7

First Applications of Suffix Trees

We will see many applications of suffix trees throughout the book. Most of these applications allow surprisingly efficient, linear-time solutions to complex string problems. Some of the most impressive applications need an additional tool, the constant-time lowest common ancestor algorithm, and so are deferred until that algorithm has been discussed (in Chapter 8). Other applications arise in the context of specific problems that will be discussed in detail later. But there are many applications we can now discuss that illustrate the power and utility of suffix trees. In this chapter and in the exercises at its end, several of these applications will be explored.

Perhaps the best way to appreciate the power of suffix trees is for the reader to spend some time trying to solve the problems discussed below, without using suffix trees. Without this effort or without some historical perspective, the availability of suffix trees may make certain of the problems appear trivial, even though linear-time algorithms for those problems were unknown before the advent of suffix trees. The *longest common substring problem* discussed in Section 7.4 is one clear example, where Knuth had conjectured that a linear-time algorithm would not be possible [24, 278], but where such an algorithm is immediate with the use of suffix trees. Another classic example is the *longest prefix repeat problem* discussed in the exercises, where a linear-time solution using suffix trees is easy, but where the best prior method ran in $O(n \log n)$ time.

7.1. APL1: Exact string matching

There are three important variants of this problem depending on which string P or T is known first and held fixed. We have already discussed (in Section 5.3) the use of suffix trees in the exact string matching problem when the pattern and the text are both known to the algorithm at the same time. In that case the use of a suffix tree achieves the same worst-case bound, $O(n + m)$, as the Knuth-Morris-Pratt or Boyer–Moore algorithms.

But the exact matching problem often occurs in the situation when the text T is known first and kept fixed for some time. After the text has been preprocessed, a long sequence of patterns is input, and for each pattern P in the sequence, the search for all occurrences of P in T must be done as quickly as possible. Let n denote the length of P and k denote the number of occurrences of P in T. Using a suffix tree for T, all occurrences can be found in $O(n + k)$ time, totally independent of the size of T. That any pattern (unknown at the preprocessing stage) can be found in time proportional to its length alone, and after only spending linear time preprocessing T, is amazing and was the prime motivation for developing suffix trees. In contrast, algorithms that preprocess the pattern would take $O(n + m)$ time during the search for any single pattern P.

The reverse situation – when the pattern is first fixed and can be preprocessed before the text is known – is the classic situation handled by Knuth-Morris-Pratt or Boyer–Moore, rather than by suffix trees. Those algorithms spend $O(n)$ preprocessing time so that the search can be done in $O(m)$ time whenever a text T is specified. Can suffix trees be used in this scenario to achieve the same time bounds? Although it is not obvious, the answer is "yes". This reverse use of suffix trees will be discussed along with a more general problem in Section 7.8. Thus for the exact matching problem (single pattern), suffix trees can be used to achieve the same time and space bounds as Knuth-Morris-Pratt and Boyer–Moore when the pattern is known first or when the pattern and text are known together, but they achieve vastly superior performance in the important case that the text is known first and held fixed, while the patterns vary.

7.2. APL2: Suffix trees and the exact set matching problem

Section 3.4 discussed the *exact set matching problem*, the problem of finding all occurrences from a set of strings $\mathcal{P}$ in a text T, where the set is input all at once. There we developed a linear-time solution due to Aho and Corasick. Recall that set $\mathcal{P}$ is of total length n and that text T is of length m. The Aho–Corasick method finds all occurrences in T of any pattern from $\mathcal{P}$ in $O(n+m+k)$ time, where k is the number of occurrences. This same time bound is easily achieved using a suffix tree $\mathcal{T}$ for T. In fact, we saw in the previous section that when T is first known and fixed and the pattern P varies, all occurrences of any specific P (of length n) in T can be found in $O(n+k_P)$ time, where k_P is the number of occurrences of P. Thus the exact set matching problem is actually a simpler case because the set $\mathcal{P}$ is input at the same time the text is known. To solve it, we build suffix tree $\mathcal{T}$ for T in $O(m)$ time and then use this tree to successively search for all occurrences of each pattern in $\mathcal{P}$. The total time needed in this approach is $O(n+m+k)$.

