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 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 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

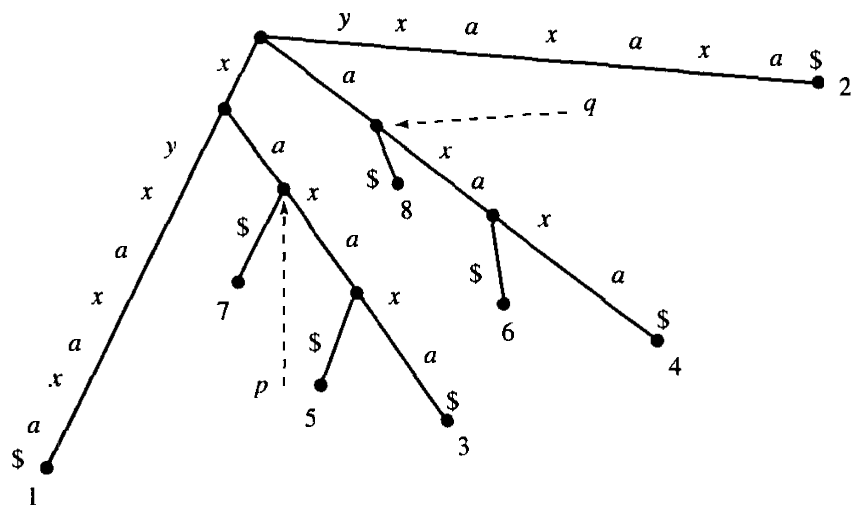

Figure 7.1: Suffix tree for string *xyxaxaxa* without suffix links shown.

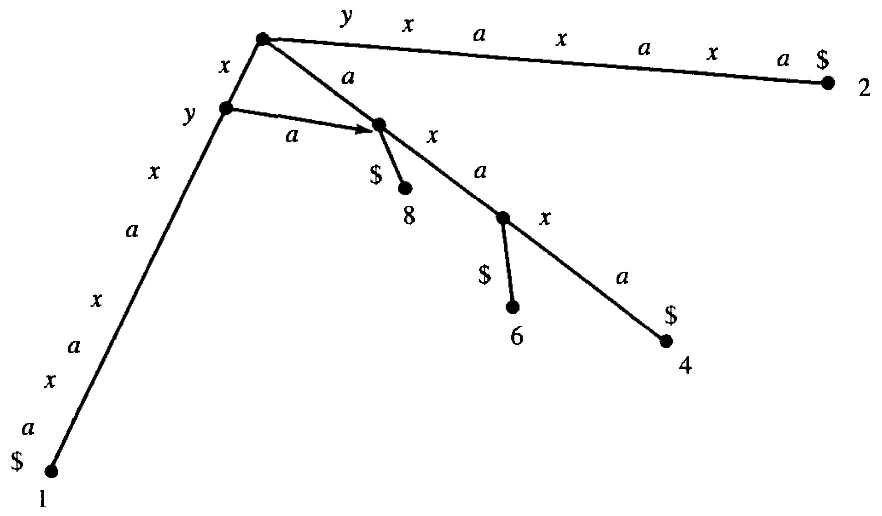

Figure 7.2: A directed acyclic graph used to recognize substrings of *xyxaxaxa*.

be used to solve the exact matching problem in the same way a suffix tree is used. The algorithm matches characters of the pattern against a unique path from the root of the graph; the pattern occurs somewhere in the text if and only if all the characters of the pattern are matched along the path. However, the leaf numbers reachable from the end of the path may no longer give the exact starting positions of the occurrences. This issue will be addressed in Exercise 10.

Since the graph is a DAG after the first merge, the algorithm must know how to merge nodes in a DAG as well as in a tree. The general merge operation for both trees and DAGs is stated in the following way:

A merge of node p *into* node q means that all edges out of p are removed, that the edges into p are directed to q but have their original respective edge-labels, and that any part of the graph that is now unreachable from the root is removed.

Although the merges generally occur in a DAG, the criteria used to determine which nodes to merge remain tied to the original suffix tree – node p can be merged into q if the edge-labeled subtree of p is isomorphic to the edge-labeled subtree of q in the suffix tree. Moreover, p can be merged into q, or q into p, only if the two subtrees are isomorphic. So the key algorithmic issue is how to find isomorphic subtrees in the suffix tree. There are general algorithms for subtree isomorphism but suffix trees have additional structure making isomorphism detection much simpler.

Theorem 7.7.1. *In a suffix tree $\mathcal{T}$ the edge-labeled subtree below a node p is isomorphic to the subtree below a node q if and only if there is a directed path of suffix links from one node to the other node, and the number of leaves in the two subtrees is equal.*

PROOF First suppose p has a direct suffix link to q and those two nodes have the same number of leaves in their subtrees. Since there is a suffix link from p to q, node p has path-label $x\alpha$ while q has path-label α. For every leaf numbered i in the subtree of p there is a leaf numbered $i+1$ in the subtree of q, since the suffix of T starting at i begins with $x\alpha$ only if the suffix of T starting at $i+1$ begins with α. Therefore, for every (labeled) path from p to a leaf in its subtree, there is an identical path (with the same labeled edges) from q to a leaf in its subtree. Now the numbers of leaves in the subtrees of p and q are assumed to be equal, so *every* path out of q is identical to some path out of p, and hence the two subtrees are isomorphic.

By the same reasoning, if there is a path of suffix links from p to q going through a node v, then the number of leaves in the subtree of v must be at least as large as the number in the subtree of p and no larger than the number in the subtree of q. It follows that if p and q have the same number of leaves in their subtrees, then all the subtrees below nodes on the path have the same number of leaves, and all these subtrees are isomorphic to each other.

For the converse side, suppose that the subtrees of p and q are isomorphic. Clearly then they have the same number of leaves. We will show that there is a directed path of suffix links between p and q. Let α be the path-label of p and β be the path-label of q and assume that $|\beta| \leq |\alpha|$.

Since $\beta \neq \alpha$, if β is a suffix of α it must be a proper suffix. And, if β is a proper suffix of α, then by the properties of suffix links, there is a directed path of suffix links from p to q, and the theorem would be proved. So we will prove, by contradiction, that β must be a suffix of α.

Suppose β is not a suffix of α. Consider any occurrence of α in T and let γ be the suffix of T just to the right of that occurrence of α. That means that $\alpha\gamma$ is a suffix of T and there is a path labeled γ running from node p to a leaf in the suffix tree. Now since β is not a suffix of α, no suffix of T that starts just after an occurrence of β can have length $|\gamma|$, and therefore there is no path of length $|\gamma|$ from q to a leaf. But that implies that the subtrees rooted at p and at q are not isomorphic, which is a contradiction. □

Definition Let Q be the set of all pairs (p, q) such that a) there exists a suffix link from p to q in $\mathcal{T}$, and b) p and q have the same number of leaves in their respective subtrees.

The entire procedure to compact a suffix tree can now be described.

Suffix tree compaction

```
begin

Identify the set $Q$ of pairs $(p, q)$ such that there is a suffix link from $p$ to $q$ and the
number of leaves in their respective subtrees is equal.

    While there is a pair $(p, q)$ in $Q$ and both $p$ and $q$ are in the current DAG,
    Merge node $p$ into $q$.

end.
```

The "correctness" of the resulting DAG is stated formally in the following theorem.

