mstrdb: A Suffix-Tree-Based Database for Sequence Alignment

Nathan Panzer
Colorado School of Mines
Golden, USA
nathanpanzer@mines.edu

Abstract

Sequence alignment is a common task in the field of bioinformatics. mstrdb (Mines Suffix Tree Database) is a database built to perform efficient sequence alignment/matching through the use of an on-disk suffix tree structure. mstrdb, built as a Python API, implements target sequence ingestion, suffix tree construction, serialization to disk, and on-disk suffix tree searching. However, mstrdb is plagued by algorithmic and design pitfalls that severely limit its usability for string matching beyond very short target strings. Many improvements are proposed, which, if implemented, may be enough to bring mstrdb up to the level of comparable database tools, but as it stands suffix trees have not brought the hoped-for efficiency to string matching.

1 Introduction

1.1 Motivation

One cannot get very far in the field of bioinformatics without running into the concept of *sequence alignment*. Sequence alignment is the process of analyzing two given strings to identify similar regions (Figure 1).

```
ACTACTAGATTACTTACGGATCAGGTACTTTAGAGGCTTGCAACCA
          |||| |||||| |||||||||||||||
          TACTCACGGATGAGGTACTTTAGAGGC
```

Figure 1: Example sequence alignment of two nucleotide (DNA) sequences.

Sequence alignment, which reveals similarities between strings of nucleotides (the building blocks of DNA) or amino acids (the building blocks of proteins), can shed light on relationships between the sequences. Similar regions may have similar biological functionality or might result in the same physical protein structure. Differences between regions can highlight mutations and evolutionary differences between organisms [4]. Being able to effectively align these sequences is an essential task in the ever-growing discipline of bioinformatics, the intersection between biology and computer science [2].

At its core, biological sequence alignment is just searching for overlap between strings. However, it poses unique challenges compared to traditional string-searching. The complete human genome consists of 3.1 billion base pairs—that's a string of 3.1 billion characters [3]. The goal of being able to efficiency perform exact and partial matches on target strings of this size has led to the development of many specialized tools and algorithms [7].

1.2 Key Contributions

The primary contribution of mstrdb is its investigation of using suffix trees (described below in 2.2) as a more efficient sequence alignment search tool. The tool is designed so users can load a target sequence into the database, which will construct the suffix tree. Users can then perform query sequence searches on the suffix tree, finding the indices in the target sequence that the query sequence appears at.

Ultimately, mstrdb establishes that its somewhat-naïve suffix tree implementation has practically no application in sequence alignment. In fact, I'd go so far as to call this whole thing a stupid idea, and I'd call *me* stupid for ever thinking it would work. Building a modified suffix tree to match every possible substring of a sequence is *incredibly* inefficient, both in space and time.

The coolest contribution of mstrdb is its suffix tree representation on-disk as a binary file, enabling efficient suffix tree serialization and deserialization to- and from-memory, plus suffix tree traversal on-disk without ever having to load more than one tree node into memory.

2 Related Work

2.1 Bioinformatics Databases

The most basic approach to sequence alignment is the Smith-Waterman algorithm, which uses dynamic programming to construct an optimal alignment of sequences [6]. The Smith-Waterman algorithm has a high time and space complexity, but produces optimal results.

One of the most popular sequence alignment database tools is the Basic Local Alignment Search Tool (BLAST) [1]. BLAST uses a "good-enough" heuristic approach to find regions of similarity between sequences, making it much more efficient for large sequences.

2.2 Suffix Trees

Suffix trees are a tree data structure used for efficient string matching, having applications in various areas of bioinformatics and text processing. A suffix tree is a compact representation of the suffixes of a given string, enabling easy traversal and matching [8]. While using suffix trees for efficient biological sequence matching is far from a novel idea, there isn't a lot of database- or systems-oriented research related to this topic.

3 Technical Aspects

mstrdb is implemented in Python. Its logic is relatively low-level and self-contained, so it has no package dependencies or similar. mstrdb is accessed by writing a Python script to use its API interface. The code can be found at the GitHub repository: https://github.com/TheNathanSpace/mstrdb.

3.1 Components

mstrdb consists of a few components, discussed in the following sections. mstrdb usage takes the form of two main parts, seen in Figures 2 and 3.

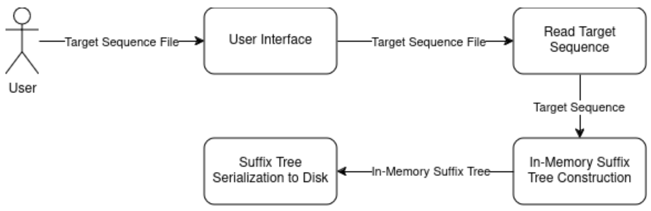


Figure 2: Suffix tree construction.

