Algorithms on Strings, Trees, and Sequences

COMPUTER SCIENCE AND COMPUTATIONAL BIOLOGY

Dan Gusfield
University of California, Davis

CAMBRIDGE
UNIVERSITY PRESS

Contents

Preface

History and motivation

Although I didn't know it at the time, I began writing this book in the summer of 1988 when I was part of a computer science (early bioinformatics) research group at the Human Genome Center of Lawrence Berkeley Laboratory.[1] Our group followed the standard assumption that biologically meaningful results could come from considering DNA as a one-dimensional character string, abstracting away the reality of DNA as a flexible three-dimensional molecule, interacting in a dynamic environment with protein and RNA, and repeating a life-cycle in which even the classic linear chromosome exists for only a fraction of the time. A similar, but stronger, assumption existed for protein, holding, for example, that all the information needed for correct three-dimensional folding is contained in the protein sequence itself, essentially independent of the biological environment the protein lives in. This assumption has recently been modified, but remains largely intact [297].

For nonbiologists, these two assumptions were (and remain) a god send, allowing rapid entry into an exciting and important field. Reinforcing the importance of sequence-level investigation were statements such as:

> The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology. [352]

and

> In a very real sense, molecular biology is all about sequences. First, it tries to reduce complex biochemical phenomena to interactions between defined sequences . . . .[449]

and

> The ultimate rationale behind all purposeful structures and behavior of living things is embodied in the sequence of residues of nascent polypeptide chains . . . In a real sense it is at this level of organization that the secret of life (if there is one) is to be found. [330]

So without worrying much about the more difficult chemical and biological aspects of DNA and protein, our computer science group was empowered to consider a variety of biologically important problems defined primarily on *sequences*, or (more in the computer science vernacular) on *strings*: reconstructing long strings of DNA from overlapping string fragments; determining physical and genetic maps from probe data under various experimental protocols; storing, retrieving, and comparing DNA strings; comparing two or more strings for similarities; searching databases for related strings and substrings; defining and exploring different notions of string relationships; looking for new or ill-defined patterns occurring frequently in DNA; looking for structural patterns in DNA and

[1] The other long-term members were William Chang, Gene Lawler, Dalit Naor, and Frank Olken.

protein; determining secondary (two-dimensional) structure of RNA; finding conserved, but faint, patterns in many DNA and protein sequences; and more.

We organized our efforts into two high-level tasks. First, we needed to learn the relevant biology, laboratory protocols, and existing algorithmic methods used by biologists. Second we sought to canvass the computer science literature for ideas and algorithms that weren't already used by biologists, but which *might plausibly* be of use either in current problems or in problems that we could anticipate arising when vast quantities of sequenced DNA or protein become available.

Our problem

None of us was an expert on string algorithms. At that point I had a textbook knowledge of Knuth-Morris-Pratt and a deep confusion about Boyer–Moore (under what circumstances it was a linear time algorithm and how to do *strong* preprocessing in linear time). I understood the use of dynamic programming to compute edit distance, but otherwise had little exposure to specific string algorithms in biology. My general background was in combinatorial optimization, although I had a prior interest in algorithms for building evolutionary trees and had studied some genetics and molecular biology in order to pursue that interest.

What we needed then, but didn't have, was a comprehensive cohesive text on string algorithms to guide our education. There were at that time several computer science texts containing a chapter or two on strings, usually devoted to a rigorous treatment of Knuth-Morris-Pratt and a cursory treatment of Boyer–Moore, and possibly an elementary discussion of matching with errors. There were also some good survey papers that had a somewhat wider scope but didn't treat their topics in much depth. There were several texts and edited volumes from the biological side on uses of computers and algorithms for sequence analysis. Some of these were wonderful in exposing the potential benefits and the pitfalls of using computers in biology, but they generally lacked algorithmic rigor and covered a narrow range of techniques. Finally, there was the seminal text *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* edited by D. Sankoff and J. Kruskal, which served as a bridge between algorithms and biology and contained many applications of dynamic programming. However, it too was much narrower than our focus and was a bit dated.

Moreover, most of the available sources from either community focused on string *matching*, the problem of searching for an exact or "nearly exact" copy of a pattern in a given text. Matching problems are central, but as detailed in this book, they constitute only a part of the many important computational problems defined on strings. Thus, we recognized that summer a need for a rigorous and fundamental treatment of the *general* topic of algorithms that operate on strings, along with a rigorous treatment of *specific* string algorithms of greatest current and potential import in computational biology. This book is an attempt to provide such a dual, and integrated, treatment.

Why mix computer science and computational biology in one book?

My interest in computational biology began in 1980, when I started reading papers on building evolutionary trees. That side interest allowed me an occasional escape from the hectic, hyper competitive "hot" topics that theoretical computer science focuses on. At that point, computational molecular biology was a largely undiscovered area for computer science, although it was an active area for statisticians and mathematicians (notably Michael Waterman and David Sankoff who have largely framed the field). Early on, seminal papers on computational issues in biology (such as the one by Buneman [83]) did not appear in mainstream computer science venues but in obscure places such as conferences on *computational archeology* [226]. But seventeen years later, computational biology *is* hot, and many computer scientists are now entering the (now more hectic, more competitive) field [280]. What should they learn?

The problem is that the emerging field of computational molecular biology is not well defined and its definition is made more difficult by rapid changes in molecular biology itself. Still, algorithms that operate on molecular sequence data (strings) are at the heart of computational molecular biology. The big-picture question in computational molecular biology is how to "do" as much "real biology" as possible by exploiting molecular sequence data (DNA, RNA, and protein). Getting sequence data is relatively cheap and fast (and getting more so) compared to more traditional laboratory investigations. The use of sequence data is already central in several subareas of molecular biology and the full impact of having extensive sequence data is yet to be seen. Hence, algorithms that operate on strings will continue to be the area of closest intersection and interaction between computer science and molecular biology. Certainly then, computer scientists need to learn the string techniques that have been most successfully applied. But that is not enough.

Computer scientists need to learn *fundamental* ideas and techniques that will endure long after today's central motivating applications are forgotten. They need to study methods that prepare them to frame and tackle future problems and applications. Significant contributions to computational biology might be made by extending or adapting algorithms from computer science, even when the original algorithm has no clear utility in biology. This is illustrated by several recent sublinear-time approximate matching methods for database searching that rely on an interplay between exact matching methods from computer science and dynamic programming methods already utilized in molecular biology.

Therefore, the computer scientist who wants to enter the general field of computational molecular biology, and who learns string algorithms with that end in mind, should receive a training in string algorithms that is much broader than a tour through techniques of known present application. Molecular biology and computer science are changing much too rapidly for that kind of narrow approach. Moreover, theoretical computer scientists try to develop effective algorithms somewhat differently than other algorithmists. We rely more heavily on correctness proofs, worst-case analysis, lower bound arguments, randomized algorithm analysis, and bounded approximation results (among other techniques) to *guide* the development of practical, effective algorithms. Our "relative advantage" partly lies in the mastery and use of those skills. So even if I were to write a book for computer scientists who only want to do computational biology, I would still choose to include a broad range of algorithmic techniques from pure computer science.

In this book, I cover a wide spectrum of string techniques – well beyond those of established utility; however, I have selected from the many possible illustrations, those techniques that seem to have the greatest *potential application* in future molecular biology. Potential application, particularly of ideas rather than of concrete methods, and to *anticipated* rather than to existing problems is a matter of judgment and speculation. No doubt, some of the material contained in this book will never find direct application in biology, while other material will find uses in surprising ways. Certain string algorithms that were generally deemed to be irrelevant to biology just a few years ago have become adopted

by practicing biologists in both large-scale projects and in narrower technical problems. Techniques previously dismissed because they originally addressed (exact) string problems where *perfect data* were assumed have been incorporated as *components* of more robust techniques that handle imperfect data.

What the book is

Following the above discussion, this book is a general-purpose rigorous treatment of the entire field of deterministic algorithms that operate on strings and sequences. Many of those algorithms utilize trees as data-structures or arise in biological problems related to evolutionary trees, hence the inclusion of "trees" in the title.

The model reader is a research-level professional in computer science or a graduate or advanced undergraduate student in computer science, although there are many biologists (and of course mathematicians) with sufficient algorithmic background to read the book. The book is intended to serve as both a reference and a main text for courses in pure computer science and for computer science–oriented courses on computational biology.

Explicit discussions of biological applications appear throughout the book, but are more concentrated in the last sections of Part II and in most of Parts III and IV. I discuss a number of biological issues in detail in order to give the reader a deeper appreciation for the reasons that many biological problems have been cast as problems on strings and for the variety of (often very imaginative) technical ways that string algorithms have been employed in molecular biology.

This book covers all the classic topics and most of the important advanced techniques in the field of string algorithms, with three exceptions. It only lightly touches on probabilistic analysis and does not discuss parallel algorithms or the elegant, but very theoretical, results on algorithms for infinite alphabets and on algorithms using only constant auxiliary space.[2] The book also does not cover stochastic-oriented methods that have come out of the machine learning community, although some of the algorithms in this book are extensively used as subtools in those methods. With these exceptions, the book covers all the major styles of thinking about string algorithms. The reader who absorbs the material in this book will gain a deep and broad understanding of the field and sufficient sophistication to undertake original research.

Reflecting my background, the book rigorously discusses each of its topics, usually providing complete proofs of behavior (correctness, worst-case time, and space). More important, it emphasizes the *ideas* and *derivations* of the methods it presents, rather than simply providing an inventory of available algorithms. To better expose ideas and encourage discovery, I often present a complex algorithm by introducing a naive, inefficient version and then successively apply additional insight and implementation detail to obtain the desired result.

The book contains some new approaches I developed to explain certain classic and complex material. In particular, the preprocessing methods I present for Knuth-Morris-Pratt, Boyer–Moore and several other linear-time pattern matching algorithms differ from the classical methods, both unifying and simplifying the preprocessing tasks needed for those algorithms. I also expect that my (hopefully simpler and clearer) expositions on linear-time suffix tree constructions and on the constant-time least common ancestor algorithm will make those important methods more available and widely understood. I connect theoretical results from computer science on sublinear-time algorithms with widely used methods for biological database search. In the discussion of multiple sequence alignment I bring together the three major objective functions that have been proposed for multiple alignment and show a continuity between approximation algorithms for those three multiple alignment problems. Similarly, the chapter on evolutionary tree construction exposes the commonality of several distinct problems and solutions in a way that is not well known. Throughout the book, I discuss many computational problems concerning repeated substrings (a very widespread phenomenon in DNA). I consider several different ways to define repeated substrings and use each specific definition to explore computational problems and algorithms on repeated substrings.

In the book I try to explain in complete detail, and at a reasonable pace, many complex methods that have previously been written exclusively for the specialist in string algorithms. I avoid detailed code, as I find it rarely serves to explain interesting ideas,[3] and I provide over 400 exercises to both reinforce the material of the book and to develop additional topics.

[2] Space is a very important practical concern, and we will discuss it frequently, but constant space seems too severe a requirement in most applications of interest.

What the book is not

Let me state clearly what the book is not. It is not a *complete* text on computational molecular biology, since I believe that field concerns computations on objects other than strings, trees, and sequences. Still, computations on strings and sequences form the heart of computational molecular biology, and the book provides a deep and wide treatment of sequence-oriented computational biology. The book is also not a "how to" book on string and sequence analysis. There are several books available that survey specific computer packages, databases, and services, while also giving a general idea of how they work. This book, with its emphasis on ideas and algorithms, does not compete with those. Finally, at the other extreme, the book does not attempt a definitive history of the field of string algorithms and its contributors. The literature is vast, with many repeated, independent discoveries, controversies, and conflicts. I have made some historical comments and have pointed the reader to what I hope are helpful references, but I am much too new an arrival and not nearly brave enough to attempt a complete taxonomy of the field. I apologize in advance, therefore, to the many people whose work may not be properly recognized.

In summary

This book is a general, rigorous text on deterministic algorithms that operate on strings, trees, and sequences. It covers the full spectrum of string algorithms from classical computer science to modern molecular biology and, when appropriate, connects those two fields. It is the book I wished I had available when I began learning about string algorithms.

Acknowledgments

I would like to thank The Department of Energy Human Genome Program, The Lawrence Berkeley Laboratory, The National Science Foundation, The Program in Math and Molec-

[3] However, many of the algorithms in the book have been coded in C and are available at http://www.cse.ucdavis.edu/~gusfield/strpgms.html.

ular Biology, and The DIMACS Center for Discrete Mathematics and Computer Science special year on computational biology, for support of my work and the work of my students and postdoctoral researchers.

Individually, I owe a great debt of appreciation to William Chang, John Kececioglu, Jim Knight, Gene Lawler, Dalit Naor, Frank Olken, R. Ravi, Paul Stelling, and Lusheng Wang.

I would also like to thank the following people for the help they have given me along the way: Stephen Altschul, David Axelrod, Doug Brutlag, Archie Cobbs, Richard Cole, Russ Doolittle, Martin Farach, Jane Gitschier, George Hartzell, Paul Horton, Robert Irving, Sorin Istrail, Tao Jiang, Dick Karp, Dina Kravets, Gad Landau, Udi Manber, Marci McClure, Kevin Murphy, Gene Myers, John Nguyen, Mike Paterson, William Pearson, Pavel Pevzner, Fred Roberts, Hershel Safer, Baruch Schieber, Ron Shamir, Jay Snoddy, Elizabeth Sweedyk, Sylvia Spengler, Martin Tompa, Esko Ukkonen, Martin Vingron, Tandy Warnow, and Mike Waterman.

PART I

Exact String Matching: The Fundamental String Problem

Exact matching: what's the problem?

Given a string P called the *pattern* and a longer string T called the *text*, the **exact matching** problem is to find all occurrences, if any, of pattern P in text T.

For example, if $P = aba$ and $T = bbabaxababay$ then P occurs in T starting at locations 3, 7, and 9. Note that two occurrences of P may overlap, as illustrated by the occurrences of P at locations 7 and 9.

Importance of the exact matching problem

The practical importance of the exact matching problem should be obvious to anyone who uses a computer. The problem arises in widely varying applications, too numerous to even list completely. Some of the more common applications are in word processors; in utilities such as *grep* on Unix; in textual information retrieval programs such as Medline, Lexis, or Nexis; in library catalog searching programs that have replaced physical card catalogs in most large libraries; in internet browsers and crawlers, which sift through massive amounts of text available on the internet for material containing specific keywords;[1] in internet news readers that can search the articles for topics of interest; in the giant digital libraries that are being planned for the near future; in electronic journals that are already being "published" on-line; in telephone directory assistance; in on-line encyclopedias and other educational CD-ROM applications; in on-line dictionaries and thesauri, especially those with cross-referencing features (the *Oxford English Dictionary* project has created an electronic on-line version of the *OED* containing 50 million words); and in numerous specialized databases. In molecular biology there are several hundred specialized databases holding raw DNA, RNA, and amino acid strings, or processed patterns (called motifs) derived from the raw string data. Some of these databases will be discussed in Chapter 15.

Although the practical importance of the exact matching problem is not in doubt, one might ask whether the problem is still of any research or educational interest. Hasn't exact matching been so well solved that it can be put in a black box and taken for granted? Right now, for example, I am editing a ninety-page file using an "ancient" shareware word processor and a PC clone (486), and every exact match command that I've issued executes faster than I can blink. That's rather depressing for someone writing a book containing a large section on exact matching algorithms. So is there anything left to do on this problem?

The answer is that for typical word-processing applications there probably is little left to do. The exact matching problem is solved for those applications (although other more sophisticated string tools might be useful in word processors). But the story changes radically

[1] I just visited the Alta Vista web page maintained by the Digital Equipment Corporation. The Alta Vista database contains over 21 billion words collected from over 10 million web sites. A search for all web sites that mention "Mark Twain" took a couple of seconds and reported that twenty thousand sites satisfy the query. For another example see [392].

for other applications. Users of Melvyl, the on-line catalog of the University of California library system, often experience long, frustrating delays even for fairly simple matching requests. Even *grepping* through a large directory can demonstrate that exact matching is not yet trivial. Recently we used GCG (a very popular interface to search DNA and protein databanks) to search Genbank (the major U.S. DNA database) for a thirty-character string, which is a small string in typical uses of Genbank. The search took over four hours (on a local machine using a local copy of the database) to find that the string was not there.[2] And Genbank today is only a fraction of the size it will be when the various genome programs go into full production mode, cranking out massive quantities of sequenced DNA. Certainly there are faster, common database searching programs (for example, BLAST), and there are faster machines one can use (for example, an e-mail server is available for exact and inexact database matching running on a 4,000 processor MasPar computer). But the point is that the exact matching problem is not so effectively and universally solved that it needs no further attention. It will remain a problem of interest as the size of the databases grow and also because exact matching will continue to be a *subtask* needed for more complex searches that will be devised. Many of these will be illustrated in this book.

But perhaps the most important reason to study *exact* matching in detail is to understand the various ideas developed for it. Even assuming that the exact matching problem itself is sufficiently solved, the entire field of string algorithms remains vital and open, and the education one gets from studying exact matching may be crucial for solving less understood problems. That education takes three forms: specific algorithms, general algorithmic styles, and analysis and proof techniques. All three are covered in this book, but style and proof technique get the major emphasis.

Overview of Part I

In Chapter 1 we present naive solutions to the exact matching problem and develop the fundamental tools needed to obtain more efficient methods. Although the classical solutions to the problem will not be presented until Chapter 2, we will show at the end of Chapter 1 that the use of fundamental tools alone gives a simple linear-time algorithm for exact matching. Chapter 2 develops several classical methods for exact matching, using the fundamental tools developed in Chapter 1. Chapter 3 looks more deeply at those methods and extensions of them. Chapter 4 moves in a very different direction, exploring methods for exact matching based on arithmetic-like operations rather than character comparisons.

Although exact matching is the focus of Part I, some aspects of inexact matching and the use of wild cards are also discussed. The exact matching problem will be discussed again in Part II, where it (and extensions) will be solved using suffix trees.

Basic string definitions

We will introduce most definitions at the point where they are first used, but several definitions are so fundamental that we introduce them now.

Definition A *string* S is an ordered list of characters written contiguously from left to right. For any string S, $S[i..j]$ is the (contiguous) *substring* of S that starts at position

[2] We later repeated the test using the Boyer–Moore algorithm on our own raw copy of Genbank. The search took less than ten minutes, most of which was devoted to movement of text between the disk and the computer, with less than one minute used by the actual text search.

i and ends at position j of S. In particular, $S[1..i]$ is the *prefix* of string S that ends at position i, and $S[i..|S|]$ is the *suffix* of string S that begins at position i, where $|S|$ denotes the number of characters in string S.

Definition $S[i..j]$ is the empty string if $i > j$.

For example, *california* is a string, *lifo* is a substring, *cal* is a prefix, and *ornia* is a suffix.

Definition A *proper* prefix, suffix, or substring of S is, respectively, a prefix, suffix, or substring that is not the entire string S, nor the empty string.

Definition For any string S, $S(i)$ denotes the ith character of S.

We will usually use the symbol S to refer to an arbitrary fixed string that has no additional assumed features or roles. However, when a string is known to play the role of a pattern or the role of a text, we will refer to the string as P or T respectively. We will use lower case Greek characters ($\alpha, \beta, \gamma, \delta$) to refer to variable strings and use lower case roman characters to refer to single variable characters.

Definition When comparing two characters, we say that the characters *match* if they are equal; otherwise we say they *mismatch*.

Terminology confusion

The words "string" and "word" are often used synonymously in the computer science literature, but for clarity in this book we will never use "word" when "string" is meant. (However, we do use "word" when its colloquial English meaning is intended.)

More confusing, the words "string" and "sequence" are often used synonymously, particularly in the biological literature. This can be the source of much confusion because "*sub*strings" and "*sub*sequences" are very different objects and because algorithms for substring problems are usually very different than algorithms for the analogous subsequence problems. The characters in a substring of S must occur *contiguously* in S, whereas characters in a subsequence might be interspersed with characters not in the subsequence. Worse, in the biological literature one often sees the word "sequence" used in place of "subsequence". Therefore, for clarity, in this book we will always maintain a distinction between "subsequence" and "substring" and never use "sequence" for "subsequence". We will generally use "string" when pure computer science issues are discussed and use "sequence" or "string" interchangeably in the context of biological applications. Of course, we will also use "sequence" when its standard mathematical meaning is intended.

The first two parts of this book primarily concern problems on strings and substrings. Problems on subsequences are considered in Parts III and IV.

1

Exact Matching: Fundamental Preprocessing and First Algorithms

1.1. The naive method

Almost all discussions of exact matching begin with the *naive method*, and we follow this tradition. The naive method aligns the left end of P with the left end of T and then compares the characters of P and T left to right until either two unequal characters are found or until P is exhausted, in which case an occurrence of P is reported. In either case, P is then shifted one place to the right, and the comparisons are restarted from the left end of P. This process repeats until the right end of P shifts past the right end of T.

Using n to denote the length of P and m to denote the length of T, the worst-case number of comparisons made by this method is $\Theta(nm)$. In particular, if both P and T consist of the same repeated character, then there is an occurrence of P at each of the first $m-n+1$ positions of T and the method performs exactly $n(m-n+1)$ comparisons. For example, if $P = aaa$ and $T = aaaaaaaaaa$ then $n = 3, m = 10$, and 24 comparisons are made.

The naive method is certainly simple to understand and program, but its worst-case running time of $\Theta(nm)$ may be unsatisfactory and can be improved. Even the practical running time of the naive method may be too slow for larger texts and patterns. Early on, there were several related ideas to improve the naive method, both in practice and in worst case. The result is that the $\Theta(n \times m)$ worst-case bound can be reduced to $O(n+m)$. Changing "$\times$" to "+" in the bound is extremely significant (try $n = 1000$ and $m = 10{,}000{,}000$, which are realistic numbers in some applications).

1.1.1. Early ideas for speeding up the naive method

The first ideas for speeding up the naive method all try to shift P by more than one character when a mismatch occurs, but never shift it so far as to miss an occurrence of P in T. Shifting by more than one position saves comparisons since it moves P through T more rapidly. In addition to shifting by larger amounts, some methods try to reduce comparisons by skipping over parts of the pattern after the shift. We will examine many of these ideas in detail.

Figure 1.1 gives a flavor of these ideas, using $P = abxyabxz$ and $T = xabxyabxyabxz$. Note that an occurrence of P begins at location 6 of T. The naive algorithm first aligns P at the left end of T, immediately finds a mismatch, and shifts P by one position. It then finds that the next seven comparisons are matches and that the succeeding comparison (the ninth overall) is a mismatch. It then shifts P by one place, finds a mismatch, and repeats this cycle two additional times, until the left end of P is aligned with character 6 of T. At that point it finds eight matches and concludes that P occurs in T starting at position 6. In this example, a total of twenty comparisons are made by the naive algorithm.

A smarter algorithm might realize, after the ninth comparison, that the next three

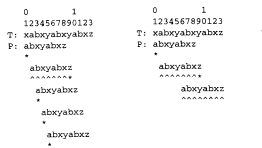


Figure 1.1: The first scenario illustrates pure naive matching, and the next two illustrate smarter shifts. A caret beneath a character indicates a match and a star indicates a mismatch made by the algorithm.

comparisons of the naive algorithm will be mismatches. This smarter algorithm skips over the next three shift/compares, immediately moving the left end of P to align with position 6 of T, thus saving three comparisons. How can a smarter algorithm do this? After the ninth comparison, the algorithm knows that the first seven characters of P match characters 2 through 8 of T. If it also knows that the first character of P (namely a) does not occur again in P until position 5 of P, it has enough information to conclude that character a does not occur again in T until position 6 of T. Hence it has enough information to conclude that there can be no matches between P and T until the left end of P is aligned with position 6 of T. Reasoning of this sort is the key to shifting by more than one character. In addition to shifting by larger amounts, we will see that certain aligned characters do not need to be compared.

An even smarter algorithm knows the next occurrence in P of the first three characters of P (namely abx) begin at position 5. Then since the first seven characters of P were found to match characters 2 through 8 of T, this smarter algorithm has enough information to conclude that when the left end of P is aligned with position 6 of T, the next three comparisons must be matches. This smarter algorithm avoids making those three comparisons. Instead, after the left end of P is moved to align with position 6 of T, the algorithm compares character 4 of P against character 9 of T. This smarter algorithm therefore saves a total of six comparisons over the naive algorithm.

The above example illustrates the kinds of ideas that allow some comparisons to be skipped, although it should still be unclear how an algorithm can efficiently implement these ideas. Efficient implementations have been devised for a number of algorithms such as the Knuth-Morris-Pratt algorithm, a real-time extension of it, the Boyer–Moore algorithm, and the Apostolico–Giancarlo version of it. All of these algorithms have been implemented to run in linear time ($O(n+m)$ time). The details will be discussed in the next two chapters.

1.2. The preprocessing approach

Many string matching and analysis algorithms are able to efficiently skip comparisons by first spending "modest" time learning about the internal structure of either the pattern P or the text T. During that time, the other string may not even be known to the algorithm. This part of the overall algorithm is called the *preprocessing* stage. Preprocessing is followed by a *search* stage, where the information found during the preprocessing stage is used to reduce the work done while searching for occurrences of P in T. In the above example, the smarter method was assumed to know that character a did not occur again until position 5, and the even smarter method was assumed to know that the pattern abx was repeated again starting at position 5. This assumed knowledge is obtained in the preprocessing stage.

For the exact matching problem, all of the algorithms mentioned in the previous section preprocess pattern P. (The opposite approach of preprocessing text T is used in other algorithms, such as those based on suffix trees. Those methods will be explained later in the book.) These preprocessing methods, as originally developed, are similar in spirit but often quite different in detail and conceptual difficulty. In this book we take a different approach and do not initially explain the originally developed preprocessing methods. Rather, we highlight the similarity of the preprocessing *tasks* needed for several different matching algorithms, by first defining a *fundamental preprocessing* of P that is independent of any particular matching algorithm. Then we show how each specific matching algorithm uses the information computed by the fundamental preprocessing of P. The result is a simpler more uniform exposition of the preprocessing needed by several classical matching methods and a simple linear time algorithm for exact matching based only on this preprocessing (discussed in Section 1.5). This approach to linear-time pattern matching was developed in [202].

1.3. Fundamental preprocessing of the pattern

Fundamental preprocessing will be described for a general string denoted by S. In specific applications of fundamental preprocessing, S will often be the pattern P, but here we use S instead of P because fundamental preprocessing will also be applied to strings other than P.

The following definition gives the key values computed during the fundamental preprocessing of a string.

Definition Given a string S and a position $i > 1$, let $Z_i(S)$ be the *length* of the longest substring of S that *starts* at i and matches a prefix of S.

In other words, $Z_i(S)$ is the length of the longest *prefix* of $S[i..|S|]$ that matches a prefix of S. For example, when $S = aabcaabxaaz$ then

$$Z_5(S) = 3 \ (aabc...aabx...),$$
$$Z_6(S) = 1 \ (aa...ab...),$$
$$Z_7(S) = Z_8(S) = 0,$$
$$Z_9(S) = 2 \ (aab...aaz).$$

When S is clear by context, we will use Z_i in place of $Z_i(S)$.

To introduce the next concept, consider the boxes drawn in Figure 1.2. Each box starts at some position $j > 1$ such that Z_j is greater than zero. The length of the box starting at j is meant to represent Z_j. Therefore, each box in the figure represents a maximal-length

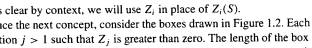


Figure 1.2: Each solid box represents a substring of S that matches a prefix of S and that starts between positions 2 and i. Each box is called a *Z-box*. We use r_i to denote the *right-most* end of any Z-box that begins at or to the left of position i and α to denote the substring in the Z-box ending at r_i. Then l_i denotes the left end of α. The copy of α that occurs as a prefix of S is also shown in the figure.

substring of S that matches a prefix of S and that does not start at position one. Each such box is called a *Z-box*. More formally, we have:

Definition For any position $i > 1$ where Z_i is greater than zero, the *Z-box* at i is defined as the interval starting at i and ending at position $i + Z_i - 1$.

Definition For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before position i. Another way to state this is: r_i is the largest value of $j + Z_j - 1$ over all $1 < j \leq i$ such that $Z_j > 0$. (See Figure 1.2.)

We use the term l_i for the value of j specified in the above definition. That is, l_i is the position of the *left end* of the Z-box that ends at r_i. In case there is more than one Z-box ending at r_i, then l_i can be chosen to be the left end of any of those Z-boxes. As an example, suppose $S = aabaabcaxaabaabcy$; then $Z_{10} = 7, r_{15} = 16$, and $l_{15} = 10$.

The linear time computation of Z values from S is the *fundamental* preprocessing task that we will use in all the classical linear-time matching algorithms that preprocess P. But before detailing those uses, we show how to do the fundamental preprocessing in linear time.

1.4. Fundamental preprocessing in linear time

The task of this section is to show how to compute all the Z_i values for S in linear time (i.e., in $O(|S|)$ time). A direct approach based on the definition would take $\Theta(|S|^2)$ time. The method we will present was developed in [307] for a different purpose.

The preprocessing algorithm computes Z_i, r_i, and l_i for each successive position i, starting from $i = 2$. All the Z values computed will be kept by the algorithm, but in any iteration i, the algorithm only needs the r_j and l_j values for $j = i - 1$. No earlier r or l values are needed. Hence the algorithm only uses a single variable, r, to refer to the most recently computed r_j value; similarly, it only uses a single variable l. Therefore, in each iteration i, if the algorithm discovers a new Z-box (starting at i), variable r will be incremented to the end of that Z-box, which is the right-most position of any Z-box discovered so far.

To begin, the algorithm finds Z_2 by explicitly comparing, left to right, the characters of $S[2..|S|]$ and $S[1..|S|]$ until a mismatch is found. Z_2 is the length of the matching string. If $Z_2 > 0$, then $r = r_2$ is set to $Z_2 + 1$ and $l = l_2$ is set to 2. Otherwise r and l are set to zero. Now assume inductively that the algorithm has correctly computed Z_i for i up to $k - 1 > 1$, and assume that the algorithm knows the current $r = r_{k-1}$ and $l = l_{k-1}$. The algorithm next computes Z_k, $r = r_k$, and $l = l_k$.

The main idea is to use the already computed Z values to accelerate the computation of Z_k. In fact, in some cases, Z_k can be deduced from the previous Z values without doing any additional character comparisons. As a concrete example, suppose $k = 121$, all the values Z_2 through Z_{120} have already been computed, and $r_{120} = 130$ and $l_{120} = 100$. That means that there is a substring of length 31 starting at position 100 and matching a prefix of S (of length 31). It follows that the substring of length 10 starting at position 121 must match the substring of length 10 starting at position 22 of S, and so Z_{22} may be very helpful in computing Z_{121}. As one case, if Z_{22} is three, say, then a little reasoning shows that Z_{121} must also be three. Thus in this illustration, Z_{121} can be deduced without any additional character comparisons. This case, along with the others, will be formalized and proven correct below.

Figure 1.3: String $S[k..r]$ is labeled β and also occurs starting at position k' of S.

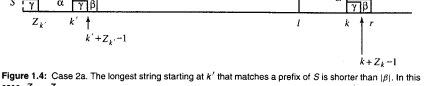


Figure 1.4: Case 2a. The longest string starting at k' that matches a prefix of S is shorter than $|\beta|$. In this case, $Z_k = Z_{k'}$.

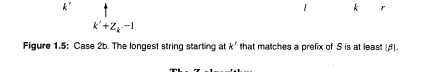


Figure 1.5: Case 2b. The longest string starting at k' that matches a prefix of S is at least $|\beta|$.

The Z algorithm

Given Z_i for all $1 < i \leq k - 1$ and the current values of r and l, Z_k and the updated r and l are computed as follows:

Begin

1. If $k > r$, then find Z_k by explicitly comparing the characters starting at position k to the characters starting at position 1 of S, until a mismatch is found. The length of the match is Z_k. If $Z_k > 0$, then set r to $k + Z_k - 1$ and set l to k.

2. If $k \leq r$, then position k is contained in a Z-box, and hence $S(k)$ is contained in substring $S[l..r]$ (call it α) such that $l > 1$ and α matches a prefix of S. Therefore, character $S(k)$ also appears in position $k' = k - l + 1$ of S. By the same reasoning, substring $S[k..r]$ (call it β) must match substring $S[k'..Z_l]$. It follows that the substring beginning at position k must match a prefix of S of length at least the *minimum* of $Z_{k'}$ and $|\beta|$ (which is $r - k + 1$). See Figure 1.3.

We consider two subcases based on the value of that minimum.

2a. If $Z_{k'} < |\beta|$ then $Z_k = Z_{k'}$ and r, l remain unchanged (see Figure 1.4).

2b. If $Z_{k'} \geq |\beta|$ then the entire substring $S[k..r]$ must be a prefix of S and $Z_k \geq |\beta| = r - k + 1$. However, Z_k might be strictly larger than $|\beta|$, so compare the characters starting at position $r + 1$ of S to the characters starting a position $|\beta| + 1$ of S until a mismatch occurs. Say the mismatch occurs at character $q \geq r + 1$. Then Z_k is set to $q - k$, r is set to $q - 1$, and l is set to k (see Figure 1.5).

End

Theorem 1.4.1. *Using Algorithm Z, value Z_k is correctly computed and variables r and l are correctly updated.*

PROOF In Case 1, Z_k is set correctly since it is computed by explicit comparisons. Also (since $k > r$ in Case 1), before Z_k is computed, no Z-box has been found that starts

between positions 2 and $k-1$ and that ends at or after position k. Therefore, when $Z_k > 0$ in Case 1, the algorithm does find a new Z-box ending at or after k, and it is correct to change r to $k + Z_k - 1$. Hence the algorithm works correctly in Case 1.

In Case 2a, the substring beginning at position k can match a prefix of S only for length $Z_{k'} < |\beta|$. If not, then the next character to the right, character $k + Z_{k'}$, must match character $1 + Z_{k'}$. But character $k + Z_{k'}$ matches character $k' + Z_{k'}$ (since $Z_{k'} < |\beta|$), so character $k' + Z_{k'}$ must match character $1 + Z_{k'}$. However, that would be a contradiction to the definition of $Z_{k'}$, for it would establish a substring longer than $Z_{k'}$ that starts at k' and matches a prefix of S. Hence $Z_k = Z_{k'}$ in this case. Further, $k + Z_k - 1 < r$, so r and l remain correctly unchanged.

In Case 2b, β must be a prefix of S (as argued in the body of the algorithm) and since any extension of this match is explicitly verified by comparing characters beyond r to characters beyond the prefix β, the full extent of the match is correctly computed. Hence Z_k is correctly obtained in this case. Furthermore, since $k + Z_k - 1 \geq r$, the algorithm correctly changes r and l. □

Corollary 1.4.1. Repeating Algorithm Z for each position $i > 2$ correctly yields all the Z_i values.

Theorem 1.4.2. *All the $Z_i(S)$ values are computed by the algorithm in $O(|S|)$ time.*

PROOF The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons. Each comparison results in either a match or a mismatch, so we next bound the number of matches and mismatches that can occur.

Each iteration that performs any character comparisons at all ends the first time it finds a mismatch; hence there are at most $|S|$ mismatches during the entire algorithm. To bound the number of matches, note first that $r_k \geq r_{k-1}$ for every iteration k. Now, let k be an iteration where $q > 0$ matches occur. Then r_k is set to $r_{k-1} + q$ at least. Finally, $r_k \leq |S|$, so the total number of matches that occur during any execution of the algorithm is at most $|S|$. □

1.5. The simplest linear-time exact matching algorithm

Before discussing the more complex (classical) exact matching methods, we show that fundamental preprocessing alone provides a simple linear-time exact matching algorithm. This is the simplest linear-time matching algorithm we know of.

Let $S = P\$T$ be the string consisting of P followed by the symbol "$" followed by T, where "$" is a character appearing in neither P nor T. Recall that P has length n and T has length m, and $n \leq m$. So, $S = P\$T$ has length $n + m + 1 = O(m)$. Compute $Z_i(S)$ for i from 2 to $n + m + 1$. Because "$" does not appear in P or T, $Z_i \leq n$ for every $i > 1$. Any value of $i > n + 1$ such that $Z_i(S) = n$ identifies an occurrence of P in T starting at position $i - (n + 1)$ of T. Conversely, if P occurs in T starting at position j of T, then $Z_{(n+1)+j}$ must be equal to n. Since all the $Z_i(S)$ values can be computed in $O(n + m) = O(m)$ time, this approach identifies all the occurrences of P in T in $O(m)$ time.

The method can be implemented to use only $O(n)$ space (in addition to the space needed for pattern and text) independent of the size of the alphabet. Since $Z_i \leq n$ for all i, position k' (determined in step 2) will always fall inside P. Therefore, there is no need to record the Z values for characters in T. Instead, we only need to record the Z values for the n characters in P and also maintain the current l and r. Those values are sufficient to compute (but not store) the Z value of each character in T and hence to identify and output any position i where $Z_i = n$.

There is another characteristic of this method worth introducing here: The method is considered an *alphabet-independent* linear-time method. That is, we never had to assume that the alphabet size was finite or that we knew the alphabet ahead of time – a character comparison only determines whether the two characters match or mismatch; it needs no further information about the alphabet. We will see that this characteristic is also true of the Knuth-Morris-Pratt and Boyer–Moore algorithms, but not of the Aho–Corasick algorithm or methods based on suffix trees.

1.5.1. Why continue?

Since function Z_i can be computed for the pattern in linear time and can be used directly to solve the exact matching problem in $O(m)$ time (with only $O(n)$ additional space), why continue? In what way are more complex methods (Knuth-Morris-Pratt, Boyer–Moore, real-time matching, Apostolico–Giancarlo, Aho–Corasick, suffix tree methods, etc.) deserving of attention?

For the exact matching problem, the Knuth-Morris-Pratt algorithm has only a marginal advantage over the direct use of Z_i. However, it has historical importance and has been generalized, in the Aho–Corasick algorithm, to solve the problem of searching for a *set* of patterns in a text in time linear in the size of the text. That problem is not nicely solved using Z_i values alone. The real-time extension of Knuth-Morris-Pratt has an advantage in situations when text is input on-line and one has to be sure that the algorithm will be ready for each character as it arrives. The Boyer–Moore method is valuable because (with the proper implementation) it also runs in linear worst-case time but typically runs in *sublinear* time, examining only a fraction of the characters of T. Hence it is the preferred method in most cases. The Apostolico–Giancarlo method is valuable because it has all the advantages of the Boyer–Moore method and yet allows a relatively simple proof of linear worst-case running time. Methods based on suffix trees typically preprocess the text rather than the pattern and then lead to algorithms in which the search time is proportional to the size of the pattern rather than the size of the text. This is an extremely desirable feature. Moreover, suffix trees can be used to solve much more complex problems than exact matching, including problems that are not easily solved by direct application of the fundamental preprocessing.

1.6. Exercises

The first four exercises use the fact that fundamental processing can be done in linear time and that all occurrences of P in T can be found in linear time.

1. Use the existence of a linear-time exact matching algorithm to solve the following problem in linear time. Given two strings α and β, determine if α is a circular (or cyclic) rotation of β, that is, if α and β have the same length and α consists of a suffix of β followed by a prefix of β. For example, *defabc* is a circular rotation of *abcdef*. This is a classic problem with a very elegant solution.
2. Similar to Exercise 1, give a linear-time algorithm to determine whether a linear string α is a substring of a *circular string* β. A circular string of length n is a string in which character n is considered to precede character 1 (see Figure 1.6). Another way to think about this

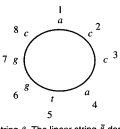


Figure 1.6: A circular string β. The linear string $\bar{\beta}$ derived from it is *accatggc*.

problem is the following. Let $\bar{\beta}$ be the *linear* string obtained from β starting at character 1 and ending at character n. Then α is a substring of circular string β if and only if α is a substring of some circular rotation of $\bar{\beta}$.

A digression on circular strings in DNA

The above two problems are mostly exercises in using the existence of a linear-time exact matching algorithm, and we don't know any critical biological problems that they address. However, we want to point out that *circular DNA* is common and important. Bacterial and mitochondrial DNA is typically circular, both in its genomic DNA and in additional small double-stranded circular DNA molecules called *plasmids*, and even some true eukaryotes (higher organisms whose cells contain a nucleus) such as yeast contain plasmid DNA in addition to their nuclear DNA. Consequently, tools for handling circular strings may someday be of use in those organisms. Viral DNA is not always circular, but even when it is linear some virus genomes exhibit circular properties. For example, in some viral populations the linear order of the DNA in one individual will be a circular rotation of the order in another individual [450]. Nucleotide mutations, in addition to rotations, occur rapidly in viruses, and a plausible problem is to determine if the DNA of two individual viruses have mutated away from each other only by a circular rotation, rather than additional mutations.

It is very interesting to note that the problems addressed in the exercises are actually "solved" in nature. Consider the special case of Exercise 2 when string α has length n. Then the problem becomes: Is α a circular rotation of $\bar{\beta}$? This problem is solved in linear time as in Exercise 1. Precisely this matching problem arises and is "solved" in *E. coli* replication under the certain experimental conditions described in [475]. In that experiment, an enzyme (RecA) and ATP molecules (for energy) are added to *E. coli* containing a single strand of one of its plasmids, called string β, and a double-stranded linear DNA molecule, one strand of which is called string α. If α is a circular rotation of $\bar{\beta}$ then the strand opposite to α (which has the DNA sequence complementary to α) hybridizes with β creating a proper double-stranded plasmid, leaving α as a single strand. This transfer of DNA may be a step in the replication of the plasmid. Thus the problem of determining whether α is a circular rotation of $\bar{\beta}$ is solved by this natural system.

Other experiments in [475] can be described as *substring* matching problems relating to circular and linear DNA in *E. coli*. Interestingly, these natural systems solve their matching problems faster than can be explained by kinetic analysis, and the molecular mechanisms used for such rapid matching remain undetermined. These experiments demonstrate the role of enzyme RecA in *E. coli* replication, but do not suggest immediate important computational problems. They do, however, provide indirect motivation for developing computational tools for handling circular strings as well as linear strings. Several other uses of circular strings will be discussed in Sections 7.13 and 16.17 of the book.

3. **Suffix-prefix matching.** Give an algorithm that takes in two strings α and β, of lengths n and m, and finds the longest suffix of α that exactly matches a prefix of β. The algorithm should run in $O(n+m)$ time.

4. **Tandem arrays.** A substring α contained in string S is called a *tandem array* of β (called the base) if α consists of more than one consecutive copy of β. For example, if $S = xyzabcabcabcabcpq$, then $\alpha = abcabcabcabc$ is a tandem array of $\beta = abc$. Note that S also contains a tandem array of *abcabc* (i.e., a tandem array with a longer base). A *maximal* tandem array is a tandem array that cannot be extended either left or right, giving its starting location in S and the number of times β is repeated. A tandem array is an example of a repeated substring (see Section 7.11.1).

   Suppose S has length n. Give an example to show that two maximal tandem arrays of a given base β can overlap.

   Now give an $O(n)$-time algorithm that takes S and β as input, finds every maximal tandem array of β, and outputs the pair (s, k) for each occurrence. Since maximal tandem arrays of a given base can overlap, a naive algorithm would establish only an $O(n^2)$-time bound.

5. If the Z algorithm finds that $Z_2 = q > 0$, all the values $Z_3, \ldots, Z_{q+1}$, Z_{q+2} can then be obtained immediately without additional character comparisons and without executing the main body of Algorithm Z. Flesh out and justify the details of this claim.

6. In Case 2b of the Z algorithm, when $Z_{k'} \geq |\beta|$, the algorithm does explicit comparisons until it finds a mismatch. This is a reasonable way to organize the algorithm, but in fact Case 2b can be refined so as to eliminate an unneeded character comparison. Argue that when $Z_{k'} > |\beta|$ then $Z_k = |\beta|$ and hence no character comparisons are needed. Therefore, explicit character comparisons are needed only in the case that $Z_{k'} = |\beta|$.

7. If Case 2b of the Z algorithm is split into two cases, one for $Z_{k'} > |\beta|$ and one for $Z_{k'} = |\beta|$, would this result in an overall speedup of the algorithm? You must consider all operations, not just character comparisons.

8. Baker [43] introduced the following matching problem and applied it to a problem of software maintenance: "The application is to track down duplication in a large software system. We want to find not only exact matches between sections of code, but parameterized matches, where a parameterized match between two sections of code means that one section can be transformed into the other by replacing the parameter names (e.g., identifiers and constants) of one section by the parameter names of the other via a one-to-one function".

   Now we present the formal definition. Let Σ and Π be two alphabets containing no symbols in common. Each symbol in Σ is called a *token* and each symbol in Π is called a *parameter*. A string can consist of any combinations of tokens and parameters from Σ and Π. For example, if Σ is the upper case English alphabet and Π is the lower case alphabet then $XYabCaCXZddW$ is a legal string over Σ and Π. Two strings S_1 and S_2 are said to *p-match* if and only if

   a. Each token in S_1 (or S_2) is opposite a matching token in S_2 (or S_1).

   b. Each parameter in S_1 (or S_2) is opposite a parameter in S_2 (or S_1).

   c. For any parameter x, if one occurrence of x in S_1 (S_2) is opposite a parameter y in S_2 (S_1), then every occurrence of x in S_1 (S_2) must be opposite an occurrence of y in S_2 (S_1). In other words, the alignment of parameters in S_1 and S_2 defines a one-one correspondence between parameter names in S_1 and parameter names in S_2.

   For example, $S_1 = XYabCaCXZddbW$ *p*-matches $S_2 = XYdxCdCXZccxW$. Notice that parameter a in S_1 maps to parameter d in S_2, while parameter d in S_1 maps to c in S_2. This does not violate the definition of *p*-matching.

   In Baker's application, a token represents a part of the program that cannot be changed,

whereas a parameter represents a program's variable, which can be renamed as long as all occurrences of the variable are renamed consistently. Thus if S_1 and S_2 p-match, then the variable names in S_1 could be changed to the corresponding variable names in S_2, making the two programs identical. If these two programs were part of a larger program, then they could both be replaced by a call to a single subroutine.

The most basic p-match problem is: Given a text T and a pattern P, each a string over Σ and Π, find all substrings of T that p-match P. Of course, one would like to find all those occurrences in $O(|P| + |T|)$ time. Let function Z_i^p for a string S be the length of the longest string starting at position i in S that p-matches a prefix of $S[1..i]$. Show how to modify algorithm Z to compute all the Z_i^p values in $O(|S|)$ time (the implementation details are slightly more involved than for function Z_i, but not too difficult). Then show how to use the modified algorithm Z to find all substrings of T that p-match P, in $O(|P| + |T|)$ time.

In [43] and [239], more involved versions of the p-match problem are solved by more complex methods.

The following three problems can be solved without the *Z* algorithm or other fancy tools. They only require thought.

9. You are given two strings of n characters each and an additional parameter k. In each string there are $n - k + 1$ substrings of length k, and so there are $\Theta(n^2)$ pairs of substrings, where one substring is from one string and one is from the other. For a pair of substrings, we define the *match-count* as the number of opposing characters that match when the two substrings of length k are aligned. The problem is to compute the match-count for each of the $\Theta(n^2)$ pairs of substrings from the two strings. Clearly, the problem can be solved with $O(kn^2)$ operations (character comparisons plus arithmetic operations). But by better organizing the computations, the time can be reduced to $O(n^2)$ operations. (From Paul Horton.)

10. A DNA molecule can be thought of as a string over an alphabet of four characters {*a, t, c, g*} (nucleotides), while a protein can be thought of as a string over an alphabet of twenty characters (amino acids). A gene, which is physically embedded in a DNA molecule, typically encodes the amino acid sequence for a particular protein. This is done as follows. Starting at a particular point in the DNA string, every three consecutive DNA characters encode a single amino acid character in the protein string. That is, three DNA nucleotides specify one amino acid. Such a coding triple is called a *codon*, and the full association of codons to amino acids is called the *genetic code*. For example, the codon *ttt* codes for the amino acid Phenylalanine (abbreviated in the single character amino acid alphabet as *F*), and the codon *gtt* codes for the amino acid Valine (abbreviated as *V*). Since there are $4^3 = 64$ possible triples but only twenty amino acids, there is a possibility that two or more triples form codons for the same amino acid and that some triples do not form codons. In fact, this is the case. For example, the amino acid Leucine is coded for by six different codons.

Problem: Suppose one is given a DNA string of n nucleotides, but you don't know the correct "reading frame". That is, you don't know if the correct decomposition of the string into codons begins with the first, second, or third nucleotide of the string. Each such "frameshift" potentially translates into a different amino acid string. (There are actually known genes where each of the three reading frames not only specifies a string in the amino acid alphabet, but each specifies a functional, yet different, protein.) The task is to produce, for each of the three reading frames, the associated amino acid string. For example, consider the string *atggacggga*. The first reading frame has three complete codons, *atg*, *gac*, and *gga*, which in the genetic code specify the amino acids *Met*, *Asp*, and *Gly*. The second reading frame has two complete codons, *tgg* and *acg*, coding for amino acids *Trp* and *Thr*. The third reading frame has two complete codons, *gga* and *cgg*, coding for amino acids *Gly* and *Arg*.

The goal is to produce the three translations, using the fewest number of character examinations of the DNA string and the fewest number of indexing steps (when using the codons to look up amino acids in a table holding the genetic code). Clearly, the three translations can be done with $3n$ examinations of characters in the DNA and $3n$ indexing steps in the genetic code table. Find a method that does the three translations in at most n character examinations and n indexing steps.

Hint: If you are acquainted with this terminology, the notion of a finite-state transducer may be helpful, although it is not necessary.

11. Let T be a text string of length m and let $\mathcal{S}$ be a *multiset* of n characters. The problem is to find all substrings in T of length n that are formed by the characters of $\mathcal{S}$. For example, let $\mathcal{S}$ = {a, a, b, c} and T = *abahgcabah*. Then *caba* is a substring of T formed from the characters of $\mathcal{S}$.

Give a solution to this problem that runs in $O(m)$ time. The method should also be able to state, for each position i, the length of the longest substring in T starting at i that can be formed from $\mathcal{S}$.

Fantasy protein sequencing. The above problem may become useful in sequencing protein from a particular organism after a large amount of the genome of that organism has been sequenced. This is most easily explained in prokaryotes, where the DNA is not interrupted by introns. In prokaryotes, the amino acid sequence for a given protein is encoded in a contiguous segment of DNA – one DNA codon for each amino acid in the protein. So assume we have the protein molecule but do not know its sequence or the location of the gene that codes for the protein. Presently, chemically determining the amino acid sequence of a protein is very slow, expensive, and somewhat unreliable. However, finding the multiset of amino acids that make up the protein is relatively easy. Now suppose that the whole DNA sequence for the genome of the organism is known. One can use that long DNA sequence to determine the amino acid sequence of a protein of interest. First, translate each codon in the DNA sequence into the amino acid alphabet (this may have to be done three times to get the proper frame) to form the string T; then chemically determine the multiset $\mathcal{S}$ of amino acids in the protein; then find all substrings in T of length $|\mathcal{S}|$ that are formed from the amino acids in $\mathcal{S}$. Any such substrings are candidates for the amino acid sequence of the protein, although it is unlikely that there will be more than one candidate. The match also locates the gene for the protein in the long DNA string.

12. Consider the two-dimensional variant of the preceding problem. The input consists of two-dimensional text (say a filled-in crossword puzzle) and a multiset of characters. The problem is to find a *connected* two-dimensional substructure in the text that matches all the characters in the multiset. How can this be done? A simpler problem is to restrict the structure to be rectangular.

13. As mentioned in Exercise 10, there are organisms (some viruses for example) containing intervals of DNA encoding not just a single protein, but three viable proteins, each read in a different reading frame. So, if each protein contains n amino acids, then the DNA string encoding those three proteins is only $n + 2$ nucleotides (characters) long. That is a very compact encoding.

(**Challenging problem?**) Give an algorithm for the following problem: The input is a protein string S_1 (over the amino acid alphabet) of length n and another protein string of length $m > n$. Determine if there is a string specifying a DNA encoding for S_2 that contains a substring specifying a DNA encoding of S_1. Allow the encoding of S_1 to begin at any point in the DNA string for S_2 (i.e., in any reading frame of that string). The problem is difficult because of the degeneracy of the genetic code and the ability to use any reading frame.

2

Exact Matching: Classical Comparison-Based Methods

2.1. Introduction

This chapter develops a number of classical comparison-based matching algorithms for the exact matching problem. With suitable extensions, all of these algorithms can be implemented to run in linear worst-case time, and all achieve this performance by preprocessing pattern P. (Methods that preprocess T will be considered in Part II of the book.) The original preprocessing methods for these various algorithms are related in spirit but are quite different in conceptual difficulty. Some of the original preprocessing methods are quite difficult.[1] This chapter does not follow the original preprocessing methods but instead exploits fundamental preprocessing, developed in the previous chapter, to implement the needed preprocessing for each specific matching algorithm.

Also, in contrast to previous expositions, we emphasize the Boyer–Moore method over the Knuth-Morris-Pratt method, since Boyer–Moore is the practical method of choice for exact matching. Knuth-Morris-Pratt is nonetheless completely developed, partly for historical reasons, but mostly because it generalizes to problems such as real-time string matching and matching against a set of patterns more easily than Boyer–Moore does. These two topics will be described in this chapter and the next.

2.2. The Boyer–Moore Algorithm

As in the naive algorithm, the Boyer–Moore algorithm successively aligns P with T and then checks whether P matches the opposing characters of T. Further, after the check is complete, P is shifted right relative to T just as in the naive algorithm. However, the Boyer–Moore algorithm contains three clever ideas not contained in the naive algorithm: the right-to-left scan, the bad character shift rule, and the good suffix shift rule. Together, these ideas lead to a method that typically examines fewer than $m + n$ characters (an expected sublinear-time method) and that (with a certain extension) runs in linear worst-case time. Our discussion of the Boyer–Moore algorithm, and extensions of it, concentrates on *provable* aspects of its behavior. Extensive experimental and practical studies of Boyer–Moore and variants have been reported in [229], [237], [409], [410], and [425].

2.2.1. Right-to-left scan

For any alignment of P with T the Boyer–Moore algorithm checks for an occurrence of P by scanning characters from *right to left* rather than from left to right as in the naive algorithm. For example, consider the alignment of P against T shown below:

```
          1         2
          12345678901234567      :
T:        xpbctbxabpqxctbpq
P:          tpabxab
```

To check whether P occurs in T at this position, the Boyer–Moore algorithm starts at the *right* end of P, first comparing $T(9)$ with $P(7)$. Finding a match, it then compares $T(8)$ with $P(6)$, etc., moving right to left until it finds a mismatch when comparing $T(5)$ with $P(3)$. At that point P is shifted *right* relative to T (the amount for the shift will be discussed below) and the comparisons begin again at the right end of P.

Clearly, if P is shifted right by one place after each mismatch, or after an occurrence of P is found, then the worst-case running time of this approach is $O(nm)$ just as in the naive algorithm. So at this point it isn't clear why comparing characters from right to left is any better than checking from left to right. However, with two additional ideas (the *bad character* and the *good suffix* rules), shifts of more than one position often occur, and in typical situations large shifts are common. We next examine these two ideas.

2.2.2. Bad character rule

To get the idea of the bad character rule, suppose that the last (right-most) character of P is y and the character in T it aligns with is $x \neq y$. When this initial mismatch occurs, if we know the right-most position in P of character x, we can safely shift P to the right so that the right-most x in P is below the mismatched x in T. Any shorter shift would only result in an immediate mismatch. Thus, the longer shift is correct (i.e., it will not shift past any occurrence of P in T). Further, if x never occurs in P, then we can shift P completely past the point of mismatch in T. In these cases, some characters of T will never be examined and the method will actually run in "sublinear" time. This observation is formalized below.

Definition For each character x in the alphabet, let $R(x)$ be the position of right-most occurrence of character x in P. $R(x)$ is defined to be zero if x does not occur in P.

It is easy to preprocess P in $O(n)$ time to collect the $R(x)$ values, and we leave that as an exercise. Note that this preprocessing does not require the fundamental preprocessing discussed in Chapter 1 (that will be needed for the more complex shift rule, the good suffix rule).

We use the R values in the following way, called the *bad character shift rule*:

Suppose for a particular alignment of P against T, the right-most $n - i$ characters of P match their counterparts in T, but the next character to the left, $P(i)$, mismatches with its counterpart, say in position k of T. The *bad character rule* says that P should be shifted right by $\max[1, i - R(T(k))]$ places. That is, if the right-most occurrence in P of character $T(k)$ is in position $j < i$ (including the possibility that $j = 0$), then shift P so that character j of P is below character k of T. Otherwise, shift P by one position.

The point of this shift rule is to shift P by more than one character when possible. In the above example, $T(5) = t$ mismatches with $P(3)$ and $R(t) = 1$, so P can be shifted right by two positions. After the shift, the comparison of P and T begins again at the right end of P.

[1] Sedgewick [401] writes "Both the Knuth-Morris-Pratt and the Boyer–Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used". In agreement with Sedgewick, I still do not understand the original Boyer–Moore preprocessing method for the *strong* good suffix rule.

Extended bad character rule

The bad character rule is a useful heuristic for mismatches near the right end of P, but it has no effect if the mismatching character from T occurs in P to the right of the mismatch point. This may be common when the alphabet is small and the text contains many similar, but not exact, substrings. That situation is typical of DNA, which has an alphabet of size four, and even protein, which has an alphabet of size twenty, often contains different regions of high similarity. In such cases, the following *extended bad character rule* is more robust:

> When a mismatch occurs at position i of P and the mismatched character in T is x, then shift P to the right so that the closest x to the left of position i in P is below the mismatched x in T.

Because the extended rule gives larger shifts, the only reason to prefer the simpler rule is to avoid the added implementation expense of the extended rule. The simpler rule uses only $O(|\Sigma|)$ space (Σ is the alphabet) for array R, and one table lookup for each mismatch. As we will see, the extended rule can be implemented to take only $O(n)$ space and at most one extra step per character comparison. That amount of added space is not often a critical issue, but it is an empirical question whether the longer shifts make up for the added time used by the extended rule. The original Boyer–Moore algorithm only uses the simpler bad character rule.

Implementing the extended bad character rule

We preprocess P so that the extended bad character rule can be implemented efficiently in both time and space. The preprocessing should discover, for each position i in P and for each character x in the alphabet, the position of the closest occurrence of x in P to the left of i. The obvious approach is to use a two-dimensional array of size n by $|\Sigma|$ to store this information. Then, when a mismatch occurs at position i of P and the mismatching character in T is x, we look up the (i, x) entry in the array. The lookup is fast, but the size of the array, and the time to build it, may be excessive. A better compromise, below, is possible.