7.2.1. Comparing suffix trees and keyword trees for exact set matching

Here we compare the relative advantages of keyword trees versus suffix trees for the exact set matching problem. Although the asymptotic time and space bounds for the two methods are the same when both the set $\mathcal{P}$ and the string T are specified together, one method may be preferable to the other depending on the relative sizes of $\mathcal{P}$ and T and on which string can be preprocessed. The Aho–Corasick method uses a keyword tree of size $O(n)$, built in $O(n)$ time, and then carries out the search in $O(m)$ time. In contrast, the suffix tree $\mathcal{T}$ is of size $O(m)$, takes $O(m)$ time to build, and is used to search in $O(n)$ time. The constant terms for the space bounds and for the search times depend on the specific way the trees are represented (see Section 6.5), but they are certainly large enough to affect practical performance.

In the case that the set of patterns is larger than the text, the suffix tree approach uses less space but takes more time to search. (As discussed in Section 3.5.1 there are applications in molecular biology where the pattern library is much larger than the typical texts presented after the library is fixed.) When the total size of the patterns is smaller than the text, the Aho–Corasick method uses less space than a suffix tree, but the suffix tree uses less search time. Hence, there is a time/space trade-off and neither method is uniformly superior to the other in time and space. Determining the relative advantages of Aho–Corasick versus suffix trees when the text is fixed and the set of patterns vary is left to the reader.

There is one way that suffix trees are better, or more robust, than keyword trees for the exact set matching problem (in addition to other problems). We will show in Section 7.8 how to use a suffix tree to solve the exact set matching problem in exactly the same time and space bounds as for the Aho–Corasick method – $O(n)$ for preprocessing and $O(m)$ for search. This is the reverse of the bounds shown above for suffix trees. The time/space trade-off remains, but a suffix tree can be used for either of the chosen time/space combinations, whereas no such choice is available for a keyword tree.

7.3. APL3: The substring problem for a database of patterns

The substring problem was introduced in Chapter 5 (page 89). In the most interesting version of this problem, a set of strings, or a database, is first known and fixed. Later, a sequence of strings will be presented and for each presented string S, the algorithm must find all the strings in the database containing S as a substring. This is the reverse of the exact set matching problem where the issue is to find which of the fixed patterns are in a substring of the input string.

In the context of databases for genomic DNA data [63, 320], the problem of finding substrings is a real one that cannot be solved by exact set matching. The DNA database contains a collection of previously sequenced DNA strings. When a new DNA string is sequenced, it could be contained in an already sequenced string, and an efficient method to check that is of value. (Of course, the opposite case is also possible, that the new string contains one of the database strings, but that is the case of exact set matching.)

One somewhat morbid application of this substring problem is a simplified version of a procedure that is in actual use to aid in identifying the remains of U.S. military personnel. Mitochondrial DNA from live military personnel is collected and a small interval of each person's DNA is sequenced. The sequenced interval has two key properties: It can be reliably isolated by the polymerase chain reaction (see the glossary page 528) and the DNA string in it is highly variable (i.e., likely differs between different people). That interval is therefore used as a "nearly unique" identifier. Later, if needed, mitochondrial DNA is extracted from the remains of personnel who have been killed. By isolating and sequencing the same interval, the string from the remains can be matched against a database of strings determined earlier (or matched against a narrower database of strings organized from missing personnel). The *substring* variant of this problem arises because the condition of the remains may not allow complete extraction or sequencing of the desired DNA interval. In that case, one looks to see if the extracted and sequenced string is a substring of one of the strings in the database. More realistically, because of errors, one might want to compute the length of the longest substring found both in the newly extracted DNA and in one of the strings in the database. That longest common substring would then narrow the possibilities for the identity of the person. The longest common substring problem will be considered in Section 7.4.