Theorem 7.7.2. *Let $\mathcal{T}$ be the suffix tree for an input string S, and let D be the DAG resulting from running the compaction algorithm on $\mathcal{T}$. Any directed path in D from the root enumerates a substring of S, and every substring of S is enumerated by some such path. Therefore, the problem of determining whether a string is a substring of S can be solved in linear time using D instead of $\mathcal{T}$.*

DAG D can be used to determine whether a pattern occurs in a text, but the graph seems to lose the location(s) where the pattern begins. It is possible, however, to add simple (linear-space) information to the graph so that the locations of all the occurrences can also be recovered when the graph is traversed. We address this issue in Exercise 10.

It may be surprising that, in the algorithm, pairs are merged in arbitrary order. We leave the correctness of this, a necessary part of the proof of Theorem 7.7.2, as an exercise. As a practical matter it makes sense to merge top-down, never merging two nodes that have ancestors in the suffix tree that can be merged.

DAGs versus DAWGs

DAG D created by the algorithm is not a DAWG as defined in [70], [71], and [115]. A DAWG represents a finite-state machine and, as such, each edge label is allowed to have only one character. Moreover, the main theoretical feature of the DAWG for a string S is that it is the finite-state machine with the fewest number of states (nodes) that recognizes suffixes of S. Of course, D can be converted to a finite-state machine by expanding any edge of D whose label has k characters into k edges labeled by one character each. But the resulting finite-state machine would not necessarily have the minimum number of states, and hence it would not necessarily be the DAWG for S.

Still, DAG D for string S has as few (or fewer) nodes and edges than does the associated DAWG for S, and so is as compact as the DAWG even though it may not be a finite-state machine. Therefore, construction of the DAWG for S is mostly of theoretical interest. In Exercises 16 and 17 we consider how to build the smallest finite-state machine that recognizes substrings of a string.

7.8. APL8: A reverse role for suffix trees, and major space reduction

We have previously shown how suffix trees can be used to solve the exact matching problem with $O(m)$ preprocessing time and space (building a suffix tree of size $O(m)$ for the text T) and $O(n+k)$ search time (where n is the length of the pattern and k is the number of occurrences). We have also seen how suffix trees are used to solve the exact set matching problem in the same time and space bounds (n is now the total size of all the patterns in the set). In contrast, the Knuth-Morris-Pratt (or Boyer–Moore) method preprocesses the pattern in $O(n)$ time and space, and then searches in $O(m)$ time. The Aho–Corasick method achieves similar bounds for the set matching problem.

Asymptotically, the suffix tree methods that preprocess the text are as efficient as the methods that preprocess the pattern – both run in $O(n+m)$ time and use $\Theta(n+m)$ space (they have to represent the strings). However, the practical constants on the time and space bounds for suffix trees often make their use unattractive compared to the other methods. Moreover, the situation sometimes arises that the pattern(s) will be given first and held fixed while the text varies. In those cases it is clearly superior to preprocess the pattern(s). So the question arises of whether we can solve those problems by building a suffix tree for the *pattern(s)*, not the text. This is the reverse of the normal use of suffix trees. In Sections 5.3 and 7.2.1 we mentioned that such a reverse role was possible, thereby using suffix trees to achieve exactly the same time and space bounds (preprocessing versus search time and space) as in the Knuth-Morris-Pratt or Aho–Corasick methods. To explain this, we will develop a result due to Chang and Lawler [94], who solved a somewhat more general problem, called the *matching statistics* problem.

7.8.1. Matching statistics: duplicating bounds and reducing space

Definition Define $ms(i)$ to be the length of the longest substring of T starting at position i that matches a substring *somewhere* (but we don't know where) in P. These values are called the *matching statistics*.

For example, if $T = abcxabcdex$ and $P = wyabcwzqabcdw$ then $ms(1) = 3$ and $ms(5) = 4$.

Clearly, there is an occurrence of P starting at position i of T if and only if $ms(i) = |P|$.

Thus the problem of finding the matching statistics is a generalization of the exact matching problem.

Matching statistics lead to space reduction

Matching statistics can be used to reduce the size of the suffix tree needed in solutions to problems more complex than exact matching. This use of matching statistics will probably be more important than their use to duplicate the preprocessing/search bounds of Knuth-Morris-Pratt and Aho–Corasick. The first example of space reduction using matching statistics will be given in Section 7.9.

Matching statistics are also used in a variety of other applications described in the book. One advertisement we give here is to say that matching statistics are central to a fast approximate matching method designed for rapid database searching. This will be detailed in Section 12.3.3. Thus matching statistics provide one bridge between exact matching methods and problems of approximate string matching.

How to compute matching statistics

We want to compute $ms(i)$, for each position i in T, in $O(m)$ time using only a suffix tree for P. First, build a suffix tree $\mathcal{T}$ for P, the fixed short string, but do not remove the suffix links used during the construction of the tree. (The suffix links are either constructed by Ukkonen's algorithm or are the reverse of the link pointers in Weiner's algorithm.) This suffix tree will then be used to find $ms(i)$ for each position i in T.

The naive way to find a single $ms(i)$ value is to match, left to right, the initial characters of $T[i..m]$ against $\mathcal{T}$, by following the unique path of matches until no further matches are possible. However, repeating this for each i would not achieve the claimed linear time bound. Instead, the suffix links are used to accelerate the entire computation, similar to the way they accelerate the construction of $\mathcal{T}$ in Ukkonen's algorithm.

To learn $ms(1)$, we match characters of string T against $\mathcal{T}$, by following the unique matching path of $T[1..m]$. The length of that matching path is $ms(1)$. Now suppose in general that the algorithm has just followed a matching path to learn $ms(i)$ for $i < |m|$. That means that the algorithm has located a point b in $\mathcal{T}$ such that the path to that point exactly matches a prefix of $T[i..m]$, but no further matches are possible (possibly because a leaf has been reached).

Having learned $ms(i)$, proceed as follows to learn $ms(i + 1)$. If b is an internal node v of $\mathcal{T}$ then the algorithm can follow its suffix link to a node $s(v)$. If b is not an internal node, then the algorithm can back up to the node v just above b. If v is the root, then the search for $ms(i + 1)$ begins at the root. But if v is not the root, then the algorithm follows the suffix link from v to $s(v)$. The path-label of v, say $x\alpha$, is a prefix of $T[i..m]$, so α must be a prefix of $T[i + 1..m]$. But $s(v)$ has path-label α, and hence the path from the root to $s(v)$ matches a prefix of $T[i + 1..m]$. Therefore, the search for $ms(i + 1)$ can start at node $s(v)$ rather than at the root.

Let β denote the string between node v and point b. Then $x\alpha\beta$ is the longest substring in P that matches a substring starting at position i of T. Hence $\alpha\beta$ is a string in P matching a substring starting at position $i + 1$ of T. Since $s(v)$ has path-label α, there must be a path labeled β out of $s(v)$. Instead of traversing that path by examining every character on it, the algorithm uses the skip/count trick (detailed in Ukkonen's algorithm; Section 6.1.3) to traverse it in time proportional to the number of nodes on the path.