During preprocessing, scan P from right to left collecting, for each character x in the alphabet, a list of the positions where x occurs in P. Since the scan is right to left, each list will be in decreasing order. For example, if $P = abacbabc$ then the list for character a is 6, 3, 1. These lists are accumulated in $O(n)$ time and of course take only $O(n)$ space. During the search stage of the Boyer–Moore algorithm if there is a mismatch at position i of P and the mismatching character in T is x, scan x's list from the top until we reach the first number less than i or discover there is none. If there is none then there is no occurrence of x before i, and all of P is shifted past the x in T. Otherwise, the found entry gives the desired position of x.

After a mismatch at position i of P the time to scan the list is at most $n - i$, which is roughly the number of characters that matched. So in worst case, this approach at most doubles the running time of the Boyer–Moore algorithm. However, in most problem settings the added time will be vastly less than double. One could also do binary search on the list in circumstances that warrant it.

2.2.3. The (strong) good suffix rule

The bad character rule by itself is reputed to be highly effective in practice, particularly for English text [229], but proves less effective for small alphabets and it does not lead to a linear worst-case running time. For that, we introduce another rule called the *strong good suffix rule*. The original preprocessing method [278] for the strong good suffix rule is generally considered quite difficult and somewhat mysterious (although a weaker version of it is easy to understand). In fact, the preprocessing for the strong rule was given incorrectly in [278] and corrected, without much explanation, in [384]. Code based on [384] is given without real explanation in the text by Baase [32], but there are no published sources that try to fully explain the method.[2] Pascal code for strong preprocessing, based on an outline by Richard Cole [107], is shown in Exercise 24 at the end of this chapter.

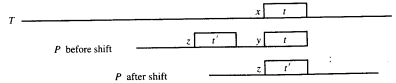


Figure 2.1: Good suffix shift rule, where character x of T mismatches with character y of P. Characters y and z of P are guaranteed to be distinct by the good suffix rule, so z has a chance of matching x.

In contrast, the fundamental preprocessing of P discussed in Chapter 1 makes the needed preprocessing very simple. That is the approach we take here. The *strong good suffix rule* is:

> Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and *the character to the left of t' in P differs from the character to the left of t in P*. Shift P to the right so that substring t' in P is below substring t in T (see Figure 2.1). If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places, that is, shift P past t in T.

For a specific example consider the alignment of P and T given below:

```
          0         1
          12345678901234567 8
T:        prstabstubabvqxrst
                      *
P:          qcabdabdab
            1234567890
```

When the mismatch occurs at position 8 of P and position 10 of T, $t = ab$ and t' occurs in P starting at position 3. Hence P is shifted right by *six* places, resulting in the following alignment:

[2] A recent plea appeared on the internet newsgroup comp. theory:

> I am looking for an elegant (easily understandable) proof of correctness for a part of the Boyer–Moore string matching algorithm. The difficult-to-prove part here is the algorithm that computes the ds_2 (good-suffix) table. I didn't find much of an understandable proof yet, so I'd much appreciate any help!

```
          0         1
          123456789012345678
T:        prstabstubabvqxrst
P:              qcabdabdab
```

Note that the extended bad character rule would have shifted P by only one place in this example.

Theorem 2.2.1. *The use of the good suffix rule never shifts P past an occurrence in T.*

PROOF Suppose the right end of P is aligned with character k of T before the shift, and suppose that the good suffix rule shifts P so its right end aligns with character $k' > k$. Any occurrence of P ending at a position l strictly between k and k' would immediately violate the selection rule for k', since it would imply either that a closer copy of t occurs in P or that a longer prefix of P matches a suffix of t. □

The original published Boyer–Moore algorithm [75] uses a simpler, weaker, version of the good suffix rule. That version just requires that the shifted P agree with the t and does not specify that the next characters to the left of those occurrences of t be different. An explicit statement of the weaker rule can be obtained by deleting the italics phrase in the first paragraph of the statement of the strong good suffix rule. In the previous example, the weaker shift rule shifts P by three places rather than six. When we need to distinguish the two rules, we will call the simpler rule the *weak* good suffix rule and the rule stated above the *strong* good suffix rule. For the purpose of proving that the search part of Boyer–Moore runs in linear worst-case time, the weak rule is not sufficient, and in this book the strong version is assumed unless stated otherwise.

2.2.4. Preprocessing for the good suffix rule

We now formalize the preprocessing needed for the Boyer–Moore algorithm.

Definition For each i, $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. $L(i)$ is defined to be zero if there is no position satisfying the conditions. For each i, $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding that suffix is not equal to $P(i-1)$. $L'(i)$ is defined to be zero if there is no position satisfying the conditions.

For example, if $P = cabdabdab$, then $L(8) = 6$ and $L'(8) = 3$.

$L(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, whereas $L'(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, with the stronger, added condition that its preceding character is unequal to $P(i-1)$. So, in the strong-shift version of the Boyer–Moore algorithm, if character $i-1$ of P is involved in a mismatch and $L'(i) > 0$, then P is shifted right by $n - L'(i)$ positions. The result is that if the right end of P was aligned with position k of T before the shift, then position $L'(i)$ is now aligned with position k.

During the preprocessing stage of the Boyer–Moore algorithm $L'(i)$ (and $L(i)$, if desired) will be computed for each position i in P. This is done in $O(n)$ time via the following definition and theorem.

Definition For string P, $N_j(P)$ is the length of the longest *suffix* of the substring $P[1..j]$ that is also a *suffix* of the full string P.

For example, if $P = cabdabdab$, then $N_3(P) = 2$ and $N_6(P) = 5$.

Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and matches a prefix of S. Clearly, N is the reverse of Z, that is, if P' denotes the string obtained by reversing P, then $N_j(P) = Z_{n-j+1}(P')$. Hence the $N_j(P)$ values can be obtained in $O(n)$ time by using *Algorithm* Z on P'. The following theorem is then immediate.

Theorem 2.2.2. *$L(i)$ is the largest index j less than n such that $N_j(P) \geq |P[i..n]|$ (which is $n-i+1$). $L'(i)$ is the largest index j less than n such that $N_j(P) = |P[i..n]| = (n-i+1)$.*

Given Theorem 2.2.2, it follows immediately that all the $L'(i)$ values can be accumulated in linear time from the N values using the following algorithm:

Z-based Boyer–Moore

```
for i := 1 to n do L'(i) := 0;
for j := 1 to n − 1 do
    begin
    i := n − N_j(P) + 1;
    L'(i) := j;
    end;
```

The $L(i)$ values (if desired) can be obtained by adding the following lines to the above pseudocode:

```
L(2) := L'(2);
for i := 3 to n do L(i) := max[L(i − 1), L'(i)];
```

Theorem 2.2.3. *The above method correctly computes the L values.*

PROOF $L(i)$ marks the right end-position of the right-most substring of P that matches $P[i..n]$ and is not a suffix of $P[1..n]$. Therefore, that substring begins at position $L(i)-n+i$, which we will denote by j. We will prove that $L(i) = \max[L(i-1), L'(i)]$ by considering what character $j-1$ is. First, if $j = 1$ then character $j-1$ doesn't exist, so $L(i-1) = 0$ and $L'(i) = 1$. So suppose that $j > 1$. If character $j-1$ equals character $i-1$ then $L(i) = L(i-1)$. If character $j-1$ does not equal character $i-1$ then $L(i) = L'(i)$. Thus, in all cases, $L(i)$ must either be $L'(i)$ or $L(i-1)$.

However, $L(i)$ must certainly be greater than or equal to both $L'(i)$ and $L(i-1)$. In summary, $L(i)$ must either be $L'(i)$ or $L(i-1)$, and yet it must be greater or equal to both of them; hence $L(i)$ must be the maximum of $L'(i)$ and $L(i-1)$. □

Final preprocessing detail

The preprocessing stage must also prepare for the case when $L'(i) = 0$ or when an occurrence of P is found. The following definition and theorem accomplish that.

Definition Let $l'(i)$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of P, if one exists. If none exists, then let $l'(i)$ be zero.

Theorem 2.2.4. *$l'(i)$ equals the largest $j \leq |P[i..n]|$, which is $n-i+1$, such that $N_j(P) = j$.*

We leave the proof, as well as the problem of how to accumulate the $l'(i)$ values in linear time, as a simple exercise. (Exercise 9 of this chapter)

2.2.5. The good suffix rule in the search stage of Boyer–Moore

Having computed $L'(i)$ and $l'(i)$ for each position i in P, these preprocessed values are used during the search stage of the algorithm to achieve larger shifts. If, during the search stage, a mismatch occurs at position $i-1$ of P and $L'(i) > 0$, then the good suffix rule shifts P by $n - L'(i)$ places to the right, so that the $L'(i)$-length prefix of the shifted P aligns with the $L'(i)$-length suffix of the unshifted P. In the case that $L'(i) = 0$, the good suffix rule shifts P by $n - l'(i)$ places. When an occurrence of P is found, then the rule shifts P by $n - l'(2)$ places. Note that the rules work correctly even when $l'(i) = 0$.

One special case remains. When the first comparison is a mismatch (i.e., $P(n)$ mismatches) then P should be shifted one place to the right.

2.2.6. The complete Boyer–Moore algorithm

We have argued that neither the good suffix rule nor the bad character rule shift P so far as to miss any occurrence of P. So the Boyer–Moore algorithm shifts by the largest amount given by either of the rules. We can now present the complete algorithm.

The Boyer–Moore algorithm

```
{Preprocessing stage}
    Given the pattern P,
    Compute L'(i) and l'(i) for each position i of P,
    and compute R(x) for each character x ∈ Σ.
{Search stage}
    k := n;
    while k ≤ m do
        begin
        i := n;
        h := k;
        while i > 0 and P(i) = T(h) do
            begin
            i := i − 1;
            h := h − 1;
            end;
        if i = 0 then
            begin
            report an occurrence of P in T ending at position k.
            k := k + n − l'(2);
            end
        else
            shift P (increase k) by the maximum amount determined by the
            (extended) bad character rule and the good suffix rule.
        end;
```

Note that although we have always talked about "shifting P", and given rules to determine by how much P should be "shifted", there is no shifting in the actual implementation. Rather, the index k is increased to the point where the right end of P would be "shifted". Hence, each act of shifting P takes constant time.

We will later show, in Section 3.2, that by using the strong good suffix rule alone, the Boyer–Moore method has a worst-case running time of $O(m)$ provided that the pattern does not appear in the text. This was first proved by Knuth, Morris, and Pratt [278], and an alternate proof was given by Guibas and Odlyzko [196]. Both of these proofs were quite difficult and established worst-case time bounds no better than $5m$ comparisons. Later, Richard Cole gave a much simpler proof [108] establishing a bound of $4m$ comparisons and also gave a difficult proof establishing a tight bound of $3m$ comparisons. We will present Cole's proof of $4m$ comparisons in Section 3.2.

When the pattern does appear in the text then the original Boyer–Moore method runs in $\Theta(nm)$ worst-case time. However, several simple modifications to the method correct this problem, yielding an $O(m)$ time bound in all cases. The first of these modifications was due to Galil [168]. After discussing Cole's proof, in Section 3.2, for the case that P doesn't occur in T, we use a variant of Galil's idea to achieve the linear time bound in all cases.

At the other extreme, if we only use the bad character shift rule, then the worst-case running time is $O(nm)$, but assuming randomly generated strings, the expected running time is sublinear. Moreover, in typical string matching applications involving natural language text, a sublinear running time is almost always observed in practice. We won't discuss random string analysis in this book but refer the reader to [184].

Although Cole's proof for the linear worst case is vastly simpler than earlier proofs, and is important in order to complete the full story of Boyer–Moore, it is not trivial. However, a fairly simple extension of the Boyer–Moore algorithm, due to Apostolico and Giancarlo [26], gives a "Boyer–Moore–like" algorithm that allows a fairly direct proof of a $2m$ worst-case bound on the number of comparisons. The Apostolico–Giancarlo variant of Boyer–Moore is discussed in Section 3.1.

2.3. The Knuth-Morris-Pratt algorithm

The best known linear-time algorithm for the exact matching problem is due to Knuth, Morris, and Pratt [278]. Although it is rarely the method of choice, and is often much inferior in practice to the Boyer–Moore method (and others), it can be simply explained, and its linear time bound is (fairly) easily proved. The algorithm also forms the basis of the well-known Aho–Corasick algorithm, which efficiently finds all occurrences in a text of any pattern from a *set* of patterns.[3]

2.3.1. The Knuth-Morris-Pratt shift idea

For a given alignment of P with T, suppose the naive algorithm matches the first i characters of P against their counterparts in T and then mismatches on the next comparison. The naive algorithm would shift P by just *one* place and begin comparing again from the left end of P. But a larger shift may often be possible. For example, if $P = abcxabcde$ and, in the present alignment of P with T, the mismatch occurs in position 8 of P, then it is easily deduced (and we will prove below) that P can be shifted by four places without passing over any occurrences of P in T. Notice that this can be deduced without even knowing what string T is or exactly how P is aligned with T. Only the location of the mismatch in P must be known. The Knuth-Morris-Pratt algorithm is based on this kind of reasoning to make larger shifts than the naive algorithm makes. We now formalize this idea.

[3] We will present several solutions to that set problem including the Aho–Corasick method in Section 3.4. For those reasons, and for its historical role in the field, we fully develop the Knuth-Morris-Pratt method here.

Definition For each position i in pattern P, define $sp_i(P)$ to be the *length* of the longest proper *suffix* of $P[1..i]$ that matches a prefix of P.

Stated differently, $sp_i(P)$ is the length of the longest proper substring of $P[1..i]$ that ends at i and that matches a prefix of P. When the string is clear by context we will use sp_i in place of the full notation.

For example, if $P = abcaeabcabd$, then $sp_2 = sp_3 = 0$, $sp_4 = 1$, $sp_8 = 3$, and $sp_{10} = 2$. Note that by definition, $sp_1 = 0$ for any string.

An optimized version of the Knuth-Morris-Pratt algorithm uses the following values.

Definition For each position i in pattern P, define $sp'_i(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P, *with the added condition that characters* $P(i+1)$ *and* $P(sp'_i + 1)$ *are unequal.*

Clearly, $sp'_i(P) \leq sp_i(P)$ for all positions i and any string P. As an example, if $P = bbccaebbcabd$, then $sp_8 = 2$ because string bb occurs both as a proper prefix of $P[1..8]$ and as a suffix of $P[1..8]$. However, both copies of the string are followed by the same character c, and so $sp'_8 < 2$. In fact, $sp'_8 = 1$ since the single character b occurs as both the first and last character of $P[1..8]$ and is followed by character b in position 2 and by character c in position 9.

The Knuth-Morris-Pratt shift rule

We will describe the algorithm in terms of the sp' values, and leave it to the reader to modify the algorithm if only the weaker sp values are used.[4] The Knuth-Morris-Pratt algorithm aligns P with T and then compares the aligned characters from *left to right*, as the naive algorithm does.

For any alignment of P and T, if the first mismatch (comparing from left to right) occurs in position $i+1$ of P and position k of T, then shift P to the right (relative to T) so that $P[1..sp'_i]$ aligns with $T[k - sp'_i..k-1]$. In other words, shift P exactly $i + 1 - (sp'_i + 1) = i - sp'_i$ places to the right, so that character $sp'_i + 1$ of P will align with character k of T. In the case that an occurrence of P has been found (no mismatch), shift P by $n - sp'_n$ places.

The shift rule guarantees that the prefix $P[1..sp'_i]$ of the shifted P matches its opposing substring in T. The next comparison is then made between characters $T(k)$ and $P[sp'_i + 1]$. The use of the stronger shift rule based on sp'_i guarantees that the same mismatch will not occur again in the new alignment, but it does not guarantee that $T(k) = P[sp'_i + 1]$.

In the above example, where $P = abcxabcde$ and $sp'_7 = 3$, if character 8 of P mismatches then P will be shifted by $7 - 3 = 4$ places. This is true even without knowing T or how P is positioned with T.

The advantage of the shift rule is twofold. First, it often shifts P by more than just a single character. Second, after a shift, the left-most sp'_i characters of P are guaranteed to match their counterparts in T. Thus, to determine whether the newly shifted P matches its counterpart in T, the algorithm can start comparing P and T at position $sp'_i + 1$ of P (and position k of T). For example, suppose $P = abcxabcde$ as above, $T = xyabcxabcxadcdqfeg$, and the left end of P is aligned with character 3 of T. Then P and T will match for 7 characters but mismatch on character 8 of P, and P will be shifted

[4] The reader should be alerted that traditionally the Knuth-Morris-Pratt algorithm has been described in terms of *failure functions*, which are related to the sp_i values. Failure functions will be explicitly defined in Section 2.3.3.

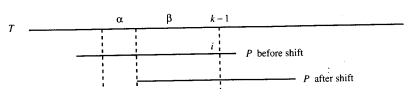


Figure 2.2: Assumed missed occurrence used in correctness proof for Knuth-Morris-Pratt.

by 4 places as shown below:

```
1         2
12345678901234567 8
xyabcxabcxadcdqfeg
  abcxabcde
     abcxabcde
```

As guaranteed, the first 3 characters of the shifted P match their counterparts in T (and their counterparts in the unshifted P).

Summarizing, we have

Theorem 2.3.1. *After a mismatch at position* $i+1$ *of* P *and a shift of* $i - sp'_i$ *places to the right, the left-most* sp'_i *characters of* P *are guaranteed to match their counterparts in* T.

Theorem 2.3.1 partially establishes the correctness of the Knuth-Morris-Pratt algorithm, but to fully prove correctness we have to show that the shift rule never shifts too far. That is, using the shift rule no occurrence of P will ever be overlooked.

Theorem 2.3.2. *For any alignment of* P *with* T, *if characters 1 through* i *of* P *match the opposing characters of* T *but character* $i+1$ *mismatches* $T(k)$, *then* P *can be shifted by* $i - sp'_i$ *places to the right without passing any occurrence of* P *in* T.

PROOF Suppose not, so that there is an occurrence of P starting strictly to the left of the shifted P (see Figure 2.2), and let α and β be the substrings shown in the figure. In particular, β is the prefix of P of length sp'_i, shown relative to the shifted position of P. The unshifted P matches T up through position i of P and position $k-1$ of T, and all characters in the (assumed) missed occurrence of P match their counterparts in T. Both of these matched regions contain the substrings α and β, so the unshifted P and the assumed occurrence of P match on the entire substring $\alpha\beta$. Hence $\alpha\beta$ is a suffix of $P[1..i]$ that matches a proper prefix of P. Now let $l = |\alpha\beta| + 1$ so that position l in the "missed occurrence" of P is opposite position k in T. Character $P(l)$ cannot be equal to $P(i+1)$ since $P(l)$ is assumed to match $T(k)$ and $P(i+1)$ does not match $T(k)$. Thus $\alpha\beta$ is a proper suffix of $P[1..i]$ that matches a prefix of P, and the next character is unequal to $P(i+1)$. But $|\alpha| > 0$ due to the assumption that an occurrence of P starts strictly before the shifted P, so $|\alpha\beta| > |\beta| = sp'_i$, contradicting the definition of sp'_i. Hence the theorem is proved. □

Theorem 2.3.2 says that the Knuth-Morris-Pratt shift rule does not miss any occurrence of P in T, and so the Knuth-Morris-Pratt algorithm will correctly find all occurrences of P in T. The time analysis is equally simple.

Theorem 2.3.3. *In the Knuth-Morris-Pratt method, the number of character comparisons is at most* $2m$.

PROOF Divide the algorithm into compare/shift phases, where a single phase consists of the comparisons done between successive shifts. After any shift, the comparisons in the phase go left to right and start either with the last character of T compared in the previous phase or with the character to its right. Since P is never shifted left, in any phase at most one comparison involves a character of T that was previously compared. Thus, the total number of character comparisons is bounded by $m + s$, where s is the number of shifts done in the algorithm. But $s < m$ since after m shifts the right end of P is certainly to the right of the right end of T, so the number of comparisons done is bounded by $2m$. □

2.3.2. Preprocessing for Knuth-Morris-Pratt

The key to the speed up of the Knuth-Morris-Pratt algorithm over the naive algorithm is the use of sp' (or sp) values. It is easy to see how to compute all the sp' and sp values from the Z values obtained during the fundamental preprocessing of P. We verify this below.

Definition Position $j > 1$ *maps to* i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j.

Theorem 2.3.4. *For any* $i > 1$, $sp'_i(P) = Z_j = i - j + 1$, *where* $j > 1$ *is the smallest position that maps to* i. *If there is no such* j *then* $sp'_i(P) = 0$. *For any* $i > 1$, $sp_i(P) = i - j + 1$, *where* j *is the smallest position in the range* $1 < j \leq i$ *that maps to* i *or beyond. If there is no such* j, *then* $sp_i(P) = 0$.

PROOF If $sp'_i(P)$ is greater than zero, then there is a proper suffix α of $P[1..i]$ that matches a prefix of P, such that $P[i+1]$ does not match $P[|\alpha|+1]$. Therefore, letting j denote the start of α, $Z_j = |\alpha| = sp'_i(P)$ and j maps to i. Hence, if there is no j in the range $1 < j \leq i$ that maps to i, then $sp'_i(P)$ must be zero.

Now suppose $sp'_i(P) > 0$ and let j be as defined above. We claim that j is the smallest position in the range 2 to i that maps to i. Suppose not, and let j^* be a position in the range $1 < j^* < j$ that maps to i. Then $P[j^*..i]$ would be a proper suffix of $P[1..i]$ that matches a prefix (call it β) of P. Moreover, by the definition of mapping, $P(i+1) \neq P(|\beta|)$, so $sp'_i(P) \geq |\beta| > |\alpha|$, contradicting the assumption that $sp'_i = \alpha$.

The proofs of the claims for $sp_i(P)$ are similar and are left as exercises. □

Given Theorem 2.3.4, all the sp' and sp values can be computed in linear time using the Z_i values as follows:

Z-based Knuth-Morris-Pratt

```
for i := 1 to n do
    sp'_i := 0;
for j := n downto 2 do
begin
    i := j + Z_j(P) - 1;
    sp'_i := Z_j;
end;
```

The sp values are obtained by adding the following:


```
sp_n(P) := sp'_n(P);
for i := n - 1 downto 2 do
    sp_i(P) := max[sp_{i+1}(P) - 1, sp'_i(P)]
```

2.3.3. A full implementation of Knuth-Morris-Pratt

We have described the Knuth-Morris-Pratt algorithm in terms of shifting P, but we never accounted for time needed to implement shifts. The reason is that shifting is only conceptual and P is never explicitly shifted. Rather, as in the case of Boyer–Moore, pointers to P and T are incremented. We use pointer p to point into P and one pointer c (for "current" character) to point into T.

Definition For each position i from 1 to $n+1$, define the failure function $F'(i)$ to be $sp'_{i-1} + 1$ (and define $F(i) = sp_{i-1} + 1$), where sp'_0 and sp_0 are defined to be zero.

We will only use the (stronger) failure function $F'(i)$ in this discussion but will refer to $F(i)$ later.

After a mismatch in position $i + 1 > 1$ of P, the Knuth-Morris-Pratt algorithm "shifts" P so that the next comparison is between the character in position c of T and the character in position $sp'_i + 1$ of P. But $sp'_i + 1 = F'(i+1)$, so a general "shift" can be implemented in constant time by just setting p to $F'(i+1)$. Two special cases remain. When the mismatch occurs in position 1 of P, then p is set to $F'(1) = 1$ and c is incremented by one. When an occurrence of P is found, then P is shifted right by $n - sp'_n$ places. This is implemented by setting $F'(n+1)$ to $sp'_n + 1$.

Putting all the pieces together gives the full Knuth-Morris-Pratt algorithm.

Knuth-Morris-Pratt algorithm

```
begin
Preprocess P to find F'(k) = sp'_{k-1} + 1 for k from 1 to n + 1.
    c := 1;
    p := 1;
    While c + (n - p) ≤ m
    do begin
        While P(p) = T(c) and p ≤ n
        do begin
            p := p + 1;
            c := c + 1;
        end;
    if p = n + 1 then
        report an occurrence of P starting at position c - n of T.
    if p := 1 then c := c + 1
    p := F'(p);
    end;
end.
```

2.4. Real-time string matching

In the search stage of the Knuth-Morris-Pratt algorithm, P is aligned against a substring of T and the two strings are compared left to right until either all of P is exhausted (in which

case an occurrence of P in T has been found) or until a mismatch occurs at some positions $i+1$ of P and k of T. In the latter case, if $sp'_i > 0$, then P is shifted right by $i - sp'_i$ positions, guaranteeing that the prefix $P[1..sp'_i]$ of the shifted pattern matches its opposing substring in T. No explicit comparison of those substrings is needed, and the next comparison is between characters $T(k)$ and $P(sp'_i + 1)$. Although the shift based on sp'_i guarantees that $P(i+1)$ differs from $P(sp'_i + 1)$, it does not guarantee that $T(k) = P(sp'_i + 1)$. Hence $T(k)$ might be compared several times (perhaps $\Omega(|P|)$ times) with differing characters in P. For that reason, the Knuth-Morris-Pratt method is not a *real-time* method.

To be real time, a method must do at most a *constant* amount of work between the time it first examines any position in T and the time it last examines that position. In the Knuth-Morris-Pratt method, if a position of T is involved in a match, it is never examined again (this is easy to verify) but, as indicated above, this is not true when the position is involved in a mismatch. Note that the definition of real time only concerns the search stage of the algorithm. Preprocessing of P need not be real time. Note also that if the search stage is real time it certainly is also linear time.

The utility of a real-time matcher is two fold. First, in certain applications, such as when the characters of the text are being sent to a small memory machine, one might need to guarantee that each character can be fully processed before the next one is due to arrive. If the processing time for each character is constant, independent of the length of the string, then such a guarantee may be possible. Second, in this particular real-time matcher, the shifts of P may be longer but never shorter than in the original Knuth-Morris-Pratt algorithm. Hence, the real-time matcher may run faster in certain problem instances.

Admittedly, arguments in favor of real-time matching algorithms over linear-time methods are somewhat tortured, and the real-time matching is more a theoretical issue than a practical one. Still, it seems worthwhile to spend a little time discussing real-time matching.

2.4.1. Converting Knuth-Morris-Pratt to a real-time method

We will use the Z values obtained during fundamental preprocessing of P to convert the Knuth-Morris-Pratt method into a real-time method. The required preprocessing of P is quite similar to the preprocessing done in Section 2.3.2 for the Knuth-Morris-Pratt algorithm. For historical reasons, the resulting real-time method is generally referred to as a *deterministic finite-state string matcher* and is often represented with a finite state machine diagram. We will not use this terminology here and instead represent the method in pseudo code.

Definition Let x denote a character of the alphabet. For each position i in pattern P, define $sp'_{(i,x)}(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P, *with the added condition that character* $P(sp'_i + 1)$ *is* x.

Knowing the $sp'_{(i,x)}$ values for each character x in the alphabet allows a shift rule that converts the Knuth-Morris-Pratt method into a real-time algorithm. Suppose P is compared against a substring of T and a mismatch occurs at characters $T(k) = x$ and $P(i+1)$. Then P should be shifted right by $i - sp'_{(i,x)}$ places. This shift guarantees that the prefix $P[1..sp'_{(i,x)}]$ matches the opposing substring in T and that $T(k)$ matches the next character in P. Hence, the comparison between $T(k)$ and $P(sp'_{(i,x)} + 1)$ can be skipped. The next needed comparison is between characters $P(sp'_{(i,x)} + 2)$ and $T(k+1)$. With this shift rule, the method becomes real time because it still never reexamines a position in T involved in a match (a feature inherited from the Knuth-Morris-Pratt algorithm), and it now also never reexamines a position involved in a mismatch. So, the search stage of this algorithm never examines a character in T more than once. It follows that the search is done in real time. Below we show how to find all the $sp'_{(i,x)}$ values in linear time. Together, this gives an algorithm that does linear preprocessing of P and real-time search of T.

It is easy to establish that the algorithm finds all occurrences of P in T, and we leave that as an exercise.

2.4.2. Preprocessing for real-time string matching

Theorem 2.4.1. *For* $P[i+1] \neq x$, $sp'_{(i,x)}(P) = i - j + 1$, *where* j *is the smallest position such that* j *maps to* i *and* $P(Z_j + 1) = x$. *If there is no such* j *then* $sp'_{(i,x)}(P) = 0$.

The proof of this theorem is almost identical to the proof of Theorem 2.3.4 (page 26) and is left to the reader. Assuming (as usual) that the alphabet is finite, the following minor modification of the preprocessing given earlier for Knuth-Morris-Pratt (Section 2.3.2) yields the needed $sp'_{(i,x)}$ values in linear time:

Z-based real-time matching

```
for i := 1 to n do
    sp'(i,x) := 0 for every character x;
for j := n downto 2 do
    begin
        i := j + Z_j(P) - 1;
        x := P(Z_j + 1);
        sp'(i,x) := Z_j;
    end;
```

Note that the linear time (and space) bound for this method require that the alphabet Σ be finite. This allows us to do $|\Sigma|$ comparisons in constant time. If the size of the alphabet is explicitly included in the time and space bounds, then the preprocessing time and space needed for the algorithm is $O(|\Sigma|n)$.

2.5. Exercises

1. In "typical" applications of exact matching, such as when searching for an English word in a book, the simple bad character rule seems to be as effective as the extended bad character rule. Give a "hand-waving" explanation for this.
2. When searching for a single word or a small phrase in a large English text, brute force (the naive algorithm) is reported [184] to run faster than most other methods. Give a hand-waving explanation for this. In general terms, how would you expect this observation to hold up with smaller alphabets (say in DNA with an alphabet size of four), as the size of the pattern grows, and when the text has many long sections of similar but not exact substrings?
3. "Common sense" and the $\Theta(nm)$ worst-case time bound of the Boyer–Moore algorithm (using only the bad character rule) both would suggest that empirical running times increase with increasing pattern length (assuming a fixed text). But when searching in actual English

texts, the Boyer–Moore algorithm runs faster in practice when given longer patterns. Thus, on an English text of about 300,000 characters, it took about five times as long to search for the word "Inter" as it did to search for "Interactively".

Give a hand-waving explanation for this. Consider now the case that the pattern length increases without bound. At what point would you expect the search times to stop decreasing? Would you expect search times to start increasing at some point?

4. Evaluate empirically the utility of the extended bad character rule compared to the original bad character rule. Perform the evaluation in combination with different choices for the two good-suffix rules. How much more is the average shift using the extended rule? Does the extra shift pay for the extra computation needed to implement it?

5. Evaluate empirically, using different assumptions about the sizes of P and T, the number of occurrences of P in T, and the size of the alphabet, the following idea for speeding up the Boyer–Moore method. Suppose that a phase ends with a mismatch and that the good suffix rule shifts P farther than the extended bad character rule. Let x and y denote the mismatching characters in T and P respectively, and let z denote the character in the shifted P below x. By the suffix rule, z will not be y, but there is no guarantee that it will be x. So rather than starting comparisons from the right of the shifted P, as the Boyer–Moore method would do, why not first compare x and z? If they are equal then a right-to-left comparison is begun from the right end of P, but if they are unequal then we apply the extended bad character rule from z in P. This will shift P again. At that point we must begin a right-to-left comparison of P against T.

6. The idea of the bad character rule in the Boyer–Moore algorithm can be generalized so that instead of examining characters in P from right to left, the algorithm compares characters in P in the order of how unlikely they are to be in T (most *unlikely* first). That is, it looks first at those characters in P that are least likely to be in T. Upon mismatching, the bad character rule or extended bad character rule is used as before. Evaluate the utility of this approach, either empirically on real data or by analysis assuming random strings.

7. Construct an example where fewer comparisons are made when the bad character rule is used alone, instead of combining it with the good suffix rule.

8. Evaluate empirically the effectiveness of the strong good suffix shift for Boyer–Moore versus the weak shift rule.

9. Give a proof of Theorem 2.2.4. Then show how to accumulate all the $l'(i)$ values in linear time.

10. If we use the weak good suffix rule in Boyer–Moore that shifts the closest copy of t under the matched suffix t, but doesn't require the next character to be different, then the preprocessing for Boyer–Moore can be based directly on sp_i values rather than on Z values. Explain this.

11. Prove that the Knuth-Morris-Pratt shift rules (either based on sp or sp') do not miss any occurrences of P in T.

12. It is possible to incorporate the bad character shift rule from the Boyer–Moore method to the Knuth-Morris-Pratt method or to the naive matching method itself. Show how to do that. Then evaluate how effective that rule is and explain why it is more effective when used in the Boyer–Moore algorithm.

13. Recall the definition of l_i on page 8. It is natural to conjecture that $sp_i = i - l_i$ for any index i, where $i \geq l_i$. Show by example that this conjecture is incorrect.

14. Prove the claims in Theorem 2.3.4 concerning $sp'(P)$.

15. Is it true that given only the sp values for a given string P, the sp' values are completely determined? Are the sp values determined from the sp' values alone?

Using *sp* values to compute *Z* values

In Section 2.3.2, we showed that one can compute all the *sp* values knowing only the *Z* values for string *S* (i.e., not knowing *S* itself). In the next five exercises we establish the converse, creating a linear-time algorithm to compute all the *Z* values from *sp* values alone. The first exercise suggests a natural method to accomplish this, and the following exercise exposes a hole in that method. The final three exercises develop a correct linear-time algorithm, detailed in [202]. We say that sp_i *maps to* k *if* $k = i - sp_i + 1$.

16. Suppose there is a position i such that sp_i maps to k, and let i be the largest such position. Prove that $Z_k = i - k + 1 = sp_i$ and that $r_k = i$.

17. Given the answer to the previous exercise, it is natural to conjecture that Z_k always equals sp_i, where i is the largest position such that sp_i maps to k. Show that this is not true. Given an example using at least three distinct characters.

    Stated another way, give an example to show that Z_k can be greater than zero even when there is *no* position i such that sp_i maps to k.

18. Recall that r_{k-1} is known at the start of iteration k of the Z algorithm (when Z_k is computed), but r_k is known only at the end of iteration k. Suppose, however, that r_k is known (somehow) at the start of iteration k. Show how the Z algorithm can then be modified to compute Z_k using *no* character comparisons. Hence this modified algorithm need not even know the string S.

19. Prove that if Z_k is greater than zero, then r_k equals the largest position i such that $k \geq i - sp_i$. Conclude that r_k can be deduced from the sp values for every position k where Z_k is not zero.

20. Combine the answers to the previous two exercises to create a linear-time algorithm that computes all the Z values for a string S given only the sp values for S and not the string S itself.

    Explain in what way the method is a "simulation" of the Z algorithm.

21. It may seem that $l'(i)$ (needed for Boyer–Moore) should be sp_n for any i. Show why this is not true.

22. In Section 1.5 we showed that all the occurrences of P in T could be found in linear time by computing the Z values on the string $S = P\$T$. Explain how the method would change if we use $S = PT$, that is, we do not use a separator symbol between P and T. Now show how to find all occurrences of P in T in linear time using $S = PT$, but with sp values in place of Z values. (This is not as simple as it might at first appear.)

23. In Boyer–Moore and Boyer–Moore–like algorithms, the search moves right to left in the pattern, although the pattern moves left to right relative to the text. That makes it more difficult to explain the methods and to combine the preprocessing for Boyer–Moore with the preprocessing for Knuth-Morris-Pratt. However, a small change to Boyer–Moore would allow an easier exposition and more uniform preprocessing. First, place the pattern at the *right* end of the text, and conduct each search *left to right* in the pattern, shifting the pattern *left* after a mismatch. Work out the details of this approach, and show how it allows a more uniform exposition of the preprocessing needed for it and for Knuth-Morris-Pratt. Argue that on average this approach has the same behavior as the original Boyer–Moore method.

24. Below is working Pascal code (in Turbo Pascal) implementing Richard Cole's preprocessing, for the strong good suffix rule. It is different than the approach based on fundamental preprocessing and is closer to the original method in [278]. Examine the code to extract the algorithm behind the program. Then explain the idea of the algorithm, prove correctness of the algorithm, and analyze its running time. The point of the exercise is that it is difficult to convey an algorithmic idea using a program.

```
program gsmatch(input,output);
{This is an implementation of Richard Cole's
preprocessing for the strong good suffix rule}
type
tstring = string[200];
indexarray = array[1..100] of integer;

const
zero = 0;

var
p:tstring;
bmshift,matchshift:indexarray;
m,i:integer;

procedure readstring(var p:tstring; var m:integer);

begin
read(p);

m:=Length(p);
writeln('the length of the string is ', m);

end;

procedure gsshift(p:tstring; var
gs_shift:indexarray;m:integer);

var
i,j,j_old,k:integer;
kmp_shift:indexarray;
go_on:boolean;

begin {1}
  for j:= 1 to m do
    gs_shift[j] := m;
  kmp_shift[m]:=1;

{stage 1}
  j:=m;

  for k:=m-1 downto 1 do
    begin {2}
       go_on:=true;
       while (p[j] <> p[k]) and go_on do
          begin {3}
          if (gs_shift[j] > j-k) then gs_shift[j] := j-k;
          if (j < m) then j:= j+kmp_shift[j+1]
          else go_on:=false;
          end; {3}
```



```
          if (p[k] = p[j]) then
             begin {3}
             kmp_shift[k]:=j-k;
           j:=j-1;
             end {3}
          else
          kmp_shift[k]:=j-k+1;
      end; {2}

  {stage 2}
   j:=j+1;
   j_old:=1;

   while (j <= m) do
     begin {2}
       for i:=j_old to j-1 do
       if (gs_shift[i] > j-1) then gs_shift[i]:=j-1;

       j_old:=j;
       j:=j+kmp_shift[j];
     end; {2}
  end; {1}

begin {main}

writeln('input a string on a single line');

readstring(p,m);
gsshift(p,matchshift,m);
writeln('the value in cell i is the number of positions to shift');
writeln('after a mismatch occurring in position i of the pattern');

for i:= 1 to m do
write(matchshift[i]:3);
writeln;
end. {main}
```

25. Prove that the shift rule used by the real-time string matcher does not miss any occurrences of P in T.

26. Prove Theorem 2.4.1.

27. In this chapter, we showed how to use Z values to compute both the sp'_i and sp_i values used in Knuth-Morris-Pratt and the sp'_{*i} values needed for its real-time extension. Instead of using Z values for the sp'_{*i} values, show how to obtain these values from the sp_i and/or sp'_i values in linear $[O(n|\Sigma|)]$ time, where n is the length of P and $|\Sigma|$ is the length of the alphabet.

28. Although we don't know how to simply convert the Boyer–Moore algorithm to be a real-time method the way Knuth-Morris-Pratt was converted, we can make similar changes to the strong shift rule to make the Boyer–Moore shift more effective. That is, when a mismatch occurs between $P(i)$ and $T(h)$ we can look for the right-most copy in P of $P[i+1..n]$ (other than $P[i+1..n]$ itself) such that the preceding character is $T(h)$. Show how to modify

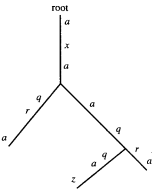


Figure 2.3: The pattern $P = aqra$ labels two subpaths of paths starting at the root. Those paths start at the root, but the subpaths containing $aqra$ do not. There is also another subpath in the tree labeled $aqra$ (it starts above the character z), but it violates the requirement that it be a subpath of a path starting at the root. Note that an edge label is displayed from the top of the edge down towards the bottom of the edge. Thus in the figure, there is an edge labeled "qra", not "arq".

the Boyer–Moore preprocessing so that the needed information is collected in linear time, assuming a fixed size alphabet.

29. Suppose we are given a tree where each edge is labeled with one or more characters, and we are given a pattern P. The label of a subpath in the tree is the concatenation of the labels on the edges in the subpath. The problem is to find all subpaths of paths starting at the root that are labeled with pattern P. Note that although the subpath must be part of a path directed from the root, the subpath itself need not start at the root (see Figure 2.3). Give an algorithm for this problem that runs in time proportional to the total number of characters on the edges of the tree plus the length of P.

3

Exact Matching: A Deeper Look at Classical Methods

3.1. A Boyer–Moore variant with a "simple" linear time bound

Apostolico and Giancarlo [26] suggested a variant of the Boyer–Moore algorithm that allows a fairly simple proof of linear worst-case running time. With this variant, no character of T will ever be compared after it is first matched with any character of P. It is then immediate that the number of comparisons is at most $2m$: Every comparison is either a match or a mismatch; there can only be m mismatches since each one results in a nonzero shift of P; and there can only be m matches since no character of T is compared again after it matches a character of P. We will also show that (in addition to the time for comparisons) the time taken for all the other work in this method is linear in m.

Given the history of very difficult and partial analyses of the Boyer–Moore algorithm, it is quite amazing that a close variant of the algorithm allows a simple linear time bound. We present here a further improvement of the Apostolico–Giancarlo idea, resulting in an algorithm that simulates *exactly* the shifts of the Boyer–Moore algorithm. The method therefore has all the rapid shifting advantages of the Boyer–Moore method as well as a simple linear worst-case time analysis.

3.1.1. Key ideas

Our version of the Apostolico–Giancarlo algorithm simulates the Boyer–Moore algorithm, finding exactly the same mismatches that Boyer–Moore would find and making exactly the same shifts. However, it infers and avoids many of the explicit matches that Boyer–Moore makes.

We take the following high-level view of the Boyer–Moore algorithm. We divide the algorithm into *compare/shift phases* numbered 1 through $q \leq m$. In a compare/shift phase, the right end of P is aligned with a character of T, and P is compared right to left with selected characters of T until either all of P is matched or until a mismatch occurs. Then, P is shifted right by some amount as given in the Boyer–Moore shift rules.

Recall from Section 2.2.4, where preprocessing for Boyer–Moore was discussed, that $N_i(P)$ is the length of the longest suffix of $P[1..i]$ that matches a suffix of P. In Section 2.2.4 we showed how to compute N_i for every i in $O(n)$ time, where n is the length of P. We assume here that vector N has been obtained during the preprocessing of P.

Two modifications of the Boyer–Moore algorithm are required. First, during the search for P in T (after the preprocessing), we maintain an m length vector M in which at most one entry is updated in every phase. Consider a phase where the right end of P is aligned with position j of T and suppose that P and T match for l places (from right to left) but no farther. Then, set $M(j)$ to a value $k \leq l$ (the rules for selecting k are detailed below). $M(j)$ records the fact that a suffix of P of length k (at least) occurs in T and ends exactly

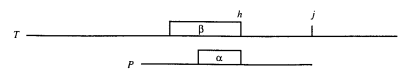


Figure 3.1: Substring α has length N_i and substring β has length $M(h) > N_i$. The two strings must match from their right ends for N_i characters, but mismatch at the next character.

at position j. As the algorithm proceeds, a value for $M(j)$ is set for every position j in T that is aligned with the right end of P; $M(j)$ is undefined for all other positions in T.

The second modification exploits the vectors N and M to speed up the Boyer–Moore algorithm by inferring certain matches and mismatches. To get the idea, suppose the Boyer–Moore algorithm is about to compare characters $P(i)$ and $T(h)$, and suppose it knows that $M(h) > N_i$ (see Figure 3.1). That means that an N_i-length substring of P ends at position i and matches a suffix of P, while an $M(h)$-length substring of T ends at position h and matches a suffix of P. So the N_i-length suffixes of those two substrings must match, and we can conclude that the next N_i comparisons (from $P(i)$ and $T(h)$ moving leftward) in the Boyer–Moore algorithm would be matches. Further, if $N_i = i$, then an occurrence of P in T has been found, and if $N_i < i$, then we can be sure that the next comparison (after the N_i matches) would be a mismatch. Hence in simulating Boyer–Moore, if $M(h) > N_i$ we can avoid at least N_i explicit comparisons. Of course, it is not always the case that $M(h) > N_i$, but all the cases are similar and are detailed below.

3.1.2. One phase in detail

As in the original Boyer–Moore algorithm, when the right end of P is aligned with a position j in T, P is compared with T from right to left. When a mismatch is found or inferred, or when an occurrence of P is found, P is shifted according to the original Boyer–Moore shift rules (either the strong or weak version) and the compare/shift phase ends. Here we will only give the details for a single phase. The phase begins with h set to j and i set to n.

Phase algorithm

1. If $M(h)$ is undefined or $M(h) = N_i = 0$, then compare $T(h)$ and $P(i)$ as follows:

   If $T(h) = P(i)$ and $i = 1$, then report an occurrence of P ending at position j of T, set $M(j) = n$, and shift as in the Boyer–Moore algorithm (ending this phase).

   If $T(h) = P(i)$ and $i > 1$, then set h to $h - 1$ and i to $i - 1$ and repeat the phase algorithm.

   If $T(h) \neq P(i)$, then set $M(j) = j - h$ and shift P according to the Boyer–Moore rules based on a mismatch occurring in position i of P (this ends the phase).

2. If $M(h) < N_i$, then P matches its counterparts in T from position n down to position $i - M(h) + 1$ of P. By the definition of $M(h)$, P might match more of T to the left, so set i to $i - M(h)$, set h to $h - M(h)$, and repeat the phase algorithm.

3. If $M(h) \geq N_i$ and $N_i = i > 0$, then declare that an occurrence of P has been found in T ending at position j. $M(j)$ must be set to a value less than or equal to n. Set $M(j)$ to $j - h$, and shift according to the Boyer–Moore rules based on finding an occurrence of P ending at j (this ends the phase).

4. If $M(h) > N_i$ and $N_i < i$, then P matches T from the right end of P down to character $i - N_i + 1$ of P, but the next pair of characters mismatch [i.e., $P(i - N_i) \neq T(h - N_i)$]. Hence P matches T for $j - h + N_i$ characters and mismatches at position $i - N_i$ of P. $M(j)$ must be set to a value less than or equal to $j - h + N_i$. Set $M(j)$ to $j - h$. Shift P by the Boyer–Moore rules based on a mismatch at position $i - N_i$ of P (this ends the phase).
5. If $M(h) = N_i$ and $0 < N_i < i$, then P and T must match for at least $M(h)$ characters to the left, but the left end of P has not yet been reached, so set i to $i - M(h)$ and set h to $h - M(h)$ and repeat the phase algorithm.

3.1.3. Correctness and linear-time analysis

Theorem 3.1.1. *Using M and N as above, the Apostolico–Giancarlo variant of the Boyer–Moore algorithm correctly finds all occurrences of P in T.*

PROOF We prove correctness by showing that the algorithm simulates the original Boyer–Moore algorithm. That is, for any given alignment of P with T, the algorithm is correct when it declares a match down to a given position and a mismatch at the next position. The rest of the simulation is correct since the shift rules are the same as in the Boyer–Moore algorithm.

Assume inductively that $M(h)$ values are valid up to some position in T. That is, wherever $M(h)$ is defined, there is an $M(h)$-length substring in T ending at position h in T that matches a suffix of P. The first such value, $M(n)$, is valid because it is found by aligning P at the left of T and making explicit comparisons, repeating rule 1 of the phase algorithm until a mismatch occurs or an occurrence of P is found. Now consider a phase where the right end of P is aligned with position j of T. The phase simulates the workings of Boyer–Moore except that in cases 2, 3, 4, and 5 certain explicit comparisons are skipped and in case 4 a mismatch is inferred, rather than observed. But whenever comparisons are skipped, they are certain to be matches by the definition of N and M and the assumption that the M values are valid thus far. Thus it is correct to skip these comparisons. In case 4, a mismatch at position $i - N_i$ of P is correctly inferred because N_i is the maximum length of any substring ending at i that matches a suffix of P, whereas $M(h)$ is less than or equal to the maximum length of any substring ending at h that matches a suffix of P. Hence this phase correctly simulates Boyer–Moore and finds exactly the same mismatch (or an occurrence of P in T) that Boyer–Moore would find. The value given to $M(j)$ is valid since in all cases it is less than or equal to the length of the suffix of P shown to match its counterpart in the substring $T[1..i]$. □

The following definitions and lemma will be helpful in bounding the work done by the algorithm.

Definition If j is a position where $M(j)$ is greater than zero then the interval $[j - M(j) + 1..j]$ is called a *covered interval* defined by j.

Definition Let $j' < j$ and suppose covered intervals are defined for both j and j'. We say that the covered intervals for j and j' *cross* if $j - M(j) + 1 \leq j'$ and $j' - M(j') + 1 < j - M(j) + 1$ (see Figure 3.2).

Lemma 3.1.1. *No covered intervals computed by the algorithm ever cross each other. Moreover, if the algorithm examines a position h of T in a covered interval, then h is at the right end of that interval.*

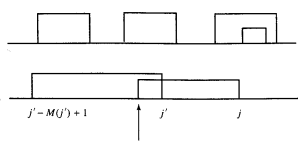


Figure 3.2: a. Diagram showing covered intervals that do not cross, although one interval can contain another. b. Two covered intervals that do cross.

PROOF The proof is by induction on the number of intervals created. Certainly the claim is true until the first interval is created, and that interval does not cross itself. Now assume that no intervals cross and consider the phase where the right end of P is aligned with position j of T.

Since $h = j$ at the start of the phase, and j is to the right of any interval, h begins outside any interval. We consider how h could first be set to a position inside an interval, other than the right end of the interval. Rule 1 is never executed when h is at the right end of an interval (since then $M(h)$ is defined and greater than zero), and after any execution of Rule 1, either the phase ends or h is decremented by one place. So an execution of Case 1 cannot cause h to move beyond the right-most character of a covered interval. This is also true for Cases 3 and 4 since the phase ends after either of those cases. So if h is ever moved into an interval in a position other than its right end, that move must follow an execution of Case 2 or 5. An execution of Case 2 or 5 moves h from the right end of some interval $I = [k..h]$ to position $k - 1$, one place to the left of I. Now suppose that $k - 1$ is in some interval I' but is not at its right end, and that this is the first time in the phase that h (presently $k - 1$) is in an interval in a position other than its right end. That means that the right end of I cannot be to the left of the right end of I' (for then position $k - 1$ would have been strictly inside I'), and the right ends of I and I' cannot be equal (since $M(h)$ has at most one value for any h). But these conditions imply that I and I' cross, which is assumed to be untrue. Hence, if no intervals cross at the start of the phase, then in that phase only the right end of any covered interval is examined.

A new covered interval gets created in the phase only after the execution of Case 1, 3, or 4. In any of these cases, the interval $[h + 1..j]$ is created after the algorithm examines position h. In Case 1, h is not in any interval, and in Cases 3 and 4, h is the right end of an interval, so in all cases $h + 1$ is either not in a covered interval or is at the left end of an interval. Since j is to the right of any interval, and $h + 1$ is either not in an interval or is the left end of one, the new interval $[h + 1..j]$ does not cross any existing interval. The previously existing intervals have not changed, so there are no crossing intervals at the end of the phase, and the induction is complete. □

Theorem 3.1.2. *The modified Apostolico–Giancarlo algorithm does at most 2m character comparisons and at most $O(m)$ additional work.*

PROOF Every phase ends if a comparison finds a mismatch and every phase, except the last, is followed by a nonzero shift of P. Thus the algorithm can find at most m mismatches. To bound the matches, observe that characters are explicitly compared only in Case 1, and

if the comparison involving $T(h)$ is a match then, at the end of the phase, $M(j)$ is set at least as large as $j - h + 1$. That means that all characters in T that matched a character of P during that phase are contained in the covered interval $[j - M(j) + 1..j]$. Now the algorithm only examines the right end of an interval, and if h is the right end of an interval then $M(h)$ is defined and greater than 0, so the algorithm never compares a character of T in a covered interval. Consequently, no character of T will ever be compared again after it is first in a match. Hence the algorithm finds at most m matches, and the total number of character comparisons is bounded by $2m$.

To bound the amount of additional work, we focus on the number of accesses of M during execution of the five cases since the amount of additional work is proportional to the number of such accesses. A character comparison is done whenever Case 1 applies. Whenever Case 3 or 4 applies, P is immediately shifted. Hence Cases 1, 3, and 4 can apply at most $O(m)$ times since there are at most $O(m)$ shifts and compares. However, it is possible that Case 2 or Case 5 can apply without an immediate shift or immediate character comparison. That is, Case 2 or 5 could apply repeatedly before a comparison or shift is done. For example, Case 5 would apply twice in a row (without a shift or character comparison) if $N_i = M(h) > 0$ and $N_{i-N_i} = M(h - M(h))$. But whenever Case 2 or 5 applies, then $j > h$ and $M(j)$ will certainly get set to $j - h + 1$ or more at the end of that phase. So position h will be in the strict interior of the covered interval defined by j. Therefore, h will never be examined again, and $M(h)$ will never be accessed again. The effect is that Cases 2 and 5 can apply at most once for any position in T, so the number of accesses made when these cases apply is also $O(m)$. □

3.2. Cole's linear worst-case bound for Boyer–Moore

Here we finally present a linear worst-case time analysis of the *original* Boyer–Moore algorithm. We consider first the use of the (strong) good suffix rule by itself. Later we will show how the analysis is affected by the addition of the bad character rule. Recall that the good suffix rule is the following:

> Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and *such that the character to the left of t' differs from the mismatched character in P*. Shift P to the right so that substring t' in P is below substring t in T (recall Figure 2.1). If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P by n places to the right.
>
> If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted pattern matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places.

We will show that by using the good suffix rule alone, the Boyer–Moore method has a worst-case running time of $O(m)$, provided that the pattern does not appear in the text. Later we will extend the Boyer–Moore method to take care of the case that P does occur in T.

As in our analysis of the Apostolico–Giancarlo algorithm, we divide the Boyer–Moore algorithm into *compare/shift* phases numbered 1 through $q \leq m$. In compare/shift phase i, a suffix of P is matched right to left with characters of T until either all of P is matched or until a mismatch occurs. In the latter case, the substring of T consisting of the matched

characters is denoted t_i, and the mismatch occurs just to the left of t_i. The pattern is then shifted right by an amount determined by the good suffix rule.

3.2.1. Cole's proof when the pattern does not occur in the text

Definition Let s_i denote the amount by which P is shifted right at the end of phase i.

Assume that P does not occur in T, so the compare part of every phase ends with a mismatch. In each compare/shift phase, we divide the comparisons into those that compare a character of T that has previously been compared (in a previous phase) and those comparisons that compare a character of T for the first time in the execution of the algorithm. Let g_i be the number of comparisons in phase i of the first type (comparisons involving a previously examined character of T), and let g_i' be the number of comparisons in phase i of the second type. Then, over the entire algorithm the number of comparisons is $\sum_{i=1}^{q}(g_i + g_i')$, and our goal is to show that this sum is $O(m)$.

Certainly, $\sum_{i=1}^{q} g_i' \leq m$ since a character can be compared for the first time only once. We will show that for any phase i, $s_i \geq g_i/3$. Then since $\sum_{i=1}^{q} s_i \leq m$ (because the total length of all the shifts is at most m) it will follow that $\sum_{i=1}^{q} g_i \leq 3m$. Hence the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g_i + g_i') \leq 4m$.

An initial lemma

We start with the following definition and a lemma that is valuable in its own right.

Definition For any string β, β^i denotes the string obtained by concatenating together i copies of β.

Lemma 3.2.1. *Let γ and δ be two nonempty strings such that $\gamma\delta = \delta\gamma$. Then $\delta = \rho^i$ and $\gamma = \rho^j$ for some string ρ and positive integers i and j.*

This lemma says that if a string is the same before and after a circular shift (so that it can be written both as $\gamma\delta$ and $\delta\gamma$, for some strings γ and δ) then γ and δ can both be written as concatenations of some single string ρ.

For example, let $\delta = abab$ and $\gamma = ababab$, so $\delta\gamma = abababab = \gamma\delta$. Then $\rho = ab$, $\delta = \rho^2$, and $\gamma = \rho^3$.

PROOF The proof is by induction on $|\delta| + |\gamma|$. For the basis, if $|\delta| + |\gamma| = 2$, it must be that $\delta = \gamma = \rho$ and $i = j = 1$. Now consider larger lengths. If $|\delta| = |\gamma|$, then again $\delta = \gamma = \rho$ and $i = j = 1$. So suppose $|\delta| < |\gamma|$. Since $\delta\gamma = \gamma\delta$ and $|\delta| < |\gamma|$, δ must be a prefix of γ, so $\gamma = \delta\delta'$ for some string δ'. Substituting this into $\delta\gamma = \gamma\delta$ gives $\delta\delta\delta' = \delta\delta'\delta$. Deleting the left copy of δ from both sides gives $\delta\delta' = \delta'\delta$. However, $|\delta| + |\delta'| = |\gamma| < |\delta| + |\gamma|$, and so by induction, $\delta = \rho^i$ and $\delta' = \rho^j$. Thus, $\gamma = \delta\delta' = \rho^k$, where $k = i + j$. □

Definition A string α is *semiperiodic* with *period* β if α consists of a nonempty suffix of a string β (possibly the entire β) followed by one or more copies of β. String α is called *periodic* with *period* β if α consists of two or more *complete* copies of β. We say that string α is *periodic* if it is periodic with some period β.

For example, *bcabcabc* is semiperiodic with period *abc*, but it is not periodic. String *abcabc* is periodic with period *abc*. Note that a periodic string is by definition also semiperiodic. Note also that a string cannot have itself as a period although a period may itself be periodic. For example, *abababab* is periodic with period *abab* and also with shorter period *ab*. An alternate definition of a semiperiodic string is sometimes useful.

Definition A string α is *prefix semiperiodic* with period γ if α consists of one or more copies of string γ followed by a nonempty prefix (possibly the entire γ) of string γ.

We use the term "prefix semiperiodic" to distinguish this definition from the definition given for "semiperiodic", but the following lemma (whose proof is simple and is left as an exercise) shows that these two definitions are really alternate reflections of the same structure.

Lemma 3.2.2. *A string α is semiperiodic with period β if and only if it is prefix semiperiodic with the same length period as β.*

For example, the string *abaabaabaabaabaab* is semiperiodic with period *aab* and is prefix semiperiodic with period *aba*.

The following useful lemma is easy to verify, and its proof is typical of the style of thinking used in dealing with overlapping matches.

Lemma 3.2.3. *Suppose pattern P occurs in text T starting at positions p and p' $> p$, where $p' - p \leq \lfloor n/2 \rfloor$. Then P is semiperiodic with period $p' - p$.*

The following lemma, called the *GCD Lemma*, is a very powerful statement about periods of strings. We won't need the lemma in our discussion of Cole's proof, but it is natural to state it here. We will prove it and use it in Section 16.17.5.

