Suffix Trees

Donovan Roseau
College of Charleston undergraduate
Student
3582 C of C Complex
843.687.5870
Roseaudc@g.cofc.edu

ABSTRACT

In this paper, we will analyze suffix trees and several real-life applications that demonstrates the efficiency of the data structure. I discuss a generalized version of a suffix tree and how it is constructed, accessed, its complexity, advantages, and disadvantages of utilizing the data structure in order to find matching patterns within a text or database. Then, I will discuss how several computer scientist have built suffix trees in order to aid them in their research, whether it be sequencing DNA and RNA[6] or scanning emails and searching for spam.[7] Many researcher have constructed the own version of a suffix tree that is more ideal for their application. So I will explain how they construct it, and show the various factors that increase and decrease the efficiency of it.

1. INTRODUCTION

1.1 Suffix Trees

Introduced by Peter Weiner in 1973, a suffix tree is a data structure and compressed trie which searches for unique patterns in a text or large database.[7] It compares a substring or characters in the database in order to look for an exact match of the suffix. The data structure is one of the fastest ways to search texts or databases and still receive accurate results.[6] Each node, unless it is the node completing the suffix tree, has a label on each branch. Each matching label is followed down the tree until the terminal character is reached. A suffix tree is similar to a suffix trie, however, the suffix tree compresses its branches that have single children into one node. The resulting data structure is requires a unique first character, but yields the same results as a binary trie but increases efficiency and reduces the run-time(O(N)) of the search.[7]

1.2 Construction

Suffix trees are constructed with one root that has a branch for each possible first character. Unlike a suffix trie, a suffix tree is compressed so that if a node has one child, and that child also has one child, they now become one node with a multi-character label.[7] The tree continues until a terminal character is reached at the leaf of the suffix tree.[6] Also, the node prior to a completed suffix, may now have a label. In general in suffix trees, the edges are usually denoted, but a tree can be built with the nodes being labeled rather than the edges and having no terminal character.[5]

1.3 Access

There are ways to traverse a suffix array and access it. One way to access it is by calling the function, root(), which returns the root node. A user can create other methods as well that can access the suffix tree, such as child(), which returns the internal and leaf nodes within the suffix tree.

1.4 Complexity

The complexity of a suffix tree can vary depending on which algorithm the user chooses to utilize when constructing it. Most real life applications utilize a compressed suffix tree.[5] This is a tree that does not explicitly store the labels and edges, but they are represented by integers that indicate the start and end positions. On the other hand, an uncompressed suffix tree, or suffix trie, is the opposite by labeling each edge with actual characters.[5] However, this method of constructing a suffix tree results in a data structure with the size O(N^2). This introduces the problem of having a tree so large that there is not enough space in memory to construct it.[6]

1.5 Advantages and Disadvantages

The utilization of suffix trees can have many advantages and disadvantages. Since the data structure uses labels on their branches they are advantageous when trying to find matching sequences in a database. For example, scientists are able to use suffix trees when searching for matches for DNA, RNA, and genome sequences in a database.[6] One disadvantage of suffix trees is that when constructing it, the data structure can become too large that there is not enough memory in the CPU for the structure. This problem can arise when the database that the suffix tree is searching is too large.[6] When Shibuyo constructs his geometric suffix tree, the database that it searches in, N, make the time to construct the tree N^2. And he has discovered that in some cases, the time to construct the tree can grow even larger and become N^3.

2. SEARCHING PROTEIN SEQUENCES USING GEOMETRIC SUFFIX TREES

2.1 Purpose

In the article written by Tetsuo Shibuyo from the University of Tokyo, his team and he develop a variation of a suffix tree called a geometric suffix tree. The purpose of the geometric suffix tree is to create an index structure sophisticated enough to index a 3-D models of protein in order to find similar protein structures within a database.[6] This will allow scientist to analyze the function of proteins, despite having different amino acid sequences. They also set out to prove that their data structure is more efficient than other indexing structures if the RMSD (root mean square deviation) threshold is no larger than 1A.[6] General suffix trees and other variations have been created in order to compare a protein structure to a database, but many fail in taking into account many global similarities such as USMRD and RMSD. But being able to compare a protein sequence to a 3-D model of a protein structure is what makes the geometric structures a key factor in determining the number of matching structures in a database.