When the end of that β path is reached, the algorithm continues to match single characters from T against characters in the tree until either a leaf is reached or until no further matches are possible. In either case, $ms(i+1)$ is the string-depth of the ending position. Note that the character comparisons done after reaching the end of the β path begin either with the same character in T that ended the search for $ms(i)$ or with the next character in T, depending on whether that search ended with a mismatch or at a leaf.

There is one special case that can arise in computing $ms(i+1)$. If $ms(i) = 1$ or $ms(i) = 0$ (so that the algorithm is at the root), and $T(i+1)$ is not in P, then $ms(i+1) = 0$.

7.8.2. Correctness and time analysis for matching statistics

The proof of correctness of the method is immediate since it merely simulates the naive method for finding each $ms(i)$. Now consider the time required by the algorithm. The analysis is very similar to that done for Ukkonen's algorithm.

Theorem 7.8.1. *Using only a suffix tree for P and a copy of T, all the m matching statistics can be found in $O(m)$ time.*

PROOF The search for any $ms(i+1)$ begins by backing up at most one edge from position b to a node v and traversing one suffix link to node $s(v)$. From $s(v)$ a β path is traversed in time proportional to the number of nodes on it, and then a certain number of additional character comparisons are done. The backup and link traversals take constant time per i and so take $O(m)$ time over the entire algorithm. To bound the total time to traverse the various β paths, recall the notion of *current node-depth* from the time analysis of Ukkonen's algorithm (page 102). There it was proved that a link traversal reduces the current depth by at most one (Lemma 6.1.2), and since each backup reduces the current depth by one, the total decrements to current depth cannot exceed $2m$. But since current depth cannot exceed m or become negative, the total increments to current depth are bounded by $3m$. Therefore, the total time used for all the β traversals is at most $3m$ since the current depth is increased at each step of any β traversal. It only remains to consider the total time used in all the character comparisons done in the "after-β" traversals. The key there is that the after-β character comparisons needed to compute $ms(i+1)$, for $i \geq 1$, begin with the character in T that ended the computation for $ms(i)$ or with the next character in T. Hence the after-β comparisons performed when computing $ms(i)$ and $ms(i+1)$ share at most one character in common. It follows that at most $2m$ comparisons in total are performed during all the after-β comparisons. That takes care of all the work done in finding all the matching statistics, and the theorem is proved. □

7.8.3. A small but important extension

The number $ms(i)$ indicates the length of the longest substring starting at position i of T that matches a substring *somewhere* in P, but it does not indicate the location of any such match in P. For some applications (such as those in Section 9.1.2) we must also know, for each i, the location of at least one such matching substring. We next modify the matching statistics algorithm so that it provides that information.

Definition For each position i in T, the number $p(i)$ specifies a starting location in P such that the substring starting at $p(i)$ matches a substring starting at position i of T for exactly $ms(i)$ places.

In order to accumulate the $p(i)$ values, first do a depth-first traversal of $\mathcal{T}$ marking each node v with the leaf number of one of the leaves in its subtree. This takes time linear in the size of $\mathcal{T}$. Then, when using $\mathcal{T}$ to find each $ms(i)$, if the search stops at a node u, the desired $p(i)$ is the suffix number written at u; otherwise (when the search stops on an edge (u, v)), $p(i)$ is the suffix number written at node v.

Back to STSs

Recall the discussion of STSs in Section 3.5.1. There it was mentioned that, because of errors, exact matching may not be an appropriate way to find STSs in new sequences. But since the number of sequencing errors is generally small, we can expect long regions of agreement between a new DNA sequence and any STS it (ideally) contains. Those regions of agreement should allow the correct identification of the STSs it contains. Using a (precomputed) generalized suffix tree for the STSs (which play the role of P), compute matching statistics for the new DNA sequence (which is T) and the set of STSs. Generally, the pointer $p(i)$ will point to the appropriate STS in the suffix tree. We leave it to the reader to flesh out the details. Note that when given a new sequence, the time for the computation is just proportional to the length of the new sequence.

7.9. APL9: Space-efficient longest common substring algorithm

In Section 7.4, we solved the problem of finding the longest common substring of S_1 and S_2 by building a generalized suffix tree for the two strings. That solution used $O(|S_1| + |S_2|)$ time and space. But because of the practical space overhead required to construct and use a suffix tree, a solution that builds a suffix tree only for the smaller of the two strings may be much more desirable, even if the worst-case space bounds remain the same. Clearly, the longest common substring has length equal to the longest matching statistic $ms(i)$. The actual substring occurs in the longer string starting at position i and in the shorter string starting at position $p(i)$. The algorithm of the previous section computes all the $ms(i)$ and $p(i)$ values using only a suffix tree for the smaller of the two strings, along with a copy of the long string. Hence, the use of matching statistics reduces the space needed to solve the longest common substring problem.

The longest common substring problem illustrates one of many space reducing applications of matching statistics to algorithms using suffix trees. Some additional applications will be mentioned in the book, but many more are possible and we will not explicitly point each one out. The reader is encouraged to examine every use of suffix trees involving more than one string, to find those places where such space reduction is possible.

7.10. APL10: All-pairs suffix-prefix matching

Here we present a more complex use of suffix trees that is interesting in its own right and that will be central in the linear-time superstring approximation algorithm to be discussed in Section 16.17.

Definition Given two strings S_i and S_j, any suffix of S_i that matches a prefix of S_j is called a *suffix-prefix match* of S_i, S_j.

Given a collection of strings $\mathcal{S} = S_1, S_2, \ldots, S_k$ of total length m, the **all-pairs suffix-prefix problem** is the problem of finding, for each ordered pair S_i, S_j in $\mathcal{S}$, the *longest* suffix-prefix match of S_i, S_j.

Motivation for the problem

The main motivation for the all-pairs suffix-prefix problem comes from its use in implementing fast approximation algorithms for the *shortest superstring problem* (to be discussed in Section 16.17). The superstring problem is itself motivated by sequencing and mapping problems in DNA that will be discussed in Chapter 16. Another motivation for the shortest superstring problem, and hence for the all-pairs suffix-prefix problem, arises in data compression; this connection will be discussed in the exercises for Chapter 16.

A different, direct application of the all-pairs suffix-prefix problem is suggested by computations reported in [190]. In that research, a set of around 1,400 ESTs (see Section 3.5.1) from the organism *C. elegans* (which is a worm) were analyzed for the presence of highly conserved substrings called *ancient conserved regions* (*ACR*s). One of the main objectives of the research was to estimate the number of ACRs that occur in the genes of *C. elegans*. Their approach was to extrapolate from the number of ACRs they observed in the set of ESTs. To describe the role of suffix-prefix matching in this extrapolation, we need to remember some facts about ESTs.

For the purposes here, we can think of an EST as a sequenced DNA substring of length around 300 nucleotides, originating in a gene of much greater length. If EST α originates in gene β, then the actual location of substring α in β is essentially random, and many different ESTs can be collected from the same gene β. However, in the common method used to collect ESTs, one does not learn the identity of the originating gene, and it is not easy to tell if two ESTs originate from the same gene. Moreover, ESTs are collected more frequently from some genes than others. Commonly, ESTs will more frequently be collected from genes that are more highly expressed (transcribed) than from genes that are less frequently expressed. We can thus consider ESTs as a biased sampling of the underlying gene sequences. Now we return to the extrapolation problem.

