A Fast Compact Indexing Algorithm for Managing Large-Scale Robinson-Foulds Distance Matrices

Beenish Jamil Tiffani L. Williams

bjamil@gmu.edu tlw@cse.tamu.edu

Abstract

Phylogenetic analysis can produce up to hundreds of thousands of trees, with each tree being a hypothesis for the evolutionary relationships between a group of organisms or taxa. The difference between these trees can be found using the Robinson-Foulds (RF) distance metric, a commonly used metric for phylogenetic tree analysis, and then stored in a matrix. However, there are currently no efficient methods of extracting data from large RF matrices, which can take up many gigabytes of disk space, due to the I/O bottlenecks that comes with dealing with such large files. One possible solution is through the use of an index. We have developed an algorithm for producing an index for RF matrices that requires minimal generation time and search time. We tested our algorithm on RF matrices ranging sizes between 3,419 to 135,135 trees and 60 to 567 taxa. We found that the largest matrix we tested (135,135 trees) took less than 14 minutes to generate, a considerably fast speed. Searching our index for queries that required the traversal of the entire matrix took 27% to 99% less time than searching the RF matrix for those same values. The applications of this index show a lot of promise and we hope that they will aid faster phylogenetic analysis.

I. INTRODUCTION

The goal of phylogenetic studies is to find the true relationships between a group of organisms or taxa. These relationships are commonly represented in the form of trees. These trees can have a variety of applications in drug and vaccine development, conservation efforts and much more [5], [3]. However, finding the correct tree for a group of organisms is an NP-hard optimization problem that requires heuristics on the tree space [3]. These heuristics can return up to tens of thousands of possible trees for a single group of organisms. Finding the correct tree from among such a large tree set is not a simple task. One of the most commonly used method for analyzing very large tree sets is constructing consensus, or summary, trees for them. However, some biologically significant data can be lost in the construction of these trees [4]. Another method of analyzing these trees is by calculating the pairwise distances between all of the trees in the tree set using the Robinson Foulds (RF) distance metric and analyzing the trees based on those distances. These matrices store a significant amount of information about the original tree

space and are an excellent source for “understanding the evolutionary relationships” represented in the generated tree sets [3], [6].

The RF matrices are t x t matrices where t is the size of the tree set. The problem arises that as tree sets get larger, these matrices grow quadratically in size and quickly become difficult to manage and search in a time efficient manner. With tree set sizes getting in the tens of thousands of trees and greater, these square matrices are becoming harder to handle and query due to the time requirements of simply reading these matrix files. We proposed a fast indexing algorithm for the RF matrix to help deal with this I/O bottleneck.

The type of searches we were interested in were those that searched for a particular RF distance in the matrix and that required searching either the entire RF matrix or a significant proportion of it. These search types are generally time consuming since they require a significant amount of time for just reading the matrix.

II. BACKGROUND

A. Phylogenetic Trees

These are trees that describe evolutionary relationships between a set of taxa. The leaf nodes are the studied taxa while the internal nodes are the “hypothetical ancestors” [1]. The edges describe the relationships between the various nodes.

Phylogenetic trees can be described in terms of bipartitions. They are obtained by removing a single edge from the tree, dividing the tree into exactly two parts [2]. There can as many bipartitions in a tree as there are edges. This two part representation is important when calculating the RF distances between two trees.

B. Robinson-Foulds Distances

The Robinson-Foulds distance metric describes the degree of dissimilarity between two trees. It can be defined as “the total number of bipartitions that differ between them” [1]. The maximum possible RF distance between two binary trees is the total number of taxa in those trees minus three.

C. Robinson-Foulds Matrices

All of the RF distances between all of the trees in a tree set can be calculated and stored in an RF matrix. These are symmetric, square matrices of size t x t, where t is the total number of trees in the studied tree set. As the tree sets get larger, it gets difficult to give a descriptive label to each tree. Instead, the row and column numbers in the matrix are used as tree identification numbers. Generally, the number of possible RF distances in a tree set is much less than the possible tree pairs (or, cells) in the matrix.

D. FastHashRF

This software is a fast method for calculating the RF matrix for a tree set [1]. It is based on HashRF, which is among the fastest RF matrix calculating software available [6].