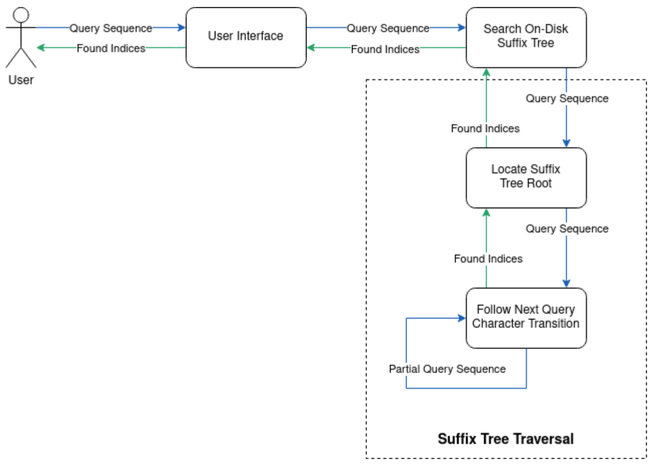


Figure 3: Suffix tree query searching.

3.1.1 Target Sequence Ingestion. First, the target sequence is read from a file provided by the user. At the moment, the entire target sequence is read into memory; however, in the future this can be easily optimized using the Seek File abstraction (described in 3.1.3) to read little-by-little from specified offsets.

3.1.2 Suffix Tree Construction. Next, a suffix tree is constructed from the target sequence. An example suffix tree is seen in Figure 4.

The suffix tree construction algorithm can be thought of in the following way. We start with a target string of characters and a root tree node. If we want to be able to find any possible substring in the tree, then the root node must have transitions to every possible substring. The way we accomplish this is by adding a branch for each possible suffix, recording at each node the start and end indices for the string built up to that point.

So, starting at the root, we scan through the target string character by character, taking each resulting suffix and adding a transition and node corresponding to its first character. We consume each suffix character-by-character, adding a transition and node for each character in the suffix.

This actually results in a tree with a straight branch for each suffix. Since taking each possible suffix naturally leads to a huge amount of overlap, we optimize it by only having one transition per character for a given node. For example, starting from the root node there will only be one "C" transition, which will lead us to the branch containing all substrings starting with a "C".

In order to enable searching for any substring, not just suffix strings, we augment the suffix tree in the following way. First, notice that when constructing any given child node, its matched substring is the same as its parent, plus one character. Therefore, we can increase the ending index of matched strings in a branch by 1 for each parent node. For a given branch, the starting index will be determined by its first node (after the root). This is demonstrated in Figure 4. Its second-from-the-right branch ("c"-"a"-"t") can only match strings starting with "c" (index 2). Even though "a"-"t" is in the branch, it can't match it—an "a"-"t" match is done by transitioning from the root along the "a" transition. So, we see that it's easy to find the start and end indices of the matched substring for any given node during its construction. (In cases where a node is shared by multiple branches, as described in the previous paragraph, it simply stores all applicable indices, all of which are valid.)

3.1.3 Suffix Tree Serialization. Now that the suffix tree is constructed in memory, it is written to disk. This serialization process is supported by a special Seek File file-access interface built on top of the Python file IO interface. The Seek File interface makes reading and writing bytes to arbitrary locations (offsets) in a file very easy. (It's not super complicated, but it's worth mentioning because it kind of made this whole process click into place.)

The suffix tree is represented on-disk as a collection of suffix tree node structures of bytes. A suffix tree node contains the number of indices the node matches, the number of children the node has, a list of its matched indices (stored as start and end integers), and a list of its children (stored as a transition character and the file offset of the child node). This serialization format is shown in Figure 5.

The suffix tree is written to disk by traversing the tree, starting with the root node. The root node writes its data (index count, child count, index start/end, child transition character, child offset) to disk—except for its child offsets. The root node doesn't yet know where in the file its children will be located. Therefore, it pads its data to reserve enough space so that it can fill in its child offsets later.

Then, the root node has its children serialize themselves to disk. They follow the exact same process, behaving as the root node of their subtree. What this amounts to is every node, starting from the root, writing most of its data to disk, except for its child offsets. Since the tree's leaf nodes don't have children, they finish their serialization completely, at which point their parent can finish its serialization (now that it knows where its child is located in the file). This travels back up the tree, until the absolute root node finishes by writing its child offsets.