The goal is to use the ACR data observed in the ESTs to estimate the number of ACRs in the entire set of genes. A simple extrapolation would be justified if the ESTs were essentially random samples selected uniformly from the entire set of *C. elegans* genes. However, genes are not uniformly sampled, so a simple extrapolation would be wrong if the prevalence of ACRs is systematically different in ESTs from frequently or infrequently expressed genes. How can that prevalence be determined? When an EST is obtained, one doesn't know the gene it comes from, or how frequently that gene is expressed, so how can ESTs from frequently sampled genes be distinguished from the others?

The approach taken in [190] is to compute the "overlap" between each pair of ESTs. Since all the ESTs are of comparable length, the heart of that computation consists of solving the all-pairs suffix-prefix problem on the set of ESTs. An EST that has no substantial overlap with another EST was considered in the study to be from an infrequently expressed (and sampled) gene, whereas an EST that has substantial overlap with one or more of the other ESTs is considered to be from a frequently expressed gene. (Because there may be some sequencing errors, and because substring containment is possible among strings of unequal length, one should also solve the all-pairs longest common substring problem.) After categorizing the ESTs in this way, it was indeed found that ACRs occur more commonly in ESTs from frequently expressed genes (more precisely, from ESTs that overlap other ESTs). To explain this, the authors [190] conclude:

> These results suggest that moderately expressed proteins have, on average, been more highly conserved in sequence over long evolutionary periods than have rarely expressed ones and in particular are more likely to contain ACRs. This is presumably attributable in part to higher selective pressures to optimize the activities and structures of those proteins ...

7.10.1. Solving the all-pairs suffix-prefix problem in linear time

For a single pair of strings, the preprocessing discussed in Section 2.2.4 will find the longest suffix-prefix match in time linear in the length of the two strings. However, applying the preprocessing to each of the k^2 pairs of strings separately gives a total bound of $O(km)$ time. Using suffix trees it is possible to reduce the computation time to $O(m+k^2)$, assuming (as usual) that the alphabet is fixed.

Definition We call an edge a *terminal edge* if it is labeled only with a string termination symbol. Clearly, every terminal edge has a leaf at one end, but not all edges touching leaves are terminal edges.

The main data structure used to solve the all-pairs suffix-prefix problem is the generalized suffix tree $\mathcal{T}(\mathcal{S})$ for the k strings in set $\mathcal{S}$. As $\mathcal{T}(\mathcal{S})$ is constructed, the algorithm also builds a list $L(v)$ for each internal node v. List $L(v)$ contains the index i if and only if v is incident with a terminal edge whose leaf is labeled by a suffix of string S_i. That is, $L(v)$ holds index i if and only if the path label to v is a complete suffix of string S_i. For example, consider the generalized suffix tree shown in Figure 6.11 (page 117). The node with path-label ba has an L list consisting of the single index 2, the node with path-label a has a list consisting of indices 1 and 2, and the node with path-label xa has a list consisting of index 1. All the other lists in this example are empty. Clearly, the lists can be constructed in linear time during (or after) the construction of $\mathcal{T}(\mathcal{S})$.

Now consider a fixed string S_j, and focus on the path from the root of $\mathcal{T}(\mathcal{S})$ to the leaf j representing the entire string S_j. The key observation is the following: If v is a node on this path and i is in $L(v)$, then the path-label of v is a suffix of S_i that matches a prefix of S_j. So for each index i, the deepest node v on the path to leaf j such that $i \in L(v)$ identifies the longest match between a suffix of S_i and a prefix of S_j. The path-label of v is the longest suffix-prefix match of (S_i, S_j). It is easy to see that by one traversal from the root to leaf j we can find the deepest nodes for all $1 \leq i \leq k$ $(i \neq j)$.

Following the above observation, the algorithm efficiently collects the needed suffix-prefix matches by traversing $\mathcal{T}(\mathcal{S})$ in a depth-first manner. As it does, it maintains k stacks, one for each string. During the depth-first traversal, when a node v is reached in a forward edge traversal, push v onto the ith stack, for each $i \in L(v)$. When a leaf j (representing the entire string S_j) is reached, scan the k stacks and record for each index i the current top of the ith stack. It is not difficult to see that the top of stack i contains the node v that defines the suffix-prefix match of (S_i, S_j). If the ith stack is empty, then there is no overlap between a suffix of string S_i and a prefix of string S_j. When the depth-first traversal backs up past a node v, we pop the top of any stack whose index is in $L(v)$.

Theorem 7.10.1. *All the k^2 longest suffix-prefix matches are found in $O(m+k^2)$ time by the algorithm. Since m is the size of the input and k^2 is the size of the output, the algorithm is time optimal.*

PROOF The total number of indices in all the lists $L(v)$ is $O(m)$. The number of edges in $\mathcal{T}(\mathcal{S})$ is also $O(m)$. Each push or pop of a stack is associated with a leaf of $\mathcal{T}(\mathcal{S})$, and each leaf is associated with at most one pop and one push; hence traversing $\mathcal{T}(\mathcal{S})$ and updating the stacks takes $O(m)$ time. Recording of each of the $O(k^2)$ answers is done in constant time per answer. □

Extensions

We note two extensions. Let $k' \leq k^2$ be the number of ordered pairs of strings that have a nonzero length suffix-prefix match. By using double links, we can maintain a linked list of the *nonempty* stacks. Then when a leaf of the tree is reached during the traversal, only the stacks on this list need be examined. In that way, all nonzero length suffix-prefix matches can be found in $O(m + k')$ time. Note that the position of the stacks in the linked list will vary, since a stack that goes from empty to nonempty must be linked at one of the ends of the list; hence we must also keep (in the stack) the name of the string associated with that stack.

At the other extreme, suppose we want to collect for every pair not just the longest suffix-prefix match, but all suffix-prefix matches no matter how long they are. We modify the above solution so that when the tops of the stacks are scanned, the entire contents of each scanned stack is read out. If the output size is k^*, then the complexity for this solution is $O(m + k^*)$.

7.11. Introduction to repetitive structures in molecular strings

Several sections of this book (Sections 7.12, 7.12.1, 9.2, 9.2.2, 9.5, 9.6, 9.6.1, 9.7, and 7.6), as well as several exercises, are devoted to discussing efficient algorithms for finding various types of *repetitive* structures in strings. (In fact, some aspects of one type of repetitive structure, *tandem repeats*, have already been discussed in the exercises of Chapter 1, and more will be discussed later in the book.) The motivation for the general topic of repetitive structures in strings comes from several sources, but our principal interest is in important repetitive structures seen in biological strings (DNA, RNA, and protein). To make this concrete, we briefly introduce some of those repetitive structures. The intent is not to write a dissertation on repetitive DNA or protein, but to motivate the algorithmic techniques we develop.