Lemma 3.2.4. *Suppose string α is semiperiodic with both a period of length p and a period of length q, and $|\alpha| \geq p + q$. Then α is semiperiodic with a period whose length is the greatest common divisor of p and q.*

Return to Cole's proof

Recall that the key thing to prove is that $s_i \geq g_i/3$ in every phase i. As noted earlier, it then follows easily that the total number of comparisons is bounded by $4m$.

Consider the ith compare/shift phase, where substring t_i of T matches a suffix of P and then P is shifted s_i places to the right. If $s_i \geq (|t_i| + 1)/3$, then $s_i \geq g_i/3$ even if all characters of T that were compared in phase i had been previously compared. Therefore, it is easy to handle phases where the shift is "relatively" large compared to the total number of characters examined during the phase. Accordingly, for the next several lemmas we consider the case when the shift is relatively small (i.e., $s_i < (|t_i| + 1)/3$ or, equivalently, $|t_i| + 1 > 3s_i$).

We need some notation at this point. Let α be the suffix of P of length s_i, and let β be the smallest substring such that $\alpha = \beta^l$ for some integer l (it may be that $\beta = \alpha$ and $l = 1$). Let $\bar{P} = P[n - |t_i| \ldots n]$ be the suffix of P of length $|t_i| + 1$, that is, that portion of P (including the mismatch) that was examined in phase i. See Figure 3.3.

Lemma 3.2.5. *If $|t_i| + 1 > 3s_i$, then both t_i and $\bar{P}$ are semiperiodic with period α and hence with period β.*

PROOF Starting from the right end of $\bar{P}$, mark off substrings of length s_i until less than s_i characters remain on the left (see Figure 3.4). There will be at least three full substrings since $|\bar{P}| = |t_i| + 1 > 3s_i$. Phase i ends by shifting P right by s_i positions. Consider how $\bar{P}$ aligns with T before and after that shift (see Figure 3.5). By definition of s_i and α, α is the part of the shifted $\bar{P}$ to the right of the original $\bar{P}$. By the good suffix rule, the portion

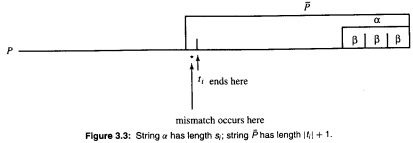


Figure 3.3: String α has length s_i; string $\bar{P}$ has length $|t_i| + 1$.

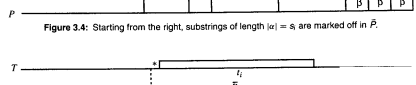


Figure 3.4: Starting from the right, substrings of length $|\alpha| = s_i$ are marked off in $\bar{P}$.

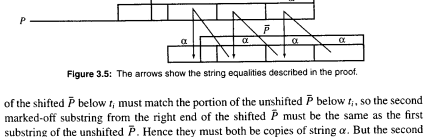


Figure 3.5: The arrows show the string equalities described in the proof.

of the shifted $\bar{P}$ below t_i must match the portion of the unshifted $\bar{P}$ below t_i, so the second marked-off substring from the right end of the shifted $\bar{P}$ must be the same as the first substring of the unshifted $\bar{P}$. Hence they must both be copies of string α. But the second substring is the same in both copies of $\bar{P}$, so continuing this reasoning we see that all the s_i-length marked substrings are copies of α and the left-most substring is a suffix of α (if it is not a complete copy of α). Hence $\bar{P}$ is semiperiodic with period α. The right-most $|t_i|$ characters of $\bar{P}$ match t_i, and so t_i is also semiperiodic with period α. Then since $\alpha = \beta^j$, $\bar{P}$ and t_i must also be semiperiodic with period β. □

Recall that we want to bound g_i, the number of characters compared in the ith phase that have been previously compared in earlier phases. All but one of the characters compared in phase i are contained in t_i, and a character in t_i could have previously been examined only during a phase where P overlaps t_i. So to bound g_i, we closely examine in what ways P could have overlapped t_i during earlier phases.

Lemma 3.2.6. *If $|t_i| + 1 > 3s_i$, then in any phase $h < i$, the right end of P could not have been aligned opposite the right end of any full copy of β in substring t_i of T.*

PROOF By Lemma 3.2.5, t_i is semiperiodic with period β. Figure 3.6 shows string t_i as a concatenation of copies of string β. In phase h, the right end of P cannot be aligned with the right end of t_i since that is the alignment of P and T in phase $i > h$, and P must have moved right between phases h and i. So, suppose, for contradiction, that in phase h the right end of P is aligned with the right end of some other full copy of β in t_i. For

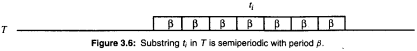


Figure 3.6: Substring t_i in T is semiperiodic with period β.

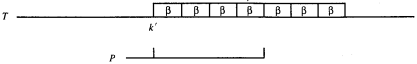


Figure 3.7: The case when the right end of P is aligned with a right end of $\bar{\beta}$ in phase h. Here $q = 3$. A mismatch must occur between $T(k')$ and $P(k)$.

concreteness, call that copy $\bar{\beta}$ and say that its right end is $q|\beta|$ places to the left of the right of t_i, where $q \geq 1$ (see Figure 3.7). We will first deduce how phase h must have ended, and then we'll use that to prove the lemma.

Let k' be the position in T just to the left of t_i (so $T(k')$ is involved in the mismatch ending phase i), and let k be the position in P opposite $T(k')$ in phase h. We claim that, in phase h, the comparison of P and T will find matches until the left end of t_i but then mismatch when comparing $T(k')$ and $P(k)$. The reason is the following: Strings $\bar{P}$ and t_i are semiperiodic with period β, and in phase h the right end of P is aligned with the right end of some β. So in phase h, P and T will certainly match until the left end of string t_i. Now $\bar{P}$ is semiperiodic with β, and in phase h, the right end of P is exactly $q|\beta|$ places to the left of the right end of t_i. Therefore, $\bar{P}(1) = \bar{P}(1 + |\beta|) = \cdots = \bar{P}(1 + q|\beta|) = P(k)$. But in phase i the mismatch occurs when comparing $T(k')$ with $\bar{P}(1)$, so $P(k) = \bar{P}(1) \neq T(k')$. Hence, if in phase h the right end of P is aligned with the right end of a β, then phase h must have ended with a mismatch between $T(k')$ and $P(k)$. This fact will be used below to prove the lemma.[1]

Now we consider the possible shifts of P done in phase h. We will show that every possible shift leads to a contradiction, so no shifts are possible and the assumed alignment of P and T in phase h is not possible, proving the lemma.

Since $h < i$, the right end of P will not be shifted in phase h past the right end of t_i; consequently, after the phase h shift a character of $\bar{P}$ is opposite character $T(k')$ (the character of T that will mismatch in phase i). Consider where the right end of P is after the phase h shift. There are two cases to consider: 1. Either the right end of P is opposite the right end of another full copy of β (in t_i) or 2. The right end of P is in the interior of a full copy of β.

Case 1 If the phase h shift aligns the right end of P with the right end of a full copy of β, then the character opposite $T(k')$ would be $P(k - r|\beta|)$ for some r. But since $\bar{P}$ is

[1] Later we will analyze the Boyer–Moore algorithm when P is in T. For that purpose we note here that when phase h is assumed to end by finding an occurrence of P, then the proof of Lemma 3.2.6 is complete at this point, having established a contradiction. That is, on the assumption that the right end of P is aligned with the right end of a β in phase h, we proved that phase h ends with a mismatch, which would contradict the assumption that h ends by finding an occurrence of P in T. So even if phase h ends by finding an occurrence of P, the right end of P could not be aligned with the right end of a β block in phase h.

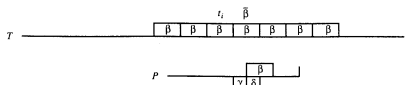


Figure 3.8: Case when the right end of P is aligned with a character in the interior of a β. Then t_i would have a smaller period than β, contradicting the definition of β.

semiperiodic with period β, $P(k)$ must be equal to $P(k - r|\beta|)$, contradicting the good suffix rule.

Case 2 Suppose the phase h shift aligns P so that its right end aligns with some character in the interior of a full copy of β. That means that, in this alignment, the right end of some β string in P is opposite a character in the interior of $\bar{\beta}$. Moreover, by the good suffix rule, the characters in the shifted P below $\bar{\beta}$ agree with $\bar{\beta}$ (see Figure 3.8). Let $\gamma\delta$ be the string in the shifted P positioned opposite $\bar{\beta}$ in t_i, where γ is the string through the end of β and δ is the remainder. Since $\bar{\beta} = \beta$, γ is a suffix of β, δ is a prefix of β, and $|\gamma| + |\delta| = |\bar{\beta}| = |\beta|$; thus $\gamma\delta = \delta\gamma$. By Lemma 3.2.1, however, $\beta = \rho^t$ for $t > 1$, which contradicts the assumption that β is the smallest string such that $\alpha = \beta^l$ for some l.

Starting with the assumption that in phase h the right end of P is aligned with the right end of a full copy of β, we reached the conclusion that no shift in phase h is possible. Hence the assumption is wrong and the lemma is proved. □

Lemma 3.2.7. *If $|t_i| + 1 > 3s_i$, then in phase $h < i$, P can match t_i in T for at most $|\beta| - 1$ characters.*

PROOF Since P is not aligned with the end of any β in phase h, if P matches t_i in T for β or more characters then the right-most β characters of P would match a string consisting of a suffix (γ) of β followed by a prefix (δ) of β. So we would again have $\beta = \gamma\delta = \delta\gamma$, and by Lemma 3.2.1, this again would lead to a contradiction to the selection of β. □

Note again that this lemma holds even if phase h is assumed to find an occurrence of P. That is, nowhere in the proof is it assumed that phase h ends with a mismatch, only that phase i does. This observation will be used later.

Lemma 3.2.8. *If $|t_i| + 1 > 3s_i$, then in phase $h < i$ if the right end of P is aligned with a character in t_i, it can only be aligned with one of the left-most $|\beta| - 1$ characters of t_i or one of the right-most $|\beta|$ characters of t_i.*

PROOF Suppose in phase h that the right end of P is aligned with a character of t_i other than one of the left-most $|\beta| - 1$ characters or the right-most $|\beta|$ characters. For concreteness, say that the right end of P is aligned with a character in copy β' of string β. Since β' is not the left-most copy of β, the right end of P is at least $|\beta|$ characters to the right of the left end of t_i, and so by Lemma 3.2.7 a mismatch would occur in phase h before the left end of t_i is reached. Say that mismatch occurs at position k'' of T. After that mismatch, P is shifted right by some amount determined by the good suffix rule. By Lemma 3.2.6, the phase-h shift cannot move the right end of P to the right end of β', and we will show that the shift will also not move the end of P past the right end of β'.

Recall that the good suffix rule shifts P (when possible) by the smallest amount so that all the characters of T that matched in phase h again match with the shifted P and so that the two characters of P aligned with $T(k'')$ before and after the shift are unequal. We claim these conditions hold when the right end of P is aligned with the right end of β'. Consider that alignment. Since $\bar{P}$ is semiperiodic with period β, that alignment of P and T would match at least until the left end of t_i and so would match at position k'' of T. Therefore, the two characters of P aligned with $T(k'')$ before and after the shift cannot be equal. Thus if the end of P were aligned with the end of β' then all the characters of T that matched in phase h would again match, and the characters of P aligned with $T(k'')$ before and after the shift would be different. Hence the good suffix rule would not shift the right end of P past the right of the end of β'.

Therefore, if the right end of P is aligned in the interior of β' in phase h, it must also be in the interior of β' in phase $h + 1$. But h was arbitrary, so the phase-$h + 1$ shift would also not move the right end of P past β'. So if the right end of P is in the interior of β' in phase h, it remains there forever. This is impossible since in phase $i > h$ the right end of P is aligned with the right end of t_i, which is to the right of β'. Hence the right end of P is not in the interior of β', and the Lemma is proved. □

Note again that Lemma 3.2.8 holds even if phase h is assumed to end by finding an occurrence of P in T. That is, the proof only needs the assumption that phase i ends with a mismatch, not that phase h does. In fact, when phase h finds an occurrence of P in T, then the proof of the lemma only needs the reasoning contained in the first two paragraphs of the above proof.

Theorem 3.2.1. *Assuming P does not occur in T, $s_i \geq g_i/3$ in every phase i.*

PROOF This is trivially true if $s_i \geq (|t_i| + 1)/3$, so assume $|t_i| + 1 > 3s_i$. By Lemma 3.2.8, in any phase $h < i$, the right end of P is opposite either one of the left-most $|\beta| - 1$ characters of t_i or one of the right-most $|\beta|$ characters of t_i (excluding the extreme right character). By Lemma 3.2.7, at most $|\beta|$ comparisons are made in phase $h < i$. Hence the only characters compared in phase i that could possibly have been compared before phase i are the left-most $|\beta| - 1$ characters of t_i, the right-most $2|\beta|$ characters of t_i, or the character just to the left of t_i. So $g_i \leq 3|\beta| = 3s_i$ when $|t_i| + 1 > 3s_i$. In both cases then, $s_i \geq g_i/3$. □

Theorem 3.2.2. [108] *Assuming that P does not occur in T, the worst-case number of comparisons made by the Boyer–Moore algorithm is at most $4m$.*

PROOF As noted before, $\sum_{i=1}^{q} g_i' \leq m$ and $\sum_{i=1}^{q} s_i \leq m$, so the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g_i + g_i') \leq (\sum_i 3s_i) + m \leq 4m$. □

3.2.2. The case when the pattern does occur in the text

Consider P consisting of n copies of a single character and T consisting of m copies of the same character. Then P occurs in T starting at every position in T except the last $n - 1$ positions, and the number of comparisons done by the Boyer–Moore algorithm is $\Theta(mn)$. The $O(m)$ time bound proved in the previous section breaks down because it was derived by showing that $g_i \leq 3s_i$, and that required the assumption that phase i ends with a mismatch. So when P does occur in T (and phases do not necessarily end with mismatches), we must modify the Boyer–Moore algorithm in order to recover the linear running time. Galil [168] gave the first such modification. Below we present a version of his idea.

The approach comes from the following observation: Suppose in phase i that the right end of P is positioned with character k of T, and that P is compared with T down

to character s of T. (We don't specify whether the phase ends by finding a mismatch or by finding an occurrence of P in T.) If the phase-i shift moves P so that its left end is to the right of character s of T, then in phase $i+1$ a *prefix* of P definitely matches the characters of T up to $T(k)$. Thus, in phase $i+1$, if the right-to-left comparisons get down to position k of T, the algorithm can conclude that an occurrence of P has been found even without explicitly comparing characters to the left of $T(k+1)$. It is easy to implement this modification to the algorithm, and we assume in the rest of this section that the Boyer–Moore algorithm includes this rule, which we call the *Galil rule*.

Theorem 3.2.3. *Using the Galil rule, the Boyer–Moore algorithm never does more than $O(m)$ comparisons, no matter how many occurrences or P there are in T.*

PROOF Partition the phases into those that do find an occurrence of P and those that do not. Let Q be the set of phases of the first type and let d_i be the number of comparisons done in phase i if $i \in Q$. Then $\sum_{i\notin Q} d_i + \sum_{i\notin Q}(|t_i|+1)$ is a bound on the total number of comparisons done in the algorithm.

The quantity $\sum_{i\notin Q}(|t_i|+1)$ is again $O(m)$. To see this, recall that the lemmas of the previous section, which proved that $g_i \le 3s_i$, only needed the assumption that phase i ends with a mismatch and that $h < i$. In particular, the analysis of how P of phase $h < i$ is aligned with P of phase i did not need the assumption that phase h ends with a mismatch. Those proofs cover both the case that h ends with a mismatch and that h ends by finding an occurrence of P. Hence it again holds that $g_i \le 3s_i$ if phase i ends with a mismatch, even though earlier phases might end with a match.

For phases in Q, we again ignore the case that $s_i \ge (n+1)/3 \ge (d_i+1)/3$, since the total number of comparisons done in such phases must be bounded by $\sum 3s_i \le 3m$. So suppose phase i ends by finding an occurrence of P in T and then shifts by less than $n/3$. By a proof essentially the same as for Lemma 3.2.5 it follows that P is semi-periodic; let β denote the shortest period of P. Hence the shift in phase i moves P right by exactly $|\beta|$ positions, and using the Galil rule in the Boyer–Moore algorithm, no character of T compared in phase $i+1$ will have ever been compared previously. Repeating this reasoning, if phase $i+1$ ends by finding an occurrence of P then P will again shift by exactly $|\beta|$ places and no comparisons in phase $i+2$ will examine a character of T compared in any earlier phase. This cycle of shifting P by exactly $|\beta|$ positions and then identifying another occurrence of P by examining only $|\beta|$ new characters of T may be repeated many times. Such a succession of overlapping occurrences of P then consists of a concatenation of copies of β (each copy of P starts exactly $|\beta|$ places to the right of the previous occurrence) and is called a *run*. Using the Galil rule, it follows immediately that in any single run the number of comparisons used to identify the occurrences of P contained in that run is exactly the length of the run. Therefore, over the entire algorithm the number of comparisons used to find those occurrences is $O(m)$. If no additional comparisons were possible with characters in a run, then the analysis would be complete. However, additional examinations are possible and we have to account for them.

A run ends in some phase $k > i$ when a mismatch is found (or when the algorithm terminates). It is possible that characters of T in the run could be examined again in phases after k. A phase that reexamines characters of the run either ends with a mismatch or ends by finding an occurrence of P that overlaps the earlier run but is not part of it. However, all comparisons in phases that end with a mismatch have already been accounted for (in the accounting for phases not in Q) and are ignored here.

Let $k' > k > i$ be a phase in which an occurrence of P is found overlapping the earlier run but is not part of that run. As an example of such an overlap, suppose $P = axaaxa$ and T contains the substring $axaaxaaxaaxaaxaaxa$. Then a run begins at the start of the substring and ends with its twelfth character, and an overlapping occurrence of P (not part of the run) begins with that character. Even with the Galil rule, characters in the run will be examined again in phase k', and since phase k' does not end with a mismatch those comparisons must still be counted.

In phase k', if the left end of the new occurrence of P in T starts at a left end of a copy of β in the run, then contiguous copies of β continue past the right end of the run. But then no mismatch would have been possible in phase k since the pattern in phase k is aligned exactly $|\beta|$ places to the right of its position in phase $k-1$ (where an occurrence of P was found). So in phase k', the left end of the new P in T must start with an interior character of some copy of β. But then if P overlaps with the run by more than $|\beta|$ characters, Lemma 3.2.1 implies that β is periodic, contradicting the selection of β. So P can overlap the run only by part of the run's left-most copy of β. Further, since phase k' ends by finding an occurrence of P, the pattern is shifted right by $s_{k'} = |\beta|$ positions. Thus any phase that finds an occurrence of P overlapping an earlier run next shifts P by a number of positions larger than the length of the overlap (and hence the number of comparisons). It follows then that over the entire algorithm the total number of such additional comparisons in overlapping regions is $O(m)$.

All comparisons are accounted for and hence $\sum_{i\in Q} d_i = O(m)$, finishing the proof of the lemma. □

3.2.3. Adding in the bad character rule

Recall that in computing a shift after a mismatch, the Boyer–Moore algorithm uses the largest shift given by either the (extended) bad character rule or the (strong) good suffix rule. It seems intuitive that if the time bound is $O(m)$ when only the good suffix rule is used, it should still be $O(m)$ when both rules are used. However, certain "interference" is plausible, and so the intuition requires a proof.

Theorem 3.2.4. *When both shift rules are used together, the worst-case running time of the modified Boyer–Moore algorithm remains $O(m)$.*

PROOF In the analysis using only the suffix rule we focused on the comparisons done in an arbitrary phase i. In phase i the right end of P was aligned with some character of T. However, we never made any assumptions about how P came to be positioned there. Rather, given an arbitrary placement of P in a phase ending with a mismatch, we deduced bounds on how many characters compared in that phase could have been compared in earlier phases. Hence all of the lemmas and analyses remain correct if P is arbitrarily picked up and moved some distance to the right at any time during the algorithm. The (extended) bad character rule only moves P to the right, so all lemmas and analyses showing the $O(m)$ bound remain correct even with its use. □

3.3. The original preprocessing for Knuth-Morris-Pratt

3.3.1. The method does not use fundamental preprocessing

In Section 1.3 we showed how to compute all the sp_i values from Z_i values obtained during fundamental preprocessing of P. The use of Z_i values was conceptually simple and allowed a uniform treatment of various preprocessing problems. However, the classical preprocessing method given in Knuth-Morris-Pratt [278] is not based on fundamental preprocessing. The approach taken there is very well known and is used or extended in several additional methods (such as the Aho–Corasick method that is discussed next). For those reasons, a serious student of string algorithms should also understand the classical algorithm for Knuth-Morris-Pratt preprocessing.

The preprocessing algorithm computes $sp_i(P)$ for each position i from $i = 2$ to $i = n$ (sp_1 is zero). To explain the method, we focus on how to compute sp_{k+1} assuming that sp_i is known for each $i \leq k$. The situation is shown in Figure 3.9, where string α is the prefix of P of length sp_k. That is, α is the longest string that occurs both as a proper prefix of P and as a substring of P ending at position k. For clarity, let α' refer to the copy of α that ends at position k.

Let x denote character $k+1$ of P, and let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1} (i.e., the prefix that the algorithm will next try to compute). Finding sp_{k+1} is equivalent to finding string $\bar{\beta}$. And clearly,

*) $\bar{\beta}$ is the longest proper prefix of $P[1..k]$ that matches a suffix of $P[1..k]$ and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.10.

Our goal is to find sp_{k+1}, or equivalently, to find $\bar{\beta}$.

3.3.2. The easy case

Suppose the character just after α is x (i.e., $P(sp_k + 1) = x$). Then, string αx is a prefix of P and also a proper suffix of $P[1..k+1]$, and thus $sp_{k+1} \geq |\alpha x| = sp_k + 1$. Can we then end our search for sp_{k+1} concluding that sp_{k+1} equals $sp_k + 1$, or is it possible for sp_{k+1} to be strictly greater than $sp_k + 1$? The next lemma settles this.

Lemma 3.3.1. *For any k, $sp_{k+1} \leq sp_k + 1$. Further, $sp_{k+1} = sp_k + 1$ if and only if the character after α is x. That is, $sp_{k+1} = sp_k + 1$ if and only if $P(sp_k + 1) = P(k + 1)$.*

PROOF Let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1}. That is, $\beta = \bar{\beta}x$ is the longest proper suffix of $P[1..k+1]$ that is a prefix of P. If sp_{k+1} is strictly greater than

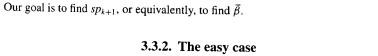


Figure 3.9: The situation after finding sp_k.

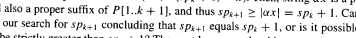


Figure 3.10: sp_{k+1} is found by finding $\bar{\beta}$.

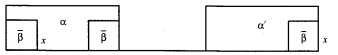


Figure 3.11: $\bar{\beta}$ must be a suffix of α.

$sp_k + 1 = |\alpha| + 1$, then $\bar{\beta}$ would be a prefix of P that is longer than α. But $\bar{\beta}$ is also a proper suffix of $P[1..k]$ (because βx is a proper suffix of $P[1..k+1]$). Those two facts would contradict the definition of sp_k (and the selection of α). Hence $sp_{k+1} \leq sp_k + 1$.

Now clearly, $sp_{k+1} = sp_k + 1$ if the character to the right of α is x, since αx would then be a prefix of P that also occurs as a proper suffix of $P[1..k+1]$. Conversely, if $sp_{k+1} = sp_k + 1$ then the character after α must be x. □

Lemma 3.3.1 identifies the largest "candidate" value for sp_{k+1} and suggests how to initially look for that value (and for string β). We should first check the character $P(sp_k+1)$, just to the right of α. If it equals $P(sp_k + 1)$ then we conclude that $\bar{\beta}$ equals α, β is αx, and sp_{k+1} equals $sp_k + 1$. But what do we do if the two characters are not equal?

3.3.3. The general case

When character $P(k + 1) \neq P(sp_k + 1)$, then $sp_{k+1} < sp_k + 1$ (by Lemma 3.3.1), so $sp_{k+1} \leq sp_k$. It follows that β must be a prefix of α, and $\bar{\beta}$ must be a *proper* prefix of α. Now substring $\beta = \bar{\beta}x$ ends at position $k + 1$ and is of length at most sp_k, whereas α' is a substring ending at position k and is of length sp_k. So $\bar{\beta}$ is a suffix of α', as shown in Figure 3.11. But since α' is a copy of α, $\bar{\beta}$ is also a suffix of α.

In summary, when $P(k + 1) \neq P(sp_k + 1)$, $\bar{\beta}$ occurs as a suffix of α and also as a proper prefix of α followed by character x. So when $P(k + 1) \neq P(sp_k + 1)$, $\bar{\beta}$ is the longest proper prefix of α that matches a suffix of α and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.11.

However, since $\alpha = P[1..sp_k]$, we can state this as

**) $\bar{\beta}$ is the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..k]$ and that is followed by character x in position $|\bar{\beta}| + 1$ of P.

The general reduction

Statements * and ** differ only by the substitution of $P[1..sp_k]$ for $P[1..k]$ and are otherwise exactly the same. Thus, when $P(sp_k + 1) \neq P(k + 1)$, the problem of finding $\bar{\beta}$ reduces to another instance of the original problem but on a smaller string ($P[1..sp_k]$ in place of $P[1..k]$). We should therefore proceed as before. That is, to search for $\bar{\beta}$ the algorithm should find the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..sp_k]$ and then check whether the character to the right of that prefix is x. By the definition of sp_k, the required prefix ends at character sp_{sp_k}. So if character $P(sp_{sp_k}+1) = x$ then we have found $\bar{\beta}$, or else we recurse again, restricting our search to ever smaller prefixes of P. Eventually, either a valid prefix is found, or the beginning of P is reached. In the latter case, $sp_{k+1} = 1$ if $P(1) = P(k + 1)$; otherwise $sp_{k+1} = 0$.

The complete preprocessing algorithm

Putting all the pieces together gives the following algorithm for finding $\bar{\beta}$ and sp_{k+1}:

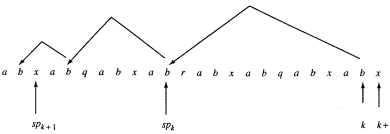


Figure 3.12: "Bouncing ball" cartoon of original Knuth-Morris-Pratt preprocessing. The arrows show the successive assignments to the variable v.

How to find sp_{k+1}

```
x := P(k + 1);
v := sp_k;
While P(v + 1) ≠ x and v ≠ 0 do
    v := sp_v;
end;
If P(v + 1) = x then
    sp_{k+1} := v + 1
else
    sp_{k+1} := 0;
```

See the example in Figure 3.12.

The entire set of sp values are found as follows:

Algorithm SP(P)

```
sp_1 = 0
For k := 1 to n − 1 do
begin
    x := P(k + 1);
    v := sp_k;
    While P(v + 1) ≠ x and v ≠ 0 do
        v := sp_v;
    end;
    If P(v + 1) = x then
        sp_{k+1} := v + 1
    else
        sp_{k+1} := 0;
end;
```

Theorem 3.3.1. *Algorithm SP finds all the $sp_i(P)$ values in $O(n)$ time, where n is the length of P.*

PROOF Note first that the algorithm consists of two nested loops, a *for* loop and a *while* loop. The *for* loop executes exactly $n - 1$ times, incrementing the value of k each time. The *while* loop executes a variable number of times each time it is entered.

The work of the algorithm is proportional to the number of times the value of v is assigned. We consider the places where the value of v is assigned and focus on how the value of v changes over the execution of the algorithm. The value of v is assigned once each time the *for* statement is reached; it is assigned a variable number of times inside the *while* loop, each time this loop is reached. Hence the number of times v is assigned is $n - 1$ plus the number of times it is assigned inside the *while* loop. How many times that can be is the key question.

Each assignment of v inside the *while* loop must decrease the value of v, and each of the $n - 1$ times v is assigned at the *for* statement, its value either increases by one or it remains unchanged (at zero). The value of v is initially zero, so the total amount that the value of v can increase (at the *for* statement) over the entire algorithm is at most $n - 1$. But since the value of v starts at zero and is never negative, the total amount that the value of v can *decrease* over the entire algorithm must also be bounded by $n - 1$, the total amount it can increase. Hence v can be assigned in the *while* loop at most $n - 1$ times, and hence the total number of times that the value of v can be assigned is at most $2(n - 1) = O(n)$, and the theorem is proved. □

3.3.4. How to compute the optimized shift values

The (stronger) sp'_i values can be easily computed from the sp_i values in $O(n)$ time using the algorithm below. For the purposes of the algorithm, character $P(n + 1)$, which does not exist, is defined to be different from any character in P.

Algorithm SP'(P)

```
sp'_1 = 0;
For i := 2 to n do
begin
    v := sp_i;
    If P(v + 1) ≠ P(i + 1) then
        sp'_i := v
    else
        sp'_i := sp'_v;
end;
```

Theorem 3.3.2. *Algorithm $SP'(P)$ correctly computes all the sp'_i values in $O(n)$ time.*

PROOF The proof is by induction on the value of i. Since $sp_1 = 0$ and $sp'_i \leq sp_i$ for all i, then $sp'_1 = 0$, and the algorithm is correct for $i = 1$. Now suppose that the value of sp'_i set by the algorithm is correct for all $i < k$ and consider $i = k$. If $P[sp_k + 1] \neq P[k + 1]$ then clearly sp'_k is equal to sp_k, since the sp_k length prefix of $P[1..k]$ satisfies all the needed requirements. Hence in this case, the algorithm correctly sets sp'_k.

If $P(sp_k + 1) = P(k + 1)$, then $sp'_k < sp_k$ and, since $P[1..sp_k]$ is a suffix $P[1..k]$, sp'_k can be expressed as the length of the longest proper prefix of $P[1..sp_k]$ that also occurs as a suffix of $P[1..sp_k]$ with the condition that $P(k + 1) \neq P(sp'_k + 1)$. But since $P(k + 1) = P(sp_k + 1)$, that condition can be rewritten as $P(sp_k + 1) \neq P(sp'_k + 1)$. By the induction hypothesis, that value has already been correctly computed as sp'_{sp_k}. So when $P(sp_k + 1) = P(k + 1)$ the algorithm correctly sets sp'_k to sp'_{sp_k}.

Because the algorithm only does constant work per position, the total time for the algorithm is $O(n)$. □

It is interesting to compare the classical method for computing sp and sp' and the method based on fundamental preprocessing (i.e., on Z values). In the classical method the (weaker) sp values are computed first and then the more desirable sp' values are derived

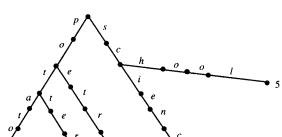


Figure 3.13: Keyword tree $\mathcal{K}$ with five patterns.

from them, whereas the order is just the opposite in the method based on fundamental preprocessing.

3.4. Exact matching with a set of patterns

An immediate and important generalization of the exact matching problem is to find all occurrences in text T of any pattern in a *set* of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$. This generalization is called the *exact set matching* problem. Let n now denote the total length of all the patterns in $\mathcal{P}$ and m be, as before, the length of T. Then, the exact set matching problem can be solved in time $O(n + zm)$ by separately using any linear-time method for each of the z patterns.

Perhaps surprisingly, the exact set matching problem can be solved faster than, $O(n + zm)$. It can be solved in $O(n + m + k)$ time, where k is the number of occurrences in T of the patterns from $\mathcal{P}$. The first method to achieve this bound is due to Aho and Corasick [9].[2] In this section, we develop the Aho–Corasick method; some of the proofs are left to the reader. An equally efficient, but more robust, method for the exact set matching problem is based on suffix trees and is discussed in Section 7.2.

Definition The *keyword tree* for set $\mathcal{P}$ is a rooted directed tree $\mathcal{K}$ satisfying three conditions: 1. each edge is labeled with exactly one character; 2. any two edges out of the same node have distinct labels; and 3. every pattern P_i in $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v exactly spell out P_i, and every leaf of $\mathcal{K}$ is mapped to by some pattern in $\mathcal{P}$.

For example, Figure 3.13 shows the keyword tree for the set of patterns {*potato, poetry, pottery, science, school*}.

Clearly, every node in the keyword tree corresponds to a prefix of one of the patterns in $\mathcal{P}$, and every prefix of a pattern maps to a distinct node in the tree.

Assuming a fixed-size alphabet, it is easy to construct the keyword tree for $\mathcal{P}$ in $O(n)$ time. Define $\mathcal{K}_i$ to be the (partial) keyword tree that encodes patterns $P_1, \ldots, P_i$ of $\mathcal{P}$.

[2] There is a more recent exposition of the Aho–Corasick method in [8], where the algorithm is used just as an "acceptor", deciding whether or not there is an occurrence in T of at least one pattern from $\mathcal{P}$. Because we will want to explicitly find all occurrences, that version of the algorithm is too limited to use here.

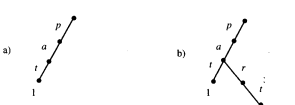


Figure 3.14: Pattern P_1 is the string *pat*. a. The insertion of pattern P_2 when P_2 is *pa*. b. The insertion when P_3 is *party*.

Tree $\mathcal{K}_1$ just consists of a single path of $|P_1|$ edges out of root r. Each edge on this path is labeled with a character of P_1 and when read from the root, these characters spell out P_1. The number 1 is written at the node at the end of this path. To create $\mathcal{K}_2$ from $\mathcal{K}_1$, first find the longest path from root r that matches the characters of P_2 in order. That is, find the longest prefix of P_2 that matches the characters on some path from r. That path either ends by exhausting P_2 or it ends at some node v in the tree where no further match is possible. In the first case, P_2 already occurs in the tree, and so we write the number 2 at the node where the path ends. In the second case, we create a new path out of v, labeled by the remaining (unmatched) characters of P_2, and write number 2 at the end of that path. An example of these two possibilities is shown in Figure 3.14.

In either of the above two cases, $\mathcal{K}_2$ will have at most one branching node (a node with more than one child), and the characters on the two edges out of the branching node will be distinct. We will see that the latter property holds inductively for any tree $\mathcal{K}_i$. That is, at any branching node v in $\mathcal{K}_i$, all edges out of v have distinct labels.

In general, to create $\mathcal{K}_{i+1}$ from $\mathcal{K}_i$, start at the root of $\mathcal{K}_i$ and follow, as far as possible, the (unique) path in $\mathcal{K}_i$ that matches the characters in P_{i+1} in order. This path is unique because, at any branching node v of $\mathcal{K}_i$, the characters on the edges out of v are distinct. If pattern P_{i+1} is exhausted (fully matched), then number the node where the match ends with the number $i + 1$. If a node v is reached where no further match is possible but P_{i+1} is not fully matched, then create a new path out of v labeled with the remaining unmatched part of P_{i+1} and number the endpoint of that path with the number $i + 1$.

During the insertion of P_{i+1}, the work done at any node is bounded by a constant, since the alphabet is finite and no two edges out of a node are labeled with the same character. Hence for any i, it takes $O(|P_{i+1}|)$ time to insert pattern P_{i+1} into $\mathcal{K}_i$, and so the time to construct the entire keyword tree is $O(n)$.

3.4.1. Naive use of keyword trees for set matching

Because no two edges out of any node are labeled with the same character, we can use the keyword tree to search for all occurrences in T of patterns from $\mathcal{P}$. To begin, consider how to search for occurrences of patterns in $\mathcal{P}$ that begin at character 1 of T: Follow the unique path in $\mathcal{K}$ that matches a prefix of T as far as possible. If a node is encountered on this path that is numbered by i, then P_i occurs in T starting from position 1. More than one such numbered node can be encountered if some patterns in $\mathcal{P}$ are prefixes of other patterns in $\mathcal{P}$.

In general, to find all patterns that occur in T, start from each position l in T and follow the unique path from r in $\mathcal{K}$ that matches a substring of T starting at character l.

Numbered nodes along that path indicate patterns in $\mathcal{P}$ that start at position l. For a fixed l, the traversal of a path of $\mathcal{K}$ takes time proportional to the minimum of m and n, so by successively incrementing l from 1 to m and traversing $\mathcal{K}$ for each l, the exact set matching problem can be solved in $O(nm)$ time. We will reduce this to $O(n + m + k)$ time below, where k is the number of occurrences.

The dictionary problem

Without any further embellishments, this simple keyword tree algorithm efficiently solves a special case of set matching, called the *dictionary problem*. In the dictionary problem, a set of strings (forming a dictionary) is initially known and preprocessed. Then a sequence of individual strings will be presented; for each one, the task is to find if the presented string is contained in the dictionary. The utility of a keyword tree is clear in this context. The strings in the dictionary are encoded into a keyword tree $\mathcal{K}$, and when an individual string is presented, a walk from the root of $\mathcal{K}$ determines if the string is in the dictionary. In this special case of exact set matching, the problem is to determine if the text T (an individual presented string) completely matches some string in $\mathcal{P}$.

We now return to the general set matching problem of determining which strings in $\mathcal{P}$ are *contained* in text T.

3.4.2. The speedup: generalizing Knuth-Morris-Pratt

The above naive approach to the exact set matching problem is analogous to the naive search we discussed before introducing the Knuth-Morris-Pratt method. Successively incrementing l by one and starting each search from root r is analogous to the naive exact match method for a single pattern, where after every mismatch the pattern is shifted by only one position, and the comparisons are always begun at the left end of the pattern. The Knuth-Morris-Pratt algorithm improves on that naive algorithm by shifting the pattern by more than one position when possible and by never comparing characters to the left of the current character in T. The Aho–Corasick algorithm makes the same kind of improvements, incrementing l by more than one and skipping over initial parts of paths in $\mathcal{K}$, when possible. The key is to generalize the function sp_i (defined on page 27 for a single pattern) to operate on a set of patterns. This generalization is fairly direct, with only one subtlety that occurs if a pattern in $\mathcal{P}$ is a proper substring of another pattern in $\mathcal{P}$. So, it is very helpful to (temporarily) make the following assumption:

Assumption No pattern in $\mathcal{P}$ is a proper substring of any other pattern in $\mathcal{P}$.

3.4.3. Failure functions for the keyword tree

Definition Each node v in $\mathcal{K}$ is *labeled* with the string obtained by concatenating in order the characters on the path from the root of $\mathcal{K}$ to node v. $\mathcal{L}(v)$ is used to denote the label on v. That is, the concatenation of characters on the path from the root to v spells out the string $\mathcal{L}(v)$.

For example, in Figure 3.15 the node pointed to by the arrow is labeled with the string *pott*.

Definition For any node v of $\mathcal{K}$, define $lp(v)$ to be the length of the longest proper suffix of string $\mathcal{L}(v)$ that is a prefix of some pattern in $\mathcal{P}$.

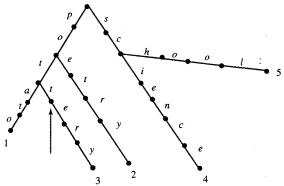


Figure 3.15: Keyword tree to illustrate the label of a node.

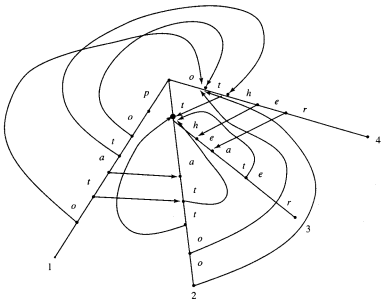


Figure 3.16: Keyword tree showing the failure links.

For example, consider the set of patterns $\mathcal{P} = \{potato, tattoo, theater, other\}$ and its keyword tree shown in Figure 3.16. Let v be the node labeled with the string *potat*. Since *tat* is prefix of *tattoo*, and it is the longest proper suffix of *potat* that is a prefix of any pattern in $\mathcal{P}$, $lp(v) = 3$.

Lemma 3.4.1. *Let α be the $lp(v)$-length suffix of string $\mathcal{L}(v)$. Then there is a unique node in the keyword tree that is labeled by string α.*

PROOF $\mathcal{K}$ encodes all the patterns in $\mathcal{P}$ and, by definition, the $lp(v)$-length suffix of $\mathcal{L}(v)$ is a prefix of some pattern in $\mathcal{P}$. So there must be a path from the root in $\mathcal{K}$ that spells out

string α. By the construction of T no two paths spell out the same string, so this path is unique and the lemma is proved. □

Definition For a node v of $\mathcal{K}$ let n_v be the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. When $lp(v) = 0$ then n_v is the root of $\mathcal{K}$.

Definition We call the ordered pair (v, n_v) a *failure link*.

Figure 3.16 shows the keyword tree for $\mathcal{P} = \{potato, tatoo, theater, other\}$. Failure links are shown as pointers from every node v to node n_v where $lp(v) > 0$. The other failure links point to the root and are not shown.

3.4.4. The failure links speed up the search

Suppose that we know the failure link $v \mapsto n_v$ for each node v in $\mathcal{K}$. (Later we will show how to efficiently find those links.) How do the failure links help speed up the search? The Aho–Corasick algorithm uses the function $v \mapsto n_v$ in a way that directly generalizes the use of the function $i \mapsto sp_i$ in the Knuth-Morris-Pratt algorithm. As before, we use l to indicate the starting position in T of the patterns being searched for. We also use pointer c into T to indicate the "current character" of T to be compared with a character on $\mathcal{K}$. The following algorithm uses the failure links to search for occurrences in T of patterns from $\mathcal{P}$:

Algorithm AC search

```
l := 1;
c := 1;
w := root of K;
repeat
    While there is an edge (w, w') labeled character T(c)
        begin
        if w' is numbered by pattern i then
            report that P_i occurs in T starting at position l;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c − lp(w);
until c > m;
```

To understand the use of the function $v \mapsto n_v$, suppose we have traversed the tree to node v but cannot continue (i.e., character $T(c)$ does not occur on any edge out of v). We know that string $\mathcal{L}(v)$ occurs in T starting at position l and ending at position $c - 1$. By the definition of the function $v \mapsto n_v$, it is guaranteed that string $\mathcal{L}(n_v)$ matches string $T[c - lp(v)..c - 1]$. That is, the algorithm could traverse $\mathcal{K}$ from the root to node n_v and be sure to match all the characters on this path with the characters in T starting from position $c - lp(v)$. So when $lp(v) \geq 0$, l can be increased to $c - lp(v)$, c can be left unchanged, and there is no need to actually make the comparisons on the path from the root to node n_v. Instead, the comparisons should begin at node n_v, comparing character c of T against the characters on the edges out of n_v.

For example, consider the text $T = xxpotattooxx$ and the keyword tree shown in Figure 3.16. When $l = 3$, the text matches the string *potat* but mismatches at the next character. At this point $c = 8$, and the failure link from the node v labeled *potat* points

to the node n_v labeled *tat*, and $lp(v) = 3$. So l is incremented to $5 = 8 - 3$, and the next comparison is between character $T(8)$ and character t on the edge below *tat*.

With this algorithm, when no further matches are possible, l may increase by more than one, avoiding the reexamination of characters of T to the left of c, and yet we may be sure that every occurrence of a pattern in $\mathcal{P}$ that *begins* at character $c - lp(v)$ of T will be correctly detected. Of course (just as in Knuth-Morris-Pratt), we have to argue that there are no occurrences of patterns of $\mathcal{P}$ starting strictly between the old l and $c - lp(v)$ in T, and thus l can be incremented to $c - lp(v)$ without missing any occurrences. With the given assumption that no pattern in $\mathcal{P}$ is a proper substring of another one, that argument is almost identical to the proof of Theorem 2.3.2 in the analysis of Knuth-Morris-Pratt, and it is left as an exercise.

When $lp(v) = 0$, then l is increased to c and the comparisons begin at the root of $\mathcal{K}$. The only case remaining is when the mismatch occurs at the root. In this case, c must be incremented by 1 and comparisons again begin at the root.

Therefore, the use of function $v \mapsto n_v$ certainly accelerates the naive search for patterns of $\mathcal{P}$. But does it improve the worst-case running time? By the same sort of argument used to analyze the search time (not the preprocessing time) of Knuth-Morris-Pratt (Theorem 2.3.3), it is easily established that the search time for Aho–Corasick is $O(m)$. We leave this as an exercise. However, we have yet to show how to precompute the function $v \mapsto n_v$ in linear time.

3.4.5. Linear preprocessing for the failure function

Recall that for any node v of $\mathcal{K}$, n_v is the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. The following algorithm finds node n_v for each node v in $\mathcal{K}$, using $O(n)$ total time. Clearly, if v is the root r or v is one character away from r, then $n_v = r$. Suppose, for some k, n_v has been computed for every node that is exactly k or fewer characters (edges) from r. The task now is to compute n_v for a node v that is $k + 1$ characters from r. Let v' be the parent of v in $\mathcal{K}$ and let x be the character on the v' to v edge, as shown in Figure 3.17.

We are looking for the node n_v and the (unknown) string $\mathcal{L}(n_v)$ labeling the path to it from the root; we know node $n_{v'}$ because v' is k characters from r. Just as in the explanation

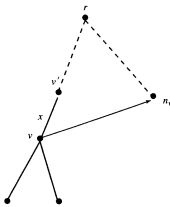


Figure 3.17: Keyword tree used to compute the failure function for node v.

of the classic preprocessing for Knuth-Morris-Pratt, $\mathcal{L}(n_v)$ must be a suffix of $\mathcal{L}(n_{v'})$ (not necessarily proper) followed by character x. So the first thing to check is whether there is an edge $(n_{v'}, w')$ out of node $n_{v'}$ labeled with character x. If that edge does exist, then n_v is node w' and we are done. If there is no such edge out of $n_{v'}$ labeled with character x, then $\mathcal{L}(n_v)$ is a *proper* suffix of $\mathcal{L}(n_{v'})$ followed by x. So we examine $n_{n_{v'}}$ next to see if there is an edge out of it labeled with character x. (Node $n_{n_{v'}}$ is known because $n_{v'}$ is k or fewer edges from the root.) Continuing in this way, with exactly the same justification as in the classic preprocessing for Knuth-Morris-Pratt, we arrive at the following algorithm for computing n_v for a node v:

Algorithm n_v

v' is the parent of v in $\mathcal{K}$;
x is the character on the edge (v', v);
$w := n_{v'}$;
While there is no edge out of w labeled x and $w \neq r$
    do $w := n_w$;
end (while);
If there is an edge (w, w') out of w labeled x then
    $n_v := w'$;
else
    $n_v := r$;

Note the importance of the assumption that n_u is already known for every node u that is k or fewer characters from r.

To find n_v for every node v, repeatedly apply the above algorithm to the nodes in $\mathcal{K}$ in a breadth-first manner starting at the root.

Theorem 3.4.1. *Let n be the total length of all the patterns in $\mathcal{P}$. The total time used by Algorithm n_v when applied to all nodes in $\mathcal{K}$ is $O(n)$.*

PROOF The argument is a direct generalization of the argument used to analyze time in the classic preprocessing for Knuth-Morris-Pratt. Consider a single pattern P in $\mathcal{P}$ of length t and its path in $\mathcal{K}$ for pattern P. We will analyze the time used in the algorithm to find the failure links for the nodes on this path, as if the path shares no nodes with paths for any other pattern in $\mathcal{P}$. That analysis will overcount the actual amount of work done by the algorithm, but it will still establish a linear time bound.

The key is to see how $lp(v)$ varies as the algorithm is executed on each successive node v down the path for P. When v is one edge from the root, then $lp(v)$ is zero. Now let v be an arbitrary node on the path for P and let v' be the parent of v. Clearly, $lp(v) \leq lp(v') + 1$, so over all executions of Algorithm n_v for nodes on the path for P, $lp()$ is increased by a total of at most t. Now consider how $lp()$ can decrease. During the computation of n_v for any node v, w starts at $n_{v'}$ and so has initial node depth equal to $lp(v')$. However, during the computation of n_v, the node depth of w decreases every time an assignment to w is made (inside the *while* loop). When n_v is finally set, $lp(v)$ equals the current depth of w, so if w is assigned k times, then $lp(v) \leq lp(v') - k$ and $lp()$ decreases by at least k. Now $lp()$ is never negative, and during all the computations along path P, $lp()$ can be increased by a total of at most t. It follows that over all the computations done for nodes on the path for P, the number of assignments made inside the *while* loop is at most t. The total time used is proportional to the number of assignments inside the loop, and hence all failure links on the path for P are set in $O(t)$ time.

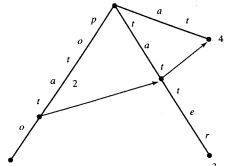


Figure 3.18: Keyword tree showing a directed path from *potat* to *at* through *tat*.

Repeating this analysis for every pattern in $\mathcal{P}$ yields the result that all the failure links are established in time proportional to the sum of the pattern lengths in $\mathcal{P}$ (i.e., in $O(n)$ total time). □

3.4.6. The full Aho–Corasick algorithm: relaxing the substring assumption

Until now we have assumed that no pattern in $\mathcal{P}$ is a substring of another pattern in $\mathcal{P}$. We now relax that assumption. If one pattern is a substring of another, and yet *Algorithm AC search* (page 56) uses the same keyword tree as before, then the algorithm may make l too large. Consider the case when $\mathcal{P} = \{acatt, ca\}$ and $T = acatg$. As given, the algorithm matches T along a path in $\mathcal{K}$ until character g is the current character. That path ends at the node v with $\mathcal{L}(v) = acat$. Now no edges out of v are labeled g, and since no proper suffix of *acat* is a prefix of *acatt* or *ac*, n_v is the root of $\mathcal{K}$. So when the algorithm gets stuck at node v it returns to the root with g as the current character, and it sets l to 5. Then after one additional comparison the current character pointer will be set to $m + 1$ and the algorithm will terminate without finding the occurrence of ca in T. This happens because the algorithm shifts (increases l) so as to match the longest *suffix* of $\mathcal{L}(v)$ with a prefix of some pattern in $\mathcal{P}$. Embedded occurrences of patterns in $\mathcal{L}(v)$ that are not suffixes of $\mathcal{L}(v)$ have no influence on how much l increases.

It is easy to repair this problem with the following observations whose proofs we leave to the reader.

Lemma 3.4.2. *Suppose in a keyword tree $\mathcal{K}$ there is a directed path of failure links (possibly empty) from a node v to a node that is numbered with pattern i. Then pattern P_i must occur in T ending at position c (the current character) whenever node v is reached during the search phase of the Aho–Corasick algorithm.*

For example, Figure 3.18 shows the keyword tree for $\mathcal{P} = \{potato, pot, tatter, at\}$ along with some of the failure links. Those links form a directed path from the node v labeled *potat* to the numbered node labeled *at*. If the traversal of $\mathcal{K}$ reaches v then T certainly contains the patterns *tat* and *at* end at the current c.

Conversely,

Lemma 3.4.3. *Suppose a node v has been reached during the algorithm. Then pattern*

P_i occurs in T ending at position c only if v is numbered i or there is a directed path of failure links from v to the node numbered i.

So the full search algorithm is

Algorithm full AC search

```
l := 1;
c := 1;
w := root;
repeat
    While there is an edge (w, w') labeled T(c)
        begin
        if w' is numbered by pattern i or there is
        a directed path of failure links from w' to a node numbered with i
            then report that P_i occurs in T ending at position c;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c - lp(w);
until c > n;
```

Implementation

Lemmas 3.4.2 and 3.4.3 specify at a high level how to find all occurrences of the patterns in the text, but specific implementation details are still needed. The goal is to be able to build the keyword tree, determine function $v \mapsto n_v$, and be able to execute the full AC search algorithm all in $O(m+k)$ time. To do this we add an additional pointer, called the *output link*, to each node of $\mathcal{K}$.

The output link (if there is one) at a node v points to that numbered node (a node associated with the end of a pattern in $\mathcal{P}$) other than v that is reachable from v by the fewest failure links. The output links can be determined in $O(n)$ time during the running of the preprocessing algorithm n_v. When the n_v value is determined, the possible output link from node v is determined as follows: If n_v is a numbered node then the output link from v points to n_v; if n_v is not numbered but has an output link to a node w, then the output link from v points to w; otherwise v has no output link. In this way, an output link points only to a numbered node, and the path of output links from any node v passes through all the numbered nodes reachable from v via a path of failure links. For example, in Figure 3.18 the nodes for *tat* and *potat* will have their output links set to the node for *at*. The work of adding output links adds only constant time per node, so the overall time for algorithm n_v remains $O(n)$.

With the output links, all occurrences in T of patterns of $\mathcal{P}$ can be detected in $O(m+k)$ time. As before, whenever a numbered node is encountered during the full AC search, an occurrence is detected and reported. But additionally, whenever a node v is encountered that has an output link from it, the algorithm must traverse the path of output links from v, reporting an occurrence ending at position c of T for each link in the path. When that path traversal reaches a node with no output link, it returns along the path to node v and continues executing the full AC search algorithm. Since no character comparisons are done during any output link traversal, over both the construction and search phases of the algorithm the number of character comparisons is still bounded by $O(n+m)$. Further, even though the number of traversals of output links can exceed that linear bound, each traversal

of an output link leads to the discovery of a pattern occurrence, so the total time for the algorithm is $O(n+m+k)$, where k is the total number of occurrences. In summary we have,

Theorem 3.4.2. *If $\mathcal{P}$ is a set of patterns with total length n and T is a text of total length m, then one can find all occurrences in T of patterns from $\mathcal{P}$ in $O(n)$ preprocessing time plus $O(m+k)$ search time, where k is the number of occurrences. This is true even without assuming that the patterns in $\mathcal{P}$ are substring free.*

In a later chapter (Section 6.5) we will discuss further implementation issues that affect the practical performance of both the Aho–Corasick method, and suffix tree methods.

3.5. Three applications of exact set matching

3.5.1. Matching against a DNA or protein library of known patterns

There are a number of applications in molecular biology where a relatively stable library of interesting or distinguishing DNA or protein substrings have been constructed. The *Sequence-tagged sites* (*STS*s) and *Expressed sequence tags* (*EST*s) provide our first important illustration.

Sequence-tagged-sites

The concept of a Sequence-tagged-site (STS) is one of the most useful by-products that has come out of the Human Genome Project [111, 234, 399]. Without going into full biological detail, an STS is intuitively a DNA string of length 200–300 nucleotides whose right and left ends, of length 20–30 nucleotides each, occur only once in the entire genome [111, 317]. Thus each STS occurs uniquely in the DNA of interest. Although this definition is not quite correct, it is adequate for our purposes. An early goal of the Human Genome Project was to select and map (locate on the genome) a set of STSs such that any substring in the genome of length 100,000 or more contains at least one of those STSs. A more refined goal is to make a map containing ESTs (expressed sequence tags), which are STSs that come from genes rather than parts of intergene DNA. ESTs are obtained from mRNA and cDNA (see Section 11.8.3 for more detail on cDNA) and typically reflect the protein coding parts of a gene sequence.

With an STS map, one can locate on the map any sufficiently long string of anonymous but sequenced DNA – the problem is just one of finding which STSs are contained in the anonymous DNA. Thus with STSs, map location of anonymous sequenced DNA becomes a string problem, an exact set matching problem. The STSs or the ESTs provide a computer-based set of indices to which new DNA sequences can be referenced. Presently, hundreds of thousands of STSs and tens of thousands of ESTs have been found and placed in computer databases [234]. Note that the total length of all the STSs and ESTs is very large compared to the typical size of an anonymous piece of DNA. Consequently, the keyword tree and the Aho–Corasick method (with a search time proportional to the length of the anonymous DNA) are of direct use in this problem for they allow very rapid identification of STSs or ESTs that occur in newly sequenced DNA.

Of course, there may be some errors in either the STS map or in the newly sequenced DNA causing trouble for this approach (see Section 16.5 for a discussion of STS maps). But in this application, the number of errors should be a small percentage of the length of the STS, and that will allow more sophisticated exact (and inexact) matching methods to succeed. We will describe some of these in Sections 7.8.3, 9.4, and 12.2 of the book.

A related application comes from the "BAC-PAC" proposal [442] for sequencing the human genome (see page 418). In that method, 600,000 strings (patterns) of length 500 would first be obtained and entered into the computer. Thousands of times thereafter, one would look for occurrences of any of these 600,000 patterns in text strings of length 150,000. Note that the total length of the patterns is 300 million characters, which is two-thousand times as large as the typical text to be searched.

3.5.2. Exact matching with wild cards

As an application of exact set matching, we return to the problem of exact matching with a single pattern, but complicate the problem a bit. We modify the exact matching problem by introducing a character ϕ, called a *wild card*, that matches any single character. Given a pattern P containing wild cards, we want to find all occurrences of P in a text T. For example, the pattern $ab\phi\phi c\phi$ occurs twice in the text $xabvccbababcax$. Note that in this version of the problem no wild cards appear in T and that each wild card matches only a single character rather than a substring of unspecified length.

The problem of matching with wild cards should need little motivating, as it is not difficult to think up realistic cases where the pattern contains wild cards. One very important case where simple wild cards occur is in DNA *transcription factors*. A transcription factor is a protein that binds to specific locations in DNA and regulates, either enhancing or suppressing, the transcription of the DNA into RNA. In this way, production of the protein that the DNA codes for is regulated. The study of transcription factors has exploded in the past decade; many transcription factors are now known and can be separated into families characterized by specific substrings containing wild cards. For example, the *Zinc Finger* is a common transcription factor that has the following signature:

$$\text{CYS}\phi\phi\text{CYS}\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\text{HIS}\phi\phi\text{HIS},$$

where CYS is the amino acid cysteine and HIS is the amino acid histidine. Another important transcription factor is the *Leucine Zipper*, which consists of four to seven leucines, each separated by six wild card amino acids.

If the number of permitted wild cards is unbounded, it is not known if the problem can be solved in linear time. However, if the number of wild cards is bounded by a fixed constant (independent of the size of P) then the following method, based on exact set pattern matching, runs in linear time:

Exact matching with wild cards

0. Let C be a vector of length $|T|$ initialized to all zeros.

1. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the (multi-)set of maximal substrings of P that do not contain any wild cards. Let $l_1, l_2, \ldots, l_k$ be the starting positions in P of each of these substrings.

{For example, if $P = ab\phi\phi c\phi ab\phi\phi$ then $\mathcal{P} = \{ab, c, ab\}$ and $l_1 = 1, l_2 = 5, l_3 = 7$.}

2. Using the Aho–Corasick algorithm (or the suffix tree approach to be discussed later), find for each string P_i in $\mathcal{P}$, all starting positions of P_i in text T. For each starting location j of P_i in T, increment the count in cell $j - l_i + 1$ of C by one.