The total length of all the strings in the database, denoted by m, is assumed to be large. What constitutes a good data structure and lookup algorithm for the substring problem? The two constraints are that the database should be stored in a small amount of space and that each lookup should be fast. A third desired feature is that the preprocessing of the database should be relatively fast.

Suffix trees yield a very attractive solution to this database problem. A generalized suffix tree $\mathcal{T}$ for the strings in the database is built in $O(m)$ time and, more importantly, requires only $O(m)$ space. Any single string S of length n is found in the database, or declared not to be there, in $O(n)$ time. As usual, this is accomplished by matching the string against a path in the tree starting from the root. The full string S is in the database if and only if the matching path reaches a leaf of $\mathcal{T}$ at the point where the last character of

S is examined. Moreover, if S is a substring of strings in the database then the algorithm can find all strings in the database containing S as a substring. This takes $O(n + k)$ time, where k is the number of occurrences of the substring. As expected, this is achieved by traversing the subtree below the end of the matched path for S. If the full string S cannot be matched against a path in $\mathcal{T}$, then S is not in the database, and neither is it contained in any string there. However, the matched path does specify the longest *prefix* of S that is contained as a substring in the database.

The substring problem is one of the classic applications of suffix trees. The results obtained using a suffix tree are dramatic and not achieved using the Knuth-Morris-Pratt, Boyer–Moore, or even the Aho–Corasick algorithm.

7.4. APL4: Longest common substring of two strings

A classic problem in string analysis is to find the longest substring common to two given strings S_1 and S_2. This is the *longest common substring problem* (different from the longest common *subsequence* problem, which will be discussed in Sections 11.6.2 and 12.5 of Part III).

For example, if $S_1 = superiorcalifornialives$ and $S_2 = sealiver$, then the longest common substring of S_1 and S_2 is *alive*.

An efficient and conceptually simple way to find a longest common substring is to build a generalized suffix tree for S_1 and S_2. Each leaf of the tree represents either a suffix from one of the two strings or a suffix that occurs in both the strings. Mark each internal node v with a 1 (2) if there is a leaf in the subtree of v representing a suffix from S_1 (S_2). The path-label of any internal node marked both 1 and 2 is a substring common to both S_1 and S_2, and the longest such string is the longest common substring. So the algorithm has only to find the node with the greatest string-depth (number of characters on the path to it) that is marked both 1 and 2. Construction of the suffix tree can be done in linear time (proportional to the total length of S_1 and S_2), and the node markings and calculations of string-depth can be done by standard linear-time tree traversal methods.

In summary, we have

Theorem 7.4.1. *The longest common substring of two strings can be found in linear time using a generalized suffix tree.*

Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible [24, 278]. We will return to this problem in Section 7.9, giving a more space efficient solution.

Now recall the problem of identifying human remains mentioned in Section 7.3. That problem reduced to finding the longest substring in one fixed string that is also in some string in a database of strings. A solution to that problem is an immediate extension of the longest common substring problem and is left to the reader.

7.5. APL5: Recognizing DNA contamination

Often the various laboratory processes used to isolate, purify, clone, copy, maintain, probe, or sequence a DNA string will cause unwanted DNA to become inserted into the string of interest or mixed together with a collection of strings. Contamination of protein in the laboratory can also be a serious problem. During cloning, contamination is often caused by a fragment (substring) of a *vector* (DNA string) used to incorporate the desired DNA in a host organism, or the contamination is from the DNA of the host itself (for example bacteria or yeast). Contamination can also come from very small amounts of undesired foreign DNA that gets physically mixed into the desired DNA and then amplified by PCR (the polymerase chain reaction) used to make copies of the desired DNA. Without going into these and other specific ways that contamination occurs, we refer to the general phenomenon as *DNA contamination.*