E. Indexing

Indexing is essentially a method for reorganizing the data of interest in a data structure for fast future retrieval. It is widely used by large databases.

III. THE COMPACT INDEXING ALGORITHM

We aimed to make our algorithm as fast as possible so that more time could be spent on searching the index rather than on generating it. In order to do this, we tried to use as few steps in our algorithm as possible since we expected these steps to possibly be repeated millions to billions of times during the course of our algorithm's run. We also tried to reduce I/O, a slowing factor in most algorithms, as much as possible for this reason as well. We made a few generalizations and base assumptions about our tree space to help towards this goal:

1. Since a RF matrix is symmetric, only the data in one side of the main diagonal needs to be recorded.
2. The users of the RF matrix are only interested in unique trees. That is to say, if two trees are zero RF distances apart, they are identical. Therefore, information about only one of those trees needs to be stored.
3. Since the number of possible RF values in large matrices is usually much less than the total number of cells in the matrix, the probability that there will be long runs of the same RF distance in a row is high.

The Index:

Our algorithm generates a compact, easy to navigate index for the RF matrix. It creates a separate file for each of the possible RF distances for the tree set. These files store all of the tree pairs that are that RF distance apart from each other. Only one value, or position ID, represents each tree pair. It is obtained by flattening the matrix into a row ordered, one dimensional array. This value can be parsed into its two tree, or tree ID, parts using the following equations:

$$tree1 = \left\lfloor \frac{position\ ID}{total\ trees} \right\rfloor \quad (1)$$

$$tree2 = position\ ID\ \%\ total_trees \quad (2)$$

If a sequence of consecutive tree pairs (or runs) in the matrix have the same RF values, then only the start and end points of that sequence is stored in the RF matrix. Each line in the index files stores a single value or range (start and end points of a run). The values in the index files are in descending order.

	0	1	2	3	4	5
0	0	1	2	2	3	3
1	**1**	0	3	1	2	1
2	**2**	**3**	0	2	2	2
3	**2**	**1**	**2**	0	2	1
4	**3**	**2**	**2**	**2**	0	1
5	**3**	**1**	**2**	**1**	**1**	0

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	**6**	7	8	9	10	11
2	**12**	**13**	14	15	16	17
3	**18**	**19**	**20**	21	22	23
4	**24**	**25**	**26**	**27**	28	29
5	**30**	**31**	**32**	**33**	**34**	35

RF 0	RF 1	RF 2	RF 3
	34-33	32	30
	31	27-25	24
	19	20	13
	6	18	
		12	

Fig1: A sample RF matrix (top left), the position id's for every cell in the matrix (top right) and the index files that would be generated for the matrix by our algorithm (bottom). Cells in the lower triangle / below the main diagonal (in black) are the only ones processed by our algorithm.

The Algorithm:

Our algorithm loops backwards through the lower triangle of the RF matrix, starting from the last row. It assigns each cell in the matrix a unique position ID, obtained by flattening the two dimensional matrix out into a row ordered one dimensional matrix. The position ID is thus:

$$position\ ID = total_trees * row + column \quad (3)$$

It is this position ID that is stored in the index (see Fig1). As it traverses the matrix, it stores the positions IDs of every cell in its matching RF value array. Once that array size reaches a set maximum value—10,000 in our runs—it writes all of those values out to the RF index file and resets the array.

```
Algorithm 1 : A Compact Indexing Algorithm
Require: A t x t RF matrix, matrix, with m taxa

  {All cells in the following arrays are initialized to 0}
  last_pos[m]
  start[m][100001]
  end[m][100000]

  for row = t to 0 d: do
    if matrix[row] is not an identical tree then
      pos_id = (row * t) + row - 1

      for col = row - 1 to 0 do
        current_rf = matrix[row][col]

        if current cell is not part of an RF run then
          store last_pos[current_rf] at the end of the start[current_rf] array
          store the current pos_id at the end of the end[current_rf] array

          if end[current_rf] is full then
            write start[current_rf] and end[current_rf] to file
            reset the start[current_rf] and end[current_rf]
          end if
        end if

        if current_rf == 0 then
          flag tree with id col as an identical tree
        end if

        pos_id -= 1
        last_pos[current_rf] = pos_id
      end for
    end if
  end for
```

If, in its traversal, it sees a cell with an RF value of zero, it flags the column associated with that cell as an identical tree and skips it when it reaches that row.