{For example, if the second copy of string ab is found in T starting at position 18, then cell 12 of C is incremented by one.}

3. Scan vector C for any cell with value k. There is an occurrence of P in T starting at position p if and only if $C(p) = k$.

Correctness and complexity of the method

Correctness Clearly, there is an occurrence of P in T starting at position p if and only if, for each i, subpattern $P_i \in \mathcal{P}$ occurs at position $j = p + l_i - 1$ of T. The above method uses this idea in reverse. If pattern $P_i \in \mathcal{P}$ is found to occur starting at position j of T, and pattern P_i starts at position l_i in P, then this provides one "witness" that P occurs at T starting at position $p = j - l_i + 1$. Hence P occurs in T starting at p if and only if similar witnesses for position p are found for each of the k strings in $\mathcal{P}$. The algorithm counts, at position p, the number of witnesses that observe an occurrence of P beginning at p. This correctly determines whether P occurs starting at p because each string in $\mathcal{P}$ can cause at most one increment to cell p of C.

Complexity The time used by the Aho–Corasick algorithm to build the keyword tree for $\mathcal{P}$ is $O(n)$. The time to search for occurrences in T of patterns from $\mathcal{P}$ is $O(m + z)$, where $|T| = m$ and z is the number of occurrences. We treat each pattern in $\mathcal{P}$ as being distinct even if there are multiple copies of it in $\mathcal{P}$. Then whenever an occurrence of a pattern from $\mathcal{P}$ is found in T, exactly one cell in C is incremented; furthermore, a cell can be incremented to at most k. Hence z must be bounded by km, and the algorithm runs in $O(km)$ time . Although the number of character comparisons used is just $O(m)$, km need not be $O(m)$ and hence the number of times C is incremented may grow faster than $O(m)$, leading to a nonlinear $O(km)$ time bound. But if k is assumed to be bounded (independent of $|P|$), then the method does run in linear time. In summary,

Theorem 3.5.1. *If the number of wild cards in pattern P is bounded by a constant, then the exact matching problem with wild cards in the Pattern can be solved in $O(n+m)$ time.*

Later, in Sections 9.3, we will return to the problem of wild cards when they occur in either the pattern, text, or both.

3.5.3. Two-dimensional exact matching

A second classic application of exact set matching occurs in a generalization of string matching to two-dimensional exact matching. Suppose we have a *rectangular* digitized picture T, where each point is given a number indicating its color and brightness. We are also given a smaller rectangular picture P, which also is digitized, and we want to find all occurrences (possibly overlapping) of the smaller picture in the larger one. We assume that the bottom edges of the two rectangles are parallel to each other. This is a two-dimensional generalization of the exact string matching problem.

Admittedly, this problem is somewhat contrived. Unlike the one-dimensional exact matching problem, which truly arises in numerous practical applications, compelling applications of two-dimensional exact matching are hard to find. Two-dimensional matching that is inexact, allowing some errors, is a more realistic problem, but its solution requires

more complex techniques of the type we will examine in Part III of the book. So for now, we view two-dimensional exact matching as an illustration of how exact set matching can be used in more complex settings and as an introduction to more realistic two-dimensional problems. The method presented follows the basic approach given in [44] and [66]. Since then, many additional methods have been presented since that improve on those papers in various ways. However, because the problem as stated is somewhat unrealistic, we will not discuss the newer, more complex, methods. For a sophisticated treatment of two-dimensional matching see [22] and [169].

Let m be the total number of points in T, let n be the number of points in P, and let n' be the number of rows in P. Just as in exact string matching, we want to find the smaller picture in the larger one in $O(n+m)$ time, where $O(nm)$ is the time for the obvious approach. Assume for now that each of the rows of P are distinct; later we will relax this assumption.

The method is divided into two phases. In the first phase, search for all occurrences of each of the rows of P among the rows of T. To do this, add an end of row marker (some character not in the alphabet) to each row of T and concatenate these rows together to form a single text string T' of length $O(m)$. Then, treating each row of P as a separate pattern, use the Aho–Corasick algorithm to search for all occurrences in T' of any row of P. Since P is rectangular, all rows have the same width, and so no row is a proper substring of another and we can use the simpler version of Aho–Corasick discussed in Section 3.4.2. Hence the first phase identifies all occurrences of complete rows of P in complete rows of T and takes $O(n+m)$ time.

Whenever an occurrence of row i of P is found starting at position (p, q) of T, write the number i in position (p, q) of another array M with the same dimensions as T. Because each row of P is assumed to be distinct and because P is rectangular, at most one number will be written in any cell of M.

In the second phase, scan each *column* of M, looking for an occurrence of the string $1, 2, \ldots, n'$ in consecutive cells in a single column. For example, if this string is found in column 6, starting at row 12 and ending at row $n'+12$, then P occurs in T when its upper left corner is at position (6,12). Phase two can be implemented in $O(n'+m) = O(n+m)$ time by applying any linear-time exact matching algorithm to each column of M.

This gives an $O(n+m)$ time solution to the two-dimensional exact set matching problem. Note the similarity between this solution and the solution to the exact matching problem with wild cards discussed in the previous section. A distinction will be discussed in the exercises.

Now suppose that the rows of P are not all distinct. Then, first find all identical rows and give them a common label (this is easily done during the construction of the keyword tree for the row patterns). For example, if rows 3, 6, and 10 are the same then we might give them all the label of 3. We do a similar thing for any other rows that are identical. Then, in phase one, only look for occurrences of row 3, and not rows 6 and 10. This ensures that a cell of M will have at most one number written in it during phase 1. In phase 2, don't look for the string $1, 2, 3, \ldots, n'$ in the columns of M, but rather for a string where 3 replaces 6 and 10, etc. It is easy to verify that this approach is correct and that it takes just $O(n+m)$ time. In summary,

Theorem 3.5.2. *If T and P are rectangular pictures with m and n cells, respectively, then all exact occurrences of P in T can be found in $O(n+m)$ time, improving upon the naive method, which takes $O(nm)$ time.*

3.6. Regular expression pattern matching

A *regular expression* is a way to specify a set of related strings, sometimes referred to as a *pattern*.[3] Many important sets of substrings (patterns) found in biosequences, particularly in proteins, can be specified as regular expressions, and several databases have been constructed to hold such patterns. The PROSITE database, developed by Amos Bairoch [41, 42], is the major regular expression database for significant patterns in proteins (see Section 15.8 for more on PROSITE).

In this section, we examine the problem of finding substrings of a text string that match one of the strings specified by a given regular expression. These matches are computed in the Unix utility *grep*, and several special programs have been developed to find matches to regular expressions in biological sequences [279, 416, 422].

It is helpful to start first with an example of a simple regular expression. A formal definition of a regular expression is given later. The following PROSITE expression specifies a set of substrings, some of which appear in a particular family of granin proteins:

[ED]-[EN]-L-[SAN]-*x*-*x*-[DE]-*x*-E-L.

Every string specified by this regular expression has ten positions, which are separated by a dash. Each capital letter specifies a single amino acid and a group of amino acids enclosed by brackets indicates that exactly one of those amino acids must be chosen. A small *x* indicates that any one of the twenty amino acids from the protein alphabet can be chosen for that position. This regular expression describes 192,000 amino acid strings, but only a few of these actually appear in any known proteins. For example, ENLSSEDEEL is specified by the regular expression and is found in human granin proteins.

3.6.1. Formal definitions

We now give a formal, recursive definition for a regular expression formed from an alphabet Σ. For simplicity, and contrary to the PROSITE example, assume that alphabet Σ does not contain any symbol from the following list: $*, +, (,), \epsilon$.

Definition A single character from Σ is a regular expression. The symbol ϵ is a regular expression. A regular expression followed by another regular expression is a regular expression. Two regular expressions separated by the symbol "+" form a regular expression. A regular expression enclosed in parentheses is a regular expression. A regular expression enclosed in parentheses and followed by the symbol "*" is a regular expression. The symbol * is called the Kleene closure.

These recursive rules are simple to follow, but may need some explanation. The symbol ϵ represents the empty string (i.e., the string of length zero). If R is a parenthesized regular expression, then R^* means that the expression R can be repeated any number of times (including zero times). The inclusion of parentheses as part of a regular expression (outside of Σ) is not standard, but is closer to the way that regular expressions are actually specified in many applications. Note that the example given above in PROSITE format does not conform to the present definition but can easily be converted to do so.

As an example, let Σ be the alphabet of lower case English characters. Then $R = (a + c + t)ykk(p + q)^* \; vdt(l + z + \epsilon)(pq)$ is a regular expression over Σ, and $S =$

[3] Note that in the context of regular expressions, the meaning of the word "pattern" is different from its previous and general meaning in this book.

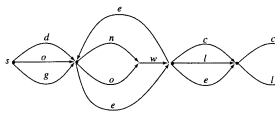


Figure 3.19: Directed graph for the regular expression $(d + o + g)((n + o)w)^*(c + l + \epsilon)(c + l)$.

aykkpqppvdtpq is a string specified by R. To specify S, the subexpression $(p+q)$ of R was repeated four times, and the empty string ϵ was the choice specified by the subexpression $(l + z + \epsilon)$.

It is very useful to represent a regular expression R by a directed graph $G(R)$ (usually called a nondeterministic, finite state automaton). An example is shown in Figure 3.19. The graph has a start node s and a termination node t, and each edge is labeled with a single symbol from $\Sigma \cup \epsilon$. Each s to t path in $G(R)$ specifies a string by concatenating the characters of Σ that label the edges of the path. The set of strings specified by all such paths is exactly the set of strings specified by the regular expression R. The rules for constructing $G(R)$ from R are simple and are left as an exercise. It is easy to show that if a regular expression R has n symbols, then $G(R)$ can be constructed using at most $2n$ edges. The details are left as an exercise and can be found in [10] and [8].

Definition A substring T' of string T *matches* the regular expression R if there is an s to t path in $G(R)$ that specifies T'.

Searching for matches

To search for a *substring* in T that matches the regular expression R, we first consider the simpler problem of determining whether some (unspecified) *prefix* of T matches R. Let $N(0)$ be the set of nodes consisting of node s plus all nodes of $G(R)$ that are reachable from node s by traversing edges labeled ϵ. In general, a node v is in set $N(i)$, for $i > 0$, if v can be reached from some node in $N(i-1)$ by traversing an edge labeled $T(i)$ followed by zero or more edges labeled ϵ. This gives a constructive rule for finding set $N(i)$ from set $N(i-1)$ and character $T(i)$. It easily follows by induction on i that a node v is in $N(i)$ if and only if there is path in $G(R)$ from s that ends at v and generates the string $T[1..i]$. Therefore, prefix $T[1..i]$ matches R if and only if $N(i)$ contains node t.

Given the above discussion, to find all prefixes of T that match R, compute the sets $N(i)$ for i from 0 to m, the length of T. If $G(R)$ contains e edges, then the time for this algorithm is $O(me)$, where m is the length of the text string T. The reason is that each iteration i [finding $N(i)$ from $N(i-1)$ and character $T(i)$] can be implemented to run in $O(e)$ time (see Exercise 29).

To search for a *nonprefix* substring of T that matches R, simply search for a prefix of T that matches the regular expression $\Sigma^* R$. Σ^* represents any number of repetitions (including zero) of any character in Σ. With this detail, we now have the following:

Theorem 3.6.1. *If T is of length m, and the regular expression R contains n symbols, then it is possible to determine whether T contains a substring matching R in $O(nm)$ time.*

3.7. Exercises

1. Evaluate empirically the speed of the Boyer–Moore method against the Apostolico–Giancarlo method under different assumptions about the text and the pattern. These assumptions should include the size of the alphabet, the "randomness" of the text or pattern, the level of periodicity of the text or pattern, etc.
2. In the Apostolico–Giancarlo method, array M is of size m, which may be large. Show how to modify the method so that it runs in the same time, but in place of M uses an array of size n.
3. In the Apostolico–Giancarlo method, it may be better to compare the characters first and then examine M and N if the two characters match. Evaluate this idea both theoretically and empirically.
4. In the Apostolico–Giancarlo method, $M(j)$ is set to be a number less than or equal to the length of the (right-to-left) match of P and T starting at position j of T. Find examples where the algorithm sets the value to be strictly less than the length of the match. Now, since the algorithm learns the exact location of the mismatch in all cases, $M(j)$ could always be set to the full length of the match, and this would seem to be a good thing to do. Argue that this change would result in a correct simulation of Boyer–Moore. Then explain why this was not done in the algorithm.
   Hint: It's the time bound.
5. Prove Lemma 3.2.2 showing the equivalence of the two definitions of semiperiodic strings.
6. For each of the n prefixes of P, we want to know whether the prefix $P[1..i]$ is a periodic string. That is, for each i we want to know the largest $k > 1$ (if there is one) such that $P[1..i]$ can be written as α^k for some string α. Of course, we also want to know the period. Show how to determine this for all n prefixes in time linear in the length of P.
   Hint: Z-algorithm.
7. Solve the same problem as above but modified to determine whether each prefix is semiperiodic and with what period. Again, the time should be linear.
8. By being more careful in the bookkeeping, establish the constant in the $O(m)$ bound from Cole's linear-time analysis of the Boyer–Moore algorithm.
9. Show where Cole's worst-case bound breaks down if only the *weak* Boyer–Moore shift rule is used. Can the argument be fixed, or is the linear time bound simply untrue when only the weak rule is used? Consider the example of T = *abababababababababab* and P = *xaaaaaaaaa* without also using the bad character rule.
10. Similar to what was done in Section 1.5, show that applying the classical Knuth-Morris-Pratt preprocessing method to the string $P\$T$ gives a linear-time method to find all occurrence of P in T. In fact, the search part of the Knuth-Morris-Pratt algorithm (after the preprocessing of P is finished) can be viewed as a slightly optimized version of the Knuth-Morris-Pratt preprocessing algorithm applied to the T part of $P\$T$. Make this precise, and quantify the utility of the optimization.
11. Using the assumption that $\mathcal{P}$ is *substring free* (i.e., that no pattern $P_i \in \mathcal{P}$ is a substring of another pattern $P_j \in \mathcal{P}$), complete the correctness proof of the Aho–Corasick algorithm. That is, prove that if no further matches are possible at a node v, then l can be set to $c - l(v)$ and the comparisons resumed at node n_v without missing any occurrences in T of patterns from $\mathcal{P}$.
12. Prove that the search phase of the Aho–Corasick algorithm runs in $O(m)$ time if no pattern in $\mathcal{P}$ is a proper substring of another, and otherwise in $O(m + k)$ time, where k is the total number of occurrences.
13. The Aho–Corasick algorithm can have the same problem that the Knuth-Morris-Pratt algorithm

has when it only uses sp values rather than sp' values. This is shown, for example, in Figure 3.16 where the edge below the character *a* in *potato* is directed to the character *a* in *tattoo*. A better failure function would avoid this situation. Give the details for computing such an improved failure function.

14. Give an example showing that k, the number of occurrences in T of patterns in set $\mathcal{P}$, can grow faster than $O(n+m)$. Be sure you account for the input size n. Try to make the growth as large as possible.

15. Prove Lemmas 3.4.2 and 3.4.3 that relate to the case of patterns that are not substring free.

16. The time analysis in the proof of Theorem 3.4.1 separately considers the path in $\mathcal{K}$ for each pattern P in $\mathcal{P}$. This results in an overcount of the time actually used by the algorithm. Perform the analysis more carefully to relate the running time of the algorithm to the number of nodes in $\mathcal{K}$.

17. Discuss the problem (and solution if you see one) of using the Aho–Corasick algorithm when **a**. wild cards are permitted in the text but not in the pattern and **b**. when wild cards are permitted in both the text and pattern.

18. Since the nonlinear time behavior of the wild card algorithm is due to duplicate copies of strings in $\mathcal{P}$, and such duplicates can be found and removed in linear time, it is tempting to "fix up" the method by first removing duplicates from $\mathcal{P}$. That approach is similar to what is done in the two-dimensional string matching problem when identical rows were first found and given a single label. Consider this approach and try to use it to obtain a linear-time method for the wild card problem. Does it work, and if not what are the problems?

19. Show how to modify the wild card method by replacing array C (which is of length $m > n$) by a list of length n, while keeping the same running time.

20. In the wild card problem we first assumed that no pattern in $\mathcal{P}$ is a substring of another one, and then we extended the algorithm to the case when that assumption does not hold. Could we instead simply reduce the case when substrings of patterns are allowed to the case when they are not? For example, perhaps we just add a new symbol to the end of each string in $\mathcal{P}$ that appears nowhere else in the patterns. Does it work? Consider both correctness and complexity issues.

21. Suppose that the wild card can match any length substring, rather than just a single character. What can you say about exact matching with these kinds of wild cards in the pattern, in the text, or in both?

22. Another approach to handling wild cards in the pattern is to modify the Knuth-Morris-Pratt or Boyer–Moore algorithms, that is, to develop shift rules and preprocessing methods that can handle wild cards in the pattern. Does this approach seem promising? Try it, and discuss the problems (and solutions if you see them).

23. Give a complete proof of the correctness and $O(n+m)$ time bound for the two-dimensional matching method described in the text (Section 3.5.3).

24. Suppose in the two-dimensional matching problem that Knuth-Morris-Pratt is used once for each pattern in $\mathcal{P}$, rather than Aho–Corasick being used. What time bound would result?

25. Show how to extend the two-dimensional matching method to the case when the bottom of the rectangular pattern is not parallel to the bottom of the large picture, but the orientation of the two bottoms is known. What happens if the pattern is not rectangular?

26. Perhaps we can omit phase two of the two-dimensional matching method as follows: Keep a counter at each cell of the large picture. When we find that row i of the small picture occurs in row j of the large picture starting at position (i', j), increment the counter for cell $(i', j-i+1)$. Then declare that P occurs in T with upper left corner in any cell whose counter becomes n' (the number of rows of P). Does this work?

Hint: No.

Why not? Can you fix it and make it run in $O(n+m)$ time?

27. Suppose we have $q > 1$ small (distinct) rectangular pictures and we want to find all occurrences of any of the q small pictures in a larger rectangular picture. Let n be the total number of points in all the small pictures and m be the number of points in the large picture. Discuss how to solve this problem efficiently. As a simplification, suppose all the small pictures have the same width. Then show that $O(n+m)$ time suffices.

28. Show how to construct the required directed graph $G(R)$ from a regular expression R. The construction should have the property that if R contains n symbols, then $G(R)$ contains at most $O(n)$ edges.

29. Since the directed graph $G(R)$ contains $O(n)$ edges when R contains n symbols, $|N(i)| = O(n)$ for any i. This suggests that the set $N(i)$ can be naively found from $N(i-1)$ and $T(i)$ in $O(ne)$ time. However, the time stated in the text for this task is $O(e)$. Explain how this reduction of time is achieved. Explain that the improvement is trivial if $G(R)$ contains no ϵ edges.

30. Explain the importance, or the utility, of ϵ edges in the graph $G(R)$. If R does not contain the closure symbol "*", can ϵ edges always be avoided? Biological strings are always finite, hence "*" can always be avoided. Explain how this simplifies the searching algorithm.

31. Wild cards can clearly be encoded into a regular expression, as defined in the text. However, it may be more efficient to modify the definition of a regular expression to explicitly include the wild card symbol. Develop that idea and explain how wild cards can be efficiently handled by an extension of the regular expression pattern matching algorithm.

32. PROSITE patterns often specify the number of times that a substring can repeat as a finite range of numbers. For example, CD(2–4) indicates that CD can repeat either two, three, or four times. The formal definition of a regular expression does not include such concise range specifications, but finite range specifications can be expressed in a regular expression. Explain how. How much do those specifications increase the length of the expression over the length of the more concise PROSITE expression? Show how such range specifications are reflected in the directed graph for the regular expression (ϵ edges are permitted). Show that one can still search for a substring of T that matches the regular expression in $O(me)$ time, where m is the length of T and e is the number of edges in the graph.

33. Theorem 3.6.1 states the time bound for determining if T contains a substring that matches a regular expression R. Extend the discussion and the theorem to cover the task of explicitly finding and outputting all such matches. State the time bound as the sum of a term that is independent of the number of matches plus a term that depends on that number.

4

Seminumerical String Matching

4.1. Arithmetic versus comparison-based methods

All of the exact matching methods in the first three chapters, as well as most of the methods that have yet to be discussed in this book, are examples of *comparison-based* methods. The main primitive operation in each of those methods is the comparison of two characters. There are, however, string matching methods based on *bit operations* or on *arithmetic*, rather than character comparisons. These methods therefore have a very different flavor than the comparison-based approaches, even though one can sometimes see character comparisons hidden at the inner level of these "seminumerical" methods. We will discuss three examples of this approach: the *Shift-And* method and its extension to a program called *agrep* to handle inexact matching; the use of the Fast Fourier Transform in string matching; and the random fingerprint method of Karp and Rabin.

4.2. The *Shift-And* method

R. Baeza-Yates and G. Gonnet [35] devised a simple, bit-oriented method that solves the exact matching problem very efficiently for relatively small patterns (the length of a typical English word for example). They call this method the *Shift-Or* method, but it seems more natural to call it *Shift-And*. Recall that pattern P is of size n and the text T is of size m.

Definition Let M be an n by $m+1$ *binary* valued array, with index i running from 1 to n and index j running from 1 to m. Entry $M(i, j)$ is 1 if and only if the first i characters of P exactly match the i characters of T ending at character j. Otherwise the entry is zero.

In other words, $M(i, j)$ is 1 if and only if $P[1..i]$ exactly matches $T[j-i+1..j]$. For example, if $T = californian$ and $P = for$, then $M(1,5) = M(2,6) = M(3,7) = 1$, whereas $M(i, j) = 0$ for all other combinations of i, j. Essentially, the entries with value 1 in row i of M show all the places in T where a copy of $P[1..i]$ ends, and column j of M shows all the prefixes of P that end at position j of T.

Clearly, $M(n, j) = 1$ if and only if an occurrence of P ends at position j of T; hence computing the last row of M solves the exact matching problem. For the algorithm to compute M it first constructs an n-length binary vector $U(x)$ for each character x of the alphabet. $U(x)$ is set to 1 for the positions in P where character x appears. For example, if $P = abacdeab$ then $U(a) = 10100010$.

Definition Define *Bit-Shift*$(j-1)$ as the vector derived by shifting the vector for *column* $j-1$ down by one position and setting that first to 1. The previous bit in position n disappears. In other words, *Bit-Shift*$(j-1)$ consists of 1 followed by the first $n-1$ bits of column $j-1$.

For example, Figure 4.1 shows a column $j-1$ before and after the bit-shift.

0	1
0	0
1	0
0	1
1	0
1	1
0	1
1	0

Figure 4.1: Column $j-1$ before and after operation *Bit-Shift*$(j-1)$.

4.2.1. How to construct array M

Array M is constructed column by column as follows: Column one of M is initialized to all zero entries if $T(1) \neq P(1)$. Otherwise, when $T(1) = P(1)$ its first entry is 1 and the remaining entries are 0. After that, the entries for column $j > 1$ are obtained from column $j-1$ and the U vector for character $T(j)$. In particular, the vector for column j is obtained by the bitwise **AND** of vector *Bit-Shift*$(j-1)$ with the U vector for character $T(j)$. More formally, if we let $M(j)$ denote the jth column of M, then $M(j) =$ *Bit-Shift*$(j-1)$ **AND** $U(T(j))$. For example, if $P = abaac$ and $T = xabxabaaxa$ then the eighth column of M is

1
0
1
0
0

because prefixes of P of lengths one and three end at position seven of T. The eighth character of T is character a, which has a U vector of

1
0
1
1
0

When the eighth column of M is shifted down and an **AND** is performed with $U(a)$, the result is

1
0
0
1
0

which is the correct ninth column of M.

To see in general why the *Shift-And* method produces the correct array entries, observe that for any $i > 1$ the array entry for cell (i, j) should be 1 if and only if the first $i-1$ characters of P match the $i-1$ characters of T ending at character $j-1$ and character $P(i)$ matches character $T(j)$. The first condition is true when the array entry for cell $(i-1, j-1)$ is 1, and the second condition is true when the ith bit of the U vector for character $T(j)$ is 1. By first shifting column $j-1$, the algorithm **AND**s together entry $(i-1, j-1)$ of column $j-1$ with entry i of the vector $U(T(j))$. Hence the algorithm computes the correct entries for array M.

4.2.2. *Shift-And* is effective for small patterns

Although the *Shift-And* method is very simple, and in worst case the number of bit operations is clearly $\Theta(mn)$, the method is very efficient if n is less than the size of a single computer word. In that case, every column of M and every U vector can be encoded into a single computer word, and both the *Bit-Shift* and the **AND** operations can be done as single-word operations. These are very fast operations in most computers and can be specified in languages such as C. Even if n is several times the size of a single computer word, only a few word operations are needed. Furthermore, only two columns of M are needed at any given time. Column j only depends on column $j-1$, so all previous columns can be forgotten. Hence, for reasonable sized patterns, such as single English words, the *Shift-And* method is very efficient in both time and space regardless of the size of the text. From a purely theoretical standpoint it is not a linear time method, but it certainly is practical and would be the method of choice in many circumstances.

4.2.3. *agrep*: The *Shift-And* method with errors

S. Wu and U. Manber [482] devised a method, packaged into a program called *agrep*, that amplifies the *Shift-And* method by finding *inexact* occurrences of a pattern in a text. By inexact we mean that the pattern either occurs exactly in the text or occurs with a "small" number of *mismatches* or *inserted* or *deleted* characters. For example, the pattern *atcgaa* occurs in the text *aatatccacaa* with two mismatches starting at position four; it also occurs with four mismatches starting at position two. In this section we will explain *agrep* and how it handles mismatches. The case of permitted insertions and deletions will be left as an exercise. For a small number of errors and for small patterns, *agrep* is very efficient and can be used in the core of more elaborate text searching methods. Inexact matching is the focus of Part III, but the ideas behind *agrep* are so closely related to the *Shift-And* method that it is appropriate to examine *agrep* at this point.

Definition For two strings P and T of lengths n and m, let M^k be a binary-valued array, where $M^k(i, j)$ is 1 if and only if at least $i-k$ of the first i characters of P match the i characters up through character j of T.

That is, $M^k(i, j)$ is the natural extension of the definition of $M(i, j)$ to allow up to k mismatches. Therefore, M^0 is the array M used in the *Shift-And* method. If $M^k(n, j) = 1$ then there is an occurrence of P in T ending at position j that contains at most k mismatches. We let $M^k(j)$ denote the jth column of M^k.

In *agrep*, the user chooses a value of k and then the arrays $M, M^1, M^2, \ldots, M^k$ are computed. The efficiency of the method depends on the size of k – the larger k is, the slower the method. For many applications, a value of k as small as 3 or 4 is sufficient, and the method is extremely fast.

4.2.4. How to compute M^k

Let k be the fixed maximum permitted number of mismatches specified by the user. The method will compute M^l for all values of l between 0 and k. There are several ways to organize the computation and its description, but for simplicity we will compute column j of each array M^l before any columns past j will be computed in any array. Further, for every j we will compute column j in arrays M^l in increasing order of l. In particular, the zero column of each array is again initialized to all zeros. Then the jth column of M^l is computed by:

$$M^l(j) = M^{l-1}(j) \textbf{ OR } [\textit{Bit-Shift}(M^l(j-1)) \textbf{ AND } U(T(j))] \textbf{ OR } M^{l-1}(j-1).$$

Intuitively, this just says that the first i characters of P will match a substring of T ending at position j, with at most l mismatches, if and only if one of the following three conditions hold:

- The first i characters of P match a substring of T ending at j, with at most $l-1$ mismatches.
- The first $i-1$ characters of P match a substring of T ending at $j-1$, with at most l mismatches, and the next pair of characters in P and T are equal.
- The first $i-1$ characters of P match a substring of T ending at $j-1$, with at most $l-1$ mismatches.

It is simple to establish that these recurrences are correct, and over the entire algorithm the number of bit operations is $O(knm)$. As in the *Shift-And* method, the practical efficiency comes from the fact that the vectors are bit vectors (again of length n) and the operations are very simple – shifting by one position and **AND**ing bit vectors. Thus when the pattern is relatively small, so that a column of any M^l fits into a few words, and k is also small, *agrep* is extremely fast.

4.3. The match-count problem and Fast Fourier Transform

If we relax the requirement that only bit operations are permitted and allow each entry of array M to hold an integer between 0 and n, then we can easily adapt the *Shift-And* method to compute for each pair i, j the number of characters of $P[1..i]$ that match $T[j-i+1..j]$. This computation is again a form of inexact matching, which is the focus of Part III. However, as was true of *agrep*, the solution is so connected to the *Shift-And* method that we consider it here. In addition, it is a natural introduction to the next topic, match-counts. For clarity, let us define a new matrix MC.

Definition The matrix MC is an n by $m+1$ integer-valued matrix, where entry $MC(i, j)$ is the number of characters of $P[1..i]$ that match $T[j-i+1..j]$.

A simple algorithm to compute matrix MC generalizes the *Shift-And* method, replacing the **AND** operation with the *increment by one* operation. The zero column of MC starts with all zeros, but each $MC(i, j)$ entry now is set to $MC(i-1, j-1)$ if $P(i) \neq T(j)$, and otherwise it is set to $MC(i-1, j-1)+1$. Any entry with value n in the last row again indicates an occurrence of P in T, but values less than n count the *exact number* of characters that match for each of different alignments of P with T. This extension uses $\Theta(nm)$ additions and comparisons, although each addition operation is particularly simple, just incrementing by one.

If we want to compute the entire MC array then $\Theta(nm)$ time is necessary, but the most important information is contained in the last row of MC. For each position $j \geq n$ in T, the last row indicates the number of characters that match when the right end of P is aligned with character j of T. The problem of finding the last row of MC is called the *match-count* problem. Match-counts are useful in several problems to be discussed later.

4.3.1. A fast worst-case method for the match-count problem?

Can any of the linear-time exact matching methods discussed in earlier chapters be adapted to solve the match-count problem in linear time? That is an open question. The extension of the *Shift-And* method discussed above solves the match-count problem but uses $\Theta(nm)$ arithmetic operations in all cases.

Surprisingly, the match-count problem can be solved with only $O(m \log m)$ arithmetic operations if we allow multiplication and division of complex numbers. The numbers remain small enough to permit the unit-time model of computation (that is, no number requires more than $O(\log m)$ bits), but the operations are still more complex than just incrementing by one. The $O(m \log m)$ method is based on the *Fast Fourier Transform* (*FFT*). This approach was developed by Fischer and Paterson [157] and independently in the biological literature by Felsenstein, Sawyer, and Kochin [152]. Other work that builds on this approach is found in [3], [58], [59], and [99]. We will reduce the match-count problem to a problem that can be efficiently solved by the FFT, but we treat the FFT itself as a black box and leave it to any interested reader to learn the details of the FFT.

4.3.2. Using Fast Fourier Transform for match-counts

The match-count problem is essentially that of finding the last row of the matrix MC. However, we will not work on MC directly, but rather we will solve a more general problem whose solution contains the desired information. For this more general problem the two strings involved will be denoted α and β rather than P and T, since the roles of the two strings will be completely symmetric. We still assume, however, that $|\alpha| = n \leq m = |\beta|$.

Definition Define $V(\alpha, \beta, i)$ to be the number of characters of α and β that match when the *left end* of string α is opposite position i of string β. Define $V(\alpha, \beta)$ to be the vector whose ith entry is $V(\alpha, \beta, i)$.

Clearly, when $\alpha = P$ and $\beta = T$ the vector $V(\alpha, \beta)$ contains the information needed for the last row of MC. But it contains more information because we allow the left end of α to be to the left of the left end of β, and we also allow the right end of α to be to the right of the right end of β. Negative numbers specify positions to the left of the left end of β, and positive numbers specify the other positions. For example, when α is aligned with β as follows,

```
                 21123456789
B:                 accctgtcc
A:               aactgccg
```

then the left end of α is aligned with position -2 of β.

Index i ranges from $-n + 1$ to m. Notice that when $i > m - n$, the right end of α is right of the right end of β. For any fixed i, $V(\alpha, \beta, i)$ can be directly computed in $O(n)$ time (for any i, just directly count the number of resulting matches and mismatches), so $V(\alpha, \beta)$ can be computed in $O(nm)$ total time.

We now show how to compute $V(\alpha, \beta)$ in $O(m \log m)$ total time by using the Fast Fourier Transform. For most problems of interest $\log m \ll n$, so this technique yields a large speedup. Further, there is specialized hardware for FFT that is very fast, suggesting a way to solve these problems quickly with hardware. The solution will work for any alphabet, but it is easiest to explain it on a small alphabet. For concreteness we use the four-letter alphabet a, t, c, g of DNA.

The high-level approach

We break up the match-count problem into four problems, one for each character in the alphabet.

Definition Define $V_a(\alpha, \beta, i)$ to be the number of matches of character 'a' that occur when the start of string α is positioned opposite position i of string β. $V_a(\alpha, \beta)$ is the $(n + m)$-length vector holding these values.

Similar definitions apply for the other three characters. With these definitions,

$$V(\alpha, \beta, i) = V_a(\alpha, \beta, i) + V_t(\alpha, \beta, i) + V_c(\alpha, \beta, i) + V_g(\alpha, \beta, i)$$

and

$$V(\alpha, \beta) = V_a(\alpha, \beta) + V_t(\alpha, \beta) + V_c(\alpha, \beta) + V_g(\alpha, \beta).$$

The problem then becomes how to compute $V_a(\alpha, \beta, i)$ for each i. Convert the two strings into binary strings $\vec{\alpha}_a$ and $\vec{\beta}_a$, respectively, where every occurrence of character a becomes a 1, and all other characters become 0s. For example, let α be *accacggaggtat* and β be *acccacgaag*. Then the binary strings $\vec{\alpha}_a$ and $\vec{\beta}_a$ are 1011000100010 and 1001001 10. To compute $V_a(\alpha, \beta, i)$, position $\vec{\beta}_a$ to start at position i of $\vec{\alpha}_a$ and count the number of columns where both bits are equal to 1. For example, if $i = 3$ then we get

```
1011000100010
  10010110
```

and the $V_a(\alpha, \beta, 3) = 2$. If $i = 9$ then we have

```
1011000100010
        10010110
```

and $V_a(\alpha, \beta, 9) = 1$.

Another way to view this is to consider each space opposite a bit to be a 0 (so both binary strings are the same length), do a bitwise **AND** operation with the strings, and then add the resulting bits.

To formalize this idea, pad the right end of $\vec{\beta}$ (the larger string) with n additional zeros and pad the right end of $\vec{\alpha}$ with m additional zeros. The two resulting strings then each have length $n + m$. Also, for convenience, renumber the indices of both strings to run from 0 to $n + m - 1$. Then

$$V_a(\alpha, \beta, i) = \sum_{j=0}^{j=n+m-1} \vec{\alpha}_a(j) \times \vec{\beta}_a(i + j),$$

where the indices in the expression are taken modulo $n + m$. The extra zeros are there to handle the cases when the left end of α is to the left end of β and, conversely, when the right end of α is to the right end of β. Enough zeros were padded so that when the right end of α is right of the right end of β, the corresponding bits in the padded $\vec{\alpha}_a$ are all opposite zeros. Hence no "illegitimate wraparound" of α and β is possible, and $V_a(\alpha, \beta, i)$ is correctly computed.

So far, all we have done is to recode the match-count problem, and this recoding doesn't suggest a way to compute $V_a(\alpha, \beta)$ more efficiently than before the binary coding and padding. This is where correlation and the FFT come in.

Cyclic correlation

Definition Let X and Y be two z-length vectors with real number components indexed from 0 to $z-1$. The *cyclic correlation* of X and Y is an z-length real vector $W(i) = \sum_{j=0}^{j=z-1} X(j) \times Y(i+j)$, where the indices in the expression are taken modulo z.

Clearly, the problem of computing vector $V_a(\alpha, \beta)$ is exactly the problem of computing the cyclic correlation of padded strings $\overline{\alpha}_a$ and $\overline{\beta}_a$. In detail, $X = \overline{\alpha}_a$, $Y = \overline{\beta}_a$, $z = n + m$, and $W = V_a(\alpha, \beta)$.

Now an algorithm based only on the definition of cyclic correlation would require $\Theta(z^2)$ operations, so again no progress is apparent. But cyclic correlation is a classic problem known to be solvable in $O(z \log z)$ time using the Fast Fourier Transform. (The FFT is more often associated with the *convolution* problem for two vectors, but cyclic correlation and convolution are very similar. In fact, cyclic correlation is solved by reversing one of the input vectors and then computing the convolution of the two resulting vectors.)

The FFT method, and its use in the solution of the cyclic correlation problem, is beyond the scope of this book, but the key is that it solves the cyclic correlation problem in $O(z \log z)$ arithmetic operations, for two vectors each of length z. Hence it solves the match-count problem using only $O(m \log m)$ arithmetic operations. This is surprisingly efficient and a definite improvement over the $\Theta(nm)$ bound given by the generalized *Shift-And* approach. However, the FFT requires operations over complex numbers and so each arithmetic step is more involved (and perhaps more costly) than in the more direct *Shift-And* method.[1]

Handling wild cards in match-counts

Recall the wild card discussion begun in Section 3.5.2, where the wild card symbol ϕ matches any other *single* character. For example, if $\alpha = ag\phi\phi ctctga$ and $\beta = agctctgt$, then $V(\alpha, \beta, 1) = 7$ (i.e., all positions are counted as a match except the last). How do we incorporate these wild card symbols into the FFT approach computing match-counts? If the wild cards only occur in one of the two strings, say β, then the solution is very direct. When computing $V_x(\alpha, \beta, i)$ for every position i and character x, simply replace each wild card symbol in β with the character x. This works because for any fixed starting point i and any position j in β, the jth position will contribute a 1 to $V_x(\alpha, \beta, i)$ for at most one character x, depending on what character is in position $i + j - 1$ of α (i.e., what character in α is opposite the j position in β).

But if wild cards occur in both α and β, then this direct approach will not work. If two wild cards are opposite each other when α starts at position i, then $V_x(\alpha, \beta, i)$ would be too large, since those two symbols will be counted as a match when computing $V_x(\alpha, \beta, i)$ for each $x = a, t, c$, and g. So if for a fixed i there are k places where two wild cards line up, then the computed $\sum_x V_x(\alpha, \beta, i)$ will be $3k$ larger than the correct $V(\alpha, \beta, i)$ value. How can we avoid this overcount?

The answer is to find what k is and then correct the overcount. The idea is to treat ϕ as a real character, and compute $V_\phi(\alpha, \beta, i)$ for each i. Then

$$V(\alpha, \beta, i) = \sum_{x \neq \phi} V_x(\alpha, \beta, i) - 3V_\phi(\alpha, \beta, i),$$

where for each character x, $V_x(\alpha, \beta, i)$ is computed by replacing each wild card with character x. In summary,

Theorem 4.3.1. *The match-count problem can be solved in $O(m \log m)$ time even if an unbounded number of wild cards are allowed in either P or T.*

Later, after discussing suffix trees and common ancestors, we will present in Section 9.3 a different, more comparison-based approach to handling wild cards that appear in both strings.

4.4. Karp–Rabin fingerprint methods for exact match

The *Shift-And* method assumes that we can efficiently shift a vector of bits, and the generalized *Shift-And* method assumes that we can efficiently increment an integer by one. If we treat a (row) bit vector as an integer number then a left shift by one bit results in the doubling of the number (assuming no bits fall off the left end). So it is not much of an extension to assume, in addition to being able to increment an integer, that we can also efficiently multiply an integer by two. With that added primitive operation we can turn the exact match problem (again without mismatches) into an arithmetic problem. The first result will be a simple linear-time method that has a very small probability of making an error. That method will then be transformed into one that never makes an error, but whose running time is only expected to be linear. We will explain these results using a binary string P and a binary text T. That is, the alphabet is first assumed to be just $\{0, 1\}$. The extension to larger alphabets is immediate and will be left to the reader.

4.4.1. Arithmetic replaces comparisons

Definition For a text string T, let T_r^n denote the n-length substring of T starting at character r. Usually, n is known by context, and T_r^n will be replaced by T_r.

Definition For the binary pattern P, let

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i).$$

Similarly, let

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1).$$

That is, consider P to be an n-bit binary number. Similarly, consider T_r^n to be an n-bit binary number. For example, if $P = 0101$ then $n = 4$ and $H(P) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$; if $T = 101101010$, $n = 4$, and $r = 2$, then $H(T_r) = 6$.

Clearly, if there is an occurrence of P starting at position r of T then $H(P) = H(T_r)$. However, the converse is also true, so

Theorem 4.4.1. *There is an occurrence of P starting at position r of T if and only if $H(P) = H(T_r)$.*

[1] A related approach [58] attempts to solve the match-count problem in $O(m \log m)$ integer (noncomplex) operations by implementing the FFT over a finite field. In practice, this approach is probably superior to the approach based on complex numbers, although in terms of pure complexity theory the claimed $O(m \log m)$ bound is not completely kosher because it uses a precomputed table of numbers that is only adequate for values of m up to a certain size.

The proof, which we leave to the reader, is an immediate consequence of the fact that every integer can be written in a unique way as the sum of positive powers of two.

Theorem 4.4.1 converts the exact match problem into a numerical problem, comparing the two numbers $H(P)$ and $H(T_r)$ rather than directly comparing characters. But unless the pattern is fairly small, the computation of $H(P)$ and $H(T_r)$ will not be efficient.[2] The problem is that the required powers of two used in the definition of $H(P)$ and $H(T_r)$ grow large too rapidly. (From the standpoint of complexity theory, the use of such large numbers violates the unit-time *random access machine* (*RAM*) model. In that model, the largest allowed numbers must be represented in $O[\log(n+m)]$ bits, but the number 2^n requires n bits. Thus the required numbers are exponentially too large.) Even worse, when the alphabet is not binary but say has t characters, then numbers as large as t^n are needed.

In 1987 R. Karp and M. Rabin [266] published a method (devised almost ten years earlier), called the *randomized fingerprint* method, that preserves the spirit of the above numerical approach, but that is extremely efficient as well, using numbers that satisfy the *RAM* model. It is a *randomized* method where the *only if* part of Theorem 4.4.1 continues to hold, but the *if* part does not. Instead, the *if* part will hold *with high probability*. This is explained in detail in the next section.

4.4.2. Fingerprints of P and T

The general idea is that, instead of working with numbers as large as $H(P)$ and $H(T_r)$, we will work with those numbers *reduced modulo* a relatively small integer p. The arithmetic will then be done on numbers requiring only a small number of bits, and so will be efficient. But the really attractive feature of this method is a proof that the probability of error can be made small if p is chosen randomly in a certain range. The following definitions and lemmas make this precise.

Definition For a positive integer p, $H_p(P)$ is defined as $H(P) \bmod p$. That is $H_p(P)$ is the remainder of $H(P)$ after division by p. Similarly, $H_p(T_r)$ is defined as $H(T_r) \bmod p$. The numbers $H_p(P)$ and $H_p(T_r)$ are called *fingerprints* of P and T_r.

Already, the utility of using fingerprints should be apparent. By reducing $H(P)$ and $H(T_r)$ modulo a number p, every fingerprint remains in the range 0 to $p-1$, so the size of a fingerprint does not violate the *RAM* model. But if $H(P)$ and $H(T_r)$ must be computed before they can be reduced modulo p, then we have the same problem of intermediate numbers that are too large. Fortunately, modular arithmetic allows one to reduce at any time (i.e., one can never reduce too much), so that the following generalization of Horner's rule holds:

Lemma 4.4.1. $H_p(P) = \{[\ldots([P(1)\times 2 \bmod p + P(2)]\times 2 \bmod p + P(3)]\times 2 \bmod p + P(4))\ldots] \bmod p + P(n)\} \bmod p$, *and no number ever exceeds* $2p$ *during the computation of* $H_p(P)$.

[2] One can more efficiently compute $H(T_{r+1})$ from $H(T_r)$ than by following the definition directly (and we will need that later on), but the time to do the updates is not the issue here.

For example, if $P = 101111$ and $p = 7$, then $H(P) = 47$ and $H_p(P) = 47 \bmod 7 = 5$. Moreover, this can be computed as follows:

$$\begin{aligned}
&1 \times 2 \bmod 7 + 0 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \times 2 \bmod 7 + 1 = 4\\
&4 \times 2 \bmod 7 + 1 = 2\\
&2 \times 2 \bmod 7 + 1 = 5\\
&5 \bmod 7 = 5.
\end{aligned}$$

The point of Horner's rule is not only that the number of multiplications and additions required is linear, but that the intermediate numbers are always kept small.

Intermediate numbers are also kept small when computing $H_p(T_r)$ for any r, since that computation can be organized the way that $H_p(P)$ was. However, even greater efficiency is possible: For $r > 1$, $H_p(T_r)$ can be computed from $H_p(T_{r-1})$ with only a small *constant* number of operations. Since

$$H_p(T_r) = H(T_r) \bmod p$$

and

$$H(T_r) = 2 \times H(T_{r-1}) - 2^n T(r-1) + T(r+n-1),$$

it follows that

$$H_p(T_r) = [(2 \times H(T_{r-1}) \bmod p) - (2^n \bmod p) \times T(r-1) + T(r+n-1)] \bmod p.$$

Further,

$$2^n \bmod p = 2 \times (2^{n-1} \bmod p) \bmod p.$$

Therefore, each successive power of two taken mod p and each successive value $H_p(T_r)$ can be computed in constant time.

Prime moduli limit false matches

Clearly, if P occurs in T starting at position r then $H_p(P) = H_p(T_r)$, but now the converse does not hold for every p. That is, we cannot necessarily conclude that P occurs in T starting at r just because $H_p(P) = H_p(T_r)$.

Definition If $H_p(P) = H_p(T_r)$ but P does not occur in T starting at position r, then we say there is a *false match* between P and T at position r. If there is *some* position r such that there is a false match between P and T at r, then we say there is a false match between P and T.

The goal will be to choose a modulus p small enough that the arithmetic is kept efficient, yet large enough that the probability of a false match between P and T is kept small. The key comes from choosing p to be a *prime* number in the proper range and exploiting properties of prime numbers. We will state the needed properties of prime numbers without proof.

Definition For a positive integer u, $\pi(u)$ is the *number* of primes that are less than or equal to u.

The following theorem is a variant of the famous *prime number theorem*.

Theorem 4.4.2. $\frac{u}{\ln(u)} \le \pi(u) \le 1.26\frac{u}{\ln(u)}$, *where* $\ln(u)$ *is the base e logarithm of* u [383].

Lemma 4.4.2. *If $u \geq 29$, then the product of all the primes that are less than or equal to u is greater than 2^u* [383].

For example, for $u = 29$ the prime numbers less than or equal to 29 are 2, 5, 7, 11, 13, 17, 19, 23, and 29. Their product is 2,156,564,410 whereas 2^{29} is 536,870,912.

Corollary 4.4.1. If $u \geq 29$ and x is any number less than or equal to 2^u, then x has fewer than $\pi(u)$ (distinct) prime divisors.

PROOF Suppose x does have $k > \pi(u)$ distinct prime divisors $q_1, q_2, \ldots, q_k$. Then $2^u \geq x \geq q_1 q_2 \ldots q_k$ (the first inequality is from the statement of the corollary, and the second from the fact that some primes in the factorization of x may be repeated). But $q_1 q_2 \ldots q_k$ is at least as large as the product of the smallest k primes, which is greater than the product of the first $\pi(u)$ primes (by assumption that $k > \pi(u)$). However, the product of the primes less than or equal to u is greater than 2^u (by Lemma 4.4.2). So the assumption that $k > \pi(u)$ leads to the contradiction that $2^u > 2^u$, and the lemma is proved. □

The central theorem

Now we are ready for the central theorem of the Karp–Rabin approach.

Theorem 4.4.3. *Let P and T be any strings such that $nm \geq 29$, where n and m are the lengths of P and T, respectively. Let I be any positive integer. If p is a randomly chosen prime number less than or equal to I, then the probability of a false match between P and T is less than or equal to $\frac{\pi(nm)}{\pi(I)}$.*

PROOF Let R be the set of positions in T where P does *not* begin. That is, $s \in R$ if and only if P does not occur in T beginning at s. For each $s \in R$, $H(P) \neq H(T_s)$. Now consider the product $\Pi_{s \in R}(|H(P) - H(T_s)|)$. That product must be at most 2^{nm} since for any s, $H(P) - H(T_s) \leq 2^n$ (recall that we have assumed a binary alphabet). Applying Corollary 4.4.1, $\Pi_{s \in R}(|H(P) - H(T_s)|)$ has at most $\pi(nm)$ distinct prime divisors.

Now suppose a false match between P and T occurs at some position r of T. That means that $H(P) \bmod p = H(T_r) \bmod p$ and that p evenly divides $H(P) - H(T_r)$. Trivially then, p evenly divides $\Pi_{s \in R}(|H(P) - H(T_s)|)$, and so p is one of the prime divisors of that product. If p allows a false match to occur between P and T, then p must be one of a set of at most $\pi(nm)$ numbers. But p was chosen randomly from a set of $\pi(I)$ numbers, so the probability that p is a prime that allows a false match between P and T is at most $\frac{\pi(nm)}{\pi(I)}$. □

Notice that Theorem 4.4.3 holds for any choice of pattern P and text T such that $nm \geq 29$. The probability in the theorem is not taken over choices of P and T but rather over choices of prime p. Thus, this theorem does not make any (questionable) assumptions about P or T being random or generated by a Markov process, etc. It works for any P and T! Moreover, the theorem doesn't just bound the probability that a false match occurs at a fixed position r, it bounds the probability that there is even a single such position r in T. It is also notable that the analysis in the proof of the theorem feels "weak". That is, it only develops a very weak property of a prime p that allows a false match, namely being one of at most $\pi(nm)$ numbers that divide $\Pi_{s \in R}(|H(P) - H(T_s)|)$. This suggests that the true probability of a false match occurring between P and T is much less than the bound established in the theorem.

Theorem 4.4.3 leads to the following random fingerprint algorithm for finding all occurrences of P in T.

Random fingerprint algorithm

1. Choose a positive integer I (to be discussed in more detail below).
2. Randomly pick a prime number less than or equal to I, and compute $H_p(P)$. (Efficient randomized algorithms exist for finding random primes [331].)
3. For each position r in T, compute $H_p(T_r)$ and test to see if it equals $H_p(P)$. If the numbers are equal, then either declare a probable match or check explicitly that P occurs in T starting at that position r.

Given the fact that each $H_p(T_r)$ can be computed in constant time from $H_p(T_{r-1})$, the fingerprint algorithm runs in $O(m)$ time, excluding any time used to explicitly check a declared match. It may, however, be reasonable not to bother explicitly checking declared matches, depending on the probability of an error. We will return to the issue of checking later. For now, to fully analyze the probability of error, we have to answer the question of what I should be.

How to choose I

The utility of the fingerprint method depends on finding a good value for I. As I increases, the probability of a false match between P and T decreases, but the allowed size of p increases, increasing the effort needed to compute $H_p(P)$ and $H_p(T_r)$. Is there a good balance? There are several good ways to choose I depending on n and m. One choice is to take $I = nm^2$. With that choice the largest number used in the algorithm requires at most $4(\log n + \log m)$ bits, satisfying the *RAM* model requirement that the numbers be kept small as a function of the size of the input. But, what of the probability of a false match?

Corollary 4.4.2. When $I = nm^2$, the probability of a false match is at most $\frac{2.53}{m}$.

PROOF By Theorem 4.4.3 and the prime number theorem (Theorem 4.4.2), the probability of a false match is bounded by

$$\frac{\pi(nm)}{\pi(nm^2)} \leq 1.26 \frac{nm}{nm^2} \frac{\ln(nm^2)}{\ln(nm)} = 1.26 \frac{1}{m} \left[\frac{\ln(n) + 2\ln(m)}{\ln(n) + \ln(m)} \right] \leq \frac{2.53}{m}. \qquad \square$$

A small example from [266] illustrates this bound. Take $n = 250$, $m = 4000$, and hence $I = 4 \times 10^9 < 2^{32}$. Then the probability of a false match is at most $\frac{2.53}{4000} < 10^{-3}$. Thus, with just a 32-bit fingerprint, for any P and T the probability that even a single one of the algorithm's declarations is wrong is bounded by 0.001.

Alternately, if $I = n^2 m$ then the probability of a false match is $O(1/n)$, and since it takes $O(n)$ time to determine whether a match is false or real, the *expected* verification time would be constant. The result would be an $O(m)$ *expected* time method that never has a false match.

Extensions

If one prime is good, why not use several? Why not pick k primes $p_1, p_2, \ldots, p_k$ randomly and compute k fingerprints? For any position r, there can be an occurrence of P starting at r *only if* $H_{p_i}(P) = H_{p_i}(T_r)$ for *every* one of the k selected primes. We now define a false match between P and T to mean that there is an r such that P does not occur in T starting at r, but $H_{p_i}(P) = H_{p_i}(T_r)$ for *each* of the k primes. What now is the probability of a false match between P and T? One bound is fairly immediate and intuitive.

Theorem 4.4.4. *When k primes are chosen randomly between 1 and I and k fingerprints are used, the probability of a false match between P and T is at most $[\frac{\pi(nm)}{\pi(I)}]^k$.*

PROOF We saw in the proof of Theorem 4.4.3 that if p is a prime that allows $H_p(P) = H_p(T_r)$ at some position r where P does not occur, then p is in a set of at most $\pi(nm)$ integers. When k fingerprints are used, a false match can occur only if each of the k primes is in that set, and since the primes are chosen randomly (independently), the bound from Theorem 4.4.3 holds for each of the primes. So the probability that all the primes are in the set is bounded by $[\frac{\pi(nm)}{\pi(I)}]^k$, and the theorem is proved. □

As an example, if $k = 4$ and n, m, and I are as in the previous example, then the probability of a false match between P and T is at most by 10^{-12}. Thus, the probability of a false match is reduced dramatically, from 10^{-3} to 10^{-12}, while the computational effort of using four primes only increases by four times. For typical values of n and m, a small choice of k will assure that the probability of an error due to a false match is less than the probability of error due to a hardware malfunction.

Even lower limits on error

The analysis in the proof of Theorem 4.4.4 is again very weak, because it just multiplies the probability that each of the k primes allows a false match *somewhere* in T. However, for the algorithm to actually make an error at some specific position r, each of the primes must *simultaneously* allow a false match at the same r. This is an even less likely event. With this observation we can reduce the probability of a false match as follows:

Theorem 4.4.5. *When k primes are chosen randomly between 1 and I and k fingerprints are used, the probability of a false match between P and T is at most $m[\frac{\pi(n)}{\pi(I)}]^k$.*

PROOF Suppose that a false match occurs at some fixed position r. That means that each prime p_i must evenly divide $|H(P) - H(T_r)|$. Since $|H(P) - H(T_r)| \leq 2^n$, there are at most $\pi(n)$ primes that divide it. So each p_i was chosen randomly from a set of $\pi(I)$ primes and by chance is part of a subset of $\pi(n)$ primes. The probability of this happening at that fixed r is therefore $[\frac{\pi(n)}{\pi(I)}]^k$. Since there are m possible choices for r, the probability of a false match between P and T (i.e., the probability that there is such an r) is at most $m[\frac{\pi(n)}{\pi(I)}]^k$, and the theorem is proved. □

Assuming, as before, that $I = nm^2$, a little arithmetic (which we leave to the reader) shows

Corollary 4.4.3. When k primes are chosen randomly and used in the fingerprint algorithm, the probability of a false match between P and T is at most $(1.26)^k m^{-(2k-1)}(1 + 0.6 \ln m)^k$.

Applying this to the running example of $n = 250$, $m = 4000$, and $k = 4$ reduces the probability of a false match to at most 2×10^{-22}.

We mention one further refinement discussed in [266]. Returning to the case where only a single prime is used, suppose the algorithm explicitly checks that P occurs in T when $H_p(P) = H_p(T_r)$, and it finds that P does not occur there. Then one may be better off by picking a new prime to use for the continuation of the computation. This makes intuitive sense. Theorem 4.4.3 randomizes over the choice of primes and bounds the probability that a randomly picked prime will allow a false match anywhere in T. But once the prime has been shown to allow a false match, it is no longer random. It may well be a prime that allows numerous false matches (a *demon seed*). Theorem 4.4.3 says nothing about how bad a particular prime can be. But by picking a new prime after each error is detected, we can apply Corollary 4.4.2 to each prime, establishing

Theorem 4.4.6. *If a new prime is randomly chosen after the detection of an error, then for any pattern and text the probability of t errors is at most $(\frac{2.53}{m})^t$.*

This probability falls so rapidly that one is effectively protected against a long series of errors on any particular problem instance. For additional probabilistic analysis of the Karp–Rabin method, see [182].

Checking for error in linear time

All the variants of the Karp–Rabin method presented above have the property that they find *all true* occurrences of P in T, but they may also find *false matches* – locations where P is declared to be in T, even though it is not there. If one checks for P at each declared location, this checking would seem to require $\Theta(nm)$ worst-case time, although the expected time can be made smaller. We present here an $O(m)$-time method, noted first by S. Muthukrishnan [336], that determines if any of the declared locations are false matches. That is, the method either verifies that the Karp–Rabin algorithm has found no false matches or it declares that there is at least one false match (but it may not be able to find all the false matches) in $O(m)$ time.

The method is related to Galil's extension of the Boyer–Moore algorithm (Section 3.2.2), but the reader need not have read that section. Consider a list $\mathcal{L}$ of (starting) locations in T where the Karp–Rabin algorithm declares P to be found. A *run* is a maximal interval of consecutive starting locations $l_1, l_2, \ldots, l_r$ in $\mathcal{L}$ such that every two successive numbers in the interval differ by at most $n/2$ (i.e., $l_{i+1} - l_i \leq n/2$). The method works on each run separately, so we first discuss how to check for false matches in a single run.