Contamination is an extremely serious problem, and there have been embarrassing occurrences of large-scale DNA sequencing efforts where the use of highly contaminated clone libraries resulted in a huge amount of wasted sequencing. Similarly, the announcement a few years ago that DNA had been successfully extracted from dinosaur bone is now viewed as premature at best. The "extracted" DNA sequences were shown, through DNA database searching, to be more similar to mammal DNA (particularly human) [2] than to bird and crockodilian DNA, suggesting that much of the DNA in hand was from human contamination and not from dinosaurs. Dr. S. Blair Hedges, one of the critics of the dinosaur claims, stated: "In looking for dinosaur DNA we all sometimes find material that at first looks like dinosaur genes but later turns out to be human contamination, so we move on to other things. But this one was published." [80]

These embarrassments might have been avoided if the sequences were examined early for signs of likely contaminants, before large-scale analysis was performed or results published. Russell Doolittle [129] writes "...On a less happy note, more than a few studies have been curtailed when a preliminary search of the sequence revealed it to be a common contaminant ... used in purification. As a rule, then, the experimentalist should search early and often".

Clearly, it is important to know whether the DNA of interest has been contaminated. Besides the general issue of the accuracy of the sequence finally obtained, contamination can greatly complicate the task of shotgun sequence assembly (discussed in Sections 16.14 and 16.15) in which short strings of sequenced DNA are assembled into long strings by looking for overlapping substrings.

Often, the DNA sequences from many of the possible contaminants are known. These include cloning vectors, PCR primers, the complete genomic sequence of the host organism (yeast, for example), and other DNA sources being worked with in the laboratory. (The dinosaur story doesn't quite fit here because there isn't yet a substantial transcript of human DNA.) A good illustration comes from the study of the nemotode *C. elegans*, one of the key model organisms of molecular biology. In discussing the need to use YACs (Yeast Artificial Chromosomes) to sequence the *C. elegans* genome, the contamination problem and its potential solution is stated as follows:

The main difficulty is the unavoidable contamination of purified YACs by substantial amounts of DNA from the yeast host, leading to much wasted time in sequencing and assembling irrelevant yeast sequences. However, this difficulty should be eliminated (using)... the complete (yeast) sequence... It will then become possible to discard instantly all sequencing reads that are recognizable as yeast DNA and focus exclusively on *C. elegans* DNA. [225]

This motivates the following computational problem:

DNA contamination problem Given a string S_1 (the newly isolated and sequenced string of DNA) and a known string S_2 (the combined sources of possible contamination), find all substrings of S_2 that occur in S_1 and that are longer than some given length l. These substrings are candidates for unwanted pieces of S_2 that have contaminated the desired DNA string.

This problem can easily be solved in linear time by extending the approach discussed above for the longest common substring of two strings. Build a generalized suffix tree for S_1 and S_2. Then mark each internal node that has in its subtree a leaf representing a suffix of S_1 and also a leaf representing a suffix of S_2. Finally, report all marked nodes that have string-depth of l or greater. If v is such a marked node, then the path-label of v is a suspicious string that may be contaminating the desired DNA string. If there are no marked nodes with string-depth above the threshold l, then one can have greater confidence (but not certainty) that the DNA has not been contaminated by the known contaminants.

More generally, one has an entire set of known DNA strings that might contaminate a desired DNA string. The problem now is to determine if the DNA string in hand has any sufficiently long substrings (say length l or more) from the known set of possible contaminants. The approach in this case is to build a generalized suffix tree for the set $\mathcal{P}$ of possible contaminants together with S_1, and then mark every internal node that has a leaf in its subtree representing a suffix from S_1 and a leaf representing a suffix from a pattern in $\mathcal{P}$. All marked nodes of string-depth l or more identify suspicious substrings.

Generalized suffix trees can be built in time proportional to the total length of the strings in the tree, and all the other marking and searching tasks described above can be performed in linear time by standard tree traversal methods. Hence suffix trees can be used to solve the contamination problem in linear time. In contrast, it is not clear if the Aho–Corasick algorithm can solve the problem in linear time, since that algorithm is designed to search for occurrences of *full* patterns from $\mathcal{P}$ in S_1, rather than for substrings of patterns.

As in the longest common substring problem, there is a more space efficient solution to the contamination problem, based on the material in Section 7.8. We leave this to the reader.

7.6. APL6: Common substrings of more than two strings

One of the most important questions asked about a set of strings is: What substrings are common to a large number of the *distinct* strings? This is in contrast to the important problem of finding substrings that occur repeatedly in a single string.

In biological strings (DNA, RNA, or protein) the problem of finding substrings common to a large number of distinct strings arises in many different contexts. We will say much more about this when we discuss database searching in Chapter 15 and multiple string comparison in Chapter 14. Most directly, the problem of finding common substrings arises because mutations that occur in DNA after two species diverge will more rapidly change those parts of the DNA or protein that are less functionally important. The parts of the DNA or protein that are critical for the correct functioning of the molecule will be more highly conserved, because mutations that occur in those regions will more likely be lethal. Therefore, finding DNA or protein substrings that occur commonly in a wide range of species helps point to regions or subpatterns that may be critical for the function or structure of the biological string.

Less directly, the problem of finding (exactly matching) common substrings in a set of distinct strings arises as a subproblem of many heuristics developed in the biological literature to *align* a set of strings. That problem, called multiple alignment, will be discussed in some detail in Section 14.10.3.

The biological applications motivate the following exact matching problem: Given a set of strings, find substrings "common" to a large number of those strings. The word "common" here means "occurring with equality". A more difficult problem is to find "similar" substrings in many given strings, where "similar" allows a small number of differences. Problems of this type will be discussed in Part III.

Formal problem statement and first method

Suppose we have K strings whose lengths sum to n.

Definition For each k between 2 and K, we define $l(k)$ to be the length of the *longest substring common to at least k of the strings.*

We want to compute a table of $K - 1$ entries, where entry k gives $l(k)$ and also points to one of the common substrings of that length. For example, consider the set of strings {*sandollar, sandlot, handler, grand, pantry*}. Then the $l(k)$ values (without pointers to the strings) are:

k	$l(k)$	one substring
2	4	sand
3	3	and
4	3	and
5	2	an

Surprisingly, the problem can be solved in linear, $O(n)$, time [236]. It really is amazing that so much information about the contents and substructure of the strings can be extracted in time proportional to the time needed just to read in the strings. The linear-time algorithm will be fully discussed in Chapter 9 after the constant-time lowest common ancestor method has been discussed.

To prepare for the $O(n)$ result, we show here how to solve the problem in $O(Kn)$ time. That time bound is also nontrivial but is achieved by a generalization of the longest common substring method for two strings. First, build a generalized suffix tree $\mathcal{T}$ for the K strings. Each leaf of the tree represents a suffix from one of the K strings and is marked with one of K unique string identifiers, 1 to K, to indicate which string the suffix is from. Each of the K strings is given a distinct termination symbol, so that identical suffixes appearing in more than one string end at distinct leaves in the generalized suffix tree. Hence, each leaf in $\mathcal{T}$ has only one string identifier.

Definition For every internal node v of $\mathcal{T}$, define $C(v)$ to be the number of *distinct* string identifiers that appear at the leaves in the subtree of v.

Once the $C(v)$ numbers are known, and the string-depth of every node is known, the desired $l(k)$ values can be easily accumulated with a linear-time traversal of the tree. That traversal builds a vector V where, for each value of k from 2 to K, $V(k)$ holds the string-depth (and location if desired) of the deepest (string-depth) node v encountered with $C(v) = k$. (When encountering a node v with $C(v) = k$, compare the string-depth of v to the current value of $V(k)$ and if v's depth is greater than $V(k)$, change $V(k)$ to the depth of v.) Essentially, $V(k)$ reports the length of the longest string that occurs *exactly* k times. Therefore, $V(k) \leq l(k)$. To find $l(k)$ simply scan V from largest to smallest index, writing into each position the maximum $V(k)$ value seen. That is, if $V(k)$ is empty or $V(k) < V(k + 1)$ then set $V(k)$ to $V(k + 1)$. The resulting vector holds the desired $l(k)$ values.

7.6.1. Computing the $C(v)$ numbers

In linear time, it is easy to compute for each internal node v the number of leaves in v's subtree. But that number may be larger than $C(v)$ since two leaves in the subtree may have the same identifier. That repetition of identifiers is what makes it hard to compute $C(v)$ in $O(n)$ time. Therefore, instead of counting the number of leaves below v, the algorithm uses $O(Kn)$ time to explicitly compute which identifiers are found below any node. For each internal node v, a K-length bit vector is created that has a 1 in bit i if there is a leaf with identifier i in the subtree of v. Then $C(v)$ is just the number of 1-bits in that vector. The vector for v is obtained by **OR**ing the vectors of the children of v. For l children, this takes lK time. Therefore over the entire tree, since there are $O(n)$ edges, the time needed to build the entire table is $O(Kn)$. We will return to this problem in Section 9.7, where an $O(n)$ time solution will be presented.

7.7. APL7: Building a smaller directed graph for exact matching

As discussed before, in many applications space is the critical constraint, and any significant reduction in space is of value. In this section we consider how to compress a suffix tree into a directed acyclic graph (DAG) that can be used to solve the exact matching problem (and others) in linear time but that uses less space than the tree. These compression techniques can also be used to build a *directed acyclic word graph* (DAWG), which is the smallest finite-state machine that can recognize suffixes of a given string. Linear-time algorithms for building DAWGs are developed in [70], [71], and [115]. Thus the method presented here to compress suffix trees can either be considered as an application of suffix trees to building DAWGs or simply as a technique to compact suffix trees.

Consider the suffix tree for a string $S = xyxaxaxa$ shown in Figure 7.1. The edge-labeled subtree below node p is *isomorphic* to the subtree below node q, except for the leaf numbers. That is, for every path from p there is a path from q with the same path-labels, and vice versa. If we only want to determine *whether* a pattern occurs in a larger text, rather than learning all the locations of the pattern occurrence(s), we could *merge* p into q by redirecting the labeled edge from p's parent to now go into q, deleting the subtree of p as shown in Figure 7.2. The resulting graph is not a tree but a directed acyclic graph.

Clearly, after merging two nodes in the suffix tree, the resulting directed graph can

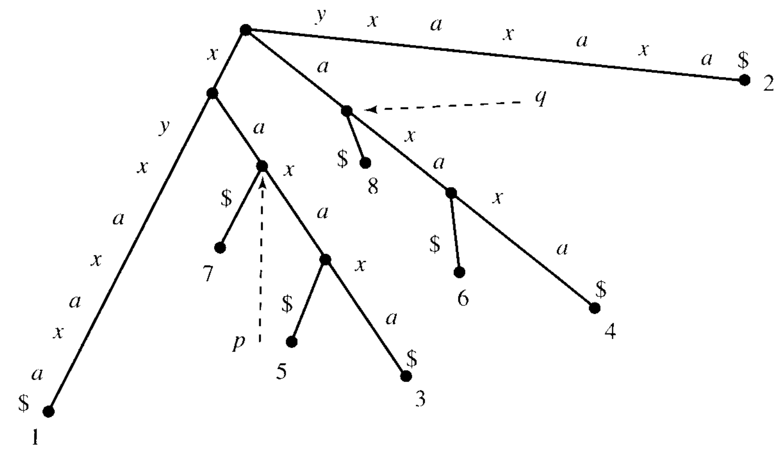

Figure 7.1: Suffix tree for string *xyxaxaxa* without suffix links shown.