3.1.4 Suffix Tree Disk Search. Now that the suffix tree is serialized to disk in a predictable, well-defined format, it can be searched by following the file offsets. Starting from the root node, we check its children for a transition corresponding to the next character in the search sequence. If not found, the search sequence doesn't exist in the target sequence. If found, we consume the search string's

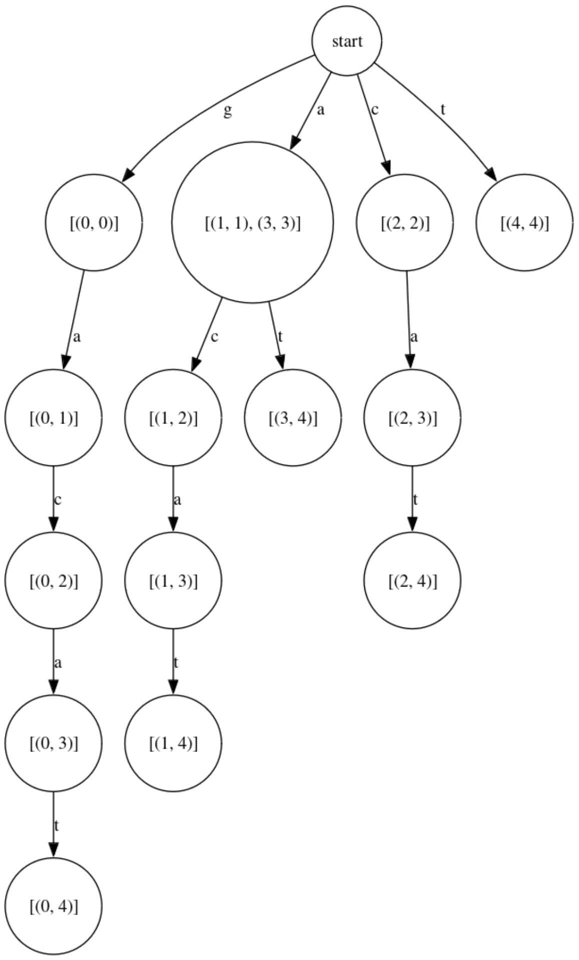
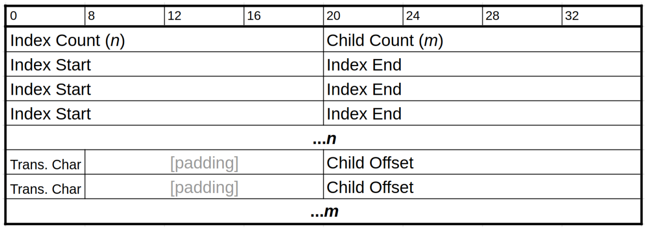


Figure 4: Example suffix tree for the string "gacat".



0	8	20
Index Count (*n*)		Child Count (*m*)
Index Start		Index End
Index Start		Index End
Index Start		Index End
...*n*		
Trans. Char	[padding]	Child Offset
Trans. Char	[padding]	Child Offset
...*m*		

Figure 5: The bytefield protocol for a suffix tree node.

character and follow the transition to the next node, repeating the process. During this traversal, at any given time it is necessary to load only one suffix tree node into memory.

4 Evaluation

4.1 Testing and Verification

I implemented functionality in `mstrdb` to output the constructed suffix trees in the DOT graph format. Then, I rendered the DOT data to a visual graph using the Graphviz tool [**ellson**]. Using this workflow, I was able to test the suffix tree construction algorithm, construction implementation, and deserialization.

I inspected the serialized suffix tree on-disk using a hex editor, verifying that each part of the suffix tree node was formatted correctly and that the offsets pointed to the same place.

4.2 Benchmarking

Minimal benchmarking against existing systems was performed. Unfortunately, `mstrdb`'s implementation is so inefficient that it could not even process larger strings. This is further discussed in sections 5 and 6.

To illustrate this point, take the following example: with the tiny input of a 120-character nucleotide sequence (120 bytes), the resulting serialized suffix tree was 144 *kilobytes*—that's a 1000x increase in size. And, as discussed in section 6, `mstrdb`can't even construct trees past a depth of 1000 characters.

Therefore, in its current state, benchmarking `mstrdb` against a system like BLAST using plausible target strings would not be practical (or possible). Further improvements and optimizations (discussed further in the mentioned sections) will be necessary to actually conduct interesting benchmarking, which may very well prove the idea of using suffix trees in this way completely useless.

5 Future Work

While I was able to build a working implementation of `mstrdb`, there are many places where it fell short of its goal, any many parts that could be significantly improved. These are described in the following sections. Others can find the code repository at: https://github.com/TheNathanSpace/mstrdb and modify the Python code under the GPLv3 license.

5.1 Search Algorithm

In contrast to exact matches, many sequence alignment tools like BLAST have functionality for *approximate* matches, where substitutions, deletions, and insertions are permitted. By default, suffix tree searching doesn't match approximate sequences, but with modifications to the search algorithm this improvement could be implemented.

