Suffix Arrays and their Applications in Biology

Alijah O'Connor (aloc4043)
alijah.oconnor@colorado.edu
CSCI 2270 Data Structures and Algorithms

INTRODUCTION

Working with biological data (e.g. protein sequences, genomic sequences, etc) in an efficient manner is at the heart of computational biology and bioinformatics. Consider the human genome and its roughly 3 billion base pairs (that is, 3 billion characters in the set {ATCG}, not including wildcard characters); without the right data structures and algorithms, processing and utilizing a genome becomes a painfully long, expensive task. Fortunately, because these datasets are simply very long strings, there are a number of structures and algorithms available that usher in practicality to this space. In particular, the suffix array has become a popular structure for solving biological problems in space- and time-effective ways. Herein, I will explain the basic properties of suffix arrays, present a comparative analysis between them and related structures, outline some of the biological problems they solve, and describe steps I've taken in constructing my own implementation of one.

SUFFIX ARRAY BASICS

The suffix array is a data structure with the ability to solve famous string-based computational problems like pattern matching, long common substring between two strings, and longest palindromic sequence amongst others. To understand suffix arrays, one must first understand strings and suffixes. A string, for our purposes, is a collection of alphanumeric symbols (encoded using a particular schema such as ASCII), and suffixes are the substrings generated from a string by removing one letter at a time. For example, the string "banana" is broken down into the suffixes {"banana", "anana", "nana", "ana", "na", "a"}. A suffix array takes these suffixes and sorts them lexographically: {"a", "ana", "anana", "banana", "na", "nana"} producing an array of the sorted indices (based on the index of the first letter of the suffix), [5, 3, 1, 0, 4, 2]. A proper suffix array structure may contain more information than the array itself; however, this simple array is the foundation for all suffix arrays and forms the basis for solving string-/biology-related problems.

CANNONICAL APPLICATIONS

The first and, arguably, most important application for a suffix array in computational biology is pattern matching; that is finding a specified sequence of characters in a larger text. For example, given genomic data, we may want to know if a particular gene or gene variation is present. In addition to just finding if a pattern exists in the text, it may be helpful to find where the pattern exists and/or how many times it appears. A number of other important biology-subproblem-solutions exist using a suffix array. Without going into the details of genome sequence construction, a vital problem to be solved is known as the longest common substring problem, which can be solved efficiently by a suffix array. These applications are discussed further later.

COMPARATIVE ANALYSIS

It should be noted that the suffix array is not the only data structure or collection of algorithms that can solve these problems. In fact, there are a number of algorithms and other data structures that can handle them. So, why the suffix array? First, let's consider an algorithm for string pattern matching: Knuth-Morris-Pratt algorithm (KMP). This famous algorithm is able to solve pattern matching O(m*n) time (including pattern preprocessing), where m is the length of the pattern and n is the length of the text. This sounds pretty good, especially considering that constructing a suffix tree is going to be at best

O(n) and searching that tree will be O(m); therefore, when doing both, we end up back at O(m*n). Therein lies the advantage though. With the suffix array, we only need to perform construction once, so if we are looking for many patterns in a particular text, we can keep running them with complexity of O(m) (which can be orders of magnitude smaller than KMP's O(m*n) for large texts).

What about other data structures? The suffix array is actually an extended form of the suffix trie and suffix tree. These data structures (particularly the suffix tree) are efficient at solving these string-based problems as well, and in some cases make for easier implementation of the methods. However, these data structures are very costly in terms of space requirements—the suffix tree increases the amount of information of the original text 10-20 fold, and it only gets worse for the suffix trie (which is essentially the uncompressed form a suffix tree) (Shrestha, 2014). A suffix tree, on the other hand, will increase the amount of information of the original text by approximately 8-fold. This space requirement can become very important when working with large datasets, such as genomic data.

SUFFIX ARRAY STRUCT & TESTING

My implementation of a suffix array will include at least three attributes. The original text, the suffix array (array of lexographically-sorted suffix indices), and an auxiliary array called a longest common prefix (LCP) array that will be used for particular functions of the suffix array. The correctness of the array construction will be evaluated using the catch framework. The suffix array itself will be tested using "banana" and a nucleotide sequence that is relatively easy to predict the results of. The indices will be easily hand-calculable, so testing will rely off those. The LCP and original text will follow the same theory in the testing scope.