7.11.1. Repetitive structures in biological strings

One of the most striking features of DNA (and to a lesser degree, protein) is the extent to which repeated substrings occur in the genome. This is particularly true of eukaryotes (higher-order organisms whose DNA is enclosed in a cell nucleus). For example, most of the human Y chromosome consists of repeated substrings, and overall

> Families of reiterated sequences account for about one third of the human genome. [317]

There is a vast[1] literature on repetitive structures in DNA, and even in protein,

> ...reports of various kinds of repeats are too common even to list. [128]

In an analysis of 3.6 million bases of DNA from *C. elegans*, over 7,000 families of repetitive sequences were identified [5]. In contrast, prokaryotes (organisms such as bacteria whose DNA is not enclosed in a nucleus) have in total little repetitive DNA, although they still possess certain highly structured small-scale repeats.

In addition to its sheer quantity, repetitive DNA is striking for the variety of repeated structures it contains, for the various proposed mechanisms explaining the origin and maintenance of repeats, and for the biological functions that some of the repeats may play (see [394] for one aspect of gene duplication). In many texts (for example, [317], [469], and [315]) on genetics or molecular biology one can find extensive discussions of repetitive strings and their hypothesized functional and evolutionary role. For an introduction to repetitive elements in human DNA, see [253] and [255].

[1] It is reported in [192] that a search of the database MEDLINE using the key (*repeat OR repetitive*) *AND* (*protein OR sequence*) turned up over 6,000 papers published in the preceding twenty years.

5′ TCGACCGGTCGA 3′
5′ TCGACCGGTCGA 3′

Figure 7.3: A palindrome in the vernacular of molecular biology. The double-stranded string is the same after reflection around both the horizontal and vertical midpoints. Each strand is a *complemented palindrome* according to the definitions used in this book.

In the following discussion of repetitive structures in DNA and protein, we divide the structures into three types: local, small-scale repeated strings whose function or origin is at least partially understood; simple repeats, both local and interspersed, whose function is less clear; and more complex interspersed repeated strings whose function is even more in doubt.

Definition A *palindrome* is a string that reads the same backwards as forwards.

For emphasis, the Random House dictionary definition of "palindrome" is: a word, sentence or verse reading the same backwards as forwards [441]. For example, the string *xyaayx* is a palindrome under this definition. Ignoring spaces, the sentence *was it a cat i saw* is another example.

Definition A *complemented palindrome* is a DNA or RNA string that becomes a palindrome if each character in one half of the string is changed to its complement character (in DNA, $A-T$ are complements and $C-G$ are complements; in RNA $A-U$ and $C-G$ are complements). For example, *AGCTCGCGAGCT* is a complemented palindrome.[2]

Small-scale local repeats whose function or origin is partially understood include: *complemented palindromes* in both DNA and RNA, which act to regulate DNA transcription (the two parts of the complemented palindrome fold and pair to form a "hairpin loop"); *nested complemented palindromes* in tRNA (transfer RNA) that allow the molecule to fold up into a cloverleaf structure by complementary base pairing; tandem arrays of repeated RNA that flank retroviruses (viruses whose primary genetic material is RNA) and facilitate the incorporation of viral DNA (produced from the RNA sequence by reverse transcription) into the host's DNA; single copy inverted repeats that flank transposable (movable) DNA in various organisms and that facilitate that movement or the inversion of the DNA orientation; short repeated substrings (both palindromic and nonpalindromic) in DNA that may help the chromosome fold into a more compact structure; repeated substrings at the ends of viral DNA (in a linear state) that allow the concatenation of many copies of the viral DNA (a molecule of this type is called a *concatamer*); copies of genes that code for important RNAs (rRNAs and tRNAs) that must be produced in large number; clustered genes that code for important proteins (such as histone) that regulate chromosome structure and must be made in large number; families of genes that code for similar proteins (hemoglobins and myoglobins for example); similar genes that probably arose through duplication and subsequent mutation (including *pseudogenes* that have mutated

[2] The use of the word "palindrome" in molecular biology does not conform to the normal English dictionary definition of the word. The easiest translation of the molecular biologist's "palindrome" to normal English is: "complemented palindrome". A more molecular view is that a palindrome is a segment of double-stranded DNA or RNA such that both strands read the same when both are read in the same direction, say in the 5′ to 3′ direction. Alternately, a palindrome is a segment of double-stranded DNA that is symmetric (with respect to reflection) around both the horizontal axis and the midpoint of the segment. (See Figure 7.3). Since the two strands are complementary, each strand defines a complemented palindrome in the sense defined above. The term "mirror repeat" is sometimes used in the molecular biology literature to refer to a "palindrome" as defined by the dictionary.

to the point that they no longer function); common exons of eukaryotic DNA that may be basic building blocks of many genes; and common functional or structural subunits in protein (motifs and domains).

Restriction enzyme cutting sites illustrate another type of small-scale, structured, repeating substring of great importance to molecular biology. A restriction enzyme is an enzyme that recognizes a specific substring in the DNA of both prokaryotes and eukaryotes and cuts (or cleaves) the DNA every place where that pattern occurs (exactly where it cuts inside the pattern varies with the pattern). There are hundreds of known restriction enzymes and their use has been absolutely critical in almost all aspects of modern molecular biology and recombinant DNA technology. For example, the surprising discovery that eukaryotic DNA contains *introns* (DNA substrings that interrupt the DNA of protein coding regions), for which Nobel prizes were awarded in 1993, was closely coupled with the discovery and use of restriction enzymes in the late 1970s.

Restriction enzyme cutting sites are interesting examples of repeats because they tend to be *complemented palindromic substrings*. For example, the restriction enzyme *EcoRI* recognizes the complemented palindrome *GAATTC* and cuts between the *G* and the adjoining *A* (the substring *TTC* when reversed and complemented is *GAA*). Other restriction enzymes recognize *separated* (or *interrupted*) complemented palindromes. For example, restriction enzyme *BglI* recognizes *GCCNNNNNGGC*, where *N* stands for any nucleotide. The enzyme cuts between the last two *N*s. The complemented palindromic structure has been postulated to allow the two halves of the complemented palindrome (separated or not) to fold and form complementary pairs. This folding then apparently facilitates either recognition or cutting by the enzyme. Because of the palindromic structure of restriction enzyme cutting sites, people have scanned DNA databases looking for common repeats of this form in order to find additional candidates for unknown restriction enzyme cutting sites.

Simple repeats that are less well understood often arise as *tandem arrays* (consecutive repeated strings, also called "direct repeats") of repeated DNA. For example, the string *TTAGGG* appears at the ends of every human chromosome in arrays containing one to two thousand copies [332]. Some tandem arrays may originate and continue to grow by a postulated mechanism of *unequal crossing over* in meiosis, although there is serious opposition to that theory. With unequal crossing over in meiosis, the likelihood that more copies will be added in a single meiosis increases as the number of existing copies increases. A number of genetic diseases (Fragile X syndrome, Huntington's disease, Kennedy's disease, myotonic dystrophy, ataxia) are now understood to be caused by increasing numbers of tandem DNA repeats of a string three bases long. These triplet repeats somehow interfere with the proper production of particular proteins. Moreover, the number of triples in the repeat increases with successive generations, which appears to explain why the disease increases in severity with each generation. Other long tandem arrays consisting of short strings are very common and are widely distributed in the genomes of mammals. These repeats are called *satellite DNA* (further subdivided into micro and mini-satellite DNA), and their existence has been heavily exploited in genetic mapping and forensics. Highly dispersed tandem arrays of length-two strings are common. In addition to tri-nucleotide repeats, other mini-satellite repeats also play a role in human genetic diseases [286].