In a single run, the method explicitly checks for the occurrence of P at the first two positions in the run, l_1 and l_2. If P does not occur in both of those locations then the method has found a false match and stops. Otherwise, when P does occur at both l_1 and l_2, the method learns that P is semiperiodic with period $l_2 - l_1$ (see Lemma 3.2.3). We use d to refer to $l_2 - l_1$, and we show that d is the smallest period of P. If d is not the smallest period, then d must be a multiple of the smallest period, say d'. (This follows easily from the GCD Theorem, which is stated in Section 16.17.1.) (page 431). But that implies that there is an occurrence of P starting at position $l_1 + d' < d_2$, and since the Karp–Rabin method *never* misses any occurrence of P, that contradicts the choice of l_2 as the second occurrence of P in the interval between l_1 and l_r. So d must be the smallest period of P, and it follows that if there are no false matches in the run, then $l_{i+1} - l_i = d$ for each i in the run. Hence, as a first check, the method verifies that $l_{i+1} - l_i = d$ for each i; it declares a false match and stops if this check fails for some i. Otherwise, as in the Galil method, to check each location in $\mathcal{L}$, it suffices to *successively* check the last d characters in each declared occurrence of P against the last d characters of P. That is, for position l_i, the method checks the d characters of T starting at position $l_i + n - d$. If any of these successive checks finds a mismatch, then the method has found a false match in the run and stops. Otherwise, P does in fact occur starting at each declared location in the run.

For the time analysis, note first that no character of T is examined more than twice during a check of a single run. Moreover, since two runs are separated by at least $n/2$ positions and each run is at least n positions long, no character of T can be examined in

more than two consecutive runs. It follows that the total time for the method, over all runs, is $O(m)$.

With the ability to check for false matches in $O(m)$ time, the Karp–Rabin algorithm can be converted from a method with a small probability of error that runs in $O(m)$ worst-case time, to one that makes no error, but runs in $O(m)$ *expected* time (a conversion from a Monte Carlo algorithm to a Las Vegas algorithm). To achieve this, simply (re)run and (re)check the Karp–Rabin algorithm until no false matches are detected. We leave the details as an exercise.

4.4.3. Why fingerprints?

The Karp–Rabin fingerprint method runs in linear worst-case time, but with a nonzero (though extremely small) chance of error. Alternatively, it can be thought of as a method that never makes an error and whose expected running time is linear. In contrast, we have seen several methods that run in linear worst-case time and never make errors. So what is the point of studying the Karp–Rabin method?

There are three responses to this question. First, from a practical standpoint, the method is simple and can be extended to other problems, such as two-dimensional pattern matching with odd pattern shapes – a problem that is more difficult for other methods. Second, the method is accompanied by concrete proofs, establishing significant properties of the method's performance. Methods similar in spirit to fingerprints (or filters) predate the Karp–Rabin method, but, unlike the Karp–Rabin method, they generally lack any theoretical analysis. Little has been proven about their performance. But the main attraction is that the method is based on very different *ideas* than the linear-time methods that guarantee no error. Thus the method is included because a central goal of this book is to present a diverse collection of ideas used in a range of techniques, algorithms, and proofs.

4.5. Exercises

1. Evaluate empirically the *Shift-And* method against methods discussed earlier. Vary the sizes of P and T.
2. Extend the *agrep* method to solve the problem of finding an "occurrence" of a pattern P inside a text T, when a small number of insertions and deletions of characters, as well as mismatches, are allowed. That is, characters can be inserted into P and characters can be deleted from P.
3. Adapt *Shift-And* and *agrep* to handle a set of patterns. Can you do better than just handling each pattern in the set independently?
4. Prove the correctness of the *agrep* method.
5. Show how to efficiently handle wild cards (both in the pattern and the text) in the *Shift-And* approach. Do the same for *agrep*. Show that the efficiency of neither method is affected by the number of wild cards in the strings.
6. Extend the *Shift-And* method to efficiently handle regular expressions that do not use the Kleene closure. Do the same for *agrep*. Explain the utility of these extensions to collections of biosequence patterns such as those in PROSITE.
7. We mentioned in Exercise 32 of Chapter 3 that PROSITE patterns often specify a range for the number of times that a subpattern repeats. Ranges of this type can be easily handled by the $O(nm)$ regular expression pattern matching method of Section 3.6. Can such range specifications be efficiently handled with the *Shift-And* method or *agrep*? The answer partly depends on the number of such specifications that appear in the expression.
8. (Open problem) Devise a purely comparison-based method to compute match-counts in $O(m \log m)$ time. Perhaps one can examine the FFT method in detail to see if complex arithmetic can be replaced with character comparisons in the case of computing match-counts.
9. Complete the proof of Corollary 4.4.3.
10. The random fingerprint approach can be extended to the two-dimensional pattern matching problem discussed in Section 3.5.3. Do it.
11. Complete the details and analysis to convert the Karp–Rabin method from a Monte-Carlo–style randomized algorithm to a Las Vegas–style randomized algorithm.
12. There are improvements possible in the method to check for false matches in the Karp–Rabin method. For example, the method can find in $O(m)$ time all those runs containing no false matches. Explain how. Also, at some point, the method needs to explicitly check for P at only l_1 and not l_2. Explain when and why.

PART II

Suffix Trees and Their Uses

5

Introduction to Suffix Trees

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

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

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

Often the text is a fixed *set* of strings, for example, a collection of STSs or ESTs (see Sections 3.5.1 and 7.10), so that the substring problem is to determine whether the input string is a substring of any of the fixed strings. Suffix trees work nicely to efficiently solve this problem as well. Superficially, this case of multiple text strings resembles the *dictionary* problem discussed in the context of the Aho–Corasick algorithm. Thus it is natural to expect that the Aho–Corasick algorithm could be applied. However, the Aho–Corasick method does not solve the substring problem in the desired time bounds, because it will only determine if the new string is a *full* string in the dictionary, not whether it is a substring of a string in the dictionary.

After presenting the algorithms, several applications and extensions will be discussed in Chapter 7. Then a remarkable result, *the constant-time least common ancestor method*, will be presented in Chapter 8. That method greatly amplifies the utility of suffix trees, as will be illustrated by additional applications in Chapter 9. Some of those applications provide a bridge to inexact matching; more applications of suffix trees will be discussed in Part III, where the focus is on inexact matching.

5.1. A short history

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

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

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

5.2. Basic definitions

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

Definition A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i..m]$.

For example, the suffix tree for the string $xabxac$ is shown in Figure 5.1. The path from the root to the leaf numbered 1 spells out the full string $S = xabxac$, while the path to the leaf numbered 5 spells out the suffix ac, which starts in position 5 of S.

As stated above, the definition of a suffix tree for S does not guarantee that a suffix tree for any string S actually exists. The problem is that if one *suffix* of S matches a *prefix* of another suffix of S then no suffix tree obeying the above definition is possible, since the path for the first suffix would not end at a leaf. For example, if the last character of $xabxac$ is removed, creating string $xabxa$, then suffix xa is a prefix of suffix $xabxa$, so the path spelling out xa would not end at a leaf.

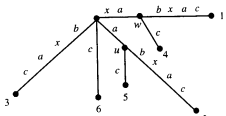


Figure 5.1: Suffix tree for string $xabxac$. The node labels u and w on the two interior nodes will be used later.

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

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

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

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

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

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

5.3. A motivating example

Before diving into the details of the methods to construct suffix trees, let's look at how a suffix tree for a string is used to solve the exact match problem: Given a pattern P of

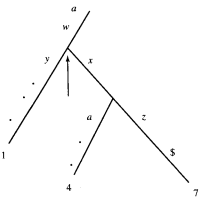


Figure 5.2: Three occurrences of aw in $awyawxawxz$. Their starting positions number the leaves in the subtree of the node with path-label aw.

length n and a text T of length m, find all occurrences of P in T in $O(n+m)$ time. We have already seen several solutions to this problem. Suffix trees provide another approach:

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

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

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

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

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

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

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

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

5.4. A naive algorithm to build a suffix tree

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

In detail, tree N_1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string S\$. Tree N_{i+1} is constructed from N_i as follows: Starting at the root of N_i find the longest path from the root whose label matches a prefix of $S[i+1..m]$\$. This path is found by successively comparing and matching characters in suffix $S[i+1..m]$\$ to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of S\$ is a prefix of any other suffix of S\$. When that point is reached, the algorithm is either at a node, w say, or it is in the middle of an edge. If it is in the middle of an edge, (u, v) say, then it breaks edge (u, v) into two edges by inserting a new node, called w, just after the last character on the edge that matched a character in $S[i+1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labeled with the part of the (u, v) label that matched with $S[i+1..m]$, and the new edge (w, v) is labeled with the remaining part of the (u, v) label. Then (whether a new node w was created or whether one already existed at the point where the match ended), the algorithm creates a new edge $(w, i+1)$ running from w to a new leaf labeled $i+1$, and it labels the new edge with the unmatched part of suffix $S[i+1..m]$\$.

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

Assuming, as usual, a bounded-size alphabet, the above naive method takes $O(m^2)$ time to build a suffix tree for the string S of length m.

6

Linear-Time Construction of Suffix Trees

We will present two methods for constructing suffix trees in detail, Ukkonen's method and Weiner's method. Weiner was the first to show that suffix trees can be built in linear time, and his method is presented both for its historical importance and for some different technical ideas that it contains. However, Ukkonen's method is equally fast and uses far less space (i.e., memory) in practice than Weiner's method. Hence Ukkonen is the method of choice for most problems requiring the construction of a suffix tree. We also believe that Ukkonen's method is easier to understand. Therefore, it will be presented first. A reader who wishes to study only one method is advised to concentrate on it. However, our development of Weiner's method does not depend on understanding Ukkonen's algorithm, and the two algorithms can be read independently (with one small shared section noted in the description of Weiner's method).

6.1. Ukkonen's linear-time suffix tree algorithm

Esko Ukkonen [438] devised a linear-time algorithm for constructing a suffix tree that may be the conceptually easiest linear-time construction algorithm. This algorithm has a space-saving improvement over Weiner's algorithm (which was achieved first in the development of McCreight's algorithm), and it has a certain "on-line" property that may be useful in some situations. We will describe that on-line property but emphasize that the main virtue of Ukkonen's algorithm is the simplicity of its description, proof, and time analysis. The simplicity comes because the algorithm can be developed as a simple but inefficient method, followed by "common-sense" implementation tricks that establish a better worst-case running time. We believe that this less direct exposition is more understandable, as each step is simple to grasp.

6.1.1. Implicit suffix trees

Ukkonen's algorithm constructs a sequence of *implicit* suffix trees, the last of which is converted to a true suffix tree of the string S.

Definition An *implicit suffix tree* for string S is a tree obtained from the suffix tree for $S\$$ by removing every copy of the terminal symbol \$ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

An implicit suffix tree for a prefix $S[1..i]$ of S is similarly defined by taking the suffix tree for $S[1..i]\$$ and deleting \$ symbols, edges, and nodes as above.

Definition We denote the implicit suffix tree of the string $S[1..i]$ by $\mathcal{I}_i$, for i from 1 to m.

The implicit suffix tree for any string S will have fewer leaves than the suffix tree for

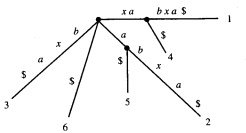


Figure 6.1: Suffix tree for string *xabxa* \$.

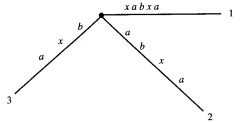


Figure 6.2: Implicit suffix tree for string *xabxa*.

string $S\$$ if and only if at least one of the suffixes of S is a prefix of another suffix. The terminal symbol \$ was added to the end of S precisely to avoid this situation. However, if S ends with a character that appears nowhere else in S, then the implicit suffix tree of S will have a leaf for each suffix and will hence be a true suffix tree.

As an example, consider the suffix tree for string $xabxa\$$ shown in Figure 6.1. Suffix xa is a prefix of suffix $xabxa$, and similarly the string a is a prefix of $abxa$. Therefore, in the suffix tree for $xabxa$ the edges leading to leaves 4 and 5 are labeled only with \$. Removing these edges creates two nodes with only one child each, and these are then removed as well. The resulting implicit suffix tree for $xabxa$ is shown in Figure 6.2. As another example, Figure 5.1 on page 91 shows a tree built for the string $xabxac$. Since character c appears only at the end of the string, the tree in that figure is both a suffix tree and an implicit suffix tree for the string.

Even though an implicit suffix tree may not have a leaf for each suffix, it does encode all the suffixes of S – each suffix is spelled out by the characters on some path from the root of the implicit suffix tree. However, if the path does not end at a leaf, there will be no marker to indicate the path's end. Thus implicit suffix trees, on their own, are somewhat less informative than true suffix trees. We will use them just as a tool in Ukkonen's algorithm to finally obtain the true suffix tree for S.

6.1.2. Ukkonen's algorithm at a high level

Ukkonen's algorithm constructs an implicit suffix tree $\mathcal{I}_i$ for each prefix $S[1..i]$ of S, starting from $\mathcal{I}_1$ and incrementing i by one until $\mathcal{I}_m$ is built. The true suffix tree for S is constructed from $\mathcal{I}_m$, and the time for the entire algorithm is $O(m)$. We will explain

Ukkonen's algorithm by first presenting an $O(m^3)$-time method to build all trees $\mathcal{I}_i$ and then optimizing its implementation to obtain the claimed time bound.

High-level description of Ukkonen's algorithm

Ukkonen's algorithm is divided into m *phases*. In phase $i+1$, tree $\mathcal{I}_{i+1}$ is constructed from $\mathcal{I}_i$. Each phase $i+1$ is further divided into $i+1$ *extensions*, one for each of the $i+1$ suffixes of $S[1..i+1]$. In *extension* j of phase $i+1$, the algorithm first finds the end of the path from the root labeled with substring $S[j..i]$. It then extends the substring by adding the character $S(i+1)$ to its end, unless $S(i+1)$ already appears there. So in phase $i+1$, string $S[1..i+1]$ is first put in the tree, followed by strings $S[2..i+1]$, $S[3..i+1]$, ... (in extensions 1,2, 3, ..., respectively). Extension $i+1$ of phase $i+1$ extends the *empty* suffix of $S[1..i]$, that is, it puts the single character string $S(i+1)$ into the tree (unless it is already there). Tree $\mathcal{I}_1$ is just the single edge labeled by character $S(1)$. Procedurally, the algorithm is as follows:

High-level Ukkonen algorithm

```
Construct tree I_1.
For i from 1 to m − 1 do
begin {phase i + 1}
    For j from 1 to i + 1
        begin {extension j}
        Find the end of the path from the root labeled S[j..i] in the
        current tree. If needed, extend that path by adding character S(i + 1),
        thus assuring that string S[j..i + 1] is in the tree.
        end;
end;
```

Suffix extension rules

To turn this high-level description into an algorithm, we must specify exactly how to perform a *suffix extension*. Let $S[j..i] = \beta$ be a suffix of $S[1..i]$. In extension j, when the algorithm finds the end of β in the current tree, it extends β to be sure the suffix $\beta S(i+1)$ is in the tree. It does this according to one of the following three rules:

Rule 1 In the current tree, path β ends at a leaf. That is, the path from the root labeled β extends to the end of some leaf edge. To update the tree, character $S(i+1)$ is added to the end of the label on that leaf edge.

Rule 2 No path from the end of string β starts with character $S(i+1)$, but at least one labeled path continues from the end of β.

In this case, a new leaf edge starting from the end of β must be created and labeled with character $S(i+1)$. A new node will also have to be created there if β ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

Rule 3 Some path from the end of string β starts with character $S(i+1)$. In this case the string $\beta S(i+1)$ is already in the current tree, so (remembering that in an implicit suffix tree the end of a suffix need not be explicitly marked) we do nothing.

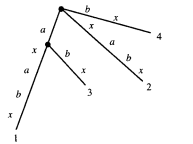


Figure 6.3: Implicit suffix tree for string $axabx$ before the sixth character, b, is added.

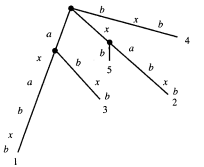


Figure 6.4: Extended implicit suffix tree after the addition of character b.

As an example, consider the implicit suffix tree for $S = axabx$ shown in Figure 6.3. The first four suffixes end at leaves, but the single character suffix x ends inside an edge. When a sixth character b is added to the string, the first four suffixes get extended by applications of Rule 1, the fifth suffix gets extended by rule 2, and the sixth by rule 3. The result is shown in Figure 6.4.

6.1.3. Implementation and speedup

Using the suffix extension rules given above, once the end of a suffix β of $S[1..i]$ has been found in the current tree, only constant time is needed to execute the extension rules (to ensure that suffix $\beta S(i+1)$ is in the tree). The key issue in implementing Ukkonen's algorithm then is how to locate the ends of all the $i+1$ suffixes of $S[1..i]$.

Naively we could find the end of any suffix β in $O(|\beta|)$ time by walking from the root of the current tree. By that approach, extension j of phase $i+1$ would take $O(i+1-j)$ time, $\mathcal{I}_{i+1}$ could be created from $\mathcal{I}_i$ in $O(i^2)$ time, and $\mathcal{I}_m$ could be created in $O(m^3)$ time. This algorithm may seem rather foolish since we already know a straightforward algorithm to build a suffix tree in $O(m^2)$ time (and another is discussed in the exercises), but it is easier to describe Ukkonen's $O(m)$ algorithm as a speedup of the $O(m^3)$ method above.

We will reduce the above $O(m^3)$ time bound to $O(m)$ time with a few observations and implementation tricks. Each trick by itself looks like a sensible heuristic to accelerate the algorithm, but acting individually these tricks do not necessarily reduce the worst-case

bound. However, taken together, they do achieve a linear worst-case time. The most important element of the acceleration is the use of *suffix links*.

Suffix links: first implementation speedup

Definition Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring. For an internal node v with path-label $x\alpha$, *if* there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a *suffix link*.

We will sometimes refer to a suffix link from v to $s(v)$ as the pair $(v, s(v))$. For example, in Figure 6.1 (on page 95) let v be the node with path-label xa and let $s(v)$ be the node whose path-label is the single character a. Then there exists a suffix link from node v to node $s(v)$. In this case, α is just a single character long.

As a special case, if α is empty, then the suffix link from an internal node with path-label $x\alpha$ goes to the root node. The root node itself is not considered internal and has no suffix link from it.

Although definition of suffix links does not imply that every internal node of an implicit suffix tree has a suffix link from it, it will, in fact, have one. We actually establish something stronger in the following lemmas and corollaries.

Lemma 6.1.1. *If a new internal node v with path-label $x\alpha$ is added to the current tree in extension j of some phase $i+1$, then either the path labeled α already ends at an internal node of the current tree or an internal node at the end of string α will be created (by the extension rules) in extension $j+1$ in the same phase $i+1$.*

PROOF A new internal node v is created in extension j (of phase $i+1$) only when extension rule 2 applies. That means that in extension j, the path labeled $x\alpha$ continued with some character other than $S(i+1)$, say c. Thus, in extension $j+1$, there is a path labeled α in the tree and it certainly has a continuation with character c (although possibly with other characters as well). There are then two cases to consider: Either the path labeled α continues only with character c or it continues with some additional character. When α is continued only by c, extension rule 2 will create a node $s(v)$ at the end of path α. When α is continued with two different characters, then there must already be a node $s(v)$ at the end of path α. The Lemma is proved in either case. □

Corollary 6.1.1. In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

PROOF The proof is inductive and is true for tree $\mathcal{I}_1$ since $\mathcal{I}_1$ contains no internal nodes. Suppose the claim is true through the end of phase i, and consider a single phase $i+1$. By Lemma 6.1.1, when a new node v is created in extension j, the correct node $s(v)$ ending the suffix link from v will be found or created in extension $j+1$. No new internal node gets created in the last extension of a phase (the extension handling the single character suffix $S(i+1)$), so all suffix links from internal nodes created in phase $i+1$ are known by the end of the phase and tree $\mathcal{I}_{i+1}$ has all its suffix links. □

Corollary 6.1.1 is similar to Theorem 6.2.5, which will be discussed during the treatment of Weiner's algorithm, and states an important fact about implicit suffix trees and ultimately about suffix trees. For emphasis, we restate the corollary in slightly different language.

Corollary 6.1.2. In any implicit suffix tree $\mathcal{I}_i$, if internal node v has path-label $x\alpha$, then *there is* a node $s(v)$ of $\mathcal{I}_i$ with path-label α.

Following Corollary 6.1.1, all internal nodes in the changing tree will have suffix links from them, except for the most recently added internal node, which will receive its suffix link by the end of the next extension. We now show how suffix links are used to speed up the implementation.

Following a trail of suffix links to build $\mathcal{I}_{i+1}$

Recall that in phase $i+1$ the algorithm locates suffix $S[j..i]$ of $S[1..i]$ in extension j, for j increasing from 1 to $i+1$. Naively, this is accomplished by matching the string $S[j..i]$ along a path from the root in the current tree. Suffix links can shortcut this walk and each extension. The first two extensions (for $j=1$ and $j=2$) in any phase $i+1$ are the easiest to describe.

The end of the full string $S[1..i]$ must end at a leaf of $\mathcal{I}_i$ since $S[1..i]$ is the longest string represented in that tree. That makes it easy to find the end of that suffix (as the trees are constructed, we can keep a pointer to the leaf corresponding to the current full string $S[1..i]$), and its suffix extension is handled by Rule 1 of the extension rules. So the first extension of any phase is special and only takes constant time since the algorithm has a pointer to the end of the current full string.

Let string $S[1..i]$ be $x\alpha$, where x is a single character and α is a (possibly empty) substring, and let $(v, 1)$ be the tree-edge that enters leaf 1. The algorithm next must find the end of string $S[2..i] = \alpha$ in the current tree derived from $\mathcal{I}_i$. The key is that node v is either the root or it is an interior node of $\mathcal{I}_i$. If it is the root, then to find the end of α the algorithm just walks down the tree following the path labeled α as in the naive algorithm. But if v is an internal node, then by Corollary 6.1.2 (since v was in $\mathcal{I}_i$) v has a suffix link out of it to node $s(v)$. Further, since $s(v)$ has a path-label that is a prefix of string α, the end of string α must end in the subtree of $s(v)$. Consequently, in searching for the end of α in the current tree, the algorithm need not walk down the entire path from the root, but can instead begin the walk from node $s(v)$. That is the main point of including suffix links in the algorithm.

To describe the second extension in more detail, let γ denote the edge-label on edge $(v, 1)$. To find the end of α, walk up from leaf 1 to node v; follow the suffix link from v to $s(v)$; and walk from $s(v)$ down the path (which may be more than a single edge) labeled γ. The end of that path is the end of α (see Figure 6.5). At the end of path α, the tree is updated following the suffix extension rules. This completely describes the first two extensions of phase $i+1$.

To extend any string $S[j..i]$ to $S[j..i+1]$ for $j > 2$, repeat the same general idea: Starting at the end of string $S[j-1..i]$ in the current tree, walk up at most one node to either the root or to a node v that has a suffix link from it; let γ be the edge-label of that edge; assuming v is not the root, traverse the suffix link from v to $s(v)$; then walk down the tree from $s(v)$, following a path labeled γ to the end of $S[j..i]$; finally, extend the suffix to $S[j..i+1]$ according to the extension rules.

There is one minor difference between extensions for $j > 2$ and the first two extensions. In general, the end of $S[j-1..i]$ may be at a node that itself has a suffix link from it, in which case the algorithm traverses that suffix link. Note that even when extension rule 2 applies in extension $j-1$ (so that the end of $S[j-1..i]$ is at a newly created internal node w), if the parent of w is not the root, then the parent of w already has a suffix link out of it, as guaranteed by Lemma 6.1.1. Thus in extension j the algorithm never walks up more than one edge.

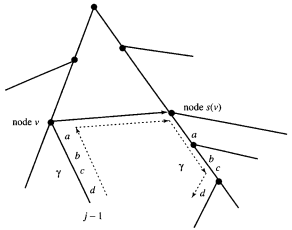


Figure 6.5: Extension $j > 1$ in phase $i + 1$. Walk up atmost one edge (labeled γ) from the end of the path labeled $S[j-1..i]$ to node v; then follow the suffix link to $s(v)$; then walk down the path specifying substring γ; then apply the appropriate extension rule to insert suffix $S[j..i+1]$.

Single extension algorithm: SEA

Putting these pieces together, when implemented using suffix links, extension $j \geq 2$ of phase $i + 1$ is:

Single extension algorithm

Begin

1. Find the first node v at or above the end of $S[j-1..i]$ that either has a suffix link from it or is the root. This requires walking up at most one edge from the end of $S[j-1..i]$ in the current tree. Let γ (possibly empty) denote the string between v and the end of $S[j-1..i]$.
2. If v is not the root, traverse the suffix link from v to node $s(v)$ and then walk down from $s(v)$ following the path for string γ. If v is the root, then follow the path for $S[j..i]$ from the root (as in the naive algorithm).
3. Using the extension rules, ensure that the string $S[j..i]S(i+1)$ is in the tree.
4. If a new internal node w was created in extension $j - 1$ (by extension rule 2), then by Lemma 6.1.1, string α must end at node $s(w)$, the end node for the suffix link from w. Create the suffix link $(w, s(w))$ from w to $s(w)$.

End.

Assuming the algorithm keeps a pointer to the current full string $S[1..i]$, the first extension of phase $i + 1$ need not do any up or down walking. Furthermore, the first extension of phase $i + 1$ always applies suffix extension rule 1.

What has been achieved so far?

The use of suffix links is clearly a practical improvement over walking from the root in each extension, as done in the naive algorithm. But does their use improve the worst-case running time?

The answer is that as described, the use of suffix links does not yet improve the time bound. However, here we introduce a trick that will reduce the worst-case time for the algorithm to $O(m^2)$. This trick will also be central in other algorithms to build and use suffix trees.

Trick number 1: skip/count trick

In Step 2 of extension $j + 1$ the algorithm walks *down* from node $s(v)$ along a path labeled γ. Recall that there surely must be such a γ path from $s(v)$. Directly implemented, this walk along γ takes time proportional to $|\gamma|$, the *number of characters* on that path. But a simple trick, called the *skip/count trick*, will reduce the traversal time to something proportional to the *number of nodes* on the path. It will then follow that the time for all the down walks in a phase is at most $O(m)$.

Trick 1 Let g denote the length of γ, and recall that no two labels of edges out of $s(v)$ can start with the same character, so the first character of γ must appear as the first character on exactly one edge out of $s(v)$. Let g' denote the number of characters on that edge. If g' is less than g, then the algorithm does not need to look at any more of the characters on that edge; it simply skips to the node at the end of the edge. There it sets g to $g - g'$, sets a variable h to $g' + 1$, and looks over the outgoing edges to find the correct next edge (whose first character matches character h of γ). In general, when the algorithm identifies the next edge on the path it compares the current value of g to the number of characters g' on that edge. When g is at least as large as g', the algorithm skips to the node at the end of the edge, sets g to $g - g'$, sets h to $h + g'$, and finds the edge whose first character is character h of γ and repeats. When an edge is reached where g is smaller than or equal to g', then the algorithm skips to character g on the edge and quits, assured that the γ path from $s(v)$ ends on that edge exactly g characters down its label. (See Figure 6.6).

Assuming simple and obvious implementation details (such as knowing the number of characters on each edge, and being able, in constant time, to extract from S the character at any given position) the effect of using the skip/count trick is to move from one node to the next node on the γ path in *constant* time.[1] The total time to traverse the path is then proportional to the number of *nodes* on it rather than the number of characters on it.

This is a useful heuristic, but what does it buy in terms of worst-case bounds? The next lemma leads immediately to the answer.

Definition Define the *node-depth* of a node u to be the number of *nodes* on the path from the root to u.

Lemma 6.1.2. *Let $(v, s(v))$ be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of v is at most one greater than the node depth of $s(v)$.*

PROOF When edge $(v, s(v))$ is traversed, any internal ancestor of v, which has path-label $x\beta$ say, has a suffix link to a node with path-label β. But $x\beta$ is a prefix of the path to v, so β is a prefix of the path to $s(v)$ and it follows that the suffix link from any internal ancestor of v goes to an ancestor of $s(v)$. Moreover, if β is nonempty then the node labeled by β is an internal node. And, because the node-depths of any two ancestors of v must differ, each ancestor of v has a suffix link to a distinct ancestor of $s(v)$. It follows that the node-depth of $s(v)$ is at least one (for the root) plus the number of internal ancestors of v who have path-labels more than one character long. The only extra ancestor that v can have (without a corresponding ancestor for $s(v)$) is an internal ancestor whose path-label

[1] Again, we are assuming a constant-sized alphabet.

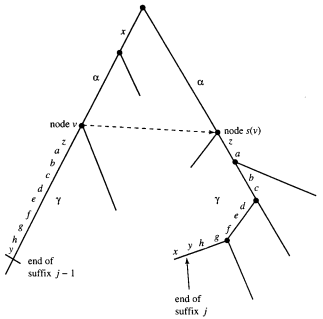


Figure 6.6: The skip/count trick. In phase $i + 1$, substring γ has length ten. There is a copy of substring γ out of node $s(v)$; it is found three characters down the last edge, after four node skips are executed.

has length one (it has label x). Therefore, v can have node-depth at most one more than $s(v)$. (See Figure 6.7). □

Definition As the algorithm proceeds, the *current node-depth* of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 6.1.1. *Using the skip/count trick, any phase of Ukkonen's algorithm takes $O(m)$ time.*

PROOF There are $i + 1 \leq m$ extensions in phase i. In a single extension the algorithm walks up at most one edge to find a node with a suffix link, traverses one suffix link, walks down some number of nodes, applies the suffix extension rules, and maybe adds a suffix link. We have already established that all the operations other than the down-walking take constant time per extension, so we only need to analyze the time for the down-walks. We do this by examining how the current node-depth can change over the phase.

The up-walk in any extension decreases the current node-depth by at most one (since it moves up at most one node), each suffix link traversal decreases the node-depth by at most another one (by Lemma 6.1.2), and each edge traversed in a down-walk moves to a node of greater node-depth. Thus over the entire phase the current node-depth is decremented at most $2m$ times, and since no node can have depth greater than m, the total possible increment to current node-depth is bounded by $3m$ over the entire phase. It follows that over the entire phase, the total number of edge traversals during down-walks is bounded by $3m$. Using the skip/count trick, the time per down-edge traversal is constant, so the total time in a phase for all the down-walking is $O(m)$, and the theorem is proved. □

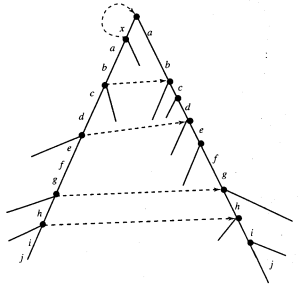


Figure 6.7: For every node v on the path $x\alpha$, the corresponding node $s(v)$ is on the path α. However, the node-depth of $s(v)$ can be one less than the node-depth of v, it can be equal, or it can be greater. For example, the node labeled *xab* has node-depth two, whereas the node-depth of *ab* is one. The node-depth of the node labeled *xabcdefg* is four, whereas the node-depth of *abcdefg* is five.

There are m phases, so the following is immediate:

Corollary 6.1.3. Ukkonen's algorithm can be implemented with suffix links to run in $O(m^2)$ time.

Note that the $O(m^2)$ time bound for the algorithm was obtained by multiplying the $O(m)$ time bound on a single phase by m (since there are m phases). This crude multiplication was necessary because the time analysis was directed to only a single phase. What is needed are some changes to the implementation allowing a time analysis that crosses phase boundaries. That will be done shortly.

At this point the reader may be a bit weary because we seem to have made no progress, since we started with a naive $O(m^2)$ method. Why all the work just to come back to the same time bound? The answer is that although we have made no progress on the time bound, we have made great conceptual progress so that with only a few more easy details, the time will fall to $O(m)$. In particular, we will need one simple implementation detail and two more little tricks.

6.1.4. A simple implementation detail

We next establish an $O(m)$ time bound for building a suffix tree. There is, however, one immediate barrier to that goal: The suffix tree may require $\Theta(m^2)$ space. As described so far, the edge-labels of a suffix tree might contain more than $\Theta(m)$ characters in total. Since the time for the algorithm is at least as large as the size of its output, that many characters makes an $O(m)$ time bound impossible. Consider the string $S = abcdefghijklmnopqrstuvwxyz$. Every suffix begins with a distinct character; hence there are 26 edges out of the root and

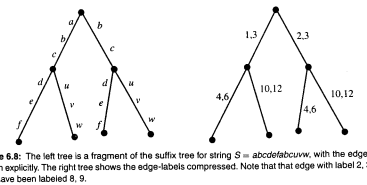


Figure 6.8: The left tree is a fragment of the suffix tree for string $S = abcdefabcuvw$, with the edge-labels written explicitly. The right tree shows the edge-labels compressed. Note that that edge with label 2, 3 could also have been labeled 8, 9.

each is labeled with a complete suffix, requiring $26 \times 27/2$ characters in all. For strings longer than the alphabet size, some characters will repeat, but still one can construct strings of arbitrary length m so that the resulting edge-labels have more than $\Theta(m)$ characters in total. Thus, an $O(m)$-time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.

Edge-label compression

A simple, alternate scheme exists for edge labeling. Instead of explicitly writing a substring on an edge of the tree, only write a *pair of indices* on the edge, specifying beginning and end positions of that substring in S (see Figure 6.8). Since the algorithm has a copy of string S, it can locate any particular character in S in constant time given its position in the string. Therefore, we may describe any particular suffix tree algorithm as if edge-labels were explicit, and yet implement that algorithm with only a constant number of symbols written on any edge (the index pair indicating the beginning and ending positions of a substring).

For example, in Ukkonen's algorithm when matching along an edge, the algorithm uses the index pair written on an edge to retrieve the needed characters from S and then performs the comparisons on those characters. The extension rules are also easily implemented with this labeling scheme. When extension rule 2 applies in a phase $i+1$, label the newly created edge with the index pair $(i+1, i+1)$, and when extension rule 1 applies (on a leaf edge), change the index pair on that leaf edge from (p, q) to $(p, q+1)$. It is easy to see inductively that q had to be i and hence the new label $(p, i+1)$ represents the correct new substring for that leaf edge.

By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most $2m - 1$, the suffix tree uses only $O(m)$ symbols and requires only $O(m)$ space. This makes it more plausible that the tree can actually be built in $O(m)$ time.[2] Although the fully implemented algorithm will not explicitly write a substring on an edge, we will still find it convenient to talk about "the substring or label on an edge or path" as if the explicit substring was written there.

[2] We make the standard RAM model assumption that a number with up to $\log m$ bits can be read, written, or compared in constant time.

6.1.5. Two more little tricks and we're done

We present two more implementation tricks that come from two observations about the way the extension rules interact in successive extensions and phases. These tricks, plus Lemma 6.1.2, will lead immediately to the desired linear time bound.

Observation 1: Rule 3 is a show stopper In any phase, if suffix extension rule 3 applies in extension j, it will also apply in all further extensions ($j+1$ to $i+1$) until the end of the phase. The reason is that when rule 3 applies, the path labeled $S[j..i]$ in the current tree must continue with character $S(i+1)$, and so the path labeled $S[j+1..i]$ does also, and rule 3 again applies in extensions $j+1, j+2, \ldots, i+1$.

When extension rule 3 applies, no work needs to be done since the suffix of interest is already in the tree. Moreover, a new suffix link needs to be added to the tree only after an extension in which extension rule 2 applies. These facts and Observation 1 lead to the following implementation trick.

Trick 2 End any phase $i+1$ the first time that extension rule 3 applies. If this happens in extension j, then there is no need to explicitly find the end of any string $S[k..i]$ for $k > j$.

The extensions in phase $i+1$ that are "done" after the first execution of rule 3 are said to be done *implicitly*. This is in contrast to any extension j where the end of $S[j..i]$ is explicitly found. An extension of that kind is called an *explicit* extension.

Trick 2 is clearly a good heuristic to reduce work, but it's not clear if it leads to a better worst-case time bound. For that we need one more observation and trick.

Observation 2: Once a leaf, always a leaf That is, if at some point in Ukkonen's algorithm a leaf is created and labeled j (for the suffix starting at position j of S), then that leaf will remain a leaf in all successive trees created during the algorithm. This is true because the algorithm has no mechanism for extending a leaf edge beyond its current leaf. In more detail, once there is a leaf labeled j, extension rule 1 will always apply to extension j in any successive phase. So once a leaf, always a leaf.

Now leaf 1 is created in phase 1, so in any phase i there is an initial sequence of consecutive extensions (starting with extension 1) where extension rule 1 or 2 applies. Let j_i denote the last extension in this sequence. Since any application of rule 2 creates a new leaf, it follows from Observation 2 that $j_i \leq j_{i+1}$. That is, the initial sequence of extensions where rule 1 or 2 applies cannot shrink in successive phases. This suggests an implementation trick that in phase $i+1$ avoids all explicit extensions 1 through j_i. Instead, only constant time will be required to do those extensions implicitly.

To describe the trick, recall that the label on any edge in an implicit suffix tree (or a suffix tree) can be represented by two indices p, q specifying the substring $S[p..q]$. Recall also that for any leaf edge of $\mathcal{I}_i$, index q is equal to i and in phase $i+1$ index q gets incremented to $i+1$, reflecting the addition of character $S(i+1)$ to the end of each suffix.

Trick 3 In phase $i+1$, when a leaf edge is first created and would normally be labeled with substring $S[p..i+1]$, instead of writing indices $(p, i+1)$ on the edge, write (p, e), where e is a symbol denoting "the current end". Symbol e is a *global* index that is set to $i+1$ once in each phase. In phase $i+1$, since the algorithm knows that rule 1 will apply in extensions 1 through j_i at least, it need do no additional explicit work to implement

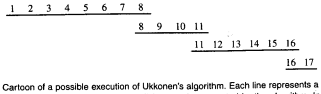


Figure 6.9: Cartoon of a possible execution of Ukkonen's algorithm. Each line represents a phase of the algorithm, and each number represents an explicit extension executed by the algorithm. In this cartoon there are four phases and seventeen explicit extensions. In any two consecutive phases, there is at most one index where the same explicit extension is executed in both phases.

those j_i extensions. Instead, it only does constant work to increment variable e, and then does explicit work for (some) extensions starting with extension $j_i + 1$.

The punch line

With Tricks 2 and 3, explicit extensions in phase $i + 1$ (using algorithm SEA) are then only required from extension $j_i + 1$ until the first extension where rule 3 applies (or until extension $i + 1$ is done). All other extensions (before and after those explicit extensions) are done implicitly. Summarizing this, phase $i + 1$ is implemented as follows:

Single phase algorithm: SPA

Begin

1. Increment index e to $i + 1$. (By Trick 3 this correctly implements all implicit extensions 1 through j_i.)
2. Explicitly compute successive extensions (using algorithm SEA) starting at $j_i + 1$ until reaching the first extension j^* where rule 3 applies or until all extensions are done in this phase. (By Trick 2, this correctly implements all the additional implicit extensions $j^* + 1$ through $i + 1$.)
3. Set j_{i+1} to $j^* - 1$, to prepare for the next phase.

End

Step 3 correctly sets j_{i+1} because the initial sequence of extensions where extension rule 1 or 2 applies must end at the point where rule 3 first applies.

The key feature of algorithm *SPA* is that phase $i + 2$ will *begin* computing explicit extensions with extension j^*, where j^* was the *last* explicit extension computed in phase $i + 1$. Therefore, two consecutive phases share *at most one* index (j^*) where an explicit extension is executed (see Figure 6.9). Moreover, phase $i + 1$ ends knowing where string $S[j^*..i + 1]$ ends, so the repeated extension of j^* in phase $i + 2$ can execute the suffix extension rule for j^* without any up-walking, suffix link traversals, or node skipping. That means the first explicit extension in any phase only takes constant time. It is now easy to prove the main result.

Theorem 6.1.2. *Using suffix links and implementation tricks* 1, 2, *and* 3, *Ukkonen's algorithm builds implicit suffix trees* $\mathcal{I}_1$ *through* $\mathcal{I}_m$ *in* $O(m)$ *total time.*

PROOF The time for all the implicit extensions in any phase is constant and so is $O(m)$ over the entire algorithm.

As the algorithm executes explicit extensions, consider an index $\bar{j}$ corresponding to the explicit extension the algorithm is currently executing. Over the entire execution of the algorithm, $\bar{j}$ never decreases, but it does remain the same between two successive phases.

Since there are only m phases, and $\bar{j}$ is bounded by m, the algorithm therefore executes only $2m$ explicit extensions. As established earlier, the time for an explicit extension is a constant plus some time proportional to the number of node skips it does during the down-walk in that extension.

To bound the total number of node skips done during all the down-walks, we consider (similar to the proof of Theorem 6.1.1) how the current node-depth changes during successive extensions, even extensions in different phases. The key is that the first explicit extension in any phase (after phase 1) begins with extension j^*, which was the last explicit extension in the previous phase. Therefore, the current node-depth does not change between the end of one extension and the beginning of the next. But (as detailed in the proof of Theorem 6.1.1), in each explicit extension the current node-depth is first reduced by at most two (up-walking one edge and traversing one suffix link), and thereafter the down-walk in that extension increases the current node-depth by one at each node skip. Since the maximum node-depth is m, and there are only $2m$ explicit extensions, it follows (as in the proof of Theorem 6.1.1) that the maximum number of node skips done during all the down-walking (and not just in a single phase) is bounded by $O(m)$. All work has been accounted for, and the theorem is proved. □

6.1.6. Creating the true suffix tree

The final implicit suffix tree $\mathcal{I}_m$ can be converted to a true suffix tree in $O(m)$ time. First, add a string terminal symbol \$ to the end of S and let Ukkonen's algorithm continue with this character. The effect is that no suffix is now a prefix of any other suffix, so the execution of Ukkonen's algorithm results in an implicit suffix tree in which each suffix ends at a leaf and so is explicitly represented. The only other change needed is to replace each index e on every leaf edge with the number m. This is achieved by an $O(m)$-time traversal of the tree, visiting each leaf edge. When these modifications have been made, the resulting tree is a true suffix tree.

In summary,

Theorem 6.1.3. *Ukkonen's algorithm builds a true suffix tree for* S, *along with all its suffix links in* $O(m)$ *time.*

6.2. Weiner's linear-time suffix tree algorithm

Unlike Ukkonen's algorithm, Weiner's algorithm starts with the entire string S. However, like Ukkonen's algorithm, it enters one suffix at a time into a growing tree, although in a very different order. In particular, it first enters string $S(m)$\$ into the tree, then string $S[m-1..m]$\$, ..., and finally, it enters the entire string S\$ into the tree.

Definition Suff_i denotes the suffix $S[i..m]$ of S starting in position i.

For example, Suff_1 is the entire string S, and Suff_m is the single character $S(m)$.

Definition Define $\mathcal{T}_i$ to be the tree that has $m - i + 2$ leaves numbered i through $m + 1$ such that the path from the root to any leaf j ($i \leq j \leq m + 1$) has label Suff_j\$. That is, $\mathcal{T}_i$ is a tree encoding all and only the suffixes of string $S[i..m]$\$, so it is a suffix tree of string $S[i..m]$\$.

Weiner's algorithm constructs trees from $\mathcal{T}_{m+1}$ *down* to $\mathcal{T}_1$ (i.e., in decreasing order of i). We will first implement the method in a straightforward inefficient way. This will

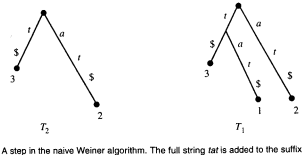


Figure 6.10: A step in the naive Weiner algorithm. The full string *tat* is added to the suffix tree for *at*. The edge labeled with the single character $ is omitted, since such an edge is part of every suffix tree.

serve to introduce and illustrate important definitions and facts. Then we will speed up the straightforward construction to obtain Weiner's linear-time algorithm.

6.2.1. A straightforward construction

The first tree $\mathcal{T}_{m+1}$ consists simply of a single edge out of the root labeled with the termination character \$. Thereafter, for each i from m down to 1, the algorithm constructs each tree $\mathcal{T}_i$ from tree $\mathcal{T}_{i+1}$ and character $S(i)$. The idea of the method is essentially the same as the idea for constructing keyword trees (Section 3.4), but for a different set of strings and without putting in backpointers. As the algorithm proceeds, each tree $\mathcal{T}_i$ will have the property that for any node v in $\mathcal{T}_i$, no two edges out of v have edge-labels beginning with the same character. Since $\mathcal{T}_{m+1}$ only has one edge out of the root, this is trivially true for $\mathcal{T}_{m+1}$. We assume inductively that this property is true for tree $\mathcal{T}_{i+1}$ and will verify that it holds for tree $\mathcal{T}_i$.

In general, to create $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, start at the root of $\mathcal{T}_{i+1}$ and walk as far as possible down a path whose label matches a prefix of Suff$_i$\$. Let R denote that path. In more detail, path R is found by starting at the root and explicitly matching successive characters of Suff$_i$\$ with successive characters along a unique path in $\mathcal{T}_{i+1}$. The matching path is unique, since at any node v of $\mathcal{T}_{i+1}$ no two edges out of v have edge-labels beginning with the same character. Thus the matching continues on at most one edge out of v. Ultimately, because no suffix is a prefix of another, no further match will be possible. If no node exists at that point, then create a new node there. In either case, refer to the node there (old or new) as w. Finally, add an edge out of w to a new leaf node labeled i, and label the new edge (w, i) with the remaining (unmatched) part of Suff$_i$\$. Since no further match had been possible, the first character on the label for edge (w, i) does not occur as the first character on any other edge out of w. Thus the claimed inductive property is maintained. Clearly, the path from the root to leaf i has label Suff$_i$\$. That is, that path exactly spells out string Suff$_i$\$, so tree $\mathcal{T}_i$ has been created.

For example, Figure 6.10 shows the transformation of $\mathcal{T}_2$ to $\mathcal{T}_1$ for the string *tat*.

Definition For any position i, *Head*(i) denotes the longest prefix of $S[i..m]$ that matches a substring of $S[i+1..m]$\$.

Note that *Head*(i) could be the empty string. In fact, *Head*(m) is always the empty string because $S[i+1..m]$ is the empty string when $i+1$ is greater than m and character $S(m) \neq$ \$.

Since a copy of string *Head*(i) begins at some position between $i+1$ and m, *Head*(i) is also a prefix of Suff$_k$ for some $k > i$. It follows that *Head*(i) is the longest prefix (possibly empty) of Suff$_i$ that is a label on some path from the root in tree $\mathcal{T}_{i+1}$.

The above straightforward algorithm to build $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ can be described as follows:

Naive Weiner algorithm

1. Find the end of the path labeled *Head*(i) in tree $\mathcal{T}_{i+1}$.
2. If there is no node at the end of *Head*(i) then create one, and let w denote the node (created or not) at the end of *Head*(i). If w is created at this point, splitting an existing edge, then split its existing edge-label so that w has node-label *Head*(i). Then, create a new leaf numbered i and a new edge (w, i) labeled with the remaining characters of Suff$_i$\$. That is, the new edge-label should be the last $m - i + 1 - |Head(i)|$ characters of Suff$_i$, followed by the termination symbol \$.

6.2.2. Toward a more efficient implementation

It should be clear that the final suffix tree $\mathcal{T} = \mathcal{T}_1$ is constructed in $O(m^2)$ time by this straightforward approach. Clearly, the difficult part of the algorithm is finding *Head*(i), since step 2 takes only constant time for any i. So, to speed up the algorithm, we will need a more efficient way to find *Head*(i). But, as in the discussion of Ukkonen's algorithm, a linear time bound is not possible if edge-labels are explicitly written on the tree. Instead, each edge-label is represented by two indices indicating the start and end positions of the labeling substring. The reader should review Section 6.1.4 at this point.

It is easy to implement Weiner's algorithm using an index pair to label an edge. When inserting Suff$_i$, suppose the algorithm has matched up to the kth character on an edge (u, z) labeled with interval $[s, t]$, but the next character is a mismatch. A new node w is created dividing (u, z) into two edges, (u, w) and (w, z), and a new edge is also created from w to leaf i. Edge (u, w) gets labeled $[s, s+k-1]$, edge (w, z) gets labeled $[s+k, t]$, and edge (w, i) gets labeled $[i + d(w), m]$\$, where $d(w)$ is the string-depth (number of characters) of the path from the root down to node w. These string-depths can easily be created and maintained as the tree is being built, since $d(w) = d(u) + k$. The string-depth of a leaf i is $m - i + 1$.

Finding *Head*(i) efficiently

We now return to the central issue of how to find *Head*(i) efficiently. The key to Weiner's algorithm are two vectors kept at each nonleaf node (including the root). The first vector is called the *indicator vector* I and the second is called the *link vector* L. Each vector is of length equal to the size of the alphabet, and each is indexed by the characters of the alphabet. For example, for the English alphabet augmented with \$, each link and indicator vector will be of length 27.

The link vector is essentially the reverse of the suffix link in Ukkonen's algorithm, and the two links are used in similar ways to accelerate traversals inside the tree.

The indicator vector is a bit vector so its entries are just 0 or 1, whereas each entry in the link vector is either null or is a pointer to a tree node. Let $I_v(x)$ specify the entry of the indicator vector at node v indexed by character x. Similarly, let $L_v(x)$ specify the entry of the link vector at node v indexed by character x.

The vectors I and L have two crucial properties that will be maintained inductively throughout the algorithm:

- For any (single) character x and any node u, $I_u(x) = 1$ in $\mathcal{T}_{i+1}$ if and only if there is a *path* from the root of $\mathcal{T}_{i+1}$ labeled $x\alpha$, where α is the path-label of node u. The path labeled $x\alpha$ need not end at a node.
- For any character x, $L_u(x)$ in $\mathcal{T}_{i+1}$ points to (internal) *node* $\overline{u}$ in $\mathcal{T}_{i+1}$ if and only if $\overline{u}$ has path-label $x\alpha$, where u has path-label α. Otherwise $L_u(x)$ is null.

For example, in the tree in Figure 5.1 (page 91) consider the two internal nodes u and w with path-labels a and xa respectively. Then $I_u(x) = 1$ for the specific character x, and $L_u(x) = w$. Also, $I_u(b) = 1$, but $L_w(b)$ is null.

Clearly, for any node u and any character x, $L_u(x)$ is nonnull only if $I_u(x) = 1$, but the converse is not true. It is also immediate that if $I_u(x) = 1$ then $I_v(x) = 1$ for every ancestor node v of u.

Tree $\mathcal{T}_m$ has only one nonleaf node, namely the root r. In this tree we set $I_r(S(m))$ to one, set $I_r(x)$ to zero for every other character x, and set all the link entries for the root to null. Hence the above properties hold for $\mathcal{T}_m$. The algorithm will maintain the vectors as the tree changes, and we will prove inductively that the above properties hold for each tree.

6.2.3. The basic idea of Weiner's algorithm

Weiner's algorithm uses the indicator and link vectors to find $Head(i)$ and to construct $\mathcal{T}_i$ more efficiently. The algorithm must take care of two degenerate cases, but these are not much different than the general "good" case where no degeneracy occurs. We first discuss how to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ in the good case, and then we handle the degenerate cases.

The algorithm in the good case

We assume that tree $\mathcal{T}_{i+1}$ has just been constructed and we now want to build $\mathcal{T}_i$. The algorithm starts at leaf $i+1$ of $\mathcal{T}_{i+1}$ (the leaf for Suff_{i+1}) and walks toward the root looking for the first node v, if it exists, such that $I_v(S(i)) = 1$. If found, it then continues from v walking upwards toward the root searching for the first node v' it encounters (possibly v) where $L_{v'}(S(i))$ is nonnull. By definition, $L_{v'}(S(i))$ is nonnull only if $I_{v'}(S(i)) = 1$, so if found, v' will also be the first node encountered on the walk from leaf $i+1$ such that $L_{v'}(S(i))$ is nonnull. In general, it may be that neither v nor v' exist or that v exists but v' does not. Note, however, that v or v' may be the root.

The "good case" is that *both* v and v' *do* exist.

Let l_i be the number of characters on the path between v' and v, and if $l_i > 0$ then let c denote the first of these l_i characters.

Assuming the good case, that both v and v' exist, we will prove below that if node v has path-label α then $Head(i)$ is precisely string $S(i)\alpha$. Further, we will prove that when $L_{v'}(S(i))$ points to node v'' in $\mathcal{T}_{i+1}$, $Head(i)$ either ends at v'', if $l_i = 0$, or else it ends *exactly* l_i characters below v'' on an edge out of v''. So in either case, $Head(i)$ can be found in constant time after v' is found.

Theorem 6.2.1. *Assume that node v has been found by the algorithm and that it has path-label α. Then the string $Head(i)$ is exactly $S(i)\alpha$.*

PROOF $Head(i)$ is the longest prefix of Suff_i that is also a prefix of Suff_k for some $k > i$. Since v was found with $I_v(S(i)) = 1$ there is a path in $\mathcal{T}_{i+1}$ that begins with $S(i)$, so $Head(i)$ is at least one character long. Therefore, we can express $Head(i)$ as $S(i)\beta$, for some (possibly empty) string β.

Suff_i and Suff_k both begin with string $Head(i) = S(i)\beta$ and differ after that. For concreteness, say Suff_i begins $S(i)\beta a$ and Suff_k begins $S(i)\beta b$. But then Suff_{i+1} begins βa and Suff_{k+1} begins βb. Both $i+1$ and $k+1$ are greater than or equal to $i+1$ and less than or equal to m, so both suffixes are represented in tree $\mathcal{T}_{i+1}$. Therefore, in tree $\mathcal{T}_{i+1}$, there must be a path from the root labeled β (possibly the empty string) that extends in two ways, one continuing with character a and the other with character b. Hence there is a node u in $\mathcal{T}_{i+1}$ with path-label β, and $I_u(S(i)) = 1$ since there is a path (namely, an initial part of the path to leaf k) labeled $S(i)\beta$ in $\mathcal{T}_{i+1}$. Further, node u must be on the path to leaf $i+1$ since β is a prefix of Suff_{i+1}.

Now $I_v(S(i)) = 1$ and v has path-label α, so $Head(i)$ must begin with $S(i)\alpha$. That means that α is a prefix of β and so node u, with path label β, must either be v or below v on the path to leaf $i+1$. However, if $u \neq v$ then u would be a node below v on the path to leaf $i+1$, and $I_u(S(i)) = 1$. This contradicts the choice of node v, so $v = u$, $\alpha = \beta$, and the theorem is proved. That is, $Head(i)$ is exactly the string $S(i)\alpha$. □

Note that in Theorem 6.2.1 and its proof we only assume that node v exists. No assumption about v' was made. This will be useful in one of the degenerate cases examined later.

Theorem 6.2.2. *Assume both v and v' have been found and $L_{v'}(S(i))$ points to node v''. If $l_i = 0$ then $Head(i)$ ends at v''; otherwise it ends after exactly l_i characters on a single edge out of v''.*

PROOF Since v' is on the path to leaf $i+1$ and $L_{v'}(S(i))$ points to node v'', the path from the root labeled $Head(i)$ must include v''. By Theorem 6.2.1 $Head(i) = S(i)\alpha$, so $Head(i)$ must end exactly l_i characters below v''. Thus, when $l_i = 0$, $Head(i)$ ends at v''. But when $l_i > 0$, there must be an edge $e = (v'', z)$ out of v'' whose label begins with character c (the first of the l_i characters on the path from v' to v) in $\mathcal{T}_{i+1}$.

Can $Head(i)$ extend down to node z (i.e., to a node below v'')? Node z must be a branching node, for if it were a leaf then some suffix Suff_k, for $k > i$, would be a prefix of Suff_i, which is not possible. Let z have path-label $S(i)\gamma$. If $Head(i)$ extends down to branching node z, then there must be two substrings starting at or after position $i+1$ of S that both begin with string γ. Therefore, there would be a node z' with path-label γ in $\mathcal{T}_{i+1}$. Node z' would then be below v' on the path to leaf $i+1$, contradicting the selection of v'. So $Head(i)$ must not reach z and must end in the interior of edge e. In particular, it ends exactly l_i characters from v'' on edge e. □

Thus when $l_i = 0$, we know $Head(i)$ ends at v'', and when $l_i > 0$, we find $Head(i)$ from v'' by examining the edges out of v'' to identify that unique edge e whose first character is c. Then $Head(i)$ ends exactly l_i characters down e from v''. Tree $\mathcal{T}_i$ is then constructed by subdividing edge e, creating a node w at this point, and adding a new edge from w to leaf i labeled with the remainder of Suff_i. The search for the correct edge out of v'' takes only constant time since the alphabet is fixed.

In summary, when v and v' exist, the above method correctly creates $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, although we must still discuss how to update the vectors. Also, it may not yet be clear at this point why this method is more efficient than the naive algorithm for finding $Head(i)$. That will come later. Let us first examine how the algorithm handles the degenerate cases when v and v' do not both exist.

The two degenerate cases

The two degenerate cases are that node v (and hence node v') does not exist or that v exists but v' does not. We will see how to find $Head(i)$ efficiently in these two cases. Recall that r denotes the root node.

Case 1 $l_v(S(i)) = 0$.

In this case the walk ends at the root and no node v was found. It follows that character $S(i)$ does not appear in any position greater than i, for if it did appear, then some suffix in that range would begin with $S(i)$, some path from the root would begin with $S(i)$, and $l_v(S(i))$ would have been 1. So when $l_v(S(i)) = 0$, $Head(i)$ is the empty string and ends at the root.

Case 2 $l_v(S(i)) = 1$ for some v (possibly the root), but v' does not exist.

In this case the walk ends at the root with $L_v(S(i))$ null. Let t_i be the number of characters from the root to v. From Theorem 6.2.1 $Head(i)$ ends exactly $t_i + 1$ characters from the root. Since v exists, there is some edge $e = (r, z)$ whose edge-label begins with character $S(i)$. This is true whether $t_i = 0$ or $t_i > 0$.

If $t_i = 0$ then $Head(i)$ ends after the first character, $S(i)$, on edge e.

Similarly, if $t_i > 0$ then $Head(i)$ ends exactly $t_i + 1$ characters from the root on edge e. For suppose $Head(i)$ extends all the way to some child z (or beyond). Then exactly as in the proof of Theorem 6.2.2, z must be a branching node and there must be a node z' below the root on the path to leaf $i + 1$ such that $L_{z'}(S(i))$ is nonnull, which would be a contradiction. So when $t_i > 0$, $Head(i)$ ends exactly $t_i + 1$ characters from the root on the edge e out of the root. This edge can be found from the root in constant time since its first character is $S(i)$.

In either of these degenerate cases (as in the good case), $Head(i)$ is found in constant time after the walk reaches the root. After the end of $Head(i)$ is found and w is created or found, the algorithm proceeds exactly as in the good case.

Note that degenerate Case 2 is very similar to the "good" case when both v and v' were found, but differs in a small detail because $Head(i)$ is found $t_i + 1$ characters down on e rather than t_i characters down (the natural analogue of the good case).