This enhancement would require a few changes. First, choose a value for the maximum allowable difference between the two strings ("edit distance"). Then, begin the search at the root node. As long as the match has not reached the maximum permitted edit distance, it can take transitions that don't match the next character of the search string. Naturally, this would cause the search space to explode, because in the beginning *every* transition would be tried. However, as paths are extended the maximum edit distance will eventually be hit, and extraneous paths will be eliminated from the returned matches.

5.2 Suffix Tree Construction/Serialization

The suffix tree construction process is one of the most inefficient parts of `mstrdb`. In order to have serious applications towards sequence alignment, suffix tree construction needs to be seriously improved. Currently, suffix tree construction follows the "naïve" algorithm, which has to scan the string for every suffix, taking n^2

time. However, more efficient construction algorithms exist. Ukkonen's Algorithm achieves a time complexity of *n*, a polynomial speedup, using *suffix links* between nodes during tree construction [8]. Implementing this more efficient algorithm would bring speedups to `mstrdb`.

Another possibility for optimization comes in the alphabet representation. Sequence alignment targets very specific alphabets: nucleotides (5 characters) and amino acids (22 characters). Instead of storing suffix tree nucleotide edge transitions as characters (8 bits), they could be stored as raw bit values (3 bits), saving space. This technique could be adapted to other alphabets, too, depending on the application.

5.3 User Interface

Right now, the `mstrdb` user interface is very barebones. No command line interface is implemented yet, so it is only accessible through its Python interface. Other improvements in logging, file input/output structure and organization, and matched alignment display could bring ease-of-use and value to the project.

6 Discussion

Many of the best solutions and biggest problems in `mstrdb` arose from the characteristics of the suffix tree data structure. Like all tree structures, each suffix tree node has just one parent. This makes it very easy to work with suffix trees recursively, because you don't need any complex graph search algorithms, just a recursive traversal starting with a root node. I took full advantage of this when building `mstrdb`, implementing most features using some form of recursion on sub-trees.

Unfortunately, recursion in Python comes with many challenges. Python has a built-in recursion limit to protect against infinite recursion and stack overflows; by default, the maximum depth is 1000 layers. So, `mstrdb` can't even construct deeper suffix trees without crashing. To solve this problem, `mstrdb` would have to be redesigned to take an iterative approach, limiting the number of stack frames active at any one time. (Python does not optimize tail recursion, eliminating that option [5].)

This fatal flaw, ingrained deep in the design of `mstrdb`, severely limits its usefulness and application—but it was still lots of fun and interesting to build!

7 Conclusion

Ultimately, `mstrdb` attempts to perform more efficient sequence alignment/matching through the use of an on-disk suffix tree structure. `mstrdb` successfully implements suffix tree construction, serialization to disk, and searching with minimal memory usage, with a user interface API. However, in its current state it is plagued by algorithmic and design pitfalls that severely limit its usability for string matching beyond very short target strings. Many improvements are proposed, which, if implemented, may be enough to bring `mstrdb` up to the level of comparable database tools, but as it stands suffix trees have not brought the hoped-for efficiency to string matching. (It was a fun project though!)

References

[1] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. 2009. BLAST+: architecture and applications. en. *BMC Bioinformatics*, 10, 1, (Dec. 2009), 421. DOI: 10.1186/1471-2105-10-421.

[2] Jeff Gauthier, Antony T Vincent, Steve J Charette, and Nicolas Derome. 2019. A brief history of bioinformatics. en. *Briefings in Bioinformatics*, 20, 6, (Nov. 2019), 1981–1996. DOI: 10.1093/bib/bby063.

[3] Peter W Harrison et al. 2024. Ensembl 2024. en. *Nucleic Acids Research*, 52, D1, (Jan. 2024), D891–D899. DOI: 10.1093/nar/gkad1049.

[4] David W. Mount. 2001. *Bioinformatics: sequence and genome analysis.* Cold Spring Harbor Laboratory Press, Cold Spring Harbor, N.Y. ISBN: 9780879696559 9780879695972 9780879696085.

[5] Guido Van Rossum. 2009. Neopythonic: Final Words on Tail Calls. (Apr. 2009). Retrieved Dec. 8, 2024 from https://neopythonic.blogspot.com/2009/04/final-words-on-tail-calls.html.

[6] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. en. *Journal of Molecular Biology*, 147, 1, (Mar. 1981), 195–197. DOI: 10.1016/0022-2836(81)90087-5.

[7] J. D. Thompson, F. Plewniak, and O. Poch. 1999. A comprehensive comparison of multiple sequence alignment programs. en. *Nucleic Acids Research*, 27, 13, (July 1999), 2682–2690. DOI: 10.1093/nar/27.13.2682.

[8] E. Ukkonen. 1995. On-line construction of suffix trees. en. *Algorithmica*, 14, 3, (Sept. 1995), 249–260. DOI: 10.1007/BF01206331.