Algorithms on Strings, Trees, and Sequences

Dan Gusfield
University of California, Davis
Cambridge University Press
1997

Introduction to Suffix Trees

A suffix tree is a data structure that exposes the internal structure of a string in a deeper way than does the fundamental preprocessing discussed in Section 1.3. Suffix trees can be used to solve the exact matching problem in linear time (achieving the same worst-case bound that the Knuth-Morris-Pratt and the foyer-Moore algorithms achieve), but the real virtue comes from their use in linear-time solutions to many string problems more complex than exact matching. Moreover (as we will detail in Chapter 9), suffix trees provide a bridge between *exact* matching problems, the focus of Pratt I, and *inexact* matching problems that are the focus of Pratt ILL

> The classic application for suffix trees is the *substring problem.* One is first given a text *T* of length *m.* Alter *0(m),* or linear, preprocessing time, one must be prepared to take in any unknown string *S* of length *n* and in *0(n)* time either find an occurrence of *S* in *T* Or determine that $ is not contained in *T.* That is, the allowed preprocessing takes time proportional to the length of the text, but thereafter, the search for *S* must be done in time proportional to the length of *S. independent* of the length of *T.* These bounds are achieved with the use of a suffix tree. The suffix tree for the text is built in *0(m)* time during a preprocessing stage; thereafter, whenever a string of length *0(n)* is input the algorithm searches for it in *0(n)* time using that suffix tree.

The *0(m)* preprocessing and *0(n)* search result for the substring problem is very surprising and extremely useful, In typical applications, a long sequence of requested strings will be input after the suffix tree is built, so the linear time bound for each search is important. That bound is *not* achievable by the Knuth-Morris-Pratt or Boyer-Moore methods – those methods would preprocess each requested string on input, and then take Θ(m) (worst-case) time to search for the string in the text. Because *m* may be huge compared to *n,* those algorithms would be impractical on any but trivial-sized texts.

Often the text is a fixed *set* of strings, for example, a collection of STSs or ESTs (see Sections 3.5.1 and 3.5.1), so that *the* substring problem is to determine whether the input string is a substring of any of the fixed strings. Suffix trees work nicely to efficiently solve this problem as well. Superficially, this ease of multiple text strings resembles the *dictionary* problem discussed in the context of the Aho-Corasick algorithm. Thus it is natural to expect that the Aho-Corasick algorithm could be applied. However, the Aho-Corasick method does not solve the substring problem in the desired time bounds, because it will only determine if the new string is *a full* string in the dictionary, not whether it is a substring of a string in the dictionary.

After presenting the algorithms, several applications and extensions will be discussed in Chapter 7. Then a remarkable result, *the constant-time least common ancestor method,* will be presented in Chapter 8. That method greatly amplifies the

utility of suffix trees, as will he illustrated by additional applications in Chapter 9. Some of those applications provide a bridge to inexact matching; more applications of suffix trees will be discussed in Part III, where the focus is on inexact matching.

5.1. A short history

The first linear-time algorithm for constructing suffix trees was given by Weiner [473] in 1973, although he called his tree a position tree. A different, more space efficient algorithm to build suffix trees in linear time was given by McCreight [318] a few years later More recently, Ukkonen [438] developed a conceptually different linear-time algorithm for building suffix trees that has all the advantages of McCreight's algorithm (and when properly viewed can be seen as a variant of McCreight's algorithm) but allows a much simpler explanation.

Although more than twenty years have passed since Weiner's original result (which Knuth is claimed to have called "the algorithm of 1973" [24]), suffix trees have not made it into the mainstream of computer science education, and they have generally received less attention and use than might have been expected. This is probably because the two original papers of the 1970s have a reputation for being extremely difficult to understand. That reputation is well deserved but unfortunate, because the algorithms, although nontrivial, are no more complicated than are many widely taught methods. And, when implemented well, the algorithms are practical and allow efficient solutions to many complex sting problems. We know of no other single data structure (other than those essentially equivalent to suffix trees) that allows efficient solutions to such a wide range of complex string problems.

Chapter 6 fully develops the linear-time algorithms of Ukkonen and Weiner and then briefly mentions the high-level organization of McCreight's algorithm and its relationship to Ukkonen's algorithm. Our approach is to introduce each algorithm at a high level, giving simple, *inefficient* implementations. Those implementations are then incrementally improved to achieve linear running times. We believe that the expositions and analyses given here, particularly for Weiner's algorithm, are much simpler and clearer than in the original papers, and we hope that these expositions result in a wider use of suffix trees in practice.