2.2 Geometric Suffix

Tetsuo Shibuya describes how there are various methods and algorithms for comparing and searching protein databases. However, these methods have proven to be inefficient due to the amount of time it takes to apply in a large database.[6] The solution to this problem is a tree that can search a protein sequence database accurately and at a more efficient run-time called a geometric suffix tree. Shibuya claims that not only is the geometric suffix tree capable of searching and matching similar protein structures, but it useful for finding structural motifs and clustering substructures. The construction of a geometric suffix tree is $(O(N^2))$, where N is the size of the database.[6] In a suffix tree the edges represent substrings of texts, however the edges of geometric represent 3-D substructures of protein. In the above table, table 1, Shibuya shows that the geometric suffix tree algorithm, the "naïve" algorithm, is five times faster than the traditional algorithm, FFT, when computing the average time it takes to match proteins structures in a query. However, this is only true when the threshold in less than or equal to 1A. When the threshold is greater than 1A, the geometric suffix algorithm is not superior to the FFT algorithm that it is compared too.[6] The reasoning behind this lack of productivity is not mentioned in the article, and Shibuya claims that this same problem arises when searching for character strings in a general suffix tree. I have concluded that the geometric suffix tree is a vital and beneficial data structure when comparing the structures of proteins in a database but only if the many of the factors utilized when searching is fixed such as the threshold and branching parameters.

Table 1. Table comparing computation time using the "Naive" algorithm and the FFT algorithm

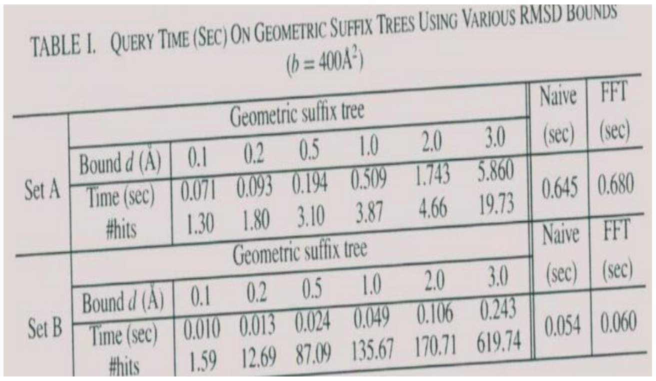

TABLE I. QUERY TIME (SEC) ON GEOMETRIC SUFFIX TREES USING VARIOUS RMSD BOUNDS ($b = 400Å^2$)

	Geometric suffix tree							Naive (sec)	FFT (sec)
Set A	Bound d (Å)	0.1	0.2	0.5	1.0	2.0	3.0		
	Time (sec)	0.071	0.093	0.194	0.509	1.743	5.860	0.645	0.680
	#hits	1.30	1.80	3.10	3.87	4.66	19.73		
	Geometric suffix tree							Naive (sec)	FFT (sec)
Set B	Bound d (Å)	0.1	0.2	0.5	1.0	2.0	3.0		
	Time (sec)	0.010	0.013	0.024	0.049	0.106	0.243	0.054	0.060
	#hits	1.59	12.69	87.09	135.67	170.71	619.74		

3. EMAIL FILTERING AND SUFFIX TREES

3.1 Purpose

In this article, researchers set out to create an algorithm that is able to filter email using a suffix tree. They want the suffix tree to be able to score different Emails that a user receives, and depending on the score, filter the Email as either spam or regular mail that the user may actually need. The necessity for this type of algorithm arose when spammers are able to bypass older methods that filter Email, such as naïve Bayes (NB), by using alternative approaches such as word salad, undistinguished messaging, intra-word characters, and embedded messaging.[5]