6.2.4. The full algorithm for creating $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$

Incorporating all the cases gives the following algorithm:

Weiner's Tree extension

1. Start at leaf $i + 1$ of $\mathcal{T}_{i+1}$ (the leaf for Suff_{i+1}) and walk toward the root searching for the first node v on the walk such that $l_v(S(i)) = 1$.
2. If the root is reached and $l_r(S(i)) = 0$, then $Head(i)$ ends at the root. Go to Step 4.
3. Let v be the node found (possibly the root) such that $l_v(S(i)) = 1$. Then continue walking upward searching for the first node v' (possibly v itself) such that $L_{v'}(S(i))$ is nonnull.

3a. If the root is reached and $L_r(S(i))$ is null, let t_i be the number of characters on the path between the root and v. Search for the edge e out of the root whose edge-label begins with $S(i)$. $Head(i)$ ends exactly $t_i + 1$ characters from the root on edge e.
Else {when the condition in 3a does not hold}

3b. If v' was found such that $L_{v'}(S(i))$ is nonnull, say v'', then follow the link (for $S(i)$) to v''. Let l_i be the number of characters on the path from v' to v and let c be the first character on this path. If $l_i = 0$ then $Head(i)$ ends at v''. Otherwise, search for the edge e out of v'' whose first character is c. $Head(i)$ ends exactly l_i characters below v'' on edge e.

4. If a node already exists at the end of $Head(i)$, then let w denote that node; otherwise, create a node w at the end of $Head(i)$. Create a new leaf numbered i; create a new edge (w, i) labeled with the remaining substring of Suff_i (i.e., the last $m - i + 1 - |Head(i)|$ characters of Suff_i), followed with the termination character \$. Tree $\mathcal{T}_i$ has now been created.

Correctness

It should be clear from the proof of Theorems 6.2.1 and 6.2.2 and the discussion of the degenerate cases that the algorithm correctly creates tree $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, although before it can create $\mathcal{T}_{i-1}$, it must update the I and L vectors.

How to update the vectors

After finding (or creating) node w, we must update the I and L vectors so that they are correct for tree $\mathcal{T}_i$. If the algorithm found a node v such that $I_v(S(i)) = 1$, then by Theorem 6.2.1 node w has path-label $S(i)\alpha$ in $\mathcal{T}_i$, where node v has path-label α. In this case, $L_v(S(i))$ should be set to point to w in $\mathcal{T}_i$. This is the only update needed for the link vectors since only one node can point via a link vector to any other node and only one new node was created. Furthermore, if node w is newly created, all its link entries for $\mathcal{T}_i$ should be null. To see this, suppose to the contrary that there is a node u in $\mathcal{T}_i$ with path-label $xHead(i)$, where w has path-label $Head(i)$. Node u cannot be a leaf because $Head(i)$ does not contain the character \$. But then there must have been a node in $\mathcal{T}_{i+1}$ with path-label $Head(i)$, contradicting the fact that node w was inserted into $\mathcal{T}_{i+1}$ to create $\mathcal{T}_i$. Consequently, there is no node in $\mathcal{T}_i$ with path-label $xHead(i)$ for any character x and all the L vector values for w should be null.

Now consider the updates needed to the indicator vectors for tree $\mathcal{T}_i$. For every node u on the path from the root to leaf $i + 1$, $I_u(S(i))$ must be set to 1 in $\mathcal{T}_i$ since there is now a path for string Suff_i in $\mathcal{T}_i$. It is easy to establish inductively that if a node v with $I_v(S(i)) = 1$ is found during the walk from leaf $i + 1$, then every node u above v on the path to the root already has $I_u(S(i)) = 1$. Therefore, only the indicator vectors for the nodes below v on the path to leaf $i + 1$ need to be set. If no node v was found, then all nodes on the path from $i + 1$ to the root were traversed and all of these nodes must have their indicator vectors updated. The needed updates for the nodes below v can be made during the search for v (i.e., no separate pass is needed). During the walk from leaf $i + 1$, $I_u(S(i))$ is set to 1 for every node u encountered on the walk. The time to set these indicator vectors is proportional to the time for the walk.

The only remaining update is to set the I vector for a newly created node w created in the interior of an edge $e = (v'', z)$.

Theorem 6.2.3. *When a new node w is created in the interior of an edge (v'', z) the indicator vector for w should be copied from the indicator vector for z.*

PROOF It is immediate that if $I_z(x) = 1$ then $I_w(x)$ must also be 1 in $\mathcal{T}_i$. But can it happen that $I_w(x)$ should be 1 and yet $I_z(x)$ is set to 0 at the moment that w is created? We will see that it cannot.

Let node z have path-label γ, and of course node w has path-label $Head(i)$, a prefix of γ. The fact that there are no nodes between u and z in $\mathcal{T}_{i+1}$ means that every suffix from Suff_m down to Suff_{i+1} that begins with string $Head(i)$ must actually begin with the longer

string γ. Hence in $\mathcal{T}_{i+1}$ there can be a path labeled $xHead(i)$ only if there is also a path labeled $x\gamma$, and this holds for any character x. Therefore, if there is a path in $\mathcal{T}_i$ labeled $xHead(i)$ (the requirement for $I_w(x)$ to be 1) but no path $x\gamma$, then the hypothesized string $xHead(i)$ must begin at character i of S. That means that Suff_{i+1} must begin with the string $Head(i)$. But since w has path-label $Head(i)$, leaf $i+1$ must be below w in $\mathcal{T}_i$ and so must be below z in $\mathcal{T}_{i+1}$. That is, z is on the root to $i+1$ path. However, the algorithm to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ starts at leaf $i+1$ and walks toward the root, and when it finds node v or reaches the root, the indicator entry for x has been set to 1 at every node on the path from the leaf $i+1$. The walk finishes before node w is created, and so it cannot be that $I_z(x) = 0$ at the time when w is created. So if path $xHead(i)$ exists in $\mathcal{T}_i$, then $I_z(x) = 1$ at the moment w is created, and the theorem is proved. □

6.2.5. Time analysis of Weiner's algorithm

The time to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ and update the vectors is proportional to the time needed during the walk from leaf $i+1$ (ending either at v' or the root). This walk moves from one node to its parent, and assuming the usual parent pointers, only constant time is used to move between nodes. Only constant time is used to follow a L link pointer, and only constant time is used after that to add w and edge (w, i). Hence the time to construct $\mathcal{T}_i$ is proportional to the number of nodes encountered on the walk from leaf $i+1$.

Recall that the node-depth of a node v is the number of nodes on the path in the tree from the root to v.

For the time analysis we imagine that as the algorithm runs we keep track of what node has most recently been encountered and what its node-depth is. Call the node-depth of the most recently encountered node the *current node-depth*. For example, when the algorithm begins, the current node-depth is one and just after $\mathcal{T}_m$ is created the current node-depth is two. Clearly, when the algorithm walks up a path from a leaf the current node-depth decreases by one at each step. Also, when the algorithm is at node v'' (or at the root) and then creates a new node w below v'' (or below the root), the current node-depth increases by one. The only question remaining is how the current node-depth changes when a link pointer is traversed from a node v' to v''.

Lemma 6.2.1. *When the algorithm traverses a link pointer from a node v' to a node v'' in $\mathcal{T}_{i+1}$, the current node-depth increases by at most one.*

PROOF Let u be a nonroot node in $\mathcal{T}_{i+1}$ on the path from the root to v'', and suppose u has path-label $S(i)x$ for some nonempty string α. All nodes on the root-to-v'' path are of this type, except for the single node (if it exists) with path-label $S(i)$. Now $S(i)\alpha$ is the prefix of Suff_i and of Suff_k for some $k > i$, and this string extends differently in the two cases. Since v' is on the path from the root to leaf $i+1$, α is a prefix of Suff_{i+1}, and there must be a node (possibly the root) with path-label α on the path to v' in $\mathcal{T}_{i+1}$. Hence the path to v' has a node corresponding to every node on the path to v'', except the node (if it exists) with path-label $S(i)$. Hence the depth of v'' is at most one more than the depth of v', although it could be less. □

We can now finish the time analysis.

Theorem 6.2.4. *Assuming a finite alphabet, Weiner's algorithm constructs the suffix tree for a string of length m in $O(m)$ time.*

PROOF The current node-depth can increase by one each time a new node is created and each time a link pointer is traversed. Hence the total number of increases in the current node-depth is at most $2m$. It follows that the current node-depth can also only decrease at most $2m$ times since the current node-depth starts at zero and is never negative. The current node-depth decreases at each move up the walk, so the total number of nodes visited during all the upward walks is at most $2m$. The time for the algorithm is proportional to the total number of nodes visited during upward walks, so the theorem is proved. □

6.2.6. Last comments about Weiner's algorithm

Our discussion of Weiner's algorithm establishes an important fact about suffix trees, regardless of how they are constructed:

Theorem 6.2.5. *If v is a node in the suffix tree labeled by the string $x\alpha$, where x is a single character, then there is a node in the tree labeled with the string α.*

This fact was also established as Corollary 6.1.2 during the discussion of Ukkonen's algorithm.

6.3. McCreight's suffix tree algorithm

Several years after Weiner published his linear-time algorithm to construct a suffix tree for a string S, McCreight [318] gave a different method that also runs in linear time but is more space efficient in practice. The inefficiency in Weiner's algorithm is the space it needs for the indicator and link vectors, I and L, kept at each node. For a fixed alphabet, this space is considered linear in the length of S, but the space used may be large in practice. McCreight's algorithm does not need those vectors and hence uses less space.

Ukkonen's algorithm also does not use the vectors I and L of Weiner's algorithm, and it has the same space efficiency as McCreight's algorithm.[3] In fact, the fully implemented version of Ukkonen's algorithm can be seen as a somewhat disguised version of McCreight's algorithm. However, the high-level organization of Ukkonen and McCreight's algorithms are quite different, and the connection between the algorithms is not obvious. That connection was suggested by Ukkonen [438] and made explicit by Giegerich and Kurtz [178]. Since Ukkonen's algorithm has all the advantages of McCreight's, and is simpler to describe, we will only introduce McCreight's algorithm at the high level.

McCreight's algorithm at the high level

McCreight's algorithm builds the suffix tree $\mathcal{T}$ for m-length string S by inserting the suffixes in order, one at a time, starting from suffix one (i.e., the complete string S). (This is opposite to the order used in Weiner's algorithm, and it is superficially different from Ukkonen's algorithm.) It builds a tree encoding all the suffixes of S starting at positions 1 through $i+1$, from the tree encoding all the suffixes of S starting at positions 1 through i.

The naive construction method is immediate and runs in $O(m^2)$ time. Using suffix links and the skip/count trick, that time can be reduced to $O(m)$. We leave this to the interested reader to work out.

[3] The space requirements for Ukkonen and McCreight's algorithms are determined by the need to represent and move around the tree quickly. We will be much more precise about space and practical implementation issues in Section 6.5.

6.4. Generalized suffix tree for a set of strings

We have so far seen methods to build a suffix tree for a single string in linear time. Those methods are easily extended to represent the suffixes of a set $\{S_1, S_2, \ldots, S_z\}$ of strings. Those suffixes are represented in a tree called a *generalized* suffix tree, which will be used in many applications.

A conceptually easy way to build a generalized suffix tree is to append a different end of string marker to each string in the set, then concatenate all the strings together, and build a suffix tree for the concatenated string. The end of string markers must be symbols that are not used in any of the strings. The resulting suffix tree will have one leaf for each suffix of the concatenated string and is built in time proportional to the sum of all the string lengths. The leaf numbers can easily be converted to two numbers, one identifying a string S_i and the other a starting position in S_i.

One defect with this way of constructing a generalized suffix tree is that the tree represents substrings (of the concatenated string) that span more than one of the original strings. These "synthetic" suffixes are not generally of interest. However, because each end of string marker is distinct and is not in any of the original strings, the label on any path from the root to an internal node must be a substring of one of the original strings. Hence by reducing the second index of the label on leaf edges, without changing any other parts of the tree, all the unwanted synthetic suffixes are removed.

Under closer examination, the above method can be simulated without first concatenating the strings. We describe the simulation using Ukkonen's algorithm and two strings S_1 and S_2, assumed to be distinct. First build a suffix tree for S_1 (assuming an added terminal character). Then starting at the root of this tree, match S_2 (again assuming the same terminal character has been added) against a path in the tree until a mismatch occurs. Suppose that the first i characters of S_2 match. The tree at this point encodes all the suffixes of S_1, and it implicitly encodes every suffix of the string $S_2[1..i]$. Essentially, the first i phases of Ukkonen's algorithm for S_2 have been executed on top of the tree for S_1. So, with that current tree, resume Ukkonen's algorithm on S_2 in phase $i+1$. That is, walk up at most one node from the end of $S_2[1..i]$, etc. When S_2 is fully processed the tree will encode all the suffixes of S_1 and all the suffixes of S_2 but will have no synthetic suffixes. Repeating this approach for each of the strings in the set creates the generalized suffix tree in time proportional to the sum of the lengths of all the strings in the set.

There are two minor subtleties with the second approach. One is that the compressed labels on different edges may refer to different strings. Hence the number of symbols per edge increases from two to three, but otherwise causes no problem. The second subtlety is that suffixes from two strings may be identical, although it will still be true that no suffix is a prefix of any other. In this case, a leaf must indicate all of the strings and starting positions of the associated suffix.

As an example, if we add the string *babxba* to the tree for *xabxa* (shown in Figure 6.1), the result is the generalized suffix tree shown in Figure 6.11.

6.5. Practical implementation issues

The implementation details already discussed in this chapter turn naive, quadratic (or even cubic) time algorithms into algorithms that run in $O(m)$ worst-case time, assuming a fixed alphabet Σ. But to make suffix trees truly practical, more attention to implementation is needed, particularly as the size of the alphabet grows. There are problems nicely solved

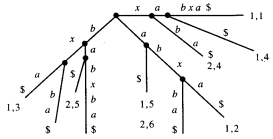


Figure 6.11: Generalized suffix tree for strings $S_1 = xabxa$ and $S_2 = babxba$. The first number at a leaf indicates the string; the second number indicates the starting position of the suffix in that string.

in theory by suffix trees, where the typical string size is in the hundreds of thousands, or even millions, and/or where the alphabet size is in the hundreds. For those problems, a "linear" time and space bound is not sufficient assurance of practicality. For large trees, paging can also be a serious problem because the trees do not have nice locality properties. Indeed, by design, suffix links allow an algorithm to move quickly from one part of the tree to a distant part of the tree. This is great for worst-case time bounds, but it is horrible for paging if the tree isn't entirely in memory. Consequently, implementing suffix trees to reduce practical space use can be a serious concern.[4] The comments made here for suffix trees apply as well to keyword trees used in the Aho–Corasick method.

The main design issues in all three algorithms are how to represent and search the branches out of the nodes of the tree and how to represent the indicator and link vectors in Weiner's algorithm. A practical design must balance the constraints of space against the need for speed, both in building the tree and in using it afterwards. We will discuss representing tree edges, since the vector issues for Weiner's algorithm are identical.

There are four basic choices possible to represent branches. The simplest is to use an *array* of size $\Theta(|\Sigma|)$ at each nonleaf node v. The array at v is indexed by single characters of the alphabet; the cell indexed by character x has a pointer to a child of v if there is an edge out of v whose edge-label begins with character x and is otherwise null. If there is such an edge, then the cell should also hold the two indices representing its edge-label. This array allows constant-time random accesses and updates and, although simple to program, it can use an impractical amount of space as $|\Sigma|$ and m get large.

An alternative to the array is to use a *linked list* at node v of characters that appear at the beginning of edge-labels out of v. When a new edge from v is added to the tree, a new character (the first character on the new edge label) is added to the list. Traversals from node v are implemented by sequentially searching the list for the appropriate character. Since the list is searched sequentially it costs no more to keep it in *sorted order*. This somewhat reduces the average time to search for a given character and thus speeds up (in practice) the construction of the tree. The key point is that it allows a faster termination of a search for a character that is not in the list. Keeping the list in sorted order will be particularly useful in some of applications of suffix trees to be discussed later.

[4] A very different approach to limiting space, based on changing the suffix tree into a different data structure called a *suffix array*, will be discussed in Section 7.14.

Keeping a linked list at node v works well if the number of children of v is small, but in worst-case adds time $|\Sigma|$ to every node operation. The $O(m)$ worst-case time bounds are preserved since $|\Sigma|$ is assumed to be fixed, but if the number of children of v is large then little space is saved over the array while noticeably degrading performance.

A third choice, a compromise between space and speed, is to implement the list at node v as some sort of *balanced tree* [10]. Additions and searches then take $O(\log k)$ time and $O(k)$ space, where k is the number of children of v. Due to the space and programming overhead of these methods, this alternative makes sense only when k is fairly large.

The final choice is some sort of *hashing scheme*. Again, the challenge is to find a scheme balancing space with speed, but for large trees and alphabets hashing is very attractive at least for some of the nodes. And, using perfect hashing techniques [167] the linear worst-case time bound can even be preserved.

When m and Σ are large enough to make implementation difficult, the best design is probably a mixture of the above choices. Nodes near the root of the tree tend to have the most children (the root has a child for every distinct character appearing in S), and so arrays are a sensible choice at those nodes. In addition, if the tree is dense for several levels below the root, then those levels can be condensed and eliminated from the explicit tree. For example, there are 20^5 possible amino acid substrings of length five. Every one of these substrings exists in some known protein sequence already in the databases. Therefore, when implementing a suffix tree for the protein database, one can replace the first five levels of the tree with a five-dimensional array (indexed by substrings of length five), where an entry of the array points to the place in the remaining tree that extends the five-tuple. The same idea has been applied [320] to depth seven for DNA data. Nodes in the suffix tree toward the leaves tend to have few children and lists there are attractive. At the extreme, if w is a leaf and v is its parent, then information about w may be brought up to v, removing the need for explicit representation of the edge (v, w) or the node w. Depending on the other implementation choices, this can lead to a large savings in space since roughly half the nodes in a suffix tree are leaves. A suffix tree whose leaves are deleted in this way is called a *position tree*. In a position tree, there is a one-to-one correspondence between leaves of the tree and substrings that are uniquely occurring in S.

For nodes in the middle of a suffix tree, hashing or balanced trees may be the best choice. Fortunately, most large suffix trees are used in applications where S is fixed (a dictionary or database) for some time and the suffix tree will be used repeatedly. In those applications, one has the time and motivation to experiment with different implementation choices. For a more in-depth look at suffix tree implementation issues, and other suggested variants of suffix trees, see [23].

Whatever implementation is selected, it is clear that a suffix tree for a string will take considerably more space than the representation of the string itself.[5] Later in the book we will discuss several problems involving two (or more) strings P and T, where two $O(|P| + |T|)$ time solutions exist, one using a suffix tree for P and one using a suffix tree for T. We will also have examples where equally time-efficient solutions exist, but where one uses a generalized suffix tree for two or more strings and the other uses just a suffix tree for the smaller string. In asymptotic worst-case time and space, neither approach is superior to the other, and usually the approach that builds the larger tree is conceptually simpler. However, when space is a serious practical concern (and in many problems, including

[5] Although, we have built suffix trees for DNA and amino acid strings more than one million characters long that can be completely contained in the main memory of a moderate-size workstation.

many in molecular biology, space is more of a constraint than is time), the size of the suffix tree for a string may dictate using the solution that builds the smaller suffix tree. So despite the added conceptual burden, we will discuss such space-reducing alternatives in some detail throughout the book.

6.5.1. Alphabet independence: all linears are equal, but some are more equal than others

The key implementation problems discussed above are all related to multiple edges (or links) at nodes. These are influenced by the size of the alphabet Σ – the larger the alphabet, the larger the problem. For that reason, some people prefer to explicitly reflect the alphabet size in the time and space bounds of keyword and suffix tree algorithms. Those people usually refer to the construction time for keyword or suffix trees as $O(m \log |\Sigma|)$, where m is the size of all the patterns in a keyword tree or the size of the string in a suffix tree. More completely, the Aho–Corasick, Weiner, Ukkonen, and McCreight algorithms all either require $\Theta(m|\Sigma|)$ space, or the $O(m)$ time bound should be replaced with the minimum of $O(m \log m)$ and $O(m \log |\Sigma|)$. Similarly, searching for a pattern P using a suffix tree can be done with $O(|P|)$ comparisons only if we use $\Theta(m|\Sigma|)$ space; otherwise we must allow the minimum of $O(|P| \log m)$ and $O(|P| \log |\Sigma|)$ comparisons during a search for P.

In contrast, the exact matching method using Z values has worst-case space and comparison requirements that are *alphabet independent* – the worst-case number of comparisons (either characters or numbers) used to compute Z values is uninfluenced by the size of the alphabet. Moreover, when two characters are compared, the method only checks whether the characters are equal or unequal, not whether one character precedes the other in some ordering. Hence no prior knowledge about the alphabet need be assumed. These properties are also true of the Knuth-Morris-Pratt and the Boyer–Moore algorithms. The alphabet independence of these algorithms makes their linear time and space bounds superior, in some people's view, to the linear time and space bounds of keyword and suffix tree algorithms: "All linears are equal but some are more equal than others". Alphabet-independent algorithms have also been developed for a number of problems other than exact matching. Two-dimensional exact matching is one such example. The method presented in Section 3.5.3 for two-dimensional matching is based on keyword trees and hence is not alphabet independent. Nevertheless, alphabet-independent solutions for that problem have been developed. Generally, alphabet-independent methods are more complex than their coarser counterparts. In this book we will not consider alphabet-independence much further, although we will discuss other approaches to reducing space that can be employed if large alphabets cause excessive space use.

6.6. Exercises

1. Construct an infinite family of strings over a fixed alphabet, where the total length of the edge-labels on their suffix trees grows faster than $\Theta(m)$ (m is the length of the string). That is, show that linear-time suffix tree algorithms would be impossible if edge-labels were written explicitly on the edges.
2. In the text, we first introduced Ukkonen's algorithm at a high level and noted that it could be implemented in $O(m^3)$ time. That time was then reduced to $O(m^2)$ with the use of suffix links and the skip/count trick. An alternative way to reduce the $O(m^3)$ time to $O(m^2)$ (without suffix links or skip/count) is to keep a pointer to the end of each suffix of $S[1..i]$.

Then Ukkonen's high-level algorithm could visit all these ends and create $\mathcal{I}_{i+1}$ from $\mathcal{I}_i$ in $O(i)$ time, so that the entire algorithm would run in $O(m^2)$ time. Explain this in detail.

3. The relationship between the suffix tree for a string S and for the reverse string S' is not obvious. However, there is a significant relationship between the two trees. Find it, state it, and prove it.

   Hint: Suffix links help.

4. Can Ukkonen's algorithm be implemented in linear time without using suffix links? The idea is to maintain, for each index i, a pointer to the node in the current implicit suffix tree that is closest to the end of suffix i.

5. In Trick 3 of Ukkonen's algorithm, the symbol "e" is used as the second index on the label of every leaf edge, and in phase $i+1$ the global variable e is set to $i+1$. An alternative to using "e" is to set the second index on any leaf edge to m (the total length of S) at the point that the leaf edge is created. In that way, no work is required to update that second index. Explain in detail why this is correct, and discuss any disadvantages there may be in this approach, compared to using the symbol "e".

6. Ukkonen's algorithm builds all the implicit suffix trees I_1 through I_m in order and *on-line*, all in $O(m)$ time. Thus it can be called a linear-time on-line algorithm to construct implicit suffix trees.

   (Open question) Find an on-line algorithm running in $O(m)$ total time that creates all the *true* suffix trees. Since the time taken to explicitly store these trees is $\Theta(m^2)$, such an algorithm would (like Ukkonen's algorithm) update each tree without saving it.

7. Ukkonen's algorithm builds all the implicit suffix trees in $O(m)$ time. This sequence of implicit suffix trees may expose more information about S than does the single final suffix tree for S. Find a problem that can be solved more efficiently with the sequence of implicit suffix trees than with the single suffix tree. Note that the algorithm cannot save the implicit suffix trees and hence the problem will have to be solved in parallel with the construction of the implicit suffix trees.

8. The naive Weiner algorithm for constructing the suffix tree of S (Section 6.2.1) can be described in terms of the Aho–Corasick algorithm of Section 3.4: Given string S of length m, append \$ and let $\mathcal{P}$ be the set of patterns consisting of the $m+1$ suffixes of string S\$. Then build a keyword tree for set $\mathcal{P}$ using the Aho–Corasick algorithm. Removing the backlinks gives the suffix tree for S. The time for this construction is $O(m^2)$. Yet, in our discussion of Aho–Corasick, that method was considered as a *linear* time method. Resolve this apparent contradiction.

9. Make explicit the relationship between link pointers in Weiner's algorithm and suffix links in Ukkonen's algorithm.

10. The time analyses of Ukkonen's algorithm and of Weiner's algorithm both rely on watching how the current node-depth changes, and the arguments are almost perfectly symmetric. Examine these two algorithms and arguments closely to make explicit the similarities and differences in the analysis. Is there some higher-level analysis that might establish the time bounds of both the algorithms at once?

11. Empirically evaluate different implementation choices for representing the branches out of the nodes and the vectors needed in Weiner's algorithm. Pay particular attention to the effect of alphabet size and string length, and consider both time and space issues in building the suffix tree and in using it afterwards.

12. By using implementation tricks similar to those used in Ukkonen's algorithm (particularly, suffix links and skip/count) give a linear-time implementation for McCreight's algorithm.

13. Flesh out the relationship between McCreight's algorithm and Ukkonen's algorithm, when they both are implemented in linear time.

14. Suppose one must dynamically maintain a suffix tree for a string that is growing or contracting. Discuss how to do this efficiently if the string is growing (contracting) on the left end, and how to do it if the string is growing (contracting) on the right end.

    Can either Weiner's algorithm or Ukkonen's algorithm efficiently handle both changes to the right and to the left ends of the string? What would be wrong in reversing the string so that a change on the left end is "simulated" by a change on the right end?

15. Consider the previous problem where the changes are in the interior of the string. If you cannot find an efficient solution to updating the suffix tree, explain what the technical issues are and why this seems like a difficult problem.

16. Consider a generalized suffix tree built for a set of k strings. Additional strings may be added to the set, or entire strings may be deleted from the set. This is the common case for maintaining a generalized suffix tree for biological sequence data [320]. Discuss the problem of maintaining the generalized suffix tree in this dynamic setting. Explain why this problem has a much easier solution than when arbitrary substrings represented in the suffix tree are deleted.

7

First Applications of Suffix Trees

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

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

7.1. APL1: Exact string matching

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

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

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

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

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

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

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

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

There is one way that suffix trees are better, or more robust, than keyword trees for the exact set matching problem (in addition to other problems). We will show in Section 7.8 how to use a suffix tree to solve the exact set matching problem in exactly the same time

and space bounds as for the Aho–Corasick method – $O(n)$ for preprocessing and $O(m)$ for search. This is the reverse of the bounds shown above for suffix trees. The time/space trade-off remains, but a suffix tree can be used for either of the chosen time/space combinations, whereas no such choice is available for a keyword tree.

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

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

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

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

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

Suffix trees yield a very attractive solution to this database problem. A generalized suffix tree $\mathcal{T}$ for the strings in the database is built in $O(m)$ time and, more importantly, requires only $O(m)$ space. Any single string S of length n is found in the database, or declared not to be there, in $O(n)$ time. As usual, this is accomplished by matching the string against a path in the tree starting from the root. The full string S is in the database if and only if the matching path reaches a leaf of $\mathcal{T}$ at the point where the last character of S is examined. Moreover, if S is a substring of strings in the database then the algorithm can find all strings in the database containing S as a substring. This takes $O(n + k)$ time, where k is the number of occurrences of the substring. As expected, this is achieved by traversing the subtree below the end of the matched path for S. If the full string S cannot be matched against a path in $\mathcal{T}$, then S is not in the database, and neither is it contained in any string there. However, the matched path does specify the longest *prefix* of S that is contained as a substring in the database.

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

7.4. APL4: Longest common substring of two strings

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

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

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

In summary, we have

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

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

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

7.5. APL5: Recognizing DNA contamination

Often the various laboratory processes used to isolate, purify, clone, copy, maintain, probe, or sequence a DNA string will cause unwanted DNA to become inserted into the string of interest or mixed together with a collection of strings. Contamination of protein in the laboratory can also be a serious problem. During cloning, contamination is often caused

by a fragment (substring) of a *vector* (DNA string) used to incorporate the desired DNA in a host organism, or the contamination is from the DNA of the host itself (for example bacteria or yeast). Contamination can also come from very small amounts of undesired foreign DNA that gets physically mixed into the desired DNA and then amplified by PCR (the polymerase chain reaction) used to make copies of the desired DNA. Without going into these and other specific ways that contamination occurs, we refer to the general phenomenon as *DNA contamination*.

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

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

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

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

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

This motivates the following computational problem:

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

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

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

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

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

7.6. APL6: Common substrings of more than two strings

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

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

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

The biological applications motivate the following exact matching problem: Given a

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

Formal problem statement and first method

Suppose we have K strings whose lengths sum to n.

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

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

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

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

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

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

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

7.6.1. Computing the $C(v)$ numbers

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

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

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

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

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

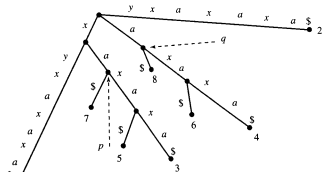


Figure 7.1: Suffix tree for string *xyxaxaxa* without suffix links shown.

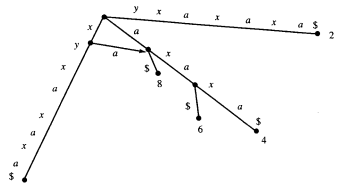


Figure 7.2: A directed acyclic graph used to recognize substrings of *xyxaxaxa*.

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 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 the 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 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 T be the suffix tree for an input string S, and let D be the DAG resulting from running the compaction algorithm on 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 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 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.

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 *GCCNNNNGGC*, 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 $S = aabxayaab$, 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 algorithm 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 $a\alpha b x\alpha y a\alpha b v x\alpha b$, substring α is a maximal repeat but not a supermaximal or a near-supermaximal repeat, whereas in $a\alpha b x\alpha y a\alpha b$, 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, $\alpha\gamma$ 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

maximal pairs, then the algorithm can be modified to run in $O(n)$ time. If only maximal pairs of a certain minimum length are requested (this would be the typical case in many applications), then the algorithm can be modified to run in $O(n + k_m)$ time, where k_m is the number of maximal pairs of length at least the required minimum. Simply stop the bottom-up traversal at any node whose string-depth falls below that minimum.

In summary, we have the following theorem:

Theorem 7.12.5. *All the maximal pairs can be found in $O(n + k)$ time, where k is the number of maximal pairs. If there are only k_m maximal pairs of length above a given threshold, then all those can be found in $O(n + k_m)$ time.*

7.13. APL12: Circular string linearization

Recall the definition of a circular string S given in Exercise 2 of Chapter 1 (page 11). The characters of S are initially numbered sequentially from 1 to n starting at an arbitrary point in S.

Definition Given an ordering of the characters in the alphabet, a string S_1 is *lexically (or lexicographically)* smaller than a string S_2 if S_1 would appear before S_2 in a normal dictionary ordering of the two strings. That is, starting from the left end of the two strings, if i is the first position where the two strings differ, then S_1 is lexically less than S_2 if and only if $S_1(i)$ precedes $S_2(i)$ in the ordering of the alphabet used in those strings.

To handle the case that S_1 is a proper prefix of S_2 (and should be considered lexically less than S_2), we follow the convention that a space is taken to be the first character of the alphabet.

The **circular string linearization problem** for a circular string S of n characters is the following: Choose a place to cut S so that the resulting linear string is the lexically smallest of all the n possible linear strings created by cutting S.

This problem arises in chemical data bases for circular molecules. Each such molecule is represented by a circular string of chemical characters; to allow faster lookup and comparisons of molecules, one wants to store each circular string by a *canonical* linear string. A single circular molecule may itself be a part of a more complex molecule, so this problem arises in the "inner loop" of more complex chemical retrieval and comparison problems.

A natural choice for canonical linear string is the one that is lexically least. With suffix trees, that string can be found in $O(n)$ time.

7.13.1. Solution via suffix trees

Arbitrarily cut the circular string S, giving a linear string L. Then, double L, creating the string LL, and build the suffix tree $\mathcal{T}$ for LL. As usual, affix the terminal symbol \$ at the end of LL, but interpret it to be lexically *greater* than any character in the alphabet used for S. (Intuitively, the purpose of doubling L is to allow efficient consideration of strings that begin with a suffix of L and end with a prefix of L.) Next, traverse tree $\mathcal{T}$ with the rule that, at every node, the traversal follows the edge whose first character is lexically smallest over all first characters on edges out of the node. This traversal continues until the traversed path has string-depth n. Such a depth will always be reached (with the proof left to the reader). Any leaf l in the subtree at that point can be used to cut the string. If $1 < l \leq n$, then cutting S between characters $l - 1$ and l creates a lexically smallest linearization of the circular string. If $l = 0$ or $l = n + 1$, then cut the circular string between character n and character 1. Each leaf in the subtree of this point gives a cutting point yielding the same linear string.

The correctness of this solution is easy to establish and is left as an exercise.

This method runs in linear time and is therefore time optimal. A different linear-time method with a smaller constant was given by Shiloach [404].

7.14. APL13: Suffix arrays – more space reduction

In Section 6.5.1, we saw that when alphabet size is included in the time and space bounds, the suffix tree for a string of length m either requires $\Theta(m|\Sigma|)$ space or the minimum of $O(m \log m)$ and $O(m \log |\Sigma|)$ time. Similarly, searching for a pattern P of length n using a suffix tree can be done in $O(n)$ time only if $\Theta(m|\Sigma|)$ space is used for the tree, or if we assume that up to $|\Sigma|$ character comparisons cost only constant time. Otherwise, the search takes the minimum of $O(n \log m)$ and $O(n \log |\Sigma|)$ comparisons. For these reasons, a suffix tree may require too much space to be practical in some applications. Hence a more space efficient approach is desired that still retains most of the advantages of searching with a suffix tree.

In the context of the substring problem (see Section 7.3) where a fixed string T will be searched many times, the key issues are the time needed for the search and the space used by the fixed data structure representing T. The space used during the preprocessing of T is of less concern, although it should still be "reasonable".

Manber and Myers [308] proposed a new data structure, called a *suffix array*, that is very space efficient and yet can be used to solve the exact matching problem or the substring problem almost as efficiently as with a suffix tree. Suffix arrays are likely to be an important contribution to certain string problems in computational molecular biology, where the alphabet can be large (we will discuss some of the reasons for large alphabets below). Interestingly, although the more formal notion of a suffix array and the basic algorithms for building and using it were developed in [308], many of the ideas were anticipated in the biological literature by Martinez [310].

After defining suffix arrays we show how to convert a suffix tree to a suffix array in linear time. It is important to be clear on the setting of the problem. String T will be held fixed for a long time, while P will vary. Therefore, the goal is to find a space-efficient representation for T (a suffix array) that will be held fixed and that facilitates search problems in T. However, the amount of space used during the construction of that representation is not so critical. In the exercises we consider a more space efficient way to build the representation itself.

Definition Given an m-character string T, a *suffix array* for T, called *Pos*, is an array of the integers in the range 1 to m, specifying the lexicographic order of the m suffixes of string T.

That is, the suffix starting at position $Pos(1)$ of T is the lexically smallest suffix, and in general suffix $Pos(i)$ of T is lexically smaller than suffix $Pos(i + 1)$.

As usual, we will affix a terminal symbol \$ to the end of S, but now we interpret it to be lexically *less* than any other character in the alphabet. This is in contrast to its interpretation in the previous section. As an example of a suffix array, if T is *mississippi*, then the suffix array *Pos* is 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3. Figure 7.4 lists the eleven suffixes in lexicographic order.

```
11:  i
 8:  ippi
 5:  issippi
 2:  ississippi
 1:  mississippi
10:  pi
 9:  ppi
 7:  sippi
 4:  sisippi
 6:  ssippi
 3:  ssissippi
```

Figure 7.4: The eleven suffixes of *mississippi* listed in lexicographic order. The starting positions of those suffixes define the suffix array *Pos*.

Notice that the suffix array holds only integers and hence contains no information about the alphabet used in string T. Therefore, the space required by suffix arrays is modest – for a string of length m, the array can be stored in exactly m computer words, assuming a word size of at least $\log m$ bits.

When augmented with an additional $2m$ values (called *Lcp* values and defined later), the suffix array can be used to find all the occurrences in T of a pattern P in $O(n + \log_2 m)$ single-character comparison and bookkeeping operations. Moreover, this bound is independent of the alphabet size. Since for most problems of interest $\log_2 m$ is $O(n)$, the substring problem is solved by using suffix arrays as efficiently as by using suffix trees.

7.14.1. Suffix tree to suffix array in linear time

We assume that sufficient space is available to build a suffix tree for T (this is done once during a preprocessing phase), but that the suffix tree cannot be kept intact to be used in the (many) subsequent searches for patterns in T. Instead, we convert the suffix tree to the more space efficient suffix array. Exercises 53, 54, and 55 develop an alternative, more space efficient (but slower) method, for *building* a suffix array.

A suffix array for T can be obtained from the suffix tree $\mathcal{T}$ for T by performing a "lexical" depth-first traversal of $\mathcal{T}$. Once the suffix array is built, the suffix tree is discarded.

Definition Define an edge (v, u) to be *lexically less* than an edge (v, w) if and only if the first character on the (v, u) edge is lexically less than the first character on (v, w). (In this application, the end of string character \$ is lexically less than any other character.)

Since no two edges out of v have labels beginning with the same character, there is a strict lexical ordering of the edges out of v. This ordering implies that the path from the root of $\mathcal{T}$ following the lexically smallest edge out of each encountered node leads to a leaf of $\mathcal{T}$ representing the lexically smallest suffix of T. More generally, a depth-first traversal of $\mathcal{T}$ that traverses the edges out of each node v in their lexical order will encounter the leaves of $\mathcal{T}$ in the lexical order of the suffixes they represent. Suffix array *Pos* is therefore just the ordered list of suffix numbers encountered at the leaves of $\mathcal{T}$ during the lexical depth-first search. The suffix tree for T is constructed in linear time, and the traversal also takes only linear time, so we have the following:

Theorem 7.14.1. *The suffix array Pos for a string T of length m can be constructed in $O(m)$ time.*

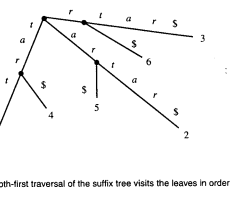


Figure 7.5: The lexical depth-first traversal of the suffix tree visits the leaves in order 5, 2, 6, 3, 4, 1.

For example, the suffix tree for $T = tartar$ is shown in Figure 7.5. The lexical depth-first traversal visits the nodes in the order 5, 2, 6, 3, 4, 1, defining the values of array *Pos*.

As an implementation detail, if the branches out of each node of the tree are organized in a *sorted* linked list (as discussed in Section 6.5, page 116) then the overhead to do a lexical depth-first search is the same as for any depth-first search. Every time the search must choose an edge out of a node v to traverse, it simply picks the next edge on v's linked list.

7.14.2. How to search for a pattern using a suffix array

The suffix array for string T allows a very simple algorithm to find all occurrences of any pattern P in T. The key is that if P occurs in T then all the locations of those occurrences will be grouped consecutively in *Pos*. For example, $P = issi$ occurs in *mississippi* starting at locations 2 and 5, which are indeed adjacent in *Pos* (see Figure 7.4). So to search for occurrences of P in T simply do binary search over the suffix array. In more detail, suppose that P is lexically less than the suffix in the middle position of *Pos* (i.e., suffix $Pos(\lceil m/2 \rceil)$). In that case, the first place in *Pos* that contains a position where P occurs in T must be in the first half of *Pos*. Similarly, if P is lexically greater than suffix $Pos(\lceil m/2 \rceil)$, then the places where P occurs in T must be in the second half of *Pos*. Using binary search, one can therefore find the smallest index i in *Pos* (if any) such that P exactly matches the first n characters of suffix $Pos(i)$. Similarly, one can find the largest index i' with that property. Then pattern P occurs in T starting at every location given by $Pos(i)$ through $Pos(i')$.

The lexical comparison of P to any suffix takes time proportional to the length of the common prefix of those two strings. That prefix has length at most n; hence

Theorem 7.14.2. *By using binary search on array Pos, all the occurrences of P in T can be found in $O(n \log m)$ time.*

Of course, the true behavior of the algorithm depends on how many long prefixes of P occur in T. If very few long prefixes of P occur in T then it will rarely happen that a specific lexical comparison actually takes $\Theta(n)$ time and generally the $O(n \log m)$ bound is quite pessimistic. In "random" strings (even on large alphabets) this method should run in $O(n + \log m)$ expected time. In cases where many long prefixes of P do occur in T, then the method can be improved with the two tricks described in the next two subsections.

7.14.3. A simple accelerant

As the binary search proceeds, let L and R denote the left and right boundaries of the "current search interval". At the start, L equals 1 and R equals m. Then in each iteration of the binary search, a query is made at location $M = \lceil (R + L)/2 \rceil$ of Pos. The search algorithm keeps track of the longest prefixes of $Pos(L)$ and $Pos(R)$ that match a prefix of P. Let l and r denote those two prefix lengths, respectively, and let $mlr = \min(l, r)$.

The value mlr can be used to accelerate the lexical comparison of P and suffix $Pos(M)$. Since array Pos gives the lexical ordering of the suffixes of T, if i is any index between L and R, the first mlr characters of suffix $Pos(i)$ must be the same as the first mlr characters of suffix $Pos(L)$ and hence of P. Therefore, the lexical comparison of P and suffix $Pos(M)$ can begin from position $mlr + 1$ of the two strings, rather than starting from the first position.

Maintaining mlr during the binary search adds little additional overhead to the algorithm but avoids many redundant comparisons. At the start of the search, when $L = 1$ and $R = m$, explicitly compare P to suffix $Pos(1)$ and suffix $Pos(m)$ to find l, r, and mlr. However, the worst-case time for this revised method is still $O(n \log m)$. Myers and Manber report that the use of mlr alone allows the search to run as fast in practice as the $O(n + \log m)$ worst-case method that we first advertised. Still, if only because of its elegance, we present the full method that guarantees that better worst-case bound.

7.14.4. A super-accelerant

Call an examination of a character in P *redundant* if that character has been examined before. The goal of the acceleration is to reduce the number of redundant character examinations to at most one per iteration of the binary search – hence $O(\log m)$ in all. The desired time bound, $O(n + \log m)$, follows immediately. The use of mlr alone does not achieve this goal. Since mlr is the minimum of l and r, whenever $l \neq r$, all characters in P from $mlr + 1$ to the maximum of l and r will have already been examined. Thus any comparisons of those characters will be redundant. What is needed is a way to begin comparisons at the *maximum* of l and r.

Definition $Lcp(i, j)$ is the length of the longest common prefix of the suffixes specified in positions i and j of Pos. That is, $Lcp(i, j)$ is the length of the longest prefix common to suffix $Pos(i)$ and suffix $Pos(j)$. The term Lcp stands for *longest common prefix*.

For example, when $T = mississippi$, suffix $Pos(3)$ is $issippi$, suffix $Pos(4)$ is $ississippi$, and so $Lcp(3, 4)$ is four (see Figure 7.4).

To speed up the search, the algorithm uses $Lcp(L, M)$ and $Lcp(M, R)$ for each triple (L, M, R) that arises during the execution of the binary search. For now, we assume that these values can be obtained in constant time when needed and show how they help the search. Later we will show how to compute the particular Lcp values needed by the binary search during the preprocessing of T.

How to use Lcp values

Simplest case In any iteration of the binary search, if $l = r$, then compare P to suffix $Pos(M)$ starting from position $mlr + 1 = l + 1 = r + 1$, as before.

General case When $l \neq r$, let us assume without loss of generality that $l > r$. Then there are three subcases:

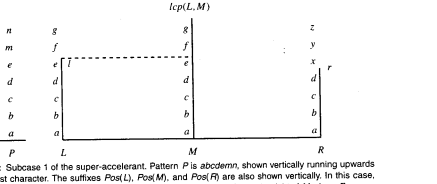


Figure 7.6: Subcase 1 of the super-accelerant. Pattern P is *abcdemn*, shown vertically running upwards from the first character. The suffixes $Pos(L)$, $Pos(M)$, and $Pos(R)$ are also shown vertically. In this case, $Lcp(L, M) > 0$ and $l > r$. Any starting location of P in T must occur in Pos to the right of M, since P agrees with suffix $Pos(M)$ only up to character l.

- If $Lcp(L, M) > l$, then the common prefix of suffix $Pos(L)$ and suffix $Pos(M)$ is longer than the common prefix of P and $Pos(L)$. Therefore, P agrees with suffix $Pos(M)$ up through character l. In other words, characters $l + 1$ of suffix $Pos(L)$ and suffix $Pos(M)$ are identical and lexically less than character $l + 1$ of P (the last fact follows since P is lexically greater than suffix $Pos(L)$). Hence all (if any) starting locations of P in T must occur to the right of position M in Pos. So in any iteration of the binary search where this case occurs, no examinations of P are needed; L just gets changed to M, and l and r remain unchanged. (See Figure 7.6.)
- If $Lcp(L, M) < l$, then the common prefix of suffix $Pos(L)$ and $Pos(M)$ is smaller than the common prefix of suffix $Pos(L)$ and P. Therefore, P agrees with suffix $Pos(M)$ up through character $Lcp(L, M)$. The $Lcp(L, M) + 1$ characters of P and suffix $Pos(L)$ are identical and lexically less than character $Lcp(L, M) + 1$ of suffix $Pos(M)$. Hence all (if any) starting locations of P in T must occur to the left of position M in Pos. So in any iteration of the binary search where this case occurs, no examinations of P are needed; r is changed to $Lcp(L, M)$, l remains unchanged, and R is changed to M.
- If $Lcp(L, M) = l$, then P agrees with suffix $Pos(M)$ up to character l. The algorithm then lexically compares P to suffix $Pos(M)$ starting from position $l + 1$. In the usual manner, the outcome of that lexical comparison determines which of L or R change, along with the corresponding change of l or r.

Theorem 7.14.3. *Using the Lcp values, the search algorithm does at most $O(n + \log m)$ comparisons and runs in that time.*

PROOF First, by simple case analysis it is easy to verify that neither l nor r ever decrease during the binary search. Also, every iteration of the binary search terminates the search, examines no characters of P, or ends after the first mismatch occurs in that iteration.

In the two cases ($l = r$ or $Lcp(L, M) = l > r$) where the algorithm examines a character during the iteration, the comparisons start with character $\max(l, r)$ of P. Suppose there are k characters of P examined in that iteration. Then there are $k - 1$ matches during the iteration, and at the end of the iteration $\max(l, r)$ increases by $k - 1$ (either l or r is changed to that value). Hence at the start of any iteration, character $\max(l, r)$ of P may have already been examined, but the next character in P has not been. That means at most one redundant comparison per iteration is done. Thus no more than $\log_2 m$ redundant comparisons are done overall. There are at most n nonredundant comparisons of characters

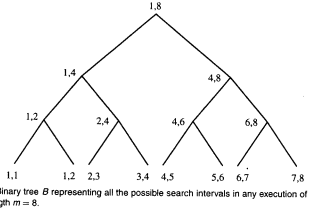


Figure 7.7: Binary tree B representing all the possible search intervals in any execution of binary search in a list of length $m = 8$.

of P, giving a total bound of $n + \log m$ comparisons. All the other work in the algorithm can clearly be done in time proportional to these comparisons. □

7.14.5. How to obtain the *Lcp* values

The *Lcp* values needed to accelerate searches are precomputed in the preprocessing phase during the creation of the suffix array. We first consider how many possible *Lcp* values are ever needed (over any possible execution of binary search). For convenience, assume m is a power of two.

Definition Let B be a complete binary tree with m leaves, where each node of B is labeled with a pair of integers (i, j), $1 \le i \le j \le m$. The root of B is labeled $(1, m)$. Every nonleaf node (i, j) has two children; the left one is labeled $(i, \lfloor (i + j)/2 \rfloor)$, and the right one is labeled $(\lfloor (i + j)/2 \rfloor, j)$. The leaves of B are labeled $(i, i + 1)$ (plus one labeled $(1, 1)$) and are ordered left to right in increasing order of i. (See Figure 7.7.)

Essentially, the node labels specify the endpoints (L, R) of all the possible search intervals that could arise in the binary search of an ordered list of length m. Since B is a binary tree with m leaves, B has $2m - 1$ nodes in total. So there are only $O(m)$ *Lcp* values that need be precomputed. It is therefore plausible that those values can be accumulated during the $O(m)$-time preprocessing of T; but how exactly? In the next lemma we show that the *Lcp* values at the leaves of B are easy to accumulate during the lexical depth-first traversal of $\mathcal{T}$.

Lemma 7.14.1. *In the depth-first traversal of $\mathcal{T}$, consider the internal nodes visited between the visits to leaf Pos(i) and leaf Pos(i + 1), that is, between the ith leaf visited and the next leaf visited. From among those internal nodes, let v denote the one that is closest to the root. Then Lcp(i, i + 1) equals the string-depth of node v.*

For example, consider again the suffix tree shown in Figure 7.5 (page 151). *Lcp*(5,6) is the string-depth of the parent of leaves 4 and 1. That string-depth is 3, since the parent of 4 and 1 is labeled with the string *tar*. The values of $Lcp(i, i + 1)$ are 2, 0, 1, 0, 3 for i from 1 to 5.

The hardest part of Lemma 7.14.1 involves parsing it. Once done, the proof is immediate from properties of suffix trees, and it is left to the reader.

If we assume that the string-depths of the nodes are known (these can be accumulated in linear time), then by the Lemma, the values $Lcp(i, i+1)$ for i from 1 to $m - 1$ are easily accumulated in $O(m)$ time. The rest of the *Lcp* values are easy to accumulate because of the following lemma:

Lemma 7.14.2. *For any pair of positions i, j, where j is greater than i + 1, Lcp(i, j) is the smallest value of Lcp(k, k + 1), where k ranges from i to j − 1.*

PROOF Suffix $Pos(i)$ and Suffix $Pos(j)$ of T have a common prefix of length $lcp(i, j)$. By the properties of lexical ordering, for every k between i and j, suffix $Pos(k)$ must also have that common prefix. Therefore, $lcp(k, k + 1) \ge lcp(i, j)$ for every k between i and $j - 1$.

Now by transitivity, $Lcp(i, i+2)$ must be at least as large as the minimum of $Lcp(i, i+1)$ and $Lcp(i + 1, i + 2)$. Extending this observation, $Lcp(i, j)$ must be at least as large as the smallest $Lcp(k, k + 1)$ for k from i to $j - 1$. Combined with the observation in the first paragraph, the lemma is proved. □

Given Lemma 7.14.2, the remaining *Lcp* values for B can be found by working up from the leaves, setting the *Lcp* value at any node v to the minimum of the *lcp* values of its two children. This clearly takes just $O(m)$ time.

In summary, the $O(n + \log m)$-time string and substring matching algorithm using a suffix array must precompute the $2m - 1$ *Lcp* values associated with the nodes of binary tree B. The leaf values can be accumulated during the linear-time, lexical, depth-first traversal of $\mathcal{T}$ used to construct the suffix array. The remaining values are computed from the leaf values in linear time by a bottom-up traversal of B, resulting in the following:

Theorem 7.14.4. *All the needed Lcp values can be accumulated in O(m) time, and all occurrences of P in T can be found using a suffix array in O(n + log m) time.*

7.14.6. Where do large alphabet problems arise?

A large part of the motivation for suffix arrays comes from problems that arise in using suffix trees when the underlying alphabet is large. So it is natural to ask where large alphabets occur.

First, there are natural languages, such as Chinese, with large "alphabets" (using some computer representation of the Chinese pictograms.) However, most large alphabets of interest to us arise because the string contains numbers, each of which is treated as a character. One simple example is a string that comes from a picture where each character in the string gives the color or gray level of a pixel.

String and substring matching problems where the alphabet contains numbers, and where P and T are large, also arise in computational problems in molecular biology. One example is the *map matching* problem. A *restriction enzyme map* for a single enzyme specifies the locations in a DNA string where copies of a certain substring (a restriction enzyme recognition site) occurs. Each such site may be separated from the next one by many thousands of bases. Hence, the restriction enzyme map for that single enzyme is represented as a string consisting of a sequence of integers specifying the distances between successive enzyme sites. Considered as a string, each integer is a character of a (huge) underlying alphabet. More generally, a map may display the sites of many different patterns of interest (whether or not they are restriction enzyme sites), so the string (map)

consists of characters from a finite alphabet (representing the known patterns of interest) alternating with integers giving the distances between such sites. The alphabet is huge because the range of integers is huge, and since distances are often known with high precision, the numbers are not rounded off. Moreover, the variety of known patterns of interest is itself large (see [435]).

It often happens that a DNA substring is obtained and studied without knowing where that DNA is located in the genome or whether that substring has been previously researched. If both the new and the previously studied DNA are fully sequenced and put in a database, then the issue of previous work or locations would be solved by exact string matching. But most DNA substrings that are studied are not fully sequenced – maps are easier and cheaper to obtain than sequences. Consequently, the following matching problem on *maps* arises and translates to an matching problem on *strings* with large alphabets:

> Given an established (restriction enzyme) map for a large DNA string and a map from a smaller string, determine if the smaller string is a substring of the larger one.

Since each map is represented as an alternating string of characters and integers, the underlying alphabet is huge. This provides one motivation for using suffix arrays for matching or substring searching in place of suffix trees. Of course, the problems become more difficult in the presence of errors, when the integers in the strings may not be exact, or when sites are missing or spuriously added. That problem, called *map alignment*, is discussed in Section 16.10.

7.15. APL14: Suffix trees in genome-scale projects

Suffix trees, generalized suffix trees and suffix arrays are now being used as the central data structures in three genome-scale projects.

Arabidopsis thaliana An *Arabidopsis thaliana* genome project,[6] at the Michigan State University and the University of Minnesota is initially creating an EST map of the *Arabidopsis* genome (see Section 3.5.1 for a discussion of ESTs and Chapter 16 for a discussion of mapping). In that project generalized suffix trees are used in several ways [63, 64, 65].

First, each sequenced fragment is checked to catch any contamination by known vector sequences. The vector sequences are kept in a generalized suffix tree, as discussed in Section 7.5.

Second, each new sequenced fragment is checked against fragments already sequenced to find duplicate sequences or regions of high similarity. The fragment sequences are kept in an expanding generalized suffix tree for this purpose. Since the project will sequence about 36,000 fragments, each of length about 400 bases, the efficiency of the searches for duplicates and for contamination is important.

Third, suffix trees are used in the search for biologically significant patterns in the obtained *Arabidopsis* sequences. Patterns of interest are often represented as regular expressions, and generalized suffix trees are used to accelerate regular expression pattern matching, where a small number of errors in a match are allowed. An approach that permits suffix trees to speed up regular expression pattern matching (with errors) is discussed in Section 12.4.

Yeast Suffix trees are also the central data structure in genome-scale analysis of *Saccharomyces cerevisiae* (brewer's yeast), done at the Max-Plank Institute [320]. Suffix trees are "particularly suitable for finding substring patterns in sequence databases" [320]. So in that project, highly optimized suffix trees called *hashed position trees* are used to solve problems of "clustering sequence data into evolutionary related protein families, structure prediction, and fragment assembly" [320]. (See Section 16.15 for a discussion of fragment assembly.)

Borrelia burgdorferi *Borrelia burgdorferi* is the bacterium causing Lyme disease. Its genome is about one million bases long, and is currently being sequenced at the Brookhaven National Laboratory using a directed sequencing approach to fill in gaps after an initial shotgun sequencing phase (see Section 16.14). Chen and Skiena [100] developed methods based on suffix trees and suffix arrays to solve the fragment assembly problem for this project. In fragment assembly, one major bottleneck is overlap detection, which requires solving a variant of the suffix-prefix matching problem (allowing some errors) for all pairs of strings in a large set (see Section 16.15.1.). The Borrelia work [100] consisted of 4,612 fragments (strings) totaling 2,032,740 bases. Using suffix trees and suffix arrays, the needed overlaps were computed in about fifteen minutes. To compare the speed and accuracy of the suffix tree methods to pure dynamic programming methods for overlap detection (discussed in Section 11.6.4 and 16.15.1), Chen and Skiena closely examined cosmid-sized data. The test established that the suffix tree approach gives a 1,000 times speedup over the (slightly) more accurate dynamic programming approach, finding 99% of the significant overlaps found by using dynamic programing.

Efficiency is critical

In all three projects, the efficiency of building, maintaining, and searching the suffix trees is extremely important, and the implementation details of Section 6.5 are crucial. However, because the suffix trees are very large (approaching 20 million characters in the case of the *Arabidopsis* project) additional implementation effort is needed, particularly in organizing the suffix tree on disk, so that the number of disk accesses is reduced. All three projects have deeply explored that issue and have found somewhat different solutions. See [320], [100] and [63] for details.

7.16. APL15: A Boyer–Moore approach to exact set matching

The Boyer–Moore algorithm for exact matching (single pattern) will often make long shifts of the pattern, examining only a small percentage of all the characters in the text. In contrast, Knuth-Morris-Pratt examines all characters in the text in order to find all occurrences of the pattern.

In the case of exact *set* matching, the Aho–Corasick algorithm is analogous to Knuth-Morris-Pratt – it examines all characters of the text. Since the Boyer–Moore algorithm for a single string is far more efficient in practice than Knuth-Morris-Pratt, one would like to have a Boyer–Moore type algorithm for the exact set matching problem, that is, a method for the exact set matching problem that typically examines only a sublinear portion of T. No known simple algorithm achieves this goal and also has a linear worst-case running

[6] *Arabidopsis thaliana* is the "fruit fly" of plant genetics, i.e., the classic model organism in studying the molecular biology of plants. Its size is about 100 million base pairs.

time. However, a synthesis of the Boyer–Moore and Aho–Corasick algorithms due to Commentz-Walter [109] solves the exact set matching problem in the spirit of the Boyer–Moore algorithm. Its shift rules allow many characters of T to go unexamined. We will not describe the Commentz–Walter algorithm but instead use suffix trees to achieve the same result more simply.

For simplicity of exposition, we will first describe a solution that uses two trees – a simple keyword tree (without back pointers) together with a suffix tree. The difficult work is done by the suffix tree. After understanding the ideas, we implement the method using only the suffix tree.