5.2. Basic definitions

When describing how to build a suffix tree for an arbitrary string, we will refer to the generic string S of length a We do not use P or T (denoting pattern and text) because suffix trees are used in a wide range of applications where the input string sometimes plays the role of a pattern, sometimes a text, sometimes both, and sometimes neither. As usual the alphabet is assumed finite and known. After discussing suffix tree algorithms for a single string 5, we will generalize the suffix tree to handle sets of strings.

Definition A suffix tree T for an rn-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the

concatenation of the edge-labels on the path from the root to leaf *i* exactly spells out the suffix of *S* that starts at position *i*. That is, it spells out *S[i..m]*.

For example, the suffix tree for the string *xabxac* is shown in Figure 51. The path from the root to the leaf numbered I spells out the full suing S = *xabsac*, while the path to the leaf numbered *5* spells out the suffix *ac*, which starts in position 5 of *S*.
As stated above, the definition of a suffix tree for *S* does not guarantee that a suffix tree for any strings actually exists. The problem is that if one *suffix* of S matches a prefix of another suffix of *S* then no suffix tree obeying the above definition is possible, since the path for the first suffix would not end at a leaf. For example, if the last character of xabxac is removed, creating string *xabxa*, then suffix *xa* is a prefix of suffix *xabxa*, so the path spelling out *xa* would not end at a leaf.

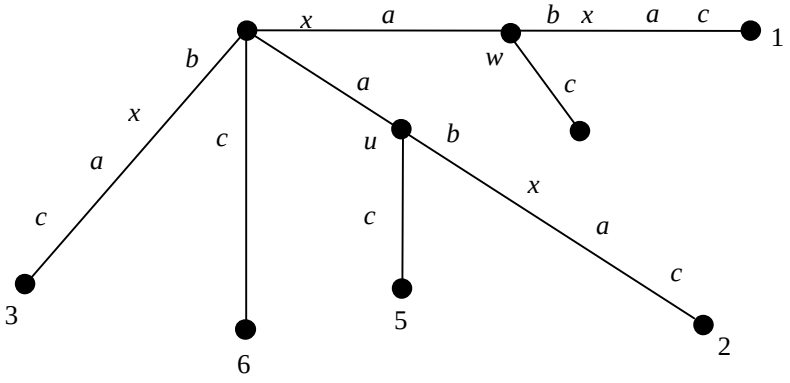

Figure 5.1: Suffix tree for string xabxac. The node labels *u* and *w* on *the* two interior nodes will be used later.

To avoid this problem, we assume (as was true in Figure 5.1) that the last character of *S* appears nowhere else in *S*. Then, no suffix of the resulting string can be a prefix of any other suffix. To achieve this in practice, we can add a character to the end of *S* that is not in the alphabet that string *S* is taken from. In this book we use $ for the "termination" character. When it is important to emphasize the fact that this termination character *has* been added, we will write it explicitly as *S*$. Much of the time, however, this reminder will not be necessary and, unless explicitly stated otherwise, every string *S* is assumed to be extended with the termination symbol $, even if *the* symbol is not explicitly shown.

A suffix tree is related to the keyword tree (without backpointers) considered in Section 3.4 (given string *S*. if set *P* is defined to be the *m* suffixes of *S*, then the suffix tree for *S* can be obtained from the keyword tree for *P* by merging any path of nonbranching nodes into a single edge. The simple algorithm given in Section 3.4 for building keyword trees could be used to construct a suffix tree for S in $0(m^2)$ time, rather than the $0(m)$ bound we will establish.

Definition The *label of a path* from the root that ends at a *node* is the concatenation, in order, of the substrings labeling the edges of that path. The *path-label of a node* is the label of the path from the root of *T* to that node.

Definition For any node *v* in a suffix tree, the *string-depth* of *v* is the number of characters in *v*'s label.

Definition A path that ends in the middle of an edge (*u, v*) splits the label on (*u, v*) at a designated point. Define the label of such a path as the label of *u* concatenated with the characters on edge (*u, v)* down to the designated split point.

For example, in Figure 5.1 string *xa* labels the internal node *w* (so node *w* has path-label *xa*), string *a* labels node *u*, and string *xabx* labels a path that ends inside edge *(w,* 1), that is, inside the leaf edge touching leaf 1.