If the algorithm encounters a continuous run of the same RF value in a row, it only stores the start and end points of that run in the associated RF value array.

Storing Position ID in Index Files:

We opted to store a single position ID for every tree pair/cell in the matrix rather than the row and column values based on preliminary indexing tests. These tests showed that, if index files stored both the row and column values (i.e. both tree IDs), then the sum of the disk space requirements of all of the index files for an RF matrix could go up to three times the disk size of the matrix itself. Storing only one position ID for each tree pair instead removed this problem.

Indexing with FastHashRF:

We implemented our algorithm as part of FastHashRF in order to avoid the file overhead and consequent program slowdown associated with reading the RF matrix from file. With FastHashRF, the matrix values only had to be obtained from a hash table created at run time and stored in the program's memory.

Index's Descending Order:

Our index is in descending order to facilitate any future search implementations for our index where all trees that are RF distance d away from a particular tree q are being searched for in the index files. That is to say, all positions in the RF matrix where q is either the row or the column value and d is the value stored in that cell. With our index, only q needs to be searched for rather than both q and d. Since our algorithm only traverses the lower triangle of the matrix, the column value will always be less than the row value for all of the values in our index. This also means that the last possible place where the query tree q can be found is in the row position. If the index were in ascending order, this would mean that the entire index file would have to be searched in order to get all search results. However, with the index files in descending order, this means that the search only has to continue until a row value lower than the query tree's tree ID is seen in the index. We believe that this ordering can save some time in such searches.

Alternatively, this ordering would not be necessary if the upper triangle of the RF matrix were traversed rather than the lower one. In this case, the search could be stopped when a row ID larger, rather than smaller, than q was found in the index.

IV. EXPERIMENTAL MATERIAL

We performed our tests on tree sets spanning 60 to 567 taxa and 3,419 to 135,135 trees. Our experiments were performed on 2.50 GHz Intel Core 2 Quad Q8300 processors with 3.9GiB memory. The Ubuntu 8.10 operating system was used.

V. METHODS

1. Algorithm's Performance:

<u>Experiment 1: Index Generation Speed Test:</u>

We tested our algorithm's speed by comparing the index generation time—i.e. the time to calculate the index values and write them all to file—with the RF matrix print-to-screen time, the only alternative if all matrix data is needed. Both the index generation algorithm and RF matrix print algorithm were implemented as part of FastHashRF. We recorded the average program runtime over a minimum of 6 trials for each tree set in our experimental group. The results are summarized in Fig2.

2. Index's Performance:

<u>Experiment 2: Searching for all Instances of x RF distance</u>

We tested our algorithm's efficiency in searching for all instances of a particular RF distance in the matrix by searching all of our tree sets that had less than 90,000 trees. We searched for all possible RF distances for the 60 taxa/3419 trees and 8 taxa/ 10395 trees tree set. We searched for 34% of the possible RF distances of the 500 taxa / 5194 trees tree set, and 8% of the 150 taxa/ 20,000 trees tree set. Three trials were conducted on the matrix as well as the index for each of the RF values searched and their average was stored. Both search algorithms were implemented in Python. Python's random number generator was employed with a seed of 0.

We compared our fastest implementation of this search type for the matrix and the index. In the matrix, this implementation returned all tree pairs that were x RF distances away while in the index only unique trees were returned. Again, our assumption was that unique trees are the only trees of interest in RF matrices. To be consistent, all searches were performed on the data below the diagonal in the RF matrix.