Repetitive DNA that is *interspersed* throughout mammalian genomes, and whose function and origin is less clear, is generally divided into SINEs (short interspersed nuclear sequences) and LINEs (long interspersed nuclear sequences). The classic example of a SINE is the *Alu* family. The Alu repeats occur about 300,000 times in the human genome and account for as much as 5% of the DNA of human and other mammalian genomes. Alu repeats are substrings of length around 300 nucleotides and occur as nearly (but not exactly) identical copies widely dispersed throughout the genome. Moreover, the interior of an Alu string itself consists of repeated substrings of length around 40, and the Alu sequence is often flanked on either side by tandem repeats of length 7–10. Those right and left flanking sequences are usually complemented palindromic copies of each other. So the Alu repeats wonderfully illustrate various kinds of phenomena that occur in repetitive DNA. For an introduction to Alu repeats see [254].

One of the most fascinating discoveries in molecular genetics is a phenomenon called *genomic* (or *gametic*) *imprinting*, whereby a particular allele of a gene is expressed only when it is inherited from one specific parent [48, 227, 391]. Sometimes the required parent is the mother and sometimes the father. The allele will be unexpressed, or expressed differently, if inherited from the "incorrect" parent. This is in contradiction to the classic Mendelian rule of *equivalence* – that chromosomes (other than the Y chromosome) have no memory of the parent they originated from, and that the same allele inherited from either parent will have the same effect on the child. In mice and humans, sixteen imprinted gene alleles have been found to date [48]. Five of these require inheritance from the mother, and the rest from the father. The DNA sequences of these sixteen imprinted genes all share the common feature that

> They contain, or are closely associated with, a region rich in direct repeats. These repeats range in size from 25 to 120 bp,[3] are unique to the respective imprinted regions, but have no obvious homology to each other or to highly repetitive mammalian sequences. The direct repeats may be an important feature of gametic imprinting, as they have been found in all imprinted genes analyzed to date, and are also evolutionarily conserved. [48]

Thus, direct repeats seem to be important in genetic imprinting, but like many other examples of repetitive DNA, the function and origin of these repeats remains a mystery.

7.11.2. Uses of repetitive structures in molecular biology

At one point, most interspersed repeated DNA was considered as a nuisance, perhaps of no functional or experimental value. But today a variety of techniques actually exploit the existence of repetitive DNA. Genetic mapping, mentioned earlier, requires the identification of features (or *markers*) in the DNA that are highly variable between individuals and that are interspersed frequently throughout the genome. Tandem repeats are just such markers. What varies between individuals is the *number* of times the substring repeats in an array. Hence the term used for this type of marker is *variable number of tandem repeats* (*VNTR*). *VNTRs* occur frequently and regularly in many genomes, including the human genome, and provide many of the markers needed for large-scale genetic mapping. These *VNTR* markers are used during the genetic-level (as opposed to the physical-level) search for specific defective genes and in forensic DNA fingerprinting (since the number of repeats is highly variable between individuals, a small set of *VNTRs* can uniquely characterize individuals in a population). Tandem repeats consisting of a very short substring, often only two characters long, are called *microsatellites* and have become the preferred marker in many genetic mapping efforts.

[3] A detail not contained in this quote is that the direct (tandem) repeats in the genes studied [48] have a total length of about 1,500 bases.

The existence of highly repetitive DNA, such as Alus, makes certain kinds of large-scale DNA sequencing more difficult (see Sections 16.11 and 16.16), but their existence can also facilitate certain cloning, mapping, and searching efforts. For example, one general approach to low-resolution physical mapping (finding on a true physical scale where features of interest are located in the genome) or to finding genes causing diseases involves inserting pieces of human DNA that may contain a feature of interest into the hamster genome. This technique is called *somatic cell hybridization*. Each resulting hybrid-hamster cell incorporates different parts of the human DNA, and these hybrid cells can be tested to identify a specific cell containing the human feature of interest. In this cell, one then has to identify the parts of the hamster's hybrid genome that are human. But what is a distinguishing feature between human and hamster DNA?

One approach exploits the Alu sequences. Alu sequences specific to human DNA are so common in the human genome that most fragments of human DNA longer than 20,000 bases will contain an Alu sequence [317]. Therefore, the fragments of human DNA in the hybrid can be identified by probing the hybrid for fragments of Alu. The same idea is used to isolate human *oncogenes* (modified growth-promoting genes that facilitate certain cancers) from human tumors. Fragments of human DNA from the tumor are first transferred to mouse cells. Cells that receive the fragment of human DNA containing the oncogene become transformed and replicate faster than cells that do not. This isolates the human DNA fragment containing the oncogene from the other human fragments, but then the human DNA has to be separated from the mouse DNA. The proximity of the oncogene to an Alu sequence is again used to identify the human part of the hybrid genome [471]. A related technique, again using proximity to Alu sequences, is described in [403].

Algorithmic problems on repeated structures

We consider specific problems concerning repeated structures in strings in several sections of the book.[4] Admittedly, not every repetitive string problem that we will discuss is perfectly motivated by a biological problem or phenomenon known today. A recurring objection is that the first repetitive string problems we consider concern *exact* repeats (although with complementation and inversion allowed), whereas most cases of repetitive DNA involve *nearly* identical copies. Some techniques for handling inexact palindromes (complemented or not) and inexact repeats will be considered in Sections 9.5 and 9.6. Techniques that handle more liberal errors will be considered later in the book. Another objection is that simple techniques suffice for small-length repeats. For example, if one seeks repeating DNA of length ten, it makes sense to first build a table of all the 4^{10} possible strings and then scan the target DNA with a length-ten template, hashing substring locations into the precomputed table.

Despite these objections, the fit of the computational problems we will discuss to biological phenomena is good enough to motivate sophisticated techniques for handling exact or nearly exact repetitions. Those techniques pass the "plausibility" test in that they, or the ideas that underlie them, may be of future use in computational biology. In this light, we now consider problems concerning exactly repeated substrings in a single string.

[4] In a sense, the longest common substring problem and the k-common substring problem (Sections 7.6 and 9.7) also concern repetitive substrings. However, the repeats in those problems occur across distinct strings, rather than inside the same string. That distinction is critical, both in the definition of the problems and for the techniques used to solve them.

7.12. APL11: Finding all maximal repetitive structures in linear time

Before developing algorithms for finding repetitive structures, we must carefully define those structures. A poor definition may lead to an avalanche of output. For example, if a string consists of n copies of the same character, an algorithm searching for all pairs of identical substrings (an initially reasonable definition of a repetitive structure) would output $\Theta(n^4)$ pairs, an undesirable result. Other poor definitions may not capture the structures of interest, or they may make reasoning about those structures difficult. Poor definitions are particularly confusing when dealing with the set of all repeats of a particular type. Accordingly, the key problem is to define repetitive structures in a way that does not generate overwhelming output and yet captures all the meaningful phenomena in a clear way. In this section, we address the issue through various notions of *maximality*. Other ways of defining and studying repetitive structures are addressed in Exercises 56, 57, and 58 in this chapter; in exercises in other chapters; and in Sections 9.5, 9.6, and 9.6.1.

