-L ID1 ... IDN

Take input as a list of taxon/sequence IDs from the command line.

-k number

Specify the minimum overlap between the query list and retrieved tree (this can be number of taxa, or sequences, Default=3)

-q inputType

T if input should be considered as taxon IDs, S if it should be considered as Sequence IDs ; default is T.

-o overlapType

T if overlap should be considered w.r.t. input Taxon IDs, S if it should be considered w.r.t. either input Sequence IDs or Sequence IDs corresponding to the Taxon IDs as the case may be; default is T.

-t treefile

Writes a treefile in valid NEXUS format, which contains one tree block and one tree description statement for each tree. See below for format details. No statistics are reported in this file.

-z labelformat

Option to select format for taxon labels in treefile: T = taxon IDs; S = sequence IDs; TS = both, using format 'ti###_gi###'

-s tablefile

Writes a file that contains a tab-delimited table with summary statistics for the search. The following columns are reported in order:

1. Node ID identifying the cluster
2. Cluster ID identifying the cluster
3. Number of sequences in the cluster
4. Number of taxa in the cluster
5. Number of overlapping taxon IDs
6. Number of overlapping sequence IDs

9. On the reduced consensus majority rule tree (RCMRT), the average value of support for splits on all non trivial splits above 50%,averaged across only those splits

10. Maximum split support value on the RCMRT

11. Number of splits above 50% (i.e, the number on the majority rule tree) on the RCMRT

COMMENTS

Gene trees with duplications will return tree descriptions with sequences from the same taxon ID. If the output format for tree descriptions is set to report taxon IDs, this will generate tree descriptions with repeated IDs, which causes problems with some (but not all) Nexus parsers.The cumulative number of taxon IDs (column 7, or seq IDs, column 8) discovered in the list will be more useful once we order the list. The value in the last row will be the total union of the sets of taxa found among all hits (or sequences found in column 8).

Notes on Nexus format for tree file:

```
#Nexus
Begin trees;
Tree Tree_NodeID1_ClusterID1 = (ID1,(ID2, ID3));
Tree Tree_NodeID2_ClusterID2 = (ID2, ID3,(ID5,ID6));
....
;
End;
```

Notice the use of NodeID# and ClusterID# to label the trees. Also, note that **SearchTree** does NOT use the 'TRANSLATE' feature of NEXUS. Instead, taxon names (ID numbers in for our case, are embedded directly in the parentheses formatted description. This is technically called the ALTNEXUS format for trees.)

APPENDIX B. LIMITATIONS IN CURRENT IMPLEMENTATION

While querying based on taxon ids, number of taxon ids in the input should be less than or equal to 1000. If the number of Taxon Ids is more than 1000, then only the first 1000 ids are considered for query processing.

Number of valid cluster ids being fed to the second stage of query processing i.e. number of clusters satisfying the minimum overlap criteria with the query set, should be less than or equal to 10,000. If the number is more than 10,000 then only the first 10,000 clusters are considered for the second step of generating MRT. The user is not informed about it currently as there is no logging facility in current implementation of **SearchTree**.

Maximum overlap of a cluster with the query can be 1000. If the overlap is more than 1000 then the cluster is not considered for generating MRT. Again, the user is not informed about this due to absence of logging facility.

BIBLIOGRAPHY

[1] AToL:Assembling the Tree of Life. *http://atol.sdsc.edu/*

[2] Maddison, D. R. and K.-S. Schulz (eds.). 2008. The Tree of Life Web Project.*http://tolweb.org*

[3] Amenta, N., Clarke, F., John, K. S., 2003. A linear-time majority tree algorithm. *In: Proc. 3rd Workshop Algs. in Bioinformatics (WABI03). Vol. 2812 of Lecture Notes in Computer Science. Springer-Verlag,* pp. 216226.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms. MIT Press, Cambridge, MA, second edition, 2001.*

[5] Zobel, J. and A. Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys*38(2):6.

[6] E.N. Adams, Consensus techniques and the comparison of taxonomic trees, *Syst. Zool. 21(1972)*,390-397

[7] Bryant, D. 2003. A classification of consensus methods for phylogenetics. *American Mathematical Society PublicationsDIMACS (Center for Discrete Mathematics and Theoretical ComputerScience), Piscataway, NJ.*

[8] T. Margush and F.R. McMorris. 1981.Consensus n-trees. *Bulletin of Mathematical Biology, 43*:239244.

[9] F.R. McMorris, D.B. Meronk, and D.A. Neumann. A view of some consensus methods for trees.1983. *In Numerical Taxonomy: Proceedings of the NATO Advanced Study Institute on Numerical Taxonomy. Springer-Verlag.*

[10] Sanderson, M. J., D. Boss, D. Chen, K. A. Cranston, and A. Wehe. 2008. The PhyLoTA Browser: processing GenBank for molecular phylogenetics research. *Syst. Biol.*, *57*:335-346.

[11] Garey, M. R., and D. S. Johnson. 1979. Computers and intractability: a guide to the theory of NPcompleteness. *W. H. Freeman, San Francisco.*

[12] David Bryant. Hunting for trees, building trees and comparing trees: theory and method in phylogenetic analysis. *PhD thesis, Dept. of Mathematics, University of Canterbury, 1997.*

[13] J. Felsenstein. Phylip (phylogeny inference package) version *3.6*, 2002.Distributed by the author. Department of Genetics, University of Washington, Seattle. The consensus tree code is in consense.c and is co-authored by Hisashi Horino, Akiko Fuseki, Sean Lamont and Andrew Keeffe.