3.2 Suffix Tree

The computer scientist conducting the experiment choose to use a suffix tree because it is a data structure that utilizes a fast technique to search databases and texts.[5] They can use it to easily track the frequency of a character in a suffix. When constructing their suffix tree, instead of labeling the edges in the tree, they decided to rather label the nodes and to not include a terminal character, such as $. The authors state that including a terminal character into their tree would be pointless because it is depth limited, so having one would neither aid nor hinder their algorithm.[5] The tree starts with the root, and then a child is created with its label being the first letter in the suffix. The algorithm then moves down the tree and creates a new node for the next character in the suffix. The resulting tree created from this is called S1. The same process is used when creating S2, however, new node is only created when none of the existing nodes are labeled as the node that algorithm is concerned with. The algorithm will continue to do this for each character in the suffix, and the result will be the suffix tree displayed in Figure1.[5] Each node in the tree also shows the frequency at which the character was used and its position. When the algorithm for the suffix tree was compared to general algorithm used to filter spam, naïve Bayes, the suffix tree yielded better results than its competitors. Researchers have also discovered that there is also risk when the threshold is changed. This causes the suffix tree to have an increase in false positives, meaning there are some emails being classified as spam when they should not be.

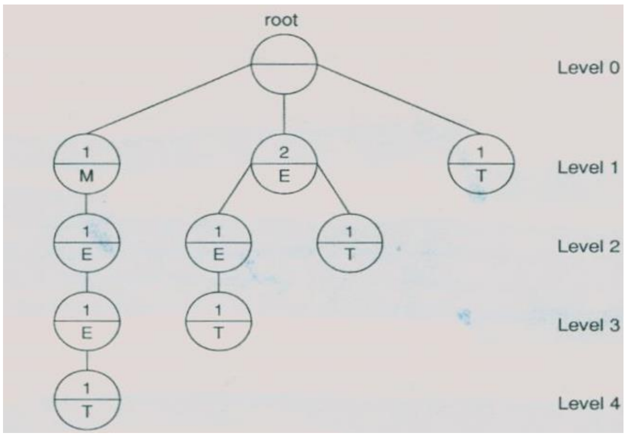

Figure 1. Displays the result when following the procedure to construct a suffix tree utilized by the researchers. Creates a branch for each character in the suffix

4. CONCLUSION AND FUTURE WORKS

Based on both articles, I have concluded that suffix trees can be a valuable data structure when needing to compare a substring to a large database and searching for matches within that database. Depending on the size of the database that the suffix tree is searching within and the method that is used when constructing the data structure, a suffix tree is an optimal tool when needing to search a database for a match quickly and accurately. But there are some drawbacks that make using a suffix tree in an algorithm pointless or the worst data structure to implement. If a programmer is building a suffix tree and the database that it is comparing to is very large, the amount of memory needed to store the tree could be exponential depending on the user's algorithm. Also higher thresholds in suffix tree algorithms can exploit flaws in using this data structure, such as slower computation times and a higher percentage of inaccurate results. There are other applications that suffix trees would be the ideal data structure to use. For example, if someone wanted to write a program that reads two text files, such as an essay for a computer science class, and compare them in order to check for plagiarism, the suffix tree would be able to compare multiple strings from one text and the text being graded. And since the text files would not be too large, the construction of the tree would not consume a significant amount of memory space. In the future, I may need this data type if I am writing a paper and want to know the frequency at which a certain word appears in my document. A program very much similar to word count in Microsoft Word.

5. References

[1] Akutsu, T., Onizuka, K.m and Ishikawa,M. 1995. New hashing techniques and their application to a protein database system. *Proc. Hawaii Int. Conf. System Sciences* 5, 197-206.

[2] Arun,K.S., Huange, T.S., And Blostein, S.D. 1987. Least-squares fitting of two 3-D point sets. *IEEE Trans Pattern Anal. Machine Intell*. 9, 698-700.

[3] Flach, P., & Lachiche, N. (2004). Naïve bayae classification of structured data. *Machine Learning*, 57(3), 233-269.

[4] Giegerich, R., & Kurtz, S. (1997). From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3), 331-353.

[5] Rajesh Pampapathi, Boris Mirkin, and Mark Levene. 2006. A suffix tree approach to anti-spam email filtering. *Mach Learn Machine Learning* 65, 1 (2006), 309–338. DOI:http://dx.doi.org/10.1007/s10994-006-9505-y

[6] Shibuya, T. 2010. Geometric suffix tree: Indexing protein 3-D structures. J. ACM 57, 3, Artticle 15,(March 2010), 17 pages. DOI = 10.1145/1706591.1706595

[7] M. Weiss., *Data Structures and Algorithms Analysis in Java.* Addison – Wesley. Reading, MA. pp. 560 – 568.