Definition A *maximal pair* (or a maximal repeated pair) in a string S is a pair of identical substrings α and β in S such that the character to the immediate left (right) of α is different from the character to the immediate left (right) of β. That is, extending α and β in either direction would destroy the equality of the two strings.

Definition A maximal pair is represented by the triple (p_1, p_2, n'), where p_1 and p_2 give the starting positions of the two substrings and n' gives their length. For a string S, we define $\mathcal{R}(S)$ to be the set of all triples describing maximal pairs in S.

For example, consider the string $S = xabcyiiizabcqabcyrxar$, where there are three occurrences of the substring abc. The first and second occurrences of abc form a maximal pair (2, 10, 3), and the second and third occurrences also form a maximal pair (10, 14, 3), whereas the first and third occurrences of abc do not form a maximal pair. The two occurrences of the string $abcy$ also form a maximal pair (2, 14, 4). Note that the definition allows the two substrings in a maximal pair to overlap each other. For example, $cxxaxxaxxb$ contains a maximal pair whose substring is $xxaxx$.

Generally, we also want to permit a prefix or a suffix of S to be part of a maximal pair. For example, two occurrences of xa in $xabcyiiizabcqabcyrxar$ should be considered as a maximal pair. To model this case, simply add a character to the start of S and one to the end of S that appear nowhere else in S. From this point on, we will assume that has been done.

It may sometimes be of interest to explicitly find and output the full set $\mathcal{R}(S)$. However, in some situations $\mathcal{R}(S)$ may be too large to be of use, and a more restricted reflection of the maximal pairs may be sufficient or even preferred.

Definition Define a *maximal repeat* α as a *substring* of S that occurs in a maximal pair in S. That is, α is a *maximal repeat* in S if there is a triple $(p_1, p_2, |\alpha|) \in \mathcal{R}(S)$ and α occurs in S starting at position p_1 and p_2. Let $\mathcal{R}'(S)$ denote the set of maximal repeats in S.

For example, with S as above, both strings abc and $abcy$ are maximal repeats. Note that no matter how many times a string participates in a maximal pair in S, it is represented only once in $\mathcal{R}'(S)$. Hence $|\mathcal{R}'(S)|$ is less than or equal to $|\mathcal{R}(S)|$ and is generally much smaller. The output is more modest, and yet it gives a good reflection of the maximal pairs.

In some applications, the definition of a maximal repeat does not properly model the desired notion of a repetitive structure. For example, in $S = a\alpha bx\alpha ya\alpha b$, substring α is a maximal repeat but so is $a\alpha b$, which is a *superstring* of string α, although not every occurrence of α is contained in that superstring. It may not always be desirable to report α as a repetitive structure, since the larger substring $a\alpha b$ that sometimes contains α may be more informative.

Definition A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat.

Maximal pairs, maximal repeats, and supermaximal repeats are only three possible ways to define exact repetitive structures of interest. Other models of exact repeats are given in the exercises. Problems related to palindromes and tandem repeats are considered in several sections throughout the book. Inexact repeats will be considered in Sections 9.5 and 9.6.1. Certain kinds of repeats are elegantly represented in graphical form in a device called a *landscape* [104]. An efficient program to construct the landscape, based essentially on suffix trees, is also described in that paper. In the next sections we detail how to efficiently find all maximal pairs, maximal repeats, and supermaximal repeats.

7.12.1. A linear-time algorithm to find all maximal repeats

The simplest problem is that of finding all maximal repeats. Using a suffix tree, it is possible to find them in $O(n)$ time for a string of length n. Moreover, there is a *compact* representation of all the maximal repeats, and it can also be constructed in $O(n)$ time, even though the total length of all the maximal repeats may be $\Omega(n^2)$. The following lemma states a necessary condition for a substring to be a maximal repeat.

Lemma 7.12.1. *Let $\mathcal{T}$ be the suffix tree for string S. If a string α is a maximal repeat in S then α is the path-label of a node v in $\mathcal{T}$.*

PROOF If α is a maximal repeat then there must be at least two copies of α in S where the character to the right of the first copy differs from the character to the right of the second copy. Hence α is the path-label of a node v in $\mathcal{T}$. □

The key point in Lemma 7.12.1 is that path α must end at a node of $\mathcal{T}$. This leads immediately to the following surprising fact:

Theorem 7.12.1. *There can be at most n maximal repeats in any string of length n.*

PROOF Since $\mathcal{T}$ has n leaves, and each internal node other than the root must have at least two children, $\mathcal{T}$ can have at most n internal nodes. Lemma 7.12.1 then implies the theorem. □

Theorem 7.12.1 would be a trivial fact if at most one substring starting at any position i could be part of a maximal pair. But that is not true. For example, in the string $S = xabcyiiizabcqabcyr$ considered earlier, both copies of substring $abcy$ participate in maximal pairs, while each copy of abc also participates in maximal pairs.

So now we know that to find maximal repeats we only need to consider strings that end at nodes in the suffix tree $\mathcal{T}$. But which specific nodes correspond to maximal repeats?

Definition For each position i in string S, character $S(i-1)$ is called the *left character* of i. The *left character of a leaf* of $\mathcal{T}$ is the left character of the suffix position represented by that leaf.

Definition A node v of $\mathcal{T}$ is called *left diverse* if at least two leaves in v's subtree have different left characters. By definition, a leaf cannot be left diverse.

Note that being left diverse is a property that propagates upward. If a node v is left diverse, so are all of its ancestors in the tree.

Theorem 7.12.2. *The string α labeling the path to a node v of $\mathcal{T}$ is a maximal repeat if and only if v is left diverse.*

PROOF Suppose first that v is left diverse. That means there are substrings $x\alpha$ and $y\alpha$ in S, where x and y represent different characters. Let the first substring be followed by character p. If the second substring is followed by any character but p, then α is a maximal repeat and the theorem is proved. So suppose that the two occurrences are $x\alpha p$ and $y\alpha p$. But since v is a (branching) node there must also be a substring αq in S for some character q that is different from p. If this occurrence of αq is preceded by character x then it participates in a maximal pair with string $y\alpha p$, and if it is preceded by y then it participates in a maximal pair with $x\alpha p$. Either way, α cannot be preceded by both x and y, so α must be part of a maximal pair and hence α must be a maximal repeat.

Conversely, if α is a maximal repeat then it participates in a maximal pair and there must be occurrences of α that have distinct left characters. Hence v must be left diverse. □

The maximal repeats can be compactly represented

Since the property of being left diverse propagates upward in $\mathcal{T}$, Theorem 7.12.2 implies that the maximal repeats of S are represented by some initial portion of the suffix tree for S. In detail, a node is called a "frontier" node in $\mathcal{T}$ if it is left diverse but none of its children are left diverse. The subtree of $\mathcal{T}$ from the root down to the frontier nodes precisely represents the maximal repeats in that every path from the root to a node at or above the frontier defines a maximal repeat. Conversely, every maximal repeat is defined by one such path. This subtree, whose leaves are the frontier nodes in $\mathcal{T}$, is a compact representation[5] of the set of all maximal repeats of S. Note that the total length of the maximal repeats could be as large as $\Theta(n^2)$, but since the representation is a subtree of $\mathcal{T}$ it has $O(n)$ total size (including the symbols used to represent edge labels). So if the left diverse nodes can be found in $O(n)$ time, then a tree representation for the set of maximal repeats can be constructed in $O(n)$ time, even though the total length of those maximal repeats could be $\Omega(n^2)$. We now describe an algorithm to find the left diverse nodes in $\mathcal{T}$.