[14] David Detlefs,Al Dosser amd Benjamin Zorn 1994. Memory Allocation Costs in Large C and C++ Programs. *SOFTWAREPRACTICE AND EXPERIENCE, VOL. 24(6), 527542*

[15] Blanchette, M., E. D. Green, W. Miller, and D. Haussler. 2004. Reconstructing large regions of an ancestral mammalian genome in silico. *Genome Research 14:2412-2423.*

[16] Renner, S. S. 2005. Relaxed molecular clocks for dating historical plant dispersal events. *Trends in Plant Science 10:550-558.*

[17] McKenna, D. D., and B. D. Farrell. 2006. Tropical forests are both evolutionary cradles and museums of leaf beetle diversity. *Proceedings of the National Academy of Sciences of the United States of America 103:10947-10951.*

[18] Gilbert, M. T. P., A. Rambaut, G. Wlasiuk, T. J. Spira, A. E. Pitchenik, and M. Worobey. 2007. The emergence of HIV/AIDS in the Americas and beyond. *Proceedings of the National Academy of Sciences of the United States of America 104:18566-18570.*

[19] Webb, C. O., and M. J. Donoghue. 2005. Phylomatic: tree assembly for applied phylogenetics. *Molecular Ecology Notes 5:181-183.*

[20] Stevens, P. F. 2007. Angiosperm Phylogeny Website, v. 8. *http://www.mobot.org/MOBOT/research/APweb/*

[21] Bininda-Emonds, O. R. P., M. Cardillo, K. E. Jones, R. D. E. MacPhee, R. M. D. Beck, R. Grenyer,S. A. Price, R. A. Vos, J. L. Gittleman, and A. Purvis. 2007. The delayed rise of present-day mammals. *Nature 446:507-512.*

[22] Driskell, A. C., C. An, J. G. Burleigh, M. M. McMahon, B. OMeara, and M. J. Sanderson. 2004. Prospects for building the tree of life from large sequence databases. *Science 306:1172-1174.*

[23] Sanderson, M. J., C. An, O. Eulenstein, D. Fernndez-Baca, J. Kim, M. M. McMahon, and R.Piaggio-Talice. 2007. Fragmentation of large data sets in phylogenetic analysis *in* Reconstructing evolution. *New mathematical and computational advances (O. Gascuel, and M. Steel, eds.).Oxford University Press, Oxford.*

[24] Sanderson, M. J. 2003. Molecular data from 27 proteins do not support a Precambrian origin of land plants. *Amer. J. Bot. 90:954-956.*

[25] McMahon, M. M., and M. J. Sanderson. 2006. Phylogenetic supermatrix analysis of GenBank sequences from 2228 papilionoid legumes. *Syst. Biol. 55:818-836.*

[26] Chang, W.-H., and O. Eulenstein. 2006. Reconciling gene trees with apparent polytomies. *COCOON 2006:235-244.*

[27] Moncalvo, J. M., R. Vilgalys, S. A. Redhead, J. E. Johnson, T. Y. James, M. C. Aime, V. Hofstetter, S. J. W. Verduin, E. Larsson, T. J. Baroni, R. G. Thorn, S. Jacobsson, H. Clemencon, and O. K. Miller. 2002. *One hundred and seventeen clades of euagarics. Molecular Phylogenetics and Evolution 23:357-400.*

[28] Hibbett, D., R. Nilsson, M. Snyder, M. Fonseca, J. Costanzo, and M. Shonfeld. 2005. Automated phylogenetic taxonomy: An example in the homobasidiomycetes (mushroom-forming fungi). *Syst. Biol. 54:660-668.*

[29] Ley, R. E., J. K. Harris, J. Wilcox, J. R. Spear, S. R. Miller, B. M. Bebout, J. A. Maresca, D. A. Bryant, M. L. Sogin, and N. R. Pace. 2006. Unexpected diversity and complexity of the Guerrero Negro hypersaline microbial mat. *Applied and Environmental Microbiology 72:3685-3695.*

[30] Dan Gusfield. Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, 1997.Chapter 8.

[31] Sayers EW, Barrett T, Benson DA, Bryant SH, Canese K, Chetvernin V, Church DM, DiCuccio M, Edgar R, Federhen S, Feolo M, Geer LY, Helmberg W, Kapustin Y, Landsman D, Lipman DJ, Madden TL, Maglott DR, Miller V, Mizrachi I, Ostell J, Pruitt KD, Schuler GD, Sequeira E, Sherry ST, Shumway M, Sirotkin K, Souvorov A, Starchenko G, Tatusova TA, Wagner L, Yaschenko E, Ye J (2009). Database resources of the National Center for Biotechnology Information. Nucleic Acids Res. 2009 Jan;37(Database issue):*D5-15.* Epub 2008 Oct 21.

[32] Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Sayers EW (2009). GenBank. Nucleic Acids Res. 2009 Jan;37(Database issue):*D26-31.* Epub 2008 Oct 21.

[33] Day, W. H. E., and H. Edelsbrunner. 1984. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification 1:7-24*

[34] Thompson, J. D., D. G. Higgins, and T. J. Gibson. 1994. Clustal-W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research 22:4673-4680.*

[35] Edgar, R. C. 2004. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research 32:1792-1797.*

[36] Thompson, J. D., F. Plewniak, and O. Poch. 1999. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research 27:2682-2690.*

[37] Felsenstein, J. 1985a. Confidence limits on phylogenies: An approach using the bootstrap. *Evolution 39:783-791.*