Definition Let P^r denote the reverse of a pattern P, and let $\mathcal{P}^r$ be the set of strings obtained by reversing every pattern P from an input set $\mathcal{P}$.

As usual, the algorithm preprocesses the set of patterns and then uses the result of the preprocessing to accelerate the search. The following exposition interleaves the descriptions of the search method and the preprocessing that supports the search.

7.16.1. The search

Recall that in the Boyer–Moore algorithm, when the end of the pattern is placed against a position i in T, the comparison of individual characters proceeds *right to left*. However, index i is *increased* at each iteration. These high-level features of Boyer–Moore will also hold in the algorithm we present for exact set matching.

In the case of multiple patterns, the search is carried out on a simple keyword tree $\mathcal{K}^r$ (without backpointers) that encodes the patterns in $\mathcal{P}^r$. The search again chooses increasing values of index i and determines for each chosen i whether there is a pattern in set $\mathcal{P}$ ending at position i of text T. Details are given below.

The preprocessing time needed to build $\mathcal{K}^r$ is only $O(n)$, the total length of all the patterns in $\mathcal{P}$. Moreover, because no backpointers are needed, the preprocessing is particularly simple. The algorithm to build $\mathcal{K}^r$ successively inserts each pattern into the tree, following as far as possible a matching path from the root, etc. Recall that each leaf of $\mathcal{K}^r$ specifies one of the patterns in $\mathcal{P}^r$.

The test at position i

Tree $\mathcal{K}^r$ can be used to test, for any specific position i in T, whether one of the patterns in $\mathcal{P}$ ends at position i. To make this test, simply follow a path from the root of $\mathcal{K}^r$, matching characters on the path with characters in T, starting with $T(i)$ and moving right to left as in Boyer–Moore. If a leaf of $\mathcal{K}^r$ is reached before the left end of T is reached, then the pattern number written at the leaf specifies a pattern that must occur in T ending at position i. Conversely, if the matched path ends before reaching a leaf and cannot be extended, then no pattern in $\mathcal{P}$ occurs in T ending at position i.

The first test begins with position i equal to the length of the smallest pattern in $\mathcal{P}$. The entire algorithm ends when i is set larger than m, the length of T.

When the test for a specific position i is finished, the algorithm increases i and returns to the root of $\mathcal{K}^r$ to begin another test. Increasing i is analogous to shifting the single pattern in the original Boyer–Moore algorithm. But by how much should i be increased? Increasing i by one is analogous to the naive algorithm for exact matching. With a shift of only one position, no occurrences of any pattern will be missed, but the resulting computation will be inefficient. In the worst case, it will take $\Theta(nm)$ time, where n is the total size of the

patterns and m is the size of the text. The more efficient algorithm will increase i by more than one whenever possible, using rules that are analogous to the bad character and good suffix rules of Boyer–Moore. Of course, no shift can be greater than the length of the shortest pattern P in $\mathcal{P}$, for such a shift could miss occurrences of P in T.

7.16.2. Bad character rule

The bad character rule from Boyer–Moore can easily be adapted to the set matching problem. Suppose the test matches some path in $\mathcal{K}^r$ against the characters from i down to $j < i$ in T but cannot extend the path to match character $T(j-1)$. A direct generalization of the bad character rule increases i to the smallest index $i_1 > i$ (if it exists) such that some pattern $\bar{P}$ from $\mathcal{P}$ has character $T(j-1)$ exactly $i_1 - j + 2$ positions from its right end. (See Figures 7.8 and 7.9.) With this rule, if i_1 exists, then when the right end of every pattern in $\mathcal{P}$ is aligned with position i_1 of T, character $j-1$ of T will be opposite a matching character in string $\bar{P}$ from $\mathcal{P}$. (There is a special case to consider if the test fails on the first comparison, i.e., at the root of $\mathcal{K}^r$. In that case, set $j = i+1$ before applying the shift rule.)

The above generalization of the bad character rule from the two-string case is not quite correct. The problem arises because of patterns in $\mathcal{P}$ that are smaller than $\bar{P}$. It may happen that i_1 is so large that if the right ends of all the patterns are aligned with it, then the left end of the smallest pattern $P_{\min}$ in $\mathcal{P}$ would be aligned with a position greater than j in T. If that happens, it is possible that some occurrence of $P_{\min}$ (with its left end opposite a position before $j+1$ in T) will be missed. Hence, using only the bad character information (not the suffix rules to come next), i should not be set larger than $j - 1 + |P_{\min}|$. In summary, the bad character rule for a set of patterns is:

> If i_1 does not exist, then increase i to $j - 1 + |P_{\min}|$; otherwise increase i to the *minimum* of i_1 and $j - 1 + |P_{\min}|$.

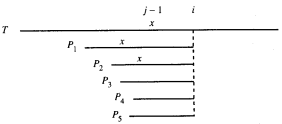


Figure 7.8: No further match is possible at position $j-1$ of T.

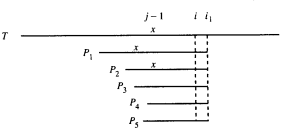


Figure 7.9: Shift when the bad character rule is applied

The preprocessing needed to implement the bad character rule is simple and is left to the reader.

The generalization of the bad character rule to set matching is easy but, unlike the case of a single pattern, use of the bad character rule alone may not be very effective. As the number of patterns grows, the typical size of $i_1 - i$ is likely to decrease, particularly if the alphabet is small. This is because *some* pattern is likely to have character $T(j-1)$ close to, but left of, the point where the previous matches end. As noted earlier, in some applications in molecular biology the total length of the patterns in $\mathcal{P}$ is larger than the size of T, making the bad character rule almost useless. A bad character rule analogous to the simpler, unextended bad character rule for a single pattern would be even less useful. Therefore, in the set matching case, a rule analogous to the good suffix rule is crucial in making a Boyer–Moore approach effective.

7.16.3. Good suffix rule

To adapt the (weak) good suffix rule to the set matching problem we reason as follows: After a matched path in $\mathcal{K}^r$ is found (either finding an occurrence of a pattern, or not) let j be the left-most character in T that was matched along the path, and let $\alpha = T[j..i]$ be the substring of T that was matched by the traversal (but found in reverse order). A direct generalization (to set matching) of the two-string good suffix rule would shift the right ends of all the patterns in $\mathcal{P}$ to the smallest value $i_2 > i$ (if it exists) such that $T[j..i]$ matches a substring of some pattern $\bar{P}$ in $\mathcal{P}$. Pattern $\bar{P}$ must contain the substring α beginning exactly $i_2 - j + 1$ positions from its right end. This shift is analogous to the good suffix shift for two patterns, but unlike the two-pattern case, that shift may be too large. The reason is again due to patterns that are smaller than $\bar{P}$.

When there are patterns smaller than $\bar{P}$, if the right end of every pattern moves to i_2, it may happen that the left end of the smallest pattern $P_{\min}$ would be placed more than one position to the right of i. In that case, an occurrence of $P_{\min}$ in T could be missed. Even if that doesn't happen, there is another problem. Suppose that a prefix β of some pattern $P' \in \mathcal{P}$ matches a suffix of α. If P' is smaller than $\bar{P}$, then shifting the right end of P' to i_2 may shift the prefix β of P' past the substring β in T. If that happens, then an occurrence of P in T could be missed. So let i_3 be the smallest index greater than i (if i_3 exists) such that when all the patterns in $\mathcal{P}$ are aligned with position i_3 of T, a prefix of at least one pattern is aligned opposite a suffix of α in T. Notice that because $\mathcal{P}$ contains more than one pattern, that overlap might not be the largest overlap between a prefix of a pattern in $\mathcal{P}$ and a suffix of α. Then the good suffix rule is:

> Increase i to the *minimum* of i_2, i_3, or $i + |P_{\min}|$. Ignore i_2 and/or i_3 in this rule, if either or both are nonexistent.

7.16.4. How to determine i_2 and i_3

The question now is how to efficiently determine i_2 and i_3 when needed during the search. We will first discuss i_2. Recall that α denotes the substring of T that was matched in the search just ended.

To determine i_2, we need to find which pattern P in $\mathcal{P}$ contains a copy of α ending closest to its right end, but not occurring as a suffix of P. If that copy of α ends r places

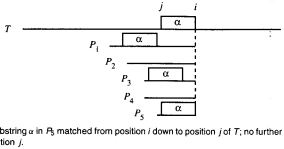


Figure 7.10: Substring α in P_3 matched from position i down to position j of T; no further match is possible to the left of position j.

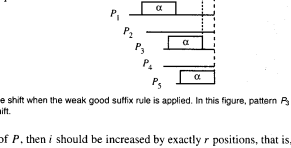


Figure 7.11: The shift when the weak good suffix rule is applied. In this figure, pattern P_3 determines the amount of the shift.

from the end of P, then i should be increased by exactly r positions, that is, i_2 should be set to $i + r$. (See Figure 7.10 and Figure 7.11.)

We will solve the problem of finding i_2, if it exists, using a suffix tree obtained by preprocessing set $\mathcal{P}^r$. The key involves using the suffix tree to search for a pattern P' in $\mathcal{P}^r$ containing a copy of α^r starting closest to its left end but not occurring as a prefix of P'. If that occurrence of α^r starts at position z of pattern P', then an occurrence of α ends $r = z - 1$ positions from the end of P.

During the preprocessing phase, build a generalized suffix tree $\mathcal{T}^r$ for the set of patterns $\mathcal{P}^r$. Recall that in a generalized suffix tree each leaf is associated with both a pattern $P' \in \mathcal{P}^r$ and a number z specifying the starting position of a suffix of P'.

> **Definition** For each internal node v of $\mathcal{T}^r$, z_v denotes the smallest number z greater than 1 (if any) such that z is a suffix position number written at a leaf in the subtree of v. If no such leaf exists, then z_v is undefined.

With this suffix tree $\mathcal{T}^r$, determine the number z_v for each internal node v. These two preprocessing tasks are easily accomplished in linear time by standard methods and are left to the reader.

As an example of the preprocessing, consider the set $\mathcal{P} = \{wxa,\ xaqq,\ qxax\}$ and the generalized suffix tree for $\mathcal{P}^r$ shown in Figure 7.12. The first number on each leaf refers to a string in $\mathcal{P}^r$, and the second number refers to a suffix starting position in that string. The number z_v is the first (or only) number written at every internal node (the second number will be introduced later).

We can now describe how $\mathcal{T}^r$ is used during the search to determine value i_2, if it exists. After matching α^r along a path in $\mathcal{K}^r$, traverse the path labeled α^r from the root of $\mathcal{T}^r$. That path exists because α is a suffix of some pattern in $\mathcal{P}$ (that is what the search in $\mathcal{K}^r$

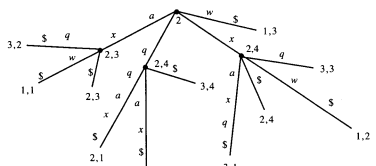


Figure 7.12: Generalized suffix tree $\mathcal{T}'$ for the set $\mathcal{P} = \{wxa, xaqq, qxax\}$.

determined), so α' is a prefix of some pattern in $\mathcal{P}'$. Let v be the first node at or below the end of that path in $\mathcal{T}'$. If z_v is defined, then i_2 can be obtained from it: The leaf defining z_v (i.e., the leaf where $z = z_v$) is associated with a string $P' \in \mathcal{P}'$ that contains a copy of α' starting to the right of position one. Over all such occurrences of α' in the strings of $\mathcal{P}'$, P' contains the copy of α' starting closest to its left end. That means that P contains a copy of α that is not a suffix of P, and over all such occurrences of α, P contains the copy of α ending closest to its right end. P is then the string in $\mathcal{P}$ that should be used to set i_2. Moreover, α ends in P exactly $z_v - 1$ characters from the end of P. Hence, as argued above, i should be increased by $z_v - 1$ positions. In summary, we have

Theorem 7.16.1. *If the first node v in $\mathcal{T}'$ at or below the end of path α' has a defined value z_v, then i_2 equals $i + z_v - 1$.*

Using suffix tree $\mathcal{T}'$, the determination of i_2 takes $O(|\alpha|)$ time, only doubling the time of the search used to find α. However, with proper preprocessing, the search used to find i_2 can be eliminated. The details will be given below in Section 7.16.5.

Now we turn to the computation of i_3. This is again easy assuming the proper preprocessing of $\mathcal{P}$. Again we use the generalized suffix tree $\mathcal{T}'$ for $\mathcal{P}'$. To get the idea of the method let $P \in \mathcal{P}$ be any pattern such that a suffix of α is a prefix of P. That means that a prefix of α' is a suffix of P'. Now consider the path labeled α' in $\mathcal{T}'$. Since some suffix of α is a prefix of P, some initial portion of the α' path in $\mathcal{T}'$ describes a suffix of P'. There thus must be a leaf edge (u, z) branching off that path, where leaf z is associated with pattern P' and the label of edge (u, z) is just the terminal character \$. Conversely, let (u, z) be any edge branching off the α' path and labeled with the single symbol \$. Then the pattern P associated with z must have a prefix matching a suffix of α. These observations lead to the following preprocessing and search methods.

In the preprocessing phase when $\mathcal{T}'$ is built, identify every edge (u, z) in $\mathcal{T}'$ that is labeled only by the terminal character \$. (The number z is used both as a leaf name and as the starting position of the suffix associated with that leaf.) For each such node u, set a variable d_u to z. For example, in Figure 7.12, d_u is the second number written at each node u (d_u is not defined for the root node). In the search phase, after matching a string α in $\mathcal{T}$, the value of i_3 (if needed) can be found as follows:

Theorem 7.16.2. *The value of i_3 should be set to $i + d_w - 1$, where d_w is the smallest d value at a node on the α' path in $\mathcal{T}'$. If no node on that path has a d value defined, then i_3 is undefined.*

The proof is immediate and is left to the reader. Clearly, i_3 can be found during the traversal of the α' path in $\mathcal{T}'$ used to search for i_2. If neither i_2 nor i_3 exist, then i should be increased by the length of the smallest pattern in $\mathcal{P}$.

7.16.5. An implementation eliminating redundancy

The implementation above builds two trees in time and space proportional to the total size of the patterns in $\mathcal{P}$. In addition, every time a string α' is matched in $\mathcal{K}'$ only $O(|\alpha|)$ additional time is used to search for i_2 and i_3. Thus the time to implement the shifts using the two trees is proportional to the time used to find the matches. From an asymptotic standpoint the two trees are as small as one, and the two traversals are as fast as one. But clearly there is superfluous work in this implementation – a single tree and a single traversal per search phase should suffice. Here's how.

Preprocessing for Boyer–Moore exact set matching

Begin

1. Build a generalized suffix tree $\mathcal{T}'$ for the strings in $\mathcal{P}'$. (Each leaf in the tree is numbered both by a specific pattern P' in $\mathcal{P}'$ and by a specific starting position z of a suffix in P'.)
2. Identify and mark every node in $\mathcal{T}'$, *including leaves*, that is an ancestor of a leaf numbered by suffix position one (for some pattern P' in $\mathcal{P}'$). Note that a node is considered to be an ancestor of itself.
3. For each marked node v, set z_v to be the smallest suffix position number z greater than one (if there is one) of any leaf in v's subtree.
4. Find every leaf edge (u, z) of $\mathcal{T}'$ that is labeled only by the terminal character \$, and set $d_u = z$.
5. For each node v in $\mathcal{T}'$ set d'_v equal to the smallest value of d_u for any ancestor (including v) of v.
6. Remove the subtree rooted at any unmarked node (including leaves) of $\mathcal{T}$. (Nodes were marked in step 2.)

End.

The above preprocessing tasks are easily accomplished in linear time by standard tree traversal methods.

Using $\mathcal{L}$ in the search phase

Let $\mathcal{L}$ denote the tree at the end of the preprocessing. Tree $\mathcal{L}$ is essentially the familiar keyword tree $\mathcal{K}'$ but is more compacted: Any path of nodes with only one descendent has been replaced with a single edge. Hence, for any i, the test to see if a pattern of $\mathcal{P}$ ends at position i can be executed using tree $\mathcal{L}$ rather than $\mathcal{K}'$. Moreover, unlike $\mathcal{K}'$, each node v in $\mathcal{L}$ now has associated with it the values needed to compute i_2 and i_3 in *constant* time. In detail, after the algorithm matches a string α in T by following the path α' in $\mathcal{L}$, the algorithm checks the first node v at or beneath the end of the path in $\mathcal{L}$. If z_v is defined there, then i_2 exists and equals $i + z_v - 1$. Next the algorithm checks the first node v at or above the end of the matched path. If d'_v is defined there then i_3 exists and equals $i + d'_v - 1$.

The search phase will not miss any occurrence of a pattern if either the good suffix rule or the bad character rule is used by itself. However, the two rules can be combined to increment i by the largest amount specified by either of the two rules.

Figure 7.13 shows tree $\mathcal{L}$ corresponding to the tree $\mathcal{T}'$ shown in Figure 7.12.

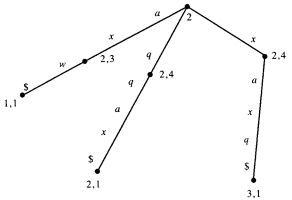


Figure 7.13: Tree $\mathcal{L}$ corresponding to tree $\mathcal{T}'$ for the set $\mathcal{P} = \{wxa, xaqq, qxax\}$.

1234567890	1234567890	1234567890	1234567890
qxaxtqqpst	qxaxtqqpst	qxaxtqqpst	qxaxtqqpst
wxa	wxa	wxa	wxa
xaqq	xaqq	xaqq	xaqq
qxax	qxax	qxax	qxax

Figure 7.14: The first comparisons start at position 3 of T and match ax. The value of z_v is equal to two, so a shift of one position occurs. String qxax matches; z_v is undefined, but d'_v is defined and equals 4, so a shift of three is made. The string qq matches, followed by a mismatch; z_v is undefined, but d'_v is defined to be four, so a shift of three is made, after which no further matches are found and the algorithm halts.

To see how $\mathcal{L}$ is used during the search, let T be $qxaxtqqpst$. The shifts of the $\mathcal{P}$ are shown in Figure 7.14.

7.17. APL16: Ziv–Lempel data compression

Large text or graphics files are often compressed in order to save storage space or to speed up transmission when the file is shipped. Most operating systems have compression utilities, and some file transfer programs automatically compress, ship, and uncompress the file, without user intervention. The field of text compression is itself the subject of several books (for example, see [423]) and will not be handled in depth here. However, a popular compression method due to Ziv–Lempel [487, 488] has an efficient implementation using suffix trees [382], providing another illustration of their utility.

The Ziv–Lempel compression method is widely used (it is the basis for the Unix utility *compress*), although there are actually several variants of the method that go by the same name (see [487] and [488]). In this section, we present a variant of the basic method and an efficient implementation of it using suffix trees.

Definition For any position i in a string S of length m, define the substring $Prior_i$ to be the longest prefix of $S[i..m]$ that also occurs as a substring of $S[1..i-1]$.

For example, if $S = abaxcabaxabz$ then $Prior_7$ is bax.

Definition For any position i in S, define l_i as the length of $Prior_i$. For $l_i > 0$, define s_i as the starting position of the left-most copy of $Prior_i$.

In the above example, $l_7 = 3$ and $s_7 = 2$.

Note that when $l_i > 0$, the copy of $Prior_i$ starting at s_i is totally contained in $S[1..i-1]$.

The Ziv–Lempel method uses some of the l_i and s_i values to construct a compressed representation of string S. The basic insight is that if the text $S[1..i-1]$ has been represented (perhaps in compressed form) and l_i is greater than zero, then the next l_i characters of S (substring $Prior_i$) need not be explicitly described. Rather, that substring can be described by the pair (s_i, l_i), pointing to an earlier occurrence of the substring. Following this insight, a compression method could process S left to right, outputting the pair (s_i, l_i) in place of the explicit substring $S[i..i+l_i-1]$ when possible, and outputting the character $S(i)$ when needed. Full details are given in the algorithm below.

Compression algorithm 1

```
begin
i := 1
Repeat
    compute l_i and s_i
    if l_i > 0 then
    begin
        output (s_i, l_i)
        i := i + l_i
    end
    else
    begin
        output S(i)
        i := i + 1
    end
Until i > n
end.
```

For example, $S = abacabaxabz$ can be described as $ab(1,1)c(1,3)x(1,2)z$. Of course, in this example the number of symbols used to represent S did not decrease, but rather increased! That's typical of small examples. But as the string length increases, providing more opportunity for repeating substrings, the compression improves. Moreover, the algorithm could choose to output character $S(i)$ explicitly whenever l_i is "small" (the actual rule depends on bit-level considerations determined by the size of the alphabet, etc.). For a small example where positive compression is observed, consider the contrived string $S = abababababababababababababab$, represented as $ab(1,2)(1,4)(1,8)(1,16)$. That representation uses 24 symbols in place of the original 32 symbols. If we extend this example to contain k repeated copies of ab, then the compressed representation contains approximately $5 \log_2 k$ symbols – a dramatic reduction in space.

To decompress a compressed string, process the compressed string left to right, so that any pair (s_i, l_i) in the representation points to a substring that has already been fully decompressed. That is, assume inductively that the first j terms (single characters or s, l pairs) of the compressed string have been processed, yielding characters 1 through $i-1$ of the original string S. The next term in the compressed string is either character $S(i+1)$, or it is a pair (s_i, l_i) pointing to a substring of S strictly before i. In either case, the algorithm has the information needed to decompress the jth term, and since the first

term in the compressed string is the first character of S, we conclude by induction that the decompression algorithm can obtain the original string S.

7.17.1. Implementation using suffix trees

The key implementation question is how to compute l_i and s_i each time the algorithm requests those values for a position i. The algorithm compresses S left to right and does not request (s_i, l_i) for any position i already in the compressed part of S. The compressed substrings are therefore nonoverlapping, and if each requested pair (s_i, l_i) can be found in $O(l_i)$ time, then the entire algorithm would run in $O(m)$ time. Using a suffix tree for S, the $O(l_i)$ time bound is easily achieved for any request.

Before beginning the compression, the algorithm first builds a suffix tree $\mathcal{T}$ for S and then numbers each node v with the number c_v. This number equals the smallest suffix (position) number of any leaf in v's subtree, and it gives the left-most starting position in S of any copy of the substring that labels the path from r to v. The tree can be built in $O(m)$ time, and all the node numbers can be obtained in $O(m)$ time by any standard tree traversal method (or bottom-up propagation).

When the algorithm needs to compute (s_i, l_i) for some position i, it traverses the unique path in $\mathcal{T}$ that matches a prefix of $S[i..m]$. The traversal ends at point p (not necessarily a node) when i equals the string-depth of point p plus the number c_v, where v is the first node at or below p. In either case, the path from the root to p describes the longest prefix of $S[i..m]$ that also occurs in $S[1..i]$. So, s_i equals c_v and l_i equals the string-depth of p. Exploiting the fact that the alphabet is fixed, the time to find (s_i, l_i) is $O(l_i)$. Thus the entire compression algorithm runs in $O(m)$ time.

7.17.2. A one-pass version

In the implementation above we assumed S to be known ahead of time and that a suffix tree for S could be built before compression begins. That works fine in many contexts, but the method can also be modified to operate on-line as S is being input, one character at a time. Essentially, the algorithm is implemented so that the compaction of S is interwoven with the construction of $\mathcal{T}$. The easiest way to see how to do this is with Ukkonen's linear-time suffix tree algorithm.

Ukkonen's algorithm builds implicit suffix trees on-line as characters are added to the right end of the growing string. Assume that the compaction has been done for $S[1..i-1]$ and that implicit suffix tree $\mathcal{I}_{i-1}$ for string $S[1..i-1]$ has been constructed. At that point, the compaction algorithm needs to know (s_i, l_i). It can obtain that pair in exactly the same way that is done in the above implementation *if* the c_v values have been written at each node v in $\mathcal{I}_{i-1}$. However, unlike the above implementation, which establishes those c_v values in a linear time traversal of $\mathcal{T}$, the algorithm cannot traverse each of the implicit suffix trees, since that would take more than linear time overall. Instead, whenever a new internal node v is created in Ukkonen's algorithm by splitting an edge (u, w), c_v is set to c_w, and whenever a new leaf v is created, c_v is just the suffix number associated with leaf v. In this way, only constant time is needed to update the c_v values when a new node is added to the tree. In summary, we have

Theorem 7.17.1. *Compression algorithm* 1 *can be implemented to run in linear time as a one-pass, on-line algorithm to compress any input string* S.

7.17.3. The real Ziv–Lempel

The compression scheme given in *Compression algorithm* 1 although not the actual Ziv–Lempel method, does resemble it and capture its spirit. The real Ziv–Lempel method is a one-pass algorithm whose output differs from the output of *Compression algorithm* 1 in that, whenever it outputs a pair (s_i, l_i), it then explicitly outputs $S(i+l_i)$, the character following the substring. For example, $S = abababababababababababababab$ would be compressed to $ab(1,2)a(2,4)b(1,10)a(2,12)b$, rather than as $ab(1,2)(1,4)(1,8)(1,16)$. The one-pass version of *Compression algorithm* 1 can trivially be converted to implement Ziv–Lempel in linear time.

It is not completely clear why the Ziv–Lempel algorithm outputs the extra character. Certainly for compaction purposes, this character is not needed and seems extraneous. One suggested reason for outputting an explicit character after each (s, l) pair is that $(s_i, l_i)S(i+l_i)$ defines the shortest substring starting at position i that does not appear anywhere earlier in the string, whereas (s_i, l_i) defines the longest substring starting at i that does appear earlier. Historically, it may have been easier to reason about shortest substrings that do not appear earlier in the string than to reason about longest substrings that do appear earlier.

7.18. APL17: Minimum length encoding of DNA

Recently, several molecular biology and computer science research groups have used the Ziv–Lempel method to compress DNA strings, not for the purpose of obtaining efficient storage, but rather to compute a measure of the "complexity" or "information content" of the strings [14, 146, 325, 326, 386]. Without fully defining the central technical terms "complexity", "information", "entropy", etc., we state the basic idea, which is that substrings of greatest biological significance should be more compressable than substrings that are essentially random. One expects that random strings will have too little structure to allow high compression, since high compression is based on finding repetitive segments in the string. Therefore, by searching for substrings that are more compressable than random strings, one may be able to find strings that have a definite biological function. (On the other hand, most repetitive DNA occurs outside of exons.)

Compression has also been used to study the "relatedness"[7] of two strings S_1 and S_2 of DNA [14, 324]. Essentially, the idea is to build a suffix tree for S_1 and then compress string S_2 using only the suffix tree for S_1. That compression of S_2 takes advantage of substrings in S_2 that appear in S_1, but does not take advantage of repeated substrings in S_2 alone. Similarly, S_1 can be compressed using only a suffix tree for S_2. These compressions reflect and estimate the "relatedness" of S_1 and S_2. If the two strings are highly related, then both computations should significantly compress the string at hand.

Another biological use for Ziv–Lempel–like algorithms involves estimating the "entropy" of short strings in order to discriminate between exons and introns in eukaryotic DNA [146]. Farach et al. [146] report that the average compression of introns does not differ significantly from the average compression of exons, and hence compression by itself does not distinguish exons from introns. However, they also report the following extension of that approach to be effective in distinguishing exons from introns.

[7] Other, more common ways to study the relatedness or similarity of strings of two strings are extensively discussed in Part III.

Definition For any position i in string S, let $ZL(i)$ denote the length of the longest substring beginning at i that appears somewhere in the string $S[1..i]$.

Definition Given a DNA string S partitioned into exons and introns, the *exon-average ZL value* is the average $ZL(i)$ taken over every position i in the exons of S. Similarly, the *intron-average ZL* is the average $ZL(i)$ taken over positions in introns of S.

It should be intuitive at this point that the exon-average ZL value and the intron-average ZL value can be computed in $O(n)$ time, by using suffix trees to compute all the $ZL(i)$ values. The technique, resembles the way matching statistics are computed, but is more involved since the substring starting at i must also appear to the left of position i.

The main empirical result of [146] is that the exon-average ZL value is lower than the intron-average ZL value by a statistically significant amount. That result is contrary to the expectation stated above that biologically significant substrings (exons in this case) should be more compressable than more random substrings (which introns are believed to be). Hence, the full biological significance of string compressability remains an open question.

7.19. Additional applications

Many additional applications of suffix trees appear in the exercises below, in Chapter 9, in Sections 12.2.4, 12.3, and 12.4, and in exercises of Chapter 14.

7.20. Exercises

1. Given a set S of k strings, we want to find every string in S that is a substring of some other string in S. Assuming that the total length of all the strings is n, give an $O(n)$-time algorithm to solve this problem. This result will be needed in algorithms for the shortest superstring problem (Section 16.17).
2. For a string S of length n, show how to compute the $N(i)$, $L(i)$, $L'(i)$ and sp_i values (discussed in Sections 2.2.4 and 2.3.2) in $O(n)$ time directly from a suffix tree for S.
3. We can define the suffix tree in terms of the keyword tree used in the Aho–Corasick (AC) algorithm. The input to the AC algorithm is a set of patterns $\mathcal{P}$, and the AC tree is a compact representation of those patterns. For a single string S we can think of the n suffixes of S as a set of patterns. Then one can build a suffix tree for S by first constructing the AC tree for those n patterns, and then compressing, into a single edge, any maximal path through nodes with only a single child. If we take this approach, what is the relationship between the failure links used in the keyword tree and the suffix links used in Ukkonen's algorithm? Why aren't suffix trees built in this way?
4. A suffix tree for a string S can be viewed as a keyword tree, where the strings in the keyword tree are the suffixes of S. In this way, a suffix tree is useful in efficiently building a keyword tree when the strings for the tree are only *implicitly* specified. Now consider the following implicitly specified set of strings: Given two strings S_1 and S_2, let D be the set of all substrings of S_1 that are not contained in S_2. Assuming the two strings are of length n, show how to construct a keyword tree for set D in $O(n)$ time. Next, build a keyword tree for D together with the set of substrings of S_2 that are not in S_1.
5. Suppose one has built a generalized suffix tree for a string S along with its suffix links (or link pointers). Show how to efficiently convert the suffix tree into an Aho–Corasick keyword tree.

6. Discuss the relative advantages of the Aho–Corasick method versus the use of suffix trees for the exact set matching problem, where the text is fixed and the set of patterns is varied over time. Consider preprocessing, search time, and space use. Consider both the cases when the text is larger than the set of patterns and vice versa.
7. In what way does the suffix tree more deeply expose the structure of a string compared to the Aho–Corasick keyword tree or the preprocessing done for the Knuth-Morris-Pratt or Boyer–Moore methods? That is, the sp values give some information about a string, but the suffix tree gives much more information about the structure of the string. Make this precise. Answer the same question about suffix trees and Z values.
8. Give an algorithm to take in a set of k strings and to find the longest common substring of each of the $\binom{k}{2}$ pairs of strings. Assume each string is of length n. Since the longest common substring of any pair can be found in $O(n)$ time, $O(k^2 n)$ time clearly suffices. Now suppose that the string lengths are different but sum to m. Show how to find all the longest common substrings in time $O(km)$. Now try for $O(m + k^2)$ (I don't know how to achieve this last bound).
9. The problem of finding substrings common to a set of distinct strings was discussed separately from the problem of finding substrings common to a single string, and the first problem seems much harder to solve than the second. Why can't the first problem just be reduced to the second by concatenating the strings in the set to form one large string?
10. By modifying the compaction algorithm and adding a little extra (linear space) information to the resulting DAG, it is possible to use the DAG to determine not only whether a pattern occurs in the text, but to find all the occurrences of the pattern. We illustrate the idea when there is only a single merge of nodes p and q. Assume that p has larger string depth than q and that u is the parent of p before the merge. During the merge, remove the subtree of p and put a displacement number of -1 on the new u to pq edge. Now suppose we search for a pattern P in the text and determine that P is in the text. Let i be a leaf below the path labeled P (i.e., below the termination point of the search). If the search traversed the u to pq edge, then P occurs starting at position $i - 1$; otherwise it occurs starting at position i.

    Generalize this idea and work out the details for any number of node merges.
11. In some applications it is desirable to know the number of times an input string P occurs in a larger string S. After the obvious linear-time preprocessing, queries of this sort can be answered in $O(|P|)$ time using a suffix tree. Show how to preprocess the DAG in linear time so that these queries can be answered in $O(|P|)$ time using a DAG.
12. Prove the correctness of the compaction algorithm for suffix trees.
13. Let S' be the reverse of the string S. Is there a relationship between the number of nodes in the DAG for S and the DAG for S'? Prove it. Find the relationship between the DAG for S and the DAG for S' (this relationship is a bit more direct than for suffix trees).
14. In Theorem 7.7.1 we gave an easily computed condition to determine when two subtrees of a suffix tree for string S are isomorphic. An alternative condition that is less useful for efficient computation is as follows: Let α be the substring labeling a node p and β be the substring labeling a node q in the suffix tree for S. The subtrees of p and q are isomorphic if and only if the set of positions in S where occurrences of α end equals the set of positions in S where occurrences of β end.

    Prove the correctness of this alternative condition for subtree isomorphism.
15. Does Theorem 7.7.1 still hold for a generalized suffix tree (for more than a single string)? If not, can it be easily extended to hold?
16. The DAG D for a string S can be converted to a finite-state machine by expanding each edge with more than one character on its label into a series of edges labeled by one character

each. This finite-state machine will recognize substrings of S, but it will not necessarily be the smallest such finite-state machine. Give an example of this.

We now consider how to build the smallest finite-state machine to recognize substrings of S. Again start with a suffix tree for S, merge isomorphic subtrees, and then expand each edge that it labeled with more than a single character. However, the merge operation must be done more carefully than before. Moreover, we imagine there is a suffix link from each leaf i to each leaf $i+1$, for $i < n$. Then, there is a path of suffix links connecting all the leaves, and each leaf has zero leaves beneath it. Hence, all the leaves will get merged.

Recall that Q is the set of all pairs (p, q) such that there exists a suffix link from p to q in T, where p and q have the same number of leaves in their respective subtrees. Suppose (p, q) is in Q. Let v be the parent of p, let γ be the label of the edge (v, p) into p, and let δ be the label of the edge into q. Explain why $|\gamma| \geq |\delta|$. Since every edge of the DAG will ultimately be expanded into a number of edges equal to the length of its edge-label, we want to make each edge-label as small as possible. Clearly, δ is a suffix of γ, and we will exploit this fact to better merge edge-labels. During a merge of p into q, remove all out edges from p as before, but the edge from v is not necessarily directed to q. Rather, if $|\delta| > 1$, then the δ edge is split into two edges by the introduction of a new node u. The first of these edges is labeled with the first character of δ and the second one, edge (u, q), is labeled with the remaining characters of δ. Then the edge from v is directed to u rather than to q. Edge (v, u) is labeled with the first $|\gamma| - |\delta| + 1$ characters of γ.

Using this modified merge, clean up the description of the entire compaction process and prove that the resulting DAG recognizes substrings of S. The finite-state machine for S is created by expanding each edge of this DAG labeled by more than a single character. Each node in the DAG is now a state in the finite-state machine.

17. Show that the finite-state machine created above has the fewest number of states of any finite-state machine that recognizes substrings of S. The key to this proof is that a deterministic finite-state machine has the fewest number of states if no state in it is equivalent to any other state. Two states are equivalent if, starting from them, exactly the same set of strings are accepted. See [228].

18. Suppose you already have the Aho–Corasick keyword tree (with backlinks). Can you use it to compute matching statistics in linear time, or if not, in some "reasonable" nonlinear time bound? Can it be used to solve the longest common substring problem in a reasonable time bound? If not, what is the difficulty?

19. In Section 7.16 we discussed how to use a suffix tree to search for all occurrences of a set of patterns in a given text. If the length of all the patterns is n and the length of the text is m, that method takes $O(n+m)$ time and $O(m)$ space. Another view of this is that the solution takes $O(m)$ preprocessing time and $O(n)$ search time. In contrast, the Aho–Corasick method solves the problem in the same total time bound but in $O(n)$ space. Also, it needs $O(n)$ preprocessing time and $O(m)$ search time.

Because there is no definite relationship between n and m, sometimes one method will use less space or preprocessing time than the other. By using a generalized suffix tree, for the set of patterns and the reverse role for suffix trees discussed in Section 7.8, it is possible to solve the problem with a suffix tree, obtaining exactly the same time and space bounds obtained by the Aho–Corasick method. Show in detail how this is done.

20. Using the reverse role for suffix trees discussed in Section 7.8, show how to solve the general DNA contamination problem of Section 7.5 using only a suffix tree for S_1, rather than a generalized suffix tree for S_1 together with all the possible contaminants.

21. In Section 7.8.1 we used a suffix tree for the small string P to compute the matching statistics $ms(i)$ for each position i in the long text string T. Now suppose we also want to compute matching statistics $ms(j)$ for each position j in P. Number $ms(j)$ is defined as the length of the longest substring starting at position j in P that matches some substring in T. We could proceed as before, but that would require a suffix tree for the long tree T. Show how to find all the matching statistics for both T and P in $O(|T|)$ time, using only a suffix tree for P.

22. In our discussion of matching statistics, we used the suffix links created by Ukkonen's algorithm. Suffix links can also be obtained by reversing the link pointers of Weiner's algorithm, but suppose that the tree cannot be modified. Can the matching statistics be computed in linear time using the tree and link pointers as given by Weiner's algorithm?

23. In Section 7.8 we discussed the reverse use of a suffix tree to solve the exact pattern matching problem: Find all occurrences of pattern P in text T. The solution there computed the matching statistic $ms(i)$ for each position i in the text. Here is a modification of that method that solves the exact matching problem but does not compute the matching statistics: Follow the details of the matching statistic algorithm but never examine new characters in the text unless you are on the path from the root to the leaf labeled 1. That is, in each iteration, do not proceed below the string $\alpha\gamma$ in the suffix tree, until you are on the path that leads to leaf 1. When not on this path, the algorithm just follows suffix links and performs skip/count operations until it gets back on the desired path.

Prove that this modification correctly solves the exact matching problem in linear time.

What advantages or disadvantages are there to this modified method compared to computing the matching statistics?

24. There is a simple practical improvement to the previous method. Let v be a point on the path to leaf 1 where some search ended, and let v' be the node on that path that was next entered by the algorithm (after some number of iterations that visit nodes off that path). Then, create a direct shortcut link from v to v'. The point is that if any future iteration ends at v, then the shortcut link can be taken to avoid the longer indirect route back to v'.

Prove that this improvement works (i.e., that the exact matching problem is correctly solved in this way).

What is the relationship of these shortcut links to the failure function used in the Knuth-Morris-Pratt method? When the suffix tree encodes more than a single pattern, what is the relationship of these shortcut links to the backpointers used by the Aho–Corasik method?

25. We might modify the previous method even further: In each iteration, only follow the suffix link (to the end of α) and do not do any skip/count operations or character comparisons unless you are on the path to leaf 1. At that point, do all the needed skip/count computations to skip past any part of the text that has already been examined.

Fill in the details of this idea and establish whether it correctly solves the exact matching problem in linear time.

26. Recall the discussion of STSs in Section 7.8.3, page 135. Show in more detail how matching statistics can be used to identify any STSs that a string contains, assuming there are "modest" number of errors in either the STS strings or the new string.

27. Given a set of k strings of length n each, find the longest common prefix for each pair of strings. The total time should be $O(kn + p)$, where p is the number of pairs of strings having a common prefix of length greater than zero. (This can be solved using the lowest common ancestor algorithm discussed later, but a simpler method is possible.)

28. For any pair of strings, we can compute the length of the longest prefix common to the pair in time linear in their total length. This is a simple use of a suffix tree. Now suppose we are given k strings of total length n and want to compute the minimum length of all the pairwise longest common prefixes over all of the $\binom{k}{2}$ pairs of strings, that is, the smallest

length of the pairwise pairs. The obvious way to solve this is to solve the longest common prefix problem for each of the $\binom{k}{2}$ pairs of strings in $O(k^2 + kn)$ time. Show how to solve the problem in $O(n)$ time independent of k. Consider also the problem of computing the maximum length over all the pairwise common prefixes.

29. Verify that the all-pairs suffix-prefix matching problem discussed in Section 7.10 can be solved in $O(km)$ time using any linear-time string matching method. That is, the $O(km)$ time bound does not require a suffix tree. Explain why the bound does not involve a term for k^2.

30. Consider again the all-pairs suffix-prefix matching problem. It is possible to solve the problem in the same time bound without an explicit tree traversal. First, build a generalized suffix tree $\mathcal{T}(S)$ for the set of k strings S (as before), and set up a vector V of length k. Then successively initialize vector V to contain all zeros, and match each string in the set through the tree. The match using any string S_j ends at the leaf labeled with suffix 1 of string S_j. During this walk for S_j, if a node v is encountered containing index i in its list $L(v)$, then write the string-depth of node v into position i of vector V. When the walk reaches the leaf for suffix 1 of S_j, $V(i)$, for each i, specifies the length of the longest suffix of S_i that matches a prefix of S_j.

Establish the worst-case time analysis of this method. Compare any advantages or disadvantages (in practical space and/or time) of this method compared to the tree traversal method discussed in Section 7.10. Then propose modifications to the tree traversal method that maintain all of its advantages and also correct for its disadvantages.

31. A substring α is called a *prefix repeat* of string S if α is a prefix of S and has the form $\beta\beta$ for some string β. Give a linear-time algorithm to find the longest prefix repeat of an input string S. This problem was one of Weiner's motivations for developing suffix trees.

Very frequently in the sequence analysis literature, methods aimed at finding interesting features in a biological sequence begin by cataloging certain substrings of a long string. These methods almost always pick a *fixed-length window*, and then find all the distinct strings of that fixed length. The result of this window or *q-gram* approach is of course influenced by the choice of the window length. In the following three exercises, we show how suffix trees avoid this problem, providing a natural and more effective extension of the window approach. See also Exercise 26 of Chapter 14.

32. There are $m^2/2$ substrings of a string T whose length is m. Some of those substrings are identical and so occur more than once in the string. Since there are $\Theta(m^2)$ substrings, we cannot count the number of times each appears in T in $O(m)$ time. However, using a suffix tree we can get an *implicit* representation of these numbers in $O(m)$ time. In particular, when any string P of length n is specified, the implicit representation should allow us to compute the frequency of P in T in $O(n)$ time. Show how to construct the implicit frequency representation and how to use it.

33. Show how to count the number of distinct substrings of a string T in $O(m)$ time, where the length of T is m. Show how to enumerate one copy of each distinct substring in time proportional to the length of all those strings.

34. One way to hunt for "interesting" sequences in a DNA sequence database is to look for substrings in the database that appear much more often than they would be predicted to appear by chance alone. This is done today and will become even more attractive when huge amounts of anonymous DNA sequences are available.

Assuming one has a statistical model to determine how likely any particular substring would occur by chance, and a threshold above which a substring is "interesting", show how to

efficiently find all interesting substrings in the database. If the database has total length m, then the method should take time $O(m)$ plus time proportional to the number of interesting substrings.

35. **(Smallest *k*-repeat)** Given a string S and a number k, we want to find the smallest substring of S that occurs in S exactly k times. Show how to solve this problem in linear time.

36. Theorem 7.12.1, which states that there can be at most n maximal repeats in a string of length n, was established by connecting maximal repeats with suffix trees. It seems there should be a direct, simple argument to establish this bound. Try to give such an argument. Recall that it is not true that at most one maximal repeat begins at any position in S.

37. Given two strings S_1 and S_2 we want to find all *maximal common pairs* of S_1 and S_2. A common substring C is maximal if the addition to C of any character on either the right or left of C results in a string that is not common to both S_1 and S_2. For example, if $A = aayxpt$ and $B = agyxpw$ then the string yxp is a maximal common substring, whereas yx is not. A *maximal common pair* is a triple (p_1, p_2, n'), where p_1 and p_2 are positions in S_1 and S_2, respectively, and n' is the length of a maximal common substring starting at those positions. This is a generalization of the maximal pair in a single string.

Letting m denote the total length of S_1 and S_2, give an $O(m + k)$-time solution to this problem, where k is the number of triples output. Give an $O(m)$-time method just to count the number of maximal common pairs and an $O(n + l)$-time algorithm to find one copy of each maximal common substring, where l is the total length of those strings. This is a generalization of the maximal repeat problem for a single string.

38. Another, equally efficient, but less concise way to identify supermaximal repeats is as follows: A maximal repeat in S represented by the left-diverse node v in the suffix tree for S is a supermaximal repeat if and only if no proper descendant of v is left diverse and no node in v's subtree (including v) is reachable via a path of suffix links from a left diverse node other than v. Prove this.

Show how to use the above claim to find all supermaximal repeats in linear time.

39. In biological applications, we are often not only interested in repeated substrings but in occurrences of substrings where one substring is an inverted copy of the other, a complemented copy, or (almost always) both. Show how to adapt all the definitions and techniques developed for repeats (maximal repeats, maximal pairs, supermaximal repeats, near-supermaximal repeats, common substrings) to handle inversion and complementation, in the same time bounds.

40. Give a linear-time algorithm that takes in a string S and finds the longest maximal pair in which the two copies do not overlap. That is, if the two copies begin at positions $p_1 < p_2$ and are of length n', then $p_1 + n' < p_2$.

41. Techniques for handling repeats in DNA are not only motivated by repetitive structures that occur in the DNA itself but also by repeats that occur in data collected from the DNA. The paper by Leung et al. [298] gives one example. In that paper they discuss a problem of analyzing DNA sequences from *E. coli*, where the data come from more than 1,000 independently sequenced fragments stored in an *E. coli* database. Since the sequences were contributed by independent sequencing efforts, some fragments contained others, some of the fragments overlapped others, and many intervals of the *E. coli* genome were yet unsequenced. Consequently, before the desired analysis was begun, the authors wanted to "clean up" the data at hand, finding redundantly sequenced regions of the *E. coli* genome and packaging all the available sequences into a few *contigs*, i.e., strings that contain all the substrings in the data base (these contigs may or may not be the shortest possible).

Using the techniques discussed for finding repeats, suffix-prefix overlaps, and so on, how

would you go about cleaning up the data and organizing the sequences into the desired contigs?

(This application is instructive because *E. coli*, as in most prokaryotic organisms, contains little repetitive DNA. However, that does not mean that techniques for handling repetitive structures have no application to prokaryotic organisms.)

Similar clean-up problems existed in the yeast genome database, where, in addition to the kinds of problems listed above, strings from other organisms were incorrectly listed as yeast, yeast strings were incorrectly identified in the larger composite databases, and parts of cloning vectors appeared in the reported yeast strings. To further complicate the problem, more recent higher quality sequencing of yeast may yield sequences that have one tenth of the errors than sequences in the existing databases. How the new and the old sequencing data should be integrated is an unsettled issue, but clearly, any large-scale curation of the yeast database will require the kinds of computational tools discussed here.

42. *k*-cover problem. Given two input strings S_1 and S_2 and a parameter k, a *k-cover* C is a set of substrings of S_1, each of length k or greater, such that S_2 can be expressed as the concatenation of the substrings of C in some order. Note that the substrings contained in C may overlap in S_1, but not in S_2. That is, S_2 is a permutation of substrings of S_1 that are each of length k or greater. Give a linear-time algorithm to find a k-cover from two strings S_1 and S_2, or determine that no such cover exists.

If there is no k-cover, then find a set of substrings of S_1, each of length k or greater, that cover the most characters of S_2. Or, find the largest $k' < k$ (if any) such that there is a k'-cover. Give linear-time algorithms for these problems.

Consider now the problem of finding *nonoverlapping* substrings in S_1, each of length k or greater, to cover S_2, or cover it as much as possible. This is a harder problem. Grapple with it as best you can.

43. exon shuffling. In eukaryotic organisms, a gene is composed of alternating *exons*, whose concatenation specifies a single protein, and *introns*, whose function is unclear. Similar exons are often seen in a variety of genes. Proteins are often built in a modular form, being composed of distinct domains (units that have distinct functions or distinct folds that are independent of the rest of the protein), and the same domains are seen in many different proteins, although in different orders and combinations. It is natural to wonder if exons correspond to individual protein domains, and there is some evidence to support this view. Hence modular protein construction may be reflected in the DNA by modular gene construction based on the reuse and reordering of stock exons. It is estimated that all proteins sequenced to date are made up of just a few thousand exons [468]. This phenomenon of reusing exons is called *exon shuffling*, and proteins created via exon shuffling are called *mosaic proteins*. These facts suggest the following general search problem.

The problem: Given anonymous, but sequenced, strings of DNA from protein-coding regions where the exons and introns are not known, try to identify the exons by finding common regions (ideally, identical substrings) in two or more DNA strings. Clearly, many of the techniques discussed in this chapter concerning common or repeated substrings could be applied, although they would have to be tried out on real data to test their utility or limitations. No elegant analytical result should be expected. In addition to methods for repeats and common substrings, does the *k*-cover problem seem of use in studying exon shuffling? That question will surely require an empirical, rather than theoretical answer. Although it may not give an elegant worst-case result, it may be helpful to first find all the maximal common substrings of length k or more.

44. Prove Lemma 7.14.1.

45. Prove the correctness of the method presented in Section 7.13 for the *circular string linearization problem*.

46. Consider in detail whether a suffix array can be used to efficiently solve the more complex string problems considered in this chapter. The goal is to maintain the space-efficient properties of the suffix array while achieving the time-efficient properties of the suffix tree. Therefore, it would be cheating to first use the suffix array for a string to construct a suffix tree for that string.

47. Give the details of the preprocessing needed to implement the bad character rule in the Boyer–Moore approach to exact set matching.

48. In Section 7.16.3, we used a suffix tree to implement a *weak* good suffix rule for a Boyer–Moore set matching algorithm. With that implementation, the increment of index i was determined in constant time after any test, independent even of the alphabet size. Extend the suffix tree approach to implement a *strong* good suffix rule, where again the increment to i can be found in constant time. Can you remove the dependence on the alphabet in this case?

49. Prove Theorem 7.16.2.

50. In the Ziv–Lempel algorithm, when computing (s_i, l_i) for some position i, why should the traversal end at point p if the string-depth of p plus c_v equals i? What would be the problem with letting the match extend past character i?

51. Try to give some explanation for why the Ziv–Lempel algorithm outputs the extra character compared to *compression algorithm* 1.

52. Show how to compute all the n values $ZL(i)$, defined in Section 7.18, in $O(n)$ time. One solution is related to the computation of matching statistics (Section 7.8.1).

53. Successive refinement methods

Successive refinement is a general algorithmic technique that has been used for a number of string problems [114, 199, 265]. In the next several exercises, we introduce the ideas, connect successive refinement to suffix trees, and apply successive refinement to particular string problems.

Let S be a string of length n. The relation E_k is defined on pairs of suffixes of S. We say $i\,E_k\,j$ if and only if suffix i and suffix j of S agree for at least their first k characters. Note that E_k is an equivalence relation and so it partitions the elements into equivalence classes. Also, since S has n characters, every class in E_n is a singleton. Verify the following two facts:

Fact 1 For any $i \neq j$, $i\,E_{k+1}\,j$ if and only if $i\,E_k\,j$ and $i+1\,E_k\,j+1$.

Fact 2 Every E_{k+1} class is a subset of an E_k class and so the E_{k+1} partition is a refinement of the E_k partition.

We use a labeled tree T, called the *refinement tree*, to represent the successive refinements of the classes of E_k as k increases from 0 to n. The root of T represents class E_0 and contains all the n suffixes of S. Each child of the root represents a class of E_1 and contains the elements in that class. In general, each node at level l represents a class of E_l and its children represent all the E_{l+1} classes that refine it.

What is the relationship of T to the keyword tree (Section 3.4) constructed from the set of n suffixes of S?

Now modify T as follows. If node v represents the same set of suffixes as its parent node v', contract v and v' to a single node. In the new refinement tree, T', each nonleaf node has at least two children. What is the relationship of T' to the suffix tree for string S? Show how to convert a suffix tree for S into tree T' in $O(n^2)$ time.

54. Several string algorithms use successive refinement without explicitly finding or representing all the classes in the refinement tree. Instead, they construct only some of the classes or only compute the tree implicitly. The advantage is reduced use of space in practice or an algorithm that is better suited for parallel computation [116]. The original suffix array construction method [308] is such an algorithm. In that algorithm, the suffix array is obtained as a byproduct of a successive refinement computation where the E_k partitions are computed only for values of k that are a power of two. We develop that method here. First we need an extension of Fact 1:

Fact 3 For any $i \neq j$, $i\ E_{2k}\ j$ if and only if $i\ E_k\ j$ and $i+k\ E_k\ j+k$.

From Fact 2, the classes of E_{2k} refine the classes of E_k.

The algorithm of [308] starts by computing the partition E_1. Each class of E_1 simply lists all the locations in S of one specific character in the alphabet, and the classes are arranged in lexical order of those characters. For example, for S =*mississippi*$, E_1 has five classes: {12}, {2, 5, 8, 11}, {1}, {9, 10}, and {3, 4, 6, 7}. The class {2, 5, 8, 11} lists the position of all the *i*'s in S and so comes before the class for the single *m*, which comes before the class for the *s*'s, etc. The end-of-string character $ is considered to be lexically smaller than any other character.

How E_1 is obtained in practice depends on the size of the alphabet and the manner that it is represented. It certainly can be obtained with $O(n \log n)$ character comparisons.

For any $k \geq 1$, we can obtain the E_{2k} partition by refining the E_k partition, as suggested in Fact 3. However, it is not clear how to efficiently implement a direct use of Fact 3. Instead, we create the E_{2k} partition in $O(n)$ time, using a *reverse* approach to refinement. Rather than examining a class C of E_k to find how C should be refined, we use C *as a refiner* to see how it forces other E_k classes to split, or to stay together, as follows: For each number $i > k$ in C, locate and mark number $i-k$. Then, for each E_k class A, any numbers in A marked by C identify a complete E_{2k} class. The correctness of this follows from Fact 3.

Give a complete proof of the correctness of the reverse refinement approach to creating the E_{2k} partition from the E_k partition.

55. Each class of E_k, for any k, holds the starting locations of a k-length substring of S. The algorithm in [308] constructs a suffix array for S using the reverse refinement approach, with the added detail that the classes of E_k are kept in the lexical order of the strings associated with the classes.

More specifically, to obtain the E_2 partition of S = *mississippi*$, process the classes of E_1 in order, from the lexically smallest to the lexically largest class. Processing the first class, {12}, results in the creation of the E_2 class {11}. The second E_1 class {2,5,8,11} marks indices {1,4,7} and {10}, and hence it creates the three E_2 classes {1},{4,7} and {10}. Class {9,10} of E_1 creates the two classes {8} and {9}. Class {3,4,6,7} of E_1 creates classes {2,5} and {3,6} of E_2. Each class of E_2 holds the starting locations of identical substrings of length one or two. These classes, lexically ordered by the substrings they represent, are: {12},{11},{8},{2,5},{1},{10},{9},{4,7},{3,6}. The classes of E_4, in lexical order are: {12},{11},{8},{2,5},{1},{10},{9},{7},{4},{6},{3}. Note that {2,5} remain in the same E_4 class because {4,7} were in the same E_2 class. The E_2 classes of {4,7} and {3,6} are each refined in E_4. Explain why.

Although the general idea of reverse refinement should now be clear, efficient implementation requires a number of additional details. Give complete implementation details and analysis, proving that the E_{2k} classes can be obtained from the E_k classes in $O(n)$ time. Be sure to detail how the classes are kept in lexical order.

Assume n is a power of two. Note that the algorithm can stop as soon as every class is a singleton, and this must happen within $\log_2 n$ iterations. When the algorithm ends, the order of the (singleton) classes describes a permutation of the integers 1 to n. Prove that this permutation is the suffix array for string S. Conclude that the reverse refinement method creates a suffix array in $O(n \log n)$ time. What is the space advantage of this method over the $O(n)$-time method detailed in Section 7.14.1?

56. Primitive tandem arrays

Recall that a string α is called a *tandem array* if α is periodic (see Section 3.2.1), i.e., it can be written as β^l for some $l \geq 2$. When $l = 2$, the tandem array can also be called a *tandem repeat*. A tandem array $\alpha = \beta^l$ contained in a string S is called *maximal* if there are no additional copies of β before or after α.

Maximal tandem arrays were initially defined in Exercise 4 in Chapter 1 (page 13) and the importance of tandem arrays and repeats was discussed in Section 7.11.1. We are interested in identifying the maximal tandem arrays contained in a string. As discussed before, it is often best to focus on a structured subset of the strings of interest in order to limit the size of the output and to identify the most informative members. We focus here on a subset of the maximal tandem arrays that succinctly and implicitly encode all the maximal tandem arrays. (In Section 9.5, 9.6, and 9.6.1 we will discuss efficient methods to find *all* the tandem repeats in a string, and we allow the repeats to contain some errors.)

We use the pair (β, l) to describe the tandem array β^l. Now consider the tandem array α = *abababababababab*. It can be described by the pair (*ababab*, 2), or by (*abab*, 4), or by (*ab*, 8). Which description is best? Since the first two pairs can be deduced from the last, we choose the later pair. This "choice" will now be precisely defined.

A string β is said to be *primitive* if β is not periodic. For example, the string *ab* is primitive, whereas *abab* is not. The pair (*ab*, 8) is the preferred description of *abababababababab* because string *ab* is primitive. The preference for primitive strings extends naturally to the description of maximal tandem arrays that occur as substrings in larger strings. Given a string S, we use the triple (i, β, l) to mean that a tandem array (β, l) occurs in S starting at position i. A triple (i, β, l) is called a *pm-triple* if β is primitive and β^l is a maximal tandem array.

For example, the maximal tandem arrays in *mississippi* described by the pm-triples are {2,*iss*,2),(3,*s*,2),(3,*ssi*,2),(6,*s*,2) and (9,*p*,2). Note that two or more pm-triples can have the same first number, since two different maximal tandem arrays can begin at the same position. For example, the two maximal tandem arrays *ss* and *ssissi* both begin at position three of *mississippi*.

The pm-triples succinctly encode all the tandem arrays in a given string S. Crochemore [114] (with different terminology) used a successive refinement method to find all the pm-triples in $O(n \log n)$ time. This implies the very nontrivial fact that in any string of length n there can be only $O(n \log n)$ pm-triples. The method in [114] finds the E_k partition for each k. The following lemma is central:

Lemma 7.20.1. *There is a tandem repeat of a k-length substring β starting at position i of S if and only if the numbers i and $i+k$ are both contained in a single class of E_k and no numbers between i and $i+k$ are in that class.*

Prove Lemma 7.20.1. One direction is easy. The other direction is harder and it may be useful to use Lemma 3.2.1 (page 40).