VI. RESULTS

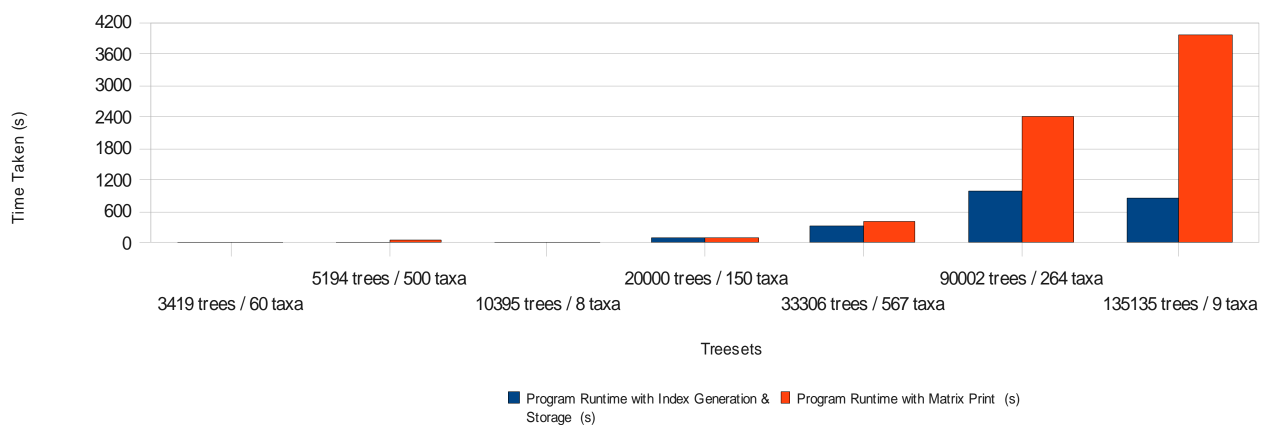

Fig1: Relative FastHashRF Runtime with Indexing vs. Printing Out the RF Matrix to File

Table 1: Percent Decrease in Program Runtime with Indexing vs. RF Matrix Print

Tree Set	Time Decrease (%)
3419 trees / 60 taxa	76.26
5194 trees / 500 taxa	58.77
10395 trees / 8 taxa	79.78
20000 trees / 150 taxa	29.76
33306 trees / 567 taxa	22.12
90002 trees / 264 taxa	59.44
135135 trees / 9 taxa	78.79

Experiment 1:

We observed that the time saving in generating our index rather than printing out the RF matrix becomes noticeable in very large (greater than 90,000 trees in size) matrices (Fig2). For RF matrices less than that size, the amount of time saved is only a couple of seconds to a minute. However, Table 1 show that our algorithm is always at least 20% and up to 79% faster than printing out the RF matrix on the matrices tested. Still, this speedup is only visible when the print time of the RF matrix is large, as it is for very large matrices. Our algorithm's speedup was mostly the result of the assumptions about the tree space listed earlier—that is:

1. Data in only one side of the diagonal is needed for a complete picture of the matrix.
2. The user is only interested in unique trees.
3. The probability of consecutive runs of the same RF distance (ranges) in a row is high in large matrices.

These assumptions decreased the amount of data that needed to be stored in our index files. Data in only half the matrix was written to file. If identical trees were observed, only the first observed instance of it was written to file. If a range was observed, only the start and end points of it were written to file.

Since writing to file was the slowest part of our algorithm, and data was written to file every time there were 100,000 values to write to file for a particular RF value. This decrease in information to save was beneficial to our algorithm's speed. It reduced the total number of writes and thus improved our algorithm's index generation speed.

Our algorithm thus performed best when there were a considerable amount of long ranges or identical trees present in the RF matrix since a lot more time was spent on processing the matrix rather than writing

the information to index files. Inversely, it performed worst when there were little to no identical trees or ranges.

Table 2: Percent Decrease in Search Time when searching the Index rather than the RF Matrix

Tree Sets	Percent Time Decrease (%)
3419 trees / 60 taxa	99.52
5194 trees / 500 taxa	80.21
10395 trees / 8 taxa	27.72
20000 trees / 150 taxa	97.23

Experiment 2:

Our index usually returned search results much faster when searching for all tree pairs that were a specific RF distance apart. Even in the worst case tested, (8 taxa / 10395 trees) searching our index proved to take more than 27% less time than searching the RF matrix on average. In the best case, searching our index took more than 99% less time than searching the RF matrix.