SUFFIX ARRAY FUNCTIONS & TESTING

In addition to the aforementioned applications of a suffix tree, I plan to implement a number of other functions in my suffix array that can be useful for computational biology. Again, the correctness of the functions will be evaluated using the catch framework (using both positive and negative cases). Below is a list of functions to implement, their potential applications, some implementation details, and test criteria:

- Pattern matching (useful as a helper function as well as a quick way to find specific nucleotide matches).
    - Input a desired pattern and return either a Bool or list of indices for matching pattern locations.
    - This abstract function will likely be broken up into two functions, one that just finds whether or not the pattern exists and one that finds the locations of all of the instances.
    - The first will be relatively straightforward to implement using a binary search.
    - The second will require taking the binary search for finding a single instance of the pattern, then using linear searches that extend toward the beginning of the array and the end of the array respectively. This process should capture all of the indices for the matching patterns.
    - Will they return True or all of the indices of the instances for the pattern in the text, respectively. Also, do they return False or empty list, respectively, if the inputted pattern doesn't exist in the text. What happens if the pattern is longer than the text itself? Same result—False/empty list.

- Searching for a "complement" or "reverse complement pattern" in biology (useful in primer design)
    - Input either a complement or reverse complement sequence and return a Boolean, list of indices, or string corresponding to the matching complement.
    - This will basically be a wrapper for the typical search/find_locations functions, but will include pattern preprocessing to allow for appropriate pattern matching.
    - Similar testing to the typical string matching.

- Finding repeating patterns (useful in neurodegenerative disease and cancer research)
    - Input the desired pattern and return a list of indices or strings corresponding to instances of the pattern that are repeated in the text.
    - This function may be difficult to make efficient just from my speculating. I think I will have to break down the steps into find all instances of a doubled-pattern (since there need to be at least two repetitions to the start of any repeating sequence), then walk through those instances to remove the non-unique repeating strings ("ARARAR" has 3 instances of "AR", but we are only concerned with the first, since this is the actual start of the repeating sequence).
    - Should test for the case above; that is, are the reported repeating sequences unique (not part of a longer repeating sequence). What's more, the return indices should point to the beginning of actually repeating sequences of the inputted patern.

- Finding patterns that are "Off by *n* (or fewer) nucleotides" (useful for point mutations)
    - Input a desired pattern to find the in the text, but return instances that not only match perfectly, but are also instances that have *n* or fewer mismatches.
    - Likely one of the more difficult functions that I wish to implement, but it seems to boil down to rewriting the strcmp method with a "threshold" for mis-matches.
    - The testing for this on will be similar to the standard search/find_locations; looking for patterns that match perfectly, with $1 < x < n$ mismatches, and $x == n$ mismatches.

- I would also like to implement the famous longest common substring between two strings function (useful for comparative evolution applications)
    - Input two separate suffix arrays returns the length of the longest common substring or a 2x2 list that contains the indices for the start and ends of that substring.
    - This function would require a couple of helper functions to handle the LCP processing of the string.
    - Test two simple substrings with a couple of common patterns (but with one that is longer than the othes). What happens when there are two common substrings that are tied for the longest length? Popular answer seems to be report the first set. Test this case. Test the case where the two substrings literally share no substrings—perhaps 'GGGG' vs 'AAAA'.

CONCLUSION

At first glance, the suffix array may seem like a very simple data structure with few applications; however, as demonstrated, it contains the unique combination of space- and time-efficient solutions to common string problems (lending to its popularity in the computational biology field). The functions associated with this data structure range from trivial to multi-stepped and difficult, making it an interesting structure for a burgeoning computational biologist to attempt to implement (helping the student learn the necessity of performance in the context of biological problems). Without suffix arrays (or the related suffix tree/trie) many applications of computational biology may still lie in the theoretical realm; therefore, their importance should not be understated.

REFERENCES

Shrestha, A. M. S., Frith, M. C., & Horton, P. (2014). A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in bioinformatics*, *15*(2), 138-154.