Finding left diverse nodes in linear time

For each node v of $\mathcal{T}$, the algorithm either records that v is left diverse or it records the character, denoted x, that is the left character of every leaf in v's subtree. The algorithm starts by recording the left character of each leaf of the suffix tree $\mathcal{T}$ for S. Then it processes the nodes in $\mathcal{T}$ bottom up. To process a node v, it examines the children of v. If any child of v has been identified as being left diverse, then it records that v is left diverse. If none of v's children are left diverse, then it examines the characters recorded at v's children. If these recorded characters are all equal, say x, then it records character x at node v. However, if they are not all x, then it records that v is left diverse. The time to check if all children of v have the same recorded character is proportional to the number of v's children. Hence the total time for the algorithrn is $O(n)$. To form the final representation of the set of maximal repeats, simply delete all nodes from $\mathcal{T}$ that are not left diverse. In summary, we have

[5] This kind of tree is sometimes referred to as a *compact trie*, but we will not use that terminology.

Theorem 7.12.3. *All the maximal repeats in S can be found in $O(n)$ time, and a tree representation for them can be constructed from suffix tree $\mathcal{T}$ in $O(n)$ time as well.*

7.12.2. Finding supermaximal repeats in linear time

Recall that a supermaximal repeat is a maximal repeat that is not a substring of any other maximal repeat. We establish here efficient criteria to find all the supermaximal repeats in a string S. To do this, we solve the more general problem of finding *near-supermaximal repeats*.

Definition A substring α of S is a *near-supermaximal repeat* if α is a maximal repeat in S that occurs *at least once* in a location where it is not contained in another maximal repeat. Such an occurrence of α is said to *witness* the near-supermaximality of α.

For example, in the string $a\alpha bx\alpha ya\alpha bx\alpha b$, substring α is a maximal repeat but not a supermaximal or a near-supermaximal repeat, whereas in $a\alpha bx\alpha ya\alpha b$, substring α is again not supermaximal, but it is near-supermaximal. The second occurrence of α witnesses that fact.

With this terminology, a supermaximal repeat α is a maximal repeat in which every occurrence of α is a witness to its near-supermaximality. Note that it is not true that the set of near-supermaximal repeats is the set of maximal repeats that are not supermaximal repeats.

The suffix tree $\mathcal{T}$ for S will be used to locate the near-supermaximal and the supermaximal repeats. Let v be a node corresponding to a maximal repeat α, and let w (possibly a leaf) be one of v's children. The leaves in the subtree of $\mathcal{T}$ rooted at w identify the locations of some (but not all) of the occurrences of substring α in S. Let $L(w)$ denote those occurrences. Do any of those occurrences of α witness the near-supermaximality of α?

Lemma 7.12.2. *If node w is an internal node in $\mathcal{T}$, then none of the occurrences of α specified by $L(w)$ witness the near-supermaximality of α.*

PROOF Let γ be the substring labeling edge (v, w). Every index in $L(w)$ specifies an occurrence of $\alpha\gamma$. But w is internal, so $|L(w)| > 1$ and $\alpha\gamma$ is the prefix of a maximal repeat. Therefore, all the occurrences of α specified by $L(w)$ are contained in a maximal repeat that begins $\alpha\gamma$, and w cannot witness the near-supermaximality of α. □

Thus no occurrence of α in $L(w)$ can witness the near-supermaximality of α unless w is a leaf. If w is a leaf, then w specifies a single particular occurrence of substring $\beta = \alpha\gamma$. We now consider that case.

Lemma 7.12.3. *Suppose w is a leaf, and let i be the (single) occurrence of β represented by leaf w. Let x be the left character of leaf w. Then the occurrence of α at position i witnesses the near-supermaximality of α if and only if x is the left character of no other leaf below v.*

PROOF If there is another occurrence of α with a preceding character x, then $x\alpha$ occurs twice and so is either a maximal repeat or is contained in one. In that case, the occurrence of α at i is contained in a maximal repeat.

If there is no other occurrence of α with a preceding x, then $x\alpha$ occurs only once in S. Now let y be the first character on the edge from v to w. Since w is a leaf, αy occurs only once in S. Therefore, the occurrence of α starting at i, which is preceded by x and succeeded by y, is not contained in a maximal repeat, and so witnesses the near-supermaximality of α. □

In summary, we can state

Theorem 7.12.4. *A left diverse internal node v represents a near-supermaximal repeat α if and only if one of v's children is a leaf (specifying position i, say) and its left character, $S(i-1)$, is the left character of no other leaf below v. A left diverse internal node v represents a supermaximal repeat α if and only if all of v's children are leaves, and each has a distinct left character.*

Therefore, all supermaximal and near-supermaximal repeats can be identified in linear time. Moreover, we can define the *degree* of near-supermaximality of α as the fraction of occurrences of α that witness its near-supermaximality. That degree of each near-supermaximal repeat can also be computed in linear time.

7.12.3. Finding all the maximal pairs in linear time

We now turn to the question of finding all the maximal pairs. Since there can be more than $O(n)$ of them, the running time of the algorithm will be stated in terms of the size of the output. The algorithm is an extension of the method given earlier to find all maximal repeats.

First, build a suffix tree for S. For each leaf specifying a suffix i, record its left character $S(i-1)$. Now traverse the tree from bottom up, visiting each node in the tree. In detail, work from the leaves upward, visiting a node v only after visiting every child of v. During the visit to v, create at most σ linked lists at each node, where σ is the size of the alphabet. Each list is indexed by a left character x. The list at v indexed by x contains all the starting positions of substrings in S that match the string on the path to v and that have the left character x. That is, the list at v indexed by x is just the list of leaf numbers below v that specify suffixes in S that are immediately preceded by character x.

Letting n denote the length of S, it is easy to create (but not keep) these lists in $O(n)$ total time, working bottom up in the tree. To create the list for character x at node v, link together (but do not copy) the lists for character x that exist for each of v's children. Because the size of the alphabet is finite, the time for all linking is constant at each node. Linking without copying is required in order to achieve the $O(n)$ time bound. Linking a list created at a node v' to some other list destroys the list for v'. Fortunately, the lists created at v' will not be needed after the lists for its parent are created.

Now we show in detail how to use the lists available at v's children to find all maximal pairs containing the string that labels the path to v. At the start of the visit to node v, before v's lists have been created, the algorithm can output all maximal pairs (p_1, p_2, α), where α is the string labeling the path to v. For each character x and each child v' of v, the algorithm forms the *Cartesian* product of the list for x at v' with the union of every list for a character other than x at a child of v other than v'. Any pair in this list gives the starting positions of a maximal pair for string α. The proof of this is essentially the same as the proof of Theorem 7.12.2.

If there are k maximal pairs, then the method works in $O(n+k)$ time. The creation of the suffix tree, its bottom up traversal, and all the list linking take $O(n)$ time. Each operation used in a Cartesian product produces a maximal pair not produced anywhere else, so $O(k)$ time is used in those operations. If we only want to count the number of