The reason why searching our index was so fast on some of the tree sets tested was due to the fact that, in tree sets with a large number of taxa, many of the possible RF distances usually don't show up on the matrix at all. However, there is no quick way of finding that out without traversing the entire matrix if the RF matrix is being used as the search structure. On the other hand, if our index is used instead, all that needs to be checked is whether there is any data for that particular RF distance or not. This can save an immense amount of time in large matrices. For example, searching the 33306 trees / 567 taxa tree matrix took an average of one and a half hours for any RF distance. This time would be required even if the RF distance being searched were not present in the RF matrix.

Using our index instead saves that entire time. Even when the files were not empty, searching our index still proved to be faster than searching the RF matrix for the same value. In the worst case tested (8 taxa / 10395 trees), there were only five possible RF distances in the matrix (number of taxa – 3), so the index files were considerably large and the speed to search them was limited by file I/O. However, the amount of file I/O for index files is always much less than that of the RF matrix since the index files contain less information than the RF matrix. This is why searching the index still took 27.72% less time than searching the RF matrix for this tree set. For this same reason, there should hypothetically be no cases where our index performs worse than the RF matrix for such searches.

Another reason searching our index was faster than searching the RF matrix was due to the fact that our index returned only unique trees while the matrix returned all trees. Only 23% of the 60

taxa/3419 trees tree set is composed of unique trees. So only that small proportion of the matrix had to be searched and returned when using the index, thereby making the search results fast.

VII. CONCLUSION AND FUTURE WORK

Phylogenetic research produces up to tens of thousands of candidate trees for a set of organisms in the search for the one correct tree to describe their evolutionary relationships. However, a lot of useful data, in the form of Robinson-Foulds matrices, for analyzing these large candidate tree sets cannot be efficiently used at the moment. It is time consuming to perform any type of queries that require a full matrix traversal of large RF matrices. We proposed an indexing algorithm that helps speed up these queries.

Our experiments tested the performance of our indexing algorithm as well as the usability of the index generated for the type of searches we were interested in. When searching for all occurrences of a particular RF value in the RF matrix, searching our algorithm proved to take 27% to 99% less time than searching the matrix for the same RF value on average. The best cases were those tree sets with a lot of taxa, and hence a lot of possible RF distances between the trees. Inversely, the worst cases were those with only a few taxa in the tree set. Our experiments were based on the assumption that only unique trees are the trees of interest. The generation time of our algorithm is consistently 20% to 79% faster than the print time of the RF matrix for our experimental set. This speedup becomes more noticeable when indexing very large RF matrices where the RF matrix print time can take a considerable amount of time.

Our work has a lot of potential of extension. Additional work is needed on optimizing the querying algorithms used on our index as well as on introducing support for other, different types of queries that users of the RF matrix might be interested in into our index. Since our algorithm performs best with long runs of the same RF value, future work could include implementing a preprocessing step to our index that would rearrange the trees in the matrix so that a maximal number and length of such runs occur in the matrix in order to speed up the indexing process.

VIII. ACKNOWLEDGMENTS

We would like to thank the Distributed Research Experiences for Undergraduates (DREU) program for making this research experience possible.

REFERENCES

[1] Seung-Jin Sul and Tiffani L. Williams, "A Randomized Algorithm for Comparing Sets of Phylogenetic Trees," Asia- Pacific Bioinformatics Conference (APBC'07), pp. 121- 130,2007.

[2] D. F. Robinson and L. R. Foulds. Comparison of Phylogenetic Trees. Mathematical Biosciences, 53:131–147, 1981.

[3] Seung-Jin Sul, Suzanne J. Matthews, and Tiffani L. Williams, "New Approaches to Compare Phylogenetic Search Heuristics", IEEE International Conference on Bioinformatics and Biomedicine (BIBM'08).

[4] D. M. Hillis, T. A. Heath, and K. S. John. Analysis and visualization of tree space. Syst. Biol, 54(3):471–482, 2004.

[5] D. Bader, B. M. Moret, and L. Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. In H. Siegel, editor, Proceedings of SPIE Commercial Applications for High-Performance Computing, volume 4528, pages 159–168, Denver, CO, Aug. 2001.

[6] Seung-Jin Sul and Tiffani L. Williams, "An Experimental Analysis of Robinson- Foulds Distance Matrix Algorithms", European Symposium on Algorithms (ESA'08).