57. Lemma 7.20.1 makes it easy to identify pm-triples. Assume that the indices in each class of E_k are sorted in increasing order. Lemma 7.20.1 implies that (i, β, l) is a pm-triple, where β is a k-length substring, if and only if some single class of E_k contains a maximal series of numbers $i, i+k, i+2k, \ldots, i+lk$, such that each consecutive pair of numbers differs by k. Explain this in detail.

By using Fact 1 in place of Fact 3, and by modifying the reverse refinement method developed in Exercises 54 and 55, show how to compute all the E_k partitions for all k (not just the powers of two) in $O(n^2)$ time. Give implementation details to maintain the indices of each class sorted in increasing order. Next, extend that method, using Lemma 7.20.1, to obtain an $O(n^2)$-time algorithm to find all the pm-triples in a string S.

58. To find all the pm-triples in $O(n \log n)$ time, Crochemore [114] used one additional idea. To introduce the idea, suppose all E_k classes except one, C say, have been used as refiners to create E_{k+1} from E_k. Let p and q be two indices that are together in some E_k class. We claim that if p and q are not together in the same E_{k+1} class, then one of them (at least) has already been placed in its proper E_{k+1} class. The reason is that by Fact 1, $p+1$ and $q+1$ cannot both be in the same E_k class. So by the time C is used as a refiner, either p or q has been marked and moved by an E_k class already used as refiners.

Now suppose that each E_k class is held in a linked list and that when a refiner identifies a number, p say, then p is removed from its current linked list and placed in the linked list for the appropriate E_{k+1} class. With that detail, if the algorithm has used all the E_k classes except C as refiners, then all the E_{k+1} classes are correctly represented by the newly created linked lists plus what remains of the original linked lists for E_k. Explain this in detail. Conclude that one E_k class need not be used as a refiner.

Being able to skip one class while refining E_k is certainly desirable, but it isn't enough to produce the stated bound. To do that we have to repeat the idea on a larger scale.

Theorem 7.20.1. *When refining E_k to create E_{k+1}, suppose that for every $k > 1$, exactly one (arbitrary) child of each E_{k-1} class is skipped (i.e., not used as a refiner). Then the resulting linked lists correctly identify the E_{k+1} classes.*

Prove Theorem 7.20.1. Note that Theorem 7.20.1 allows complete freedom in choosing which child of an E_{k-1} class to skip. This leads to the following:

Theorem 7.20.2. *If, for every $k > 1$, the largest child of each E_{k-1} class is skipped, then the total size of all the classes used as refiners is at most $n \log_2 n$.*

Prove Theorem 7.20.2. Now provide all the implementation details to find all the pm-triples in S in $O(n \log n)$ time.

59. Above, we established the bound of $O(n \log n)$ pm-triples as a byproduct of the algorithm to find them. But a direct, nonalgorithmic proof is possible, still using the idea of successive refinement and Lemma 7.20.1. In fact, the bound of $3n \log_2 n$ is fairly easy to obtain in this way. Do it.

60. Folklore has it that for any position i in S, if there are two pm-triples, (i,β,l), and (i,β',l'), and if $|\beta'| > |\beta|$, then $|\beta'| \geq 2|\beta|$. That would limit the number of pm-triples with the same first number to $\log_2 n$, and the $O(n \log n)$ bound would be immediate.

Show by example that the folklore belief is false.

61. Primer selection problem

Let $\mathcal{S}$ be a set of strings over some finite alphabet Σ. Give an algorithm (using a generalized suffix tree) to find a *shortest* string S over Σ that is a substring in *none* of the strings of $\mathcal{S}$. The algorithm should run in time proportional to the sum of the lengths of the strings in $\mathcal{S}$.

A more useful version of the problem is to find the shortest string S that is longer than a certain minimum length and is not a substring of any string of $\mathcal{S}$. Often, a string α is given along with the set $\mathcal{S}$. Now the problem becomes one of finding a shortest substring of α (if any) that does not appear as a substring of any string in $\mathcal{S}$. More generally, for every i, compute the shortest substring (if any) that begins at position i of α and does not appear as a substring of any string in $\mathcal{S}$.

The above problems can be generalized in many different directions and solved in essentially the same way. One particular generalization is the exact matching version of the *primer selection problem*. (In Section 12.2.5 we will consider a version of this problem that allows errors.)

The primer selection problem arises frequently in molecular biology. One such situation is in "chromosome walking", a technique used in some DNA sequencing methods or gene location problems. Chromosome walking was used extensively in the location of the Cystic Fibrosis gene on human chromosome 7. We discuss here only the DNA sequencing application.

In DNA sequencing, the goal is to determine the complete nucleotide sequence of a long string of DNA. To understand the application you have to know two things about existing sequencing technology. First, current common laboratory methods can only accurately sequence a small number of nucleotides, from 300 to 500, from one end of a longer string. Second, it is possible to replicate substrings of a DNA string starting at almost any point as long as you know a small number of the nucleotides, say nine, to the left of that point. This replication is done using a technology called *polymerase chain reaction* (*PCR*), which has had a tremendous impact on experimental molecular biology. Knowing as few as nine nucleotides allows one to synthesize a string that is complementary to those nine nucleotides. This complementary string can be used to create a "primer", which finds its way to the point in the long string containing the complement of the primer. It then hybridizes with the longer string at that point. This creates the conditions that allow the replication of part of the original string to the right of the primer site. (Usually PCR is done with two primers, one for each end, but here only one "variable" primer is used. The other primer is fixed and can be ignored in this discussion.)

The above two facts suggest a method to sequence a long string of DNA, assuming we know the first nine nucleotides at the very start of the string. After sequencing the first 300 (say) nucleotides, synthesize a primer complementary to the last nine nucleotides just sequenced. Then replicate a string containing the next 300 nucleotides, sequence that substring and continue. Hence the longer string gets sequenced by successively sequencing 300 nucleotides at a time, using the end of each sequenced substring to create the primer that initiates sequencing of the next substring. Compared to the shotgun sequencing method (to be discussed in Section 16.14), this *directed* method requires much less sequencing overall, but because it is an inherently sequential process it takes longer to sequence a long DNA string. (In the Cystic Fibrosis case another idea, called *gene jumping*, was used to partially parallelize this sequential process, but chromosome walking is generally laboriously sequential.)

There is a common problem with the above chromosome walking approach. What happens if the string consisting of the last nine nucleotides appears in another place in the larger string? Then the primer may not hybridize in the correct position and any sequence determined from that point would be incorrect. Since we know the sequence to the left of our current point, we can check the known sequence to see if a string complementary to the primer exists to the left. If it does, then we want to find a nine-length substring near the end of the last determined sequence that does not appear anywhere earlier. That substring can then by used to form the primer. The result will be that the next substring sequenced will resequence some known nucleotides and so sequence somewhat fewer than 300 new nucleotides.

Problem: Formalize this primer selection problem and show how to solve it efficiently using suffix trees. More generally, for each position i in string α find the shortest substring that begins at i and that appears nowhere else in α or $\mathcal{S}$.

62. In the primer selection problem, the goal of avoiding incorrect hybridizations to the *right* of the sequenced part of the string is more difficult since we don't yet know the sequence. Still, there are some known sequences that should be avoided. As discussed in Section 7.11.1, eukaryotic DNA frequently contains regions of repeated substrings, and the most commonly occurring substrings are known. On the problem that repeated substrings cause for chromosome walking, R. Weinberg[8] writes:

> They were like quicksand; anyone treading on them would be sucked in and then propelled, like Alice in Wonderland, through some vast subterranean tunnel system, only to resurface somewhere else in the genome, miles away from the starting site. The genome was riddled with these sinkholes, called "repeated sequences." They were guaranteed to slow any chromosomal walk to a crawl.

So a more general primer problem is the following: Given a substring α of 300 nucleotides (the last substring sequenced), a string β of known sequence (the part of the long string to the left of α whose sequence is known), and a set $\mathcal{S}$ of strings (the common parts of known repetitive DNA strings), find the furthest right substring in α of length nine that is not a substring of β or any string in set $\mathcal{S}$. If there is no such string, then we might seek a string of length larger than nine that does not appear in β or $\mathcal{S}$. However, a primer much larger than nine nucleotides long may falsely hybridize for other reasons. So one must balance the constraints of keeping the primer length in a certain range, making it unique, and placing it as far right as possible.

Problem: Formalize this version of the primer selection problem and show how to apply suffix trees to it.

Probe selection

A variant of the primer selection problem is the *hybridization probe* selection problem. In DNA fingerprinting and mapping (discussed in Chapter 16) there is frequent need to see which *oligomers* (short pieces of DNA) hybridize to some target piece of DNA. The purpose of the hybridization is not to create a primer for PCR but to extract some information about the target DNA. In such mapping and fingerprinting efforts, contamination of the target DNA by vector DNA is common, in which case the oligo probe may hybridize with the vector DNA instead of the target DNA. One approach to this problem is to use specifically designed oligomers whose sequences are rarely in the genome of the vector, but are frequently found in the cloned DNA of interest. This is precisely the primer (or probe) selection problem.

In some ways, the probe selection problem is a better fit than the primer problem is to the exact matching techniques discussed in this chapter. This is because when designing probes for mapping, it is desirable and feasible to design probes so that even a single mismatch will destroy the hybrization. Such stringent probes can be created under certain conditions [134, 177].

[8] Racing to the Beginning of the Road; The Search for the Origin of Cancer, Harmony Books, 1996.

8

Constant-Time Lowest Common Ancestor Retrieval

8.1. Introduction

We now begin the discussion of an amazing result that greatly extends the usefulness of suffix trees (in addition to many other applications).

Definition In a rooted tree $\mathcal{T}$, a node u is an *ancestor* of a node v if u is on the unique path from the root to v. With this definition a node is an ancestor of itself. A *proper ancestor* of v refers to an ancestor that is not v.

Definition In a rooted tree $\mathcal{T}$, the *lowest common ancestor* (*lca*) of two nodes x and y is the deepest node in $\mathcal{T}$ that is an ancestor of both x and y.

For example, in Figure 8.1 the *lca* of nodes 6 and 10 is node 5 while the *lca* of 6 and 3 is 1.

The amazing result is that after a *linear* amount of preprocessing of a rooted tree, any two nodes can then be specified and their lowest common ancestor found in *constant* time. That is, a rooted tree with n nodes is first preprocessed in $O(n)$ time, and thereafter any lowest common ancestor query takes only constant time to solve, *independent* of n. Without preprocessing, the best worst-case time bound for a single query is $\Theta(n)$, so this is a most surprising and useful result. The *lca* result was first obtained by Harel and Tarjan [214] and later simplified by Schieber and Vishkin [393]. The exposition here is based on the later approach.

8.1.1. What do ancestors have to do with strings?

The constant-time search result is particularly useful in situations where many lowest common ancestor queries must be answered for a fixed tree. That situation often arises when applying the result to string problems. To get a feel for the connection between strings and common ancestors, note that if the paths from the root to two leaves i and j in a suffix tree are identical down to a node v, then suffix i and suffix j share a common prefix consisting of the string labeling the path to v. Hence the lowest common ancestor of leaves i and j identifies the longest common prefix of suffixes i and j. The ability to find such a longest common prefix will be an important primitive in many string problems. Some of these will be detailed in Chapter 9. That chapter can be read by taking the *lca* result as a black box, if the reader prefers to review motivating examples before diving into the technical details of the *lca* result.

The statement of the *lca* result is widely known, but the details are not. The result is usually taken as a black box in the string literature, and there is a general "folk" belief that the result is only theoretical, not practical. This is certainly not true of the Schieber–Vishkin method – it is a very practical, low-overhead method, which is simple to program and very fast in practice. It should definitely be in the standard repertoire of string packages.

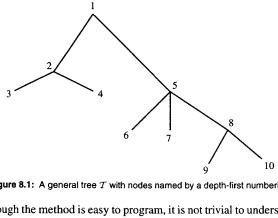


Figure 8.1: A general tree $\mathcal{T}$ with nodes named by a depth-first numbering.

However, although the method is easy to program, it is not trivial to understand at first and has been described as based on "bit magic". Nonetheless, the result has been so heavily applied in many diverse string methods, and its use is so critical in those methods, that a detailed discussion of the result is worthwhile. We hope the following exposition is a significant step toward making the method more widely understood.

8.2. The assumed machine model

Because constant retrieval time is such a demanding goal (and even linear preprocessing time requires careful attention to detail), we must be clear on what computational model is being used. Otherwise we may be accused of cheating – using an overly powerful set of primitive operations. What primitive operations are permitted in constant time? In the unit-cost RAM model when the input tree has n nodes, we certainly can allow any number with up to $O(\log n)$ bits to be written, read, or used as an address in constant time. Also, two numbers with up to $O(\log n)$ bits can be compared, added, subtracted, multiplied, or divided in constant time. Numbers that take more than $\Theta(\log n)$ bits to represent cannot be operated on in constant time. These are standard unit-cost requirements, forbidding "large" numbers from being manipulated in constant time. The *lca* result (that *lca* queries can be answered in constant time after linear-time preprocessing) can be proved in this unit-cost RAM model. However, the exposition is easier if we assume that certain additional *bit-level* operations can also be done in constant time, as long as the numbers have only $O(\log n)$ bits. In particular, we assume that the **AND**, **OR**, and **XOR** (exclusive or) of two (appropriate sized) binary numbers can be done in constant time; that a binary number can be shifted (left or right) by up to $O(\log n)$ bits in constant time; that a "mask" of consecutive 1-bits can be created in constant time; and that the position of the left-most or right-most 1-bit in a binary number can be found in constant time. On many machines, and with several high-level programming languages, these are reasonable assumptions, again assuming that the numbers involved are never more than $O(\log n)$ bits long. But for the purists, after we explain the *lca* result using these more liberal assumptions, we will explain how to achieve the same results using only the standard unit-cost RAM model.

8.3. Complete binary trees: a very simple case

We begin the discussion of the *lca* result on a particularly simple tree whose nodes are named in a very special way. The tree is a *complete binary tree* whose nodes have names

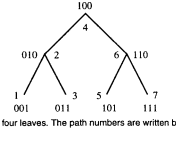


Figure 8.2: A binary tree with four leaves. The path numbers are written both in binary and in base ten.

that encode paths to them. The notation $\mathcal{B}$ will refer to this complete binary tree, and $\mathcal{T}$ will refer to an arbitrary tree.

Suppose that $\mathcal{B}$ is a *rooted complete binary tree* with p leaves ($n = 2p - 1$ nodes in total), so that every internal node has exactly two children and the number of edges on the path from the root to any leaf in $\mathcal{B}$ is $d = \log_2 p$. That is, the tree is complete and all leaves are at the same depth from the root. Each node v of $\mathcal{B}$ is assigned a $d + 1$ bit number, called its *path number*, that encodes the unique path from the root to v. Counting from the left-most bit, the ith bit of the path number for v corresponds to the ith edge on the path from the root to v: A 0 for the ith bit from the left indicates that the ith edge on the path goes to a left child, and a 1 indicates a right child.[1] For example, a path that goes left twice, right once, and then left again ends at a node whose path number begins (on the left) with 0010. The bits that describe the path are called *path bits*. Each path number is then padded out to $d+1$ bits by adding a 1 to the right of the path bits followed by as many additional 0s as needed to make $d + 1$ bits. Thus for example, if $d = 6$, the node with path bits 0010 is named by the 7-bit number 0010100. The root node for $d = 6$ would be 1000000. In fact, the root node always has a number with left bit 1 followed by d 0s. (See Figure 8.2 for an additional example.) We will refer to nodes in $\mathcal{B}$ by their path numbers.

As the tree in Figure 8.2 suggests, path numbers have another well-known description – that of *inorder* numbers. That is, when the nodes of $\mathcal{B}$ are numbered by an inorder traversal (recursively number the left child, number the root, and then recursively number the right child), the resulting node numbers are exactly the path numbers discussed above. We leave the proof of this for the reader (it has little significance in our exposition). The path number concept is preferred since it explicitly relates the number of a node to the description of the path to it from the root.

8.4. How to solve *lca* queries in $\mathcal{B}$

Definition For any two nodes i and j, we let *lca(i,j)* denote the least common ancestor of i and j.

Given two nodes i and j, we want to find *lca(i,j)* in $\mathcal{B}$ (remembering that both i and j are path numbers). First, when *lca(i,j)* is either i or j (i.e., one of these two nodes is an ancestor of the other), then this can be detected by a very simple constant-time algorithm, discussed in Exercise 3. So assume that *lca(i,j)* is neither i nor j. The algorithm begins by taking the *exclusive or* (**XOR**) of the binary number for i and the binary number for j, denoting the result by x_j. The **XOR** of two bits is 1 if and only if the two bits are different, and the **XOR** of two $d + 1$ bit numbers is obtained by independently taking the **XOR** of

[1] Note that normally when discussing binary numbers, the bits are numbered from right (least significant) to left (most significant). This is opposite the left-to-right ordering used for strings and for path numbers.

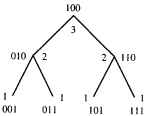


Figure 8.3: A binary tree with four leaves. The path numbers are in binary, and the position of the least-significant 1-bit is given in base ten.

each bit of the two numbers. For example, **XOR** of 00101 and 10011 is 10110. Since i and j are $O(\log n)$ bits long, **XOR** is a constant-time operation in our model.

The algorithm next finds the most significant (left-most) 1-bit in x_{ij}. If the left most 1-bit in the **XOR** of i and j is in position k (counting from the left), then the left most $k-1$ bits of i and j are the same, and the paths to i and j must agree for the first $k-1$ edges and then diverge. It follows that the path number for *lca(i,j)* consists of the left most $k-1$ bits of i (or j) followed by a 1-bit followed by $d+1-k$ zeros. For example, in Figure 8.2, the **XOR** of 101 and 111 (nodes 5 and 7) is 010, so their respective paths share one edge – the right edge out of the root. The **XOR** of 010 and 101 (nodes 2 and 5) is 111, so the paths to 2 and 5 have no agreement, and hence 100, the root, is their lowest common ancestor.

Therefore, to find *lca(i,j)*, the algorithm must **XOR** two numbers, find the left-most 1-bit in the result (say at position k), shift i right by $d+1-k$ places, set the right most bit to a 1, and shift it back left by $d+1-k$ places. By assumption, each of these operations can be done in constant time, and hence the lowest common ancestor of i and j can be found in constant time in $\mathcal{B}$.

In summary, we have

Theorem 8.4.1. *In a complete binary tree, after linear-time preprocessing to name nodes by their path numbers, any lowest common ancestor query can be answered in constant time.*

This simple case of a complete binary tree is very special, but it is presented both to develop intuition and because complete binary trees are used in the description of the general case. Moreover, by actually using complete binary trees, a very elegant and relatively simple algorithm can answer *lca* queries in constant time, if $\Theta(n \log n)$ time is allowed for preprocessing $\mathcal{T}$ and $\Theta(n \log n)$ space is available after the preprocessing. That method is explored in Exercise 12.

The *lca* algorithm we will present for general trees builds on the case of a complete binary tree. The idea (conceptually) is to *map* the nodes of a general tree $\mathcal{T}$ to the nodes of a complete binary tree $\mathcal{B}$ in such a way that *lca* retrievals on $\mathcal{B}$ will help to quickly solve *lca* queries on $\mathcal{T}$. We first describe the general *lca* algorithm assuming that the $\mathcal{T}$ to $\mathcal{B}$ mapping is explicitly used, and then we explain how explicit mapping can be avoided.

8.5. First steps in mapping $\mathcal{T}$ to $\mathcal{B}$

The first thing the preprocessing does is traverse $\mathcal{T}$ in a depth-first manner, numbering the nodes of v in the order that they are first encountered in the traversal. This is the

standard *depth-first numbering* (preorder numbering) of nodes (see Figure 8.1). With this numbering scheme, the nodes in the subtree of any node v in $\mathcal{T}$ have consecutive depth-first numbers, beginning with the number for v. That is, if there are q nodes in the subtree rooted at v, and v gets numbered k, then the numbers given to the other nodes in the subtree are $k+1$ through $k+q-1$.

For convenience, from this point on the nodes in $\mathcal{T}$ will be referred to by their depth-first numbers. That is, when we refer to node v, v is both a node and a number. Be careful not to confuse depth-first numbers used for the general tree $\mathcal{T}$ with path numbers used only for the binary tree $\mathcal{B}$.

Definition For any number k, $h(k)$ denotes the position (counting from the right) of the least-significant 1-bit in the binary representation of k.

For example, $h(8) = 4$ since 8 in binary is 1000, and $h(5) = 1$ since 5 in binary is 101. Another way to think of this is that $h(k)$ is one plus the number of consecutive zeros at the right end of k.

Definition In a complete binary tree the *height* of a node is the number of nodes on the path from it to a leaf. The height of a leaf is one.

The following lemma states a crucial fact that is easy to prove by induction on the height of the nodes.

Lemma 8.5.1. *For any node k (node with path number k) in $\mathcal{B}$, $h(k)$ equals the height of node k in $\mathcal{B}$.*

For example, node 8 (binary 1000) is at height 4, and the path from it to a leaf has four nodes (three edges).

Definition For a node v of $\mathcal{T}$, let $I(v)$ be a node w in $\mathcal{T}$ such that $h(w)$ is maximum over all nodes in the subtree of v (including v itself).

That is, over all the nodes in the subtree of v, $I(v)$ is a node (depth-first number) whose binary representation has the largest number of consecutive zeros at its right end. Figure 8.4 shows the node numbers from Figure 8.1 in binary and base 10. Then $I(1)$, $I(5)$, and $I(8)$ are all 8, $I(2)$ and $I(4)$ are both 4, and $I(v) = v$ for every other node in the figure.

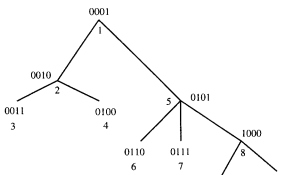


Figure 8.4: Node numbers given in four-bit binary, to illustrate the definition of $I(v)$.

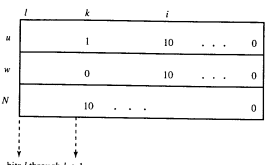


Figure 8.5: Numbers *u*, *w*, and *N*.

Clearly, if node v is an ancestor of a node w then $h(I(v)) \geq h(I(w))$. Another way to say this is that the $h(I(v))$ values never decrease along any upward path in $\mathcal{T}$. This fact will be important in several of the proofs below.

In the tree in Figure 8.4, node $I(v)$ is uniquely determined for each node v. That is, for each node v there is exactly one w in v's subtree such that $h(w)$ is maximum. This is no accident, and it will be important in the *lca* algorithm. We now prove this fact.

Lemma 8.5.2. *For any node v in $\mathcal{T}$, there is a unique node w in the subtree of v such that $h(w)$ is maximum over all nodes in v's subtree.*

PROOF Suppose not, and let u and w be two nodes in the subtree of v such that $h(u) = h(w) \geq h(q)$ for every node q in that subtree. Assume $h(u) = i$. By adding zeros to the left ends if needed, we can consider the two numbers u and w to have the same number of bits, say l. Since $u \neq w$, those two numbers must differ in some bit to the left of i (since by assumption bit i is 1 in both u and w, and all bits to the right of i are zero in both). Assume $u > w$, and let k be the left-most position where such a difference between u and w occurs. Consider the number N composed of the left-most $l - k$ bits of u followed by a 1 in bit k followed by $k - 1$ zeros (see Figure 8.5). Then N is strictly less than u and greater than w. Hence N must be the depth-first number given to some node in the subtree of v, because the depth-first numbers given to nodes below v form a consecutive interval. But $h(N) = k > i = h(u)$, contradicting the fact that $h(u) \geq h(q)$ for all nodes in the subtree of v. Hence the assumption that $h(u) = h(w)$ leads to a contradiction, and the Lemma is proved. □

The uniqueness of $I(v)$ for each v can be summarized by the following corollary:

Corollary 8.5.1. The function $v \rightarrow I(v)$ is well defined.

8.6. The mapping of $\mathcal{T}$ to $\mathcal{B}$

In mapping nodes of $\mathcal{T}$ to nodes of a binary tree $\mathcal{B}$, we want to preserve enough of the ancestry relations in $\mathcal{T}$ so that *lca* relations in $\mathcal{B}$ can be used to determine *lca* queries in $\mathcal{T}$. The function $v \rightarrow I(v)$ will be central in defining that mapping. As a first step in understanding the mapping, we partition the nodes of $\mathcal{T}$ into classes of nodes whose I value is the same.

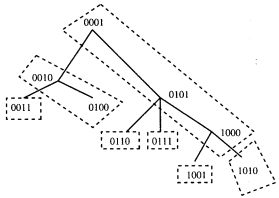


Figure 8.6: The partition of the nodes into seven runs.

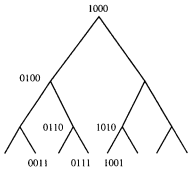


Figure 8.7: A node v in $\mathcal{B}$ is numbered if there is a node in $\mathcal{T}$ that maps to v.

Definition A *run* in $\mathcal{T}$ is a maximal subset of nodes of $\mathcal{T}$, all of which have the same I value.

That is, two nodes u and v are in the same run if and only if $I(u) = I(v)$. Figure 8.6 shows a partition of the nodes of $\mathcal{T}$ into runs.

Algorithmically we can set $I(v)$, for all nodes, using a linear-time bottom-up traversal of $\mathcal{T}$ as follows: For every leaf v, $I(v) = v$. For every internal node v, $I(v) = v$ if $h(v)$ is greater than $h(v')$ for every child v' of v. Otherwise, $I(v)$ is set to the $I(v')$ value of the child v' whose $h(I(v'))$ value is the maximum over all children of v. The result is that each run forms an upward path of nodes in $\mathcal{T}$. And, since the $h(I())$ values never decrease along any upward path in $\mathcal{T}$, it follows that

Lemma 8.6.1. *For any node v, node $I(v)$ is the deepest node in the run containing node v.*

These facts are illustrated in Figure 8.6.

Definition Define the *head* of a run to be the node of the run closest to the root.

For example, in Figure 8.6 node 1 (0001) is the head of a run of length three, node 2 (0010) is the head of a run of length two, and every remaining node (not in either of those two runs) is the head of a run consisting only of itself.

Finally, we can define the *tree map*.

Definition The *tree map* is the mapping of nodes of $\mathcal{T}$ to nodes of a complete binary tree $\mathcal{B}$ with depth $d = \lceil \log n \rceil - 1$. In particular, node v of $\mathcal{T}$ maps to node $I(v)$ of $\mathcal{B}$ (recall that nodes of $\mathcal{B}$ are named by their path numbers).

The tree map is well defined because $I(v)$ is a $d+1$ bit number, and each node of $\mathcal{B}$ is named by a distinct $d+1$ bit number. Every node in a run of $\mathcal{T}$ maps to the same node in $\mathcal{B}$, but not all nodes in $\mathcal{B}$ generally have nodes in $\mathcal{T}$ mapping to them. Figure 8.7 shows tree $\mathcal{B}$ for tree $\mathcal{T}$ from Figure 8.6. A node v in $\mathcal{B}$ is numbered if there is a node in $\mathcal{T}$ that maps to v.

8.7. The linear-time preprocessing of $\mathcal{T}$

We can now detail the linear-time preprocessing done on tree $\mathcal{T}$.

Preprocessing Algorithm for $\mathcal{T}$

Begin

1. Do a depth-first traversal of $\mathcal{T}$ to assign depth-first search numbers to the nodes. During the traversal compute $h(v)$ for each node v. For each node, set a pointer to its parent node in $\mathcal{T}$.
2. Using the bottom-up algorithm described earlier, compute $I(v)$ for each v. For each number k such that $I(v) = k$ for some node v, set $L(k)$ to point to the head (or *Leader*) of the run containing node k. {Note that after this step, the head of the run containing an arbitrary node v can be retrieved in constant time: Compute $I(v)$ and then look up $L(I(v))$.}
   {This can easily be done while computing the I values. Node v is identified as the head of its run if the I value of v's parent is not $I(v)$.}
3. Let $\mathcal{B}$ be a complete binary tree with node-depth $d = \lceil \log n \rceil - 1$. Map each node v in $\mathcal{T}$ to node $I(v)$ in $\mathcal{B}$.
   {This mapping will be useful because it preserves enough (although not all) of the ancestry relations from $\mathcal{T}$.}

   {The above three steps form the core of the preprocessing, but there is also one more technical step. For each node v in $\mathcal{T}$, we want to encode *some* information about where in $\mathcal{B}$ the ancestors of v get mapped. That information is collected in the next step. Remember that $h(I(q))$ is the height in $\mathcal{B}$ of node $I(q)$ and so it is the height in $\mathcal{B}$ of the node that q gets mapped to.}
4. For each node v in $\mathcal{T}$, create an $O(\log n)$ bit number A_v. Bit $A_v(i)$ is set to 1 if and only if node v has some ancestor in $\mathcal{T}$ that maps to height i in $\mathcal{B}$, i.e., if and only if v has an ancestor u such that $h(I(u)) = i$.

End.

This ends the description of the preprocessing of $\mathcal{T}$ and the mapping of $\mathcal{T}$ to $\mathcal{B}$. To test your understanding of A_v, verify that the number of bits set to 1 in A_v is the number of *distinct* runs encountered on the path from the root to v. Setting the A numbers is easy by a linear-time traversal of $\mathcal{T}$ after all the I values are known: If v' is the parent of v then A_v is obtained by first copying $A_{v'}$ and then setting bit $A_v(i)$ to 1 if $h(I(v)) = i$ (this last step will be redundant if v and v' are on the same run, but it is always correct). As an example, consider node 3 (0011) in Figure 8.6. $A_3 = 1101(13)$ since 3 (0011) maps to height 1, 2 (0010) maps to height 3, and 1 (0001) maps to height 4 in $\mathcal{B}$.

What is this crazy mapping doing?

In the end, the programming details of this mapping (preprocessing) are very simple, and will become simpler in Section 8.9. The mapping only requires standard linear-time traversals of tree $\mathcal{T}$ (a minor programming exercise in a sophomore-level course). However, for most readers, what exactly the mapping accomplishes is quite unintuitive, because it is a many-one mapping. Certainly, ancestry relations in $\mathcal{T}$ are not perfectly preserved by the mapping into $\mathcal{B}$ [indeed, how could they be when the depth of $\mathcal{T}$ can be n while the depth of $\mathcal{B}$ is bounded by $\Theta(\log n)$], but much ancestry information is preserved, as shown in the next key lemma. Recall that a node is defined to be an ancestor of itself.

Lemma 8.7.1. *If z is an ancestor of x in $\mathcal{T}$ then $I(z)$ is an ancestor of $I(x)$ in $\mathcal{B}$. Stated differently, if z is an ancestor of x in $\mathcal{T}$ then either z and x are on the same run in $\mathcal{T}$ or node $I(z)$ is a proper ancestor of node $I(x)$ in $\mathcal{B}$.*

Figures 8.6 and 8.7 illustrate the claim in the lemma.

PROOF OF LEMMA 8.7.1 The proof is trivial if $I(z) = I(x)$, so assume that they are unequal. Since z is an ancestor of x in $\mathcal{T}$, $h(I(z)) \geq h(I(x))$ by the definition of I, but equality is only possible if $I(z) = I(x)$. So $h(I(z)) > h(I(x))$. Now $h(I(z))$ and $h(I(x))$ are the respective heights of nodes $I(z)$ and $I(x)$ in $\mathcal{B}$, so $I(z)$ is at a height greater than the height of $I(x)$ in $\mathcal{B}$.

Let $h(I(z))$ be i. We claim that $I(z)$ and $I(x)$ are identical in all bits to the left of i (recall that bits of a binary number are numbered from the right). If not, then let $k > i$ be the left-most bit where $I(z)$ and $I(x)$ differ. Without loss of generality, assume that $I(z)$ has bit 1 and $I(x)$ has bit 0 in position k. Since k is the point of left-most difference, the bits to the left of position k are equal in the two numbers, implying that $I(z) > I(x)$. Now z is an ancestor of x in $\mathcal{T}$, so nodes $I(z)$ and $I(x)$ are both in the subtree of z in $\mathcal{T}$. Furthermore, since $I(z)$ and $I(x)$ are depth-first number of nodes in the subtree of z in $\mathcal{T}$, every number between $I(x)$ and $I(z)$ occurs as a depth-first number of some node in the subtree of z. In particular, let N be the number consisting of the bits to the left of position k in $I(z)$ (or $I(x)$) followed by 1 followed by all 0s. (Figure 8.5 helps illustrate the situation, although z plays the role of u and x plays the role of w, and bit i in $I(z)$ is unknown.) Then $I(x) < N < I(z)$; therefore N is also a node in the subtree of z. But $k > i$, so $h(N) > h(I(z))$, contradicting the definition of I. It follows that $I(z)$ and $I(x)$ must be identical in the bits to the left of bit i.

Now bit i is the right most 1-bit in $I(z)$, so the bits to the left of bit i describe the complete path in $\mathcal{B}$ to node $I(z)$. Those identical bits to the left of bit i also form the initial part of the description of the path in $\mathcal{B}$ to node $I(x)$, since $I(x)$ has a 1-bit to the right of bit i. So those bits are in the path descriptions of both $I(z)$ and $I(x)$, meaning that the path to node $I(x)$ in $\mathcal{B}$ must go through node $I(z)$. Therefore, node $I(z)$ is an ancestor of node $I(x)$ in $\mathcal{B}$, and the lemma is proved. □

Having described the preprocessing of $\mathcal{T}$ and developed some of the properties of the tree map, we can now describe the way that *lca* queries are answered.

8.8. Answering an *lca* query in constant time

Let x and y be two nodes in $\mathcal{T}$ and let z be the *lca* of x and y in $\mathcal{T}$. Suppose we know the height in $\mathcal{B}$ of the node that z is mapped to. That is, we know $h(I(z))$. Below we show, with only that limited information about z, how z can be found in constant time.

Theorem 8.8.1. *Let z denote the lca of x and y in $\mathcal{T}$. If we know $h(I(z))$, then we can find z in $\mathcal{T}$ in constant time.*

PROOF Consider the run containing z in $\mathcal{T}$. The path up $\mathcal{T}$ from x to z enters that run at some node $\bar{x}$ (possibly z) and then continues along that run until it reaches z. Similarly, the path up from y to z enters the run at some node $\bar{y}$ and continues along that run until z. It follows that z is either $\bar{x}$ or $\bar{y}$. In fact, z is the higher of those two nodes, and so by the numbering scheme, $z = \bar{x}$ if and only if $\bar{x} < \bar{y}$. For example, in Figure 8.6 when $x = 9$ (1001) and $y = 6$ (0110), then $\bar{x} = 8$ (1000) and $\bar{y} = z = 5$ (0101).

Given the above discussion, the approach to finding z from $h(I(z))$ is to use $h(I(z))$ to find $\bar{x}$ and $\bar{y}$, since those nodes determine z. We will explain how to find $\bar{x}$. Let $h(I(z)) = j$, so the height in $\mathcal{B}$ of $I(z)$ is j. By Lemma 8.7.1, node x (which is in the subtree of z in $\mathcal{T}$) maps to a node $I(x)$ in the subtree of node $I(z)$ in $\mathcal{B}$, so if $h(I(x)) = j$ then x must be on the same run as z (i.e., $x = \bar{x}$), and we are finished. Conversely, if $x = \bar{x}$, then $h(I(x))$ must be j. So assume from here on that $x \neq \bar{x}$.

Let w (which is possibly x) denote the node in $\mathcal{T}$ on the z-to-x path just below (off) the run containing z. Since x is not $\bar{x}$, x is not on the same run as z, and w exists. From $h(I(z))$ (which is assumed to be known) and A_x (which was computed during the preprocessing), we will deduce $h(I(w))$ and then $I(w)$, w, and $\bar{x}$.

Since w is in the subtree of z in $\mathcal{T}$ and is not on the same run as z, w maps to a node in $\mathcal{B}$ with height strictly less than the height of $I(z)$ (this follows from Lemma 8.7.1). In fact, by Lemma 8.7.1, among all nodes on the path from x to z that are not on z's run, w maps to a node of greatest height in $\mathcal{B}$. Thus, $h(I(w))$ (which is the height in $\mathcal{B}$ that w maps to) must be the largest position less than j such that A_x has a 1-bit in that position. That is, we can find $h(I(w))$ (even though we don't know w) by finding the most significant 1-bit of A_x in a position less than j. This can be done in constant time on the assumed machine (starting with all bits set to 1, shift right by $d - j + 1$ positions, **AND** this number together with A_x, and then find the left-most 1-bit in the resulting number.)

Let $h(I(w)) = k$. We will now find $I(w)$. Either w is x or w is a proper ancestor of x in $\mathcal{T}$, so either $I(w) = I(x)$ or node $I(w)$ is a proper ancestor of node $I(x)$ in $\mathcal{B}$. Moreover, by the path-encoding nature of the path numbers in $\mathcal{B}$, numbers $I(x)$ and $I(w)$ are identical in bits to the left of k, and $I(w)$ has a 1 in bit k and all 0s to the right. So $I(w)$ can be obtained from $I(x)$ (which we know) and k (which we obtained as above from $h(I(z))$ and A_x). Moreover, $I(w)$ can be found from $I(x)$ and $h(I(w))$ using constant-time bit operations.

Given $I(w)$ we can find w because $w = L(I(w))$. That is, w was just off the z run, so it must be the head of the run that it is on, and each node in $\mathcal{T}$ points to the head of its run. From w we find its parent $\bar{x}$ in constant time. □

In summary, assuming we know $h(I(z))$, we can find node $\bar{x}$, which is the closest ancestor of x in $\mathcal{T}$ that is on the same run as z. Similarly, we find $\bar{y}$. Then z is either $\bar{x}$ or $\bar{y}$; in fact, z is the node among those two with minimum depth-first number in $\mathcal{T}$. Of course, we must now explain how to find $j = h(I(z))$.

How to find the height of $I(z)$

Let b be the lowest common ancestor of $I(x)$ and $I(y)$ in $\mathcal{B}$. Since $\mathcal{B}$ is a complete binary tree, b can be found in constant time as described earlier. Let $h(b) = i$. Then $h(I(z))$ can be found in constant time as follows.

Theorem 8.8.2. *Let j be the smallest position greater or equal to i such that both A_x and A_y have 1-bits in position j. Then node $I(z)$ is at height j in $\mathcal{B}$, or in other words, $h(I(z)) = j$.*

PROOF Suppose $I(z)$ is at height k in $\mathcal{B}$. We will show that $k = j$. Since z is an ancestor of both x and y, both A_x and A_y have a 1-bit in position k. Furthermore, since $I(z)$ is an ancestor of both $I(x)$ and $I(y)$ in $\mathcal{B}$ (by Lemma 8.7.1), $k \geq i$, and it follows (by the selection of j) that $k \geq j$. This also establishes that a position $j \geq i$ exists where both A_x and A_y have 1-bits.

A_x has a 1-bit in position j and $j \geq i$, so x has an ancestor x' in $\mathcal{T}$ such that $I(x')$ is an ancestor of $I(x)$ in $\mathcal{B}$ and $I(x')$ is at height $\geq i$, the height of b in $\mathcal{B}$. It follows that $I(x')$ is an ancestor of b. Similarly, there is an ancestor y' of y in $\mathcal{T}$ such that $I(y')$ is at height j and is an ancestor of b in $\mathcal{B}$. But if $I(x')$ and $I(y')$ are at the same height (j) and both are ancestors of the single node b, then it must be that $I(x') = I(y')$, meaning that x' and y' are on the same run. Being on the same run, either x' is an ancestor in $\mathcal{T}$ of y' or vice versa. Say, without loss of generality, that x' is an ancestor of y' in $\mathcal{T}$. Then x' is a common ancestor of x and y, and x' is an ancestor of z in $\mathcal{T}$. Hence x' must map to the same height or higher than z in $\mathcal{B}$. That is, $j \geq k$. But $k \geq j$ was already established, so $k = j$ as claimed, and the theorem is proved. □

All the pieces for *lca* retrieval in $\mathcal{T}$ have now been described, and each takes only constant time. In summary, the lowest common ancestor z of any two nodes x and y in $\mathcal{T}$ (assuming z is neither x nor y) can be found in constant time by the following method:

Constant-time *lca* retrieval

Begin

1. Find the lowest common ancestor b in $\mathcal{B}$ of nodes $I(x)$ and $I(y)$.
2. Find the smallest position j greater than or equal to $h(b)$ such that both numbers A_x and A_y have 1-bits in position j. j is then $h(I(z))$.
3. Find node $\bar{x}$, the closest node to x on the same run as z (although we don't know z) as follows:

3a. Find the position l of the right-most 1-bit in A_x.

3b. If $l = j$, then set $\bar{x} = x$ {x and z are on the same run in $\mathcal{T}$} and go to step 4.
Otherwise (when $l < j$)

3c. Find the position k of the left-most 1-bit in A_x that is to the right of position j. Form the number consisting of the bits of $I(x)$ to the left of position k, followed by a 1-bit in position k, followed by all zeros. {That number will be $I(w)$, even though we don't yet know w.}
Look up node $L(I(w))$, which must be node w. Set node $\bar{x}$ to be the parent of node w in $\mathcal{T}$.

4. Find node $\bar{y}$, the closest node to y on the same run as z, by the same approach as in step 3.
5. If $\bar{x} < \bar{y}$ then set z to $\bar{x}$, else set z to $\bar{y}$.

End.

8.9. The binary tree is only conceptual

The astute reader will notice that the binary tree $\mathcal{B}$ can be eliminated entirely from the algorithm, although it is crucial in the exposition and the proofs. Tree $\mathcal{B}$ is certainly not

used in steps 3, 4, or 5 to obtain z from $h(I(z))$. However, it is used in step 1 to find node b from $I(x)$ and $I(y)$. But all we really need from b is $h(b)$ (step 2), and that can be gotten from the right-most common 1-bit of $I(x)$ and $I(y)$. So the mapping from $\mathcal{T}$ to $\mathcal{B}$ is only conceptual, merely used for purposes of exposition.

In summary, after the preprocessing on $\mathcal{T}$, when given nodes x and y, the algorithm finds $i = h(b)$ (without first finding b) from the right-most common 1-bit in $I(x)$ and $I(y)$. Then it finds $j = h(I(z))$ from i and A_x and A_y, and from j it finds $z = lca(x,y)$. Although the logic behind this method has been difficult to convey, a program for these operations is very easy to write.

8.10. For the purists: how to avoid bit-level operations

We have assumed that the machine can do certain bit-level operations in constant time. Many of these assumptions are reasonable for existing machines, but some, such as the ability to find the right-most 1-bit, do not seem as reasonable. How can we avoid all bit-level operations, executing the *lca* algorithm on a standard RAM model? Recall that our RAM can only read, write, address, add, subtract, multiply, and divide in constant time, and only on numbers with $O(\log n)$ bits.

The idea is that during the linear-time preprocessing of $\mathcal{T}$, we also build $O(n)$-size tables that specify the results of the needed bit-level operations. Bit-level operations on a *single* $O(\log n)$ bit number are the easiest. Shifting left by $i = O(\log n)$ bits is accomplished by multiplying by 2^i, which is a permitted constant-time operation since $2^i = O(n)$. Similarly, shifting right is accomplished by division. Now consider the problem of finding the left-most 1-bit. Construct a table with n entries, one for each $\lceil \log_2 n \rceil$ bit number. The entry for binary number 1 has a value of 1, the entries for binary numbers 2 and 3 have value 2, the next 4 entries have value 3, etc. Each entry is an $O(\log n)$ bit number, so this table can easily be generated in $O(n)$ time on a RAM model. Generating the table for right-most 1-bit is a little more involved but can be done in a related manner. An even smaller table of $\lceil \log_2 n \rceil$ entries is needed for "masks". A mask of size i consists of i 0-bits on the right end of $\lceil \log_2 n \rceil$ bits and 1s in all other positions. The i mask is used in the task of finding the right-most 1-bit to the left of position i. Mask $\lceil \log_2 n \rceil$ is the binary number 0. In general, mask i is obtained from mask $i+1$ by dividing it by two (shifting the mask to the right by one place), and then adding $2^{\lceil \log_2 n \rceil - 1}$ (which adds in the left-most 1-bit).

The tables that seem more difficult to build are the tables for *binary* operations on $O(\log n)$ bit numbers, such as **XOR**, **AND**, and **OR**. The full table for any of these binary operations is of size n^2 since there are n numbers with $\log n$ bits. One cannot construct an n^2-size table in $O(n)$ time. The trick is that each of the needed binary operations are done bitwise. For example, **XOR** of two $\lceil \log n \rceil$ bit numbers can be found by splitting each number into two numbers of roughly $\frac{\lceil \log n \rceil}{2}$ bits each, doing **XOR** twice, and then concatenating the answers (additional easy details are needed to do this in the RAM model). So it suffices to build an **XOR** table for numbers with only $\lceil \frac{\lceil \log n \rceil}{2} \rceil$ bits. But $2^{\frac{\lceil \log n \rceil}{2}} = \sqrt{n}$, so the **XOR** table for these numbers has only n entries, and hence it is plausible that the entire table can be constructed in $O(n)$ time. In particular, **XOR** can be implemented by **AND**, **OR**, and **NOT** (bit complement) operations, and tables for these can be built in $O(n)$ time (we leave this as an exercise).

8.11. Exercises

1. Using depth-first traversal, show how to construct the path numbers for the nodes of $\mathcal{B}$ in time proportional to n, the number of nodes in $\mathcal{B}$. Be careful to observe the constraints of the RAM model.
2. Prove that the path numbers in $\mathcal{B}$ are the *inorder traversal* numbers.
3. The *lca* algorithm for a complete binary tree was detailed in the case that *lca(i,j)* was neither i nor j. In the case that *lca(i,j)* is one of i or j, then a very simple constant-time algorithm can determine *lca(i,j)*. The idea is first to number the nodes of the binary tree $\mathcal{B}$ by a depth-first numbering, and to note for each node v, the number of nodes in the subtree of v (including v). Let $l(v)$ be the *dfs* number given to node v, and let $s(v)$ be the number of nodes in the subtree of v. Then node i is an ancestor of node j if and only if $l(i) \leq l(j)$ and $l(j) < l(i) + s(i)$.

   Prove that this is correct, and fill in the details to show that the needed preprocessing can be done in $O(n)$ time.

   Show that the method extends to any tree, not just complete binary trees.
4. In the special case of a complete binary tree $\mathcal{B}$, there is an alternative way to handle the situation when *lca(i,j)* is i or j. Using $h(i)$ and $h(j)$ we can determine which of the nodes i and j is higher in the tree (say i) and how many edges are on the path from the root to node i. Then we take the **XOR** of the binary for i and for j and find the left-most 1-bit as before, say in position k (counting from the left). Node i is an ancestor of j if and only if k is larger than the number of edges on the path to node i. Fill in the details of this argument and prove it is correct.
5. Explain why in the *lca* algorithm for $\mathcal{B}$, it was necessary to assume that *lca(i,j)* was neither i nor j. What would go wrong in that algorithm if the issue were ignored and that case was not checked explicitly?
6. Prove that the height of any node k in $\mathcal{B}$ is $h(k)$.
7. Write a *C* program for both the preprocessing and the *lca* retrieval. Test the program on large trees and time the results.
8. Give an explicit $O(n)$-time RAM algorithm for building the table containing the right-most 1-bit in every $\log_2 n$ bit number. Remember that the entry for binary number i must be in the ith position in the table. Give details for building tables for **AND**, **OR**, and **NOT** for $\frac{\log n}{2}$ bit numbers in $O(n)$ time.
9. It may be more reasonable to assume that the RAM can shift a word left and right in constant time than to assume that it can multiply and divide in constant time. Show how to solve the *lca* problem in constant time with linear preprocessing under those assumptions.
10. In the proof of Theorem 8.8.1 we showed how to deduce $l(w)$ from $h(l(w))$ in constant time. Can we use the same technique to deduce $l(z)$ from $h(l(z))$? If so, why doesn't the method do that rather than involving nodes w, $\bar{x}$, and $\bar{y}$?
11. The constant-time *lca* algorithm is somewhat difficult to understand and the reader might wonder whether a simpler idea works. We know how to find the *lca* in constant time in a complete binary tree after $O(n)$ preprocessing time. Now suppose we drop the assumption that the binary tree is complete. So $\mathcal{T}$ is now a binary tree, but not necessarily complete. Letting d again denote the depth of $\mathcal{T}$, we can again compute $d+1$ length path numbers that encode the paths to the nodes, and again these path numbers allow easy construction of the lowest common ancestor. Thus it might seem that even in incomplete binary trees, one can easily find the *lca* in this simple way without the need for the full *lca* algorithm. Either give the details for this or explain why it fails to find the *lca* in constant time.

If you believe that the above simple method solves the *lca* problem in constant time for any binary tree, then consider trying to use it for arbitrary trees. The idea is to use the well-known technique of converting an arbitrary tree to a binary tree by modifying every node with more than two children as follows: Suppose node v has children $v_1, v_2, \ldots, v_k$. Replace the children of v with two children v_1 and v^* and make nodes $v_2, \ldots, v_k$ children of v^*. Repeat this until each original child v_i of v has only one sibling, and place a pointer from v^* to v for every new node v^* created in this process. How is the number of nodes changed by this transformation? How does the *lca* of two nodes in this new binary tree relate to the *lca* of the two nodes in the original tree? So, assuming that the $d+1$ length path labels can be used to solve the *lca* problem in constant-time for any binary tree, does this conversion yield a constant-time *lca* search algorithm for any tree?

12. A simpler (but slower) *lca* algorithm. In Section 8.4.1 we mentioned that if $\Theta(n \log n)$ preprocessing time is allowed, and $\Theta(n \log n)$ space can be allocated during both the preprocessing and the retrieval phases, then a (conceptually) simpler constant-time *lca* retrieval method is possible. In many applications, $\Theta(n \log n)$ is an acceptable bound, which is not much worse than the $O(n)$ bound we obtained in the text. Here we sketch the idea of the $\Theta(n \log n)$ method. Your problem is to flesh out the details and prove correctness.

First we reduce the general *lca* problem to a problem of finding the smallest number in an interval of a fixed list of numbers.

The reduction of *lca* to a list problem

Step 1 Execute a depth-first traversal of tree $\mathcal{T}$ to label the nodes in depth-first order and to build a multilist L of the nodes in the order that they are visited. (For any node v other than the root, the number of times v is in L equals the degree of v.) The only property of the depth-first numbering we need is that the number given to any node is smaller than the number given to any of its proper descendants. From this point on, we refer to a node only by its *dfs* number.

For example, the list for the tree in Figure 8.1 (page 182) is

$$\{1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 7, 5, 8, 9, 8, 10, 8, 5, 1\}.$$

Notice that if $\mathcal{T}$ has n nodes, then L has $O(n)$ entries.

Step 2 The *lca* of any two nodes x and y can be gotten as follows: Find any occurrences of x and y in L; this defines an interval I in L between those occurrences of x and y. Then in L find the smallest number in interval I; that number is the *lca(x,y)*.

For example, if x is 6 and y is 9, then one interval I that they define is $\{6, 5, 7, 5, 8, 9\}$, implying that node 5 is *lca(6,9)*.

This is the end of the reduction. Now the first exercise.

a. Ignoring time complexity, prove that in general the *lca* of two nodes can be obtained as described in the two steps above.

Now we continue to describe the method. More exercises will follow.

With the above reduction, each *lca* query becomes the problem of finding the smallest number in a interval I of a fixed list L of $O(n)$ numbers. Let m denote the exact size of L. To be able to solve each *lca* query in constant time, we first do an $O(m \log m)$-time preprocessing of list L. For convenience assume that m is a power of 2.

Preprocessing of L

Step 1 Build a complete binary tree B with m leaves and number the leaves in left-to-right order (as given by an inorder traversal). Then for i from 1 to m, record the ith element of L at leaf i.

Step 2 For an arbitrary internal node v in B, let B_v denote the subtree of B rooted at v, and let $L_v = n_1, n_2, \ldots, n_z$ be an ordered list containing the elements of L written at the leaves of B_v, in the same left-to-right order as they appear in B. Create two lists, *Pmin(v)* and *Smin(v)*, for each internal node v. Each list will have size equal to the number of leaves in v's subtree. The kth entry of list *Pmin(v)* is the smallest number among $\{n_1, n_2, \ldots, n_k\}$. That is, the kth entry of *Pmin(v)* is the smallest number in the *prefix* of list L_v ending at position k. Similarly, the kth entry of list *Smin(v)* is the smallest number in the *suffix* of L_v starting at position k. This is the end of the preprocessing and exercises follow.

b. Prove that the total size of all the *Pmin* and *Smin* lists is $O(m \log m)$, and show how they can be constructed in that time bound.

After the $O(m \log m)$ preprocessing, the smallest number in any interval I can be found in constant time. Here's how. Let interval I in L have endpoints l and r and recall that these correspond to leaves of B. To find the smallest number in I, first find the *lca(l,r)*, say node v. Let v' and v'' be the left and right children of v in B, respectively. The smallest number in I can be found using one lookup in list *Smin(v')*, one lookup in *Pmin(v'')*, and one additional comparison.

c. Give complete details for how the smallest number in I is found, and fully explain why only constant time is used.

13. By refining the method developed in Exercise 12, the $\Theta(m \log m)$ preprocessing bound (time and space) can be reduced to only $\Theta(m \log \log m)$ while still maintaining constant retrieval time for any *lca* query. (It takes a pretty big value of m before the difference between $O(m)$ and $\Theta(m \log \log m)$ is appreciable!) The idea is to divide list L into $\frac{m}{\log m}$ *blocks* each of size $\log m$ and then separately preprocess each block as in Exercise 12. Also, compute the minimum number in each block, put these $\frac{m}{\log m}$ numbers in an ordered list *Lmin*, and preprocess *Lmin* as in Exercise 12.

a. Show that the above preprocessing takes $\Theta(m \log \log m)$ time and space.

Now we sketch how the retrieval is done in this faster method. Given an interval I with starting and ending positions l and r, one finds the smallest number in I as follows: If l and r are in the same block, then proceed as in Exercise 12. If they are in adjacent blocks, then find the minimum number from l to the end of l's block, find the minimum number from the start of r's block to r, and take the minimum of those two numbers. If l and r are in nonadjacent blocks, then do the above and also use *Lmin* to find the minimum number in all the blocks strictly between the block containing l and the block containing r. The smallest number in I is the minimum of those three numbers.

b. Give a detailed description of the retrieval method and justify that it takes only constant time.

14. Can the above improvement from $O(m \log m)$ preprocessing time to $O(m \log \log m)$ preprocessing time be extended to reduce the preprocessing time to $O(m \log \log \log m)$ preprocessing time? Can the improvements be continued for an arbitrary number of *logarithms*?

9

More Applications of Suffix Trees

With the ability to solve lowest common ancestor queries in constant time, suffix trees can be used to solve many additional string problems. Many of those applications move from the domain of *exact* matching to the domain of *inexact*, or approximate, matching (matching with some errors permitted). This chapter illustrates that point with several examples.

9.1. Longest common extension: a bridge to inexact matching

The *longest common extension problem* is solved as a subtask in many classic string algorithms. It is at the heart of all but the last application discussed in this chapter and is central to the *k-difference algorithm* discussed in Section 12.2.

Longest common extension problem Two strings S_1 and S_2 of total length n are first specified in a preprocessing phase. Later, a *long* sequence of index pairs is specified. For each specified index pair (i, j), we must find the length of the longest substring of S_1 starting at position i that matches a substring of S_2 starting at position j. That is, we must find the length of the longest prefix of suffix i of S_1 that matches a prefix of suffix j of S_2 (see Figure 9.1).

Of course, any time an index pair is specified, the longest common extension can be found by direct search in time proportional to the length of the match. But the goal is to compute each extension in *constant* time, independent of the length of the match. Moreover, it would be cheating to allow more than linear time to preprocess S_1 and S_2.

To appreciate the power of suffix trees combined with constant-time *lca* queries, the reader should again try first to devise a solution to the longest common extension problem without those two tools.

9.1.1. Linear-time solution

The solution to the longest common extension problem first builds the generalized suffix tree $\mathcal{T}$ for S_1 and S_2, and then prepares $\mathcal{T}$ to allow constant-time *lca* queries. During this preprocessing, it also computes the string-depth of v for every node v of $\mathcal{T}$. Building and preprocessing $\mathcal{T}$ takes $O(n)$ time.

Given any specific index pair (i, j), the algorithm finds the lowest common ancestor of the leaves of $\mathcal{T}$ that correspond to suffix i in S_1 and suffix j in S_2. Let v denote that lowest common ancestor node. The key point is that the string labeling the path to v is precisely the longest substring of S_1 starting at i that matches a substring of S_2 starting at j. Consequently, the string-depth of v is the length of the *longest common extension*. Hence each longest common extension query can be answered in constant time.

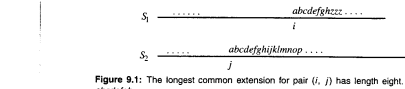


Figure 9.1: The longest common extension for pair (i, j) has length eight. The matching substring is *abcdefgh*.

9.1.2. Space-efficient longest common extension

When S_2 is much smaller than S_1, we may not wish to build the generalized suffix tree for S_1 and S_2 together. By only building the suffix tree for the smaller of the two strings, we will save considerable space, but can the longest common extension problem be efficiently solved using only this smaller tree? The answer is yes, with the aid of *matching statistics*.

Recall from Section 7.8.1 that the matching statistic $m(i)$ is the length of the longest substring of S_1 starting at position i that matches a substring starting at some position in S_2, and that $p(i)$ is one of those starting positions in S_2. In Sections 7.8.1 and 7.8.3 we showed how to compute $m(i)$ and $p(i)$ for each i in $O(|S_1|)$ total time, using only a suffix tree for S_2 and a copy of S_1.

The longest common extension query for any pair (i, j) is solved by first finding the *lca* v of leaves $p(i)$ and j in the suffix tree for S_2. The length of the longest common extension of (i, j) is then the minimum of $m(i)$ and the string-depth of node v. The proof of this is immediate and is left to the reader. Since any *lca* computation takes only constant time, we have the following:

Theorem 9.1.1. *After a preprocessing phase that takes linear time, any longest common extension query can be answered in constant time using only the space required by the suffix tree for S_2 (the smaller of the two strings) plus $2|S_1|$ words to hold the values $m(i)$ and $p(i)$.*

The space-efficient solution of the longest common extension problem usually implies a