5.3. A motivating example

Before diving into the details of the methods to construct suffix trees, let's look at how a suffix tree for a string is used to solve the exact match problem: Given a pattern *P* of length *n* and a text *T* of length *m*, find all occurrences of *P* in *T* in *0(n + m)* time. We have already seen several solutions to this problem. Suffix trees provide another approach:

Build a suffix tree *T* for text *T* on *0(m)* time. Then, match the characters of *P* along the unique path in *T* until either *P* is exhausted or no more matches are possible. In the latter case, P does not appear anywhere in *T*. In the former case, every leaf in the subtree below the point of the last match is numbered with a starting location of *P* in *T*, and every starting location of *P* in *T* numbers such a leaf.

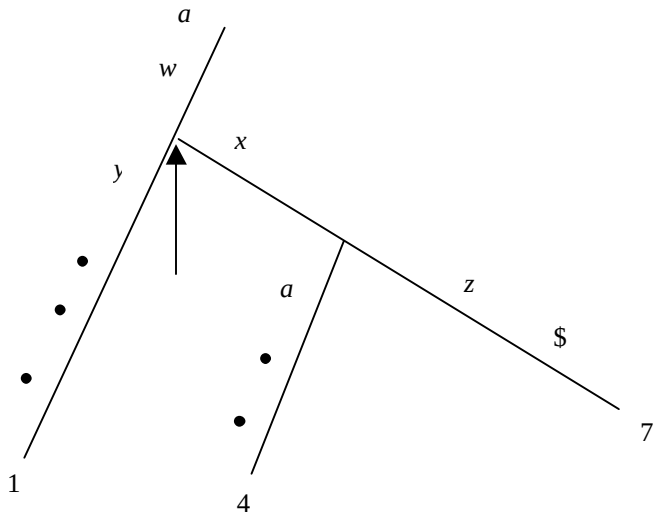

Figure 5.3: Three occurrences of aw in awyawxawxz. Their starting positions number the leaves in the subtree of the node with path-label aw.

The key to understanding the former case (when all of *P* matches a path in *T*) is to note that *P* occurs in *T* starting at position *j* if on only if *P* occurs as a prefix of *T[j..m]*. But that happens if and only if string *P* labels an initial part of the path from the root to leaf *j*. It is the initial path that will be followed by the matching algorithm.

The matching path is unique because no two edges out of a common node can have edge-labels beginning with the same character. And, because we have assumed a finite alphabet, the work at each node takes constant time and so the time

to match *P* to a path is proportional to the length of *P*.

For example, Figure 5.2 shows a fragment of the suffix tree for string *T-awyawxawxz*. Pattern *P = aw* appears three times in *T* starting at locations 1, 4, and 7. Pattern *P* matches a path down to the point shown by an arrow, and as required, the leaves below that point are numbered 1, 4 and 7.

If P fully matches some path in the tree, the algorithm can find all the starting positions of *P* in *T* by traversing the subtree below the end of the matching path, collecting position numbers written at the leaves. All occurrences of *P* in *T* can therefore, be found in *0(n + m)* time. This is the same overall time bound achieved by several algorithms considered in Part I, but the distribution of work is different. Those earlier algorithms spend *0(n)* time for preprocessing *P* and the *0(m)* time for the search. In contrast, the suffix tree approach spends *0(m)* preprocessing time and then *0(n + k)* sea4rrch time, where *k* is the number of occurrences of *P* in *T*.

To collect the *k* starting positions of *P*, traverse the subtree at the end of the matching path using any linear-time traversal (depth-first say), and note the leaf numbers encountered. Since every internal node has at least two children, the number of leaves encountered is proportional to the number of edges traversed, so the time for the traversal is *O(k), even* though the total string-depth of those *O(k)* edges may be arbitrarily larger than *k*.

If only a single occurrence of *P* is required, and the preprocessing is extended a bit, then the search time can be reduced from *O[n + k]* to *0(n)* time. The idea is to write at each node one number (say the smallest) of a leaf in its subtree This can be achieved in *0(m)* time in the preprocessing stage by a depth-first traversal of *T.* The details are straightforward and are left to the reader. Then, in the search stage, the number written on the node at or below the end of the match gives one starting position of *P* in *T*.

In Section 7.2.1 we will again consider the relative advantages of methods that preprocess the text versus methods that preprocess the pattern(s). Later, in Section 7.8, we will also show bow to use a suffix tree to solve the exact matching problem using *0(n)* preprocessing and *0(m)* search time, achieving the same bounds as in the algorithms presented in Part I.

5.4. A naive algorithm to build a suffix tree

To further solidify the definition of a suffix tree and develop the reader's intuition, we present a straightforward algorithm to build a suffix tree for string *S*. This naive method first enters a single edge for suffix *S[1..m]$* (the entire string) into the tree; then it successively enters suffix *S[i..m]$*I into the growing tree, for *i* increasing from 2 to *m*. We let *N*, denote the intermediate tree that encodes all the suffixes from 1 to *i*.

In detail, tree N_1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string *S$*. Tree N_{i+1} is constructed from *N*, as follows:

Staring at the root of *N*, find the longest path from the root whose label matches a prefix of *S[i + 1..m]$*. This path is found by successively comparing and matching characters in suffix *S[i + 1..m]$* to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of *S$* is a prefix of any other

suffix of *S$*. When that point is reached, the algorithm is either at a node, to say, or it is in the middle of an edge. If it is in the middle of an edge, (*u, v*) say, then it breaks edge (*u, v)* into two edges by inserting a new node, called *w,* just after the last character on the edge that matched a character in *S[i* + I *m]* and just before the first character on the edge that mismatched. The new edge (*u, w*) is labeled with the part of the (*u, v*) label that matched with *S[i + 1..m],* and the new edge *(w, v)* is labeled with the remaining part of the (*u, v*) label. Then (whether a new node *w,* was created or whether one already existed at the point where the match ended), the algorithm creates a new edge (*w, i + 1*) running from *w* to a new leaf labeled *i + 1* and it labels the new edge with the unmatched part of suffix *S(i + 1..m)$.*

The tree now contains a unique path from the root to leaf *i* + 1, and this path has the label *S[i + 1. .m]$.* Note that all edges out of the new node *w* have labels that begin with different first characters, and so it follows inductively that no two edges out of a node have labels with the same first character,

Assuming, as usual, a bounded-size alphabet, the above naive method takes *0(m2)* time to build a suffix tree for the strings of length *m.*

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-tine 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 tees. 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, 2181, 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 $0(n + m)$ 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 $0(n)$ preprocessing time so that the search can be done in $0(m)$ 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 he 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 ease 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 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 P is of total length a and that text T is of length m. The Aho-Corasick method finds all occurrences in T of any pattern from 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 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 $O(i + k_p)$ 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 P is input at the same time the text is known. To solve it, we build suffix tree T for T in $0(m)$ time and then use this tree to successively search for all occurrences of each pattern in P. The total time needed in this approach is $0(n + in + k)$.

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

Here we compare the relative advantages of keyword trees versus suffix frees for the exact set matching problem. Although the asymptotic time and space bounds for the two methods are the same when both the set P and the string T are specified

together, one method may be preferable to the other depending on the relative sizes of *P* and *T* and on which string can be preprocessed. The Aho-Corasick method uses a keyword tree of size *0(n),* built in *0(n)* time, and then carries out the search in *0(m)* time. In contrast, the suffix tree *T* is of size *0(m),* takes *0(m)* time to build, and is used to search in *0(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 hut 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 frees when the text is fixed and the set of patterns vary is left to the wader.

There is one way that suffix trees arc 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 in 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 – *0(n)* for preprocessing and *0(m)* for search. This is the reverse of the bounds shown above for suffix trees. The time/space tradeoff 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 watched 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 stings in the database, denoted by *m*, is assumed to be large. What constitutes a good data structure and lockup 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 *T* for the strings in the database is built in *0(m)* time and, more importantly, requires only *0(m)* space. Any single string *S* of length *n* is found in the database, or declared not to be there, in *0(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 *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 sting *S* cannot be matched against a path in *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 I and 2 is a substring common to both S_1 and S_2, and the longest such sting 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 I 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 7.41. *The longest common substring of two strings con 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, 218]. We will return to this problem in Section V.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.

75. 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 *1*. These substrings are candidates for unwanted pieces of S_2 that have contaminated the desired DNA sting.

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 strings 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 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 *P*. All marked modes of string-depth *l* or more identify suspicions 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 fu1 patterns from *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 sting.

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 *1(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	$1(k)$	one substring
2	4	sand
3	3	and
4	3	and
5	2	an

Surprisingly, the problem can be solved in linear, $0(n)$, 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 $0(n)$ 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 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 T has only one string identifier

Definition For every internal node v of 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 u encountered with $C(v) = k$. (When encountering a node v with $C(v) = k$, compare the string-dept 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 $in)$ Essentially, V(k) reports the length of the longest string that occurs *exactly k* times. Therefore, $V(k) \leq 1(k)$. 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 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 $0(n)$ time. Therefore, instead of counting the number of leaves below v, the algorithm uses $O(n)$ 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 *0(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 *0(n)* time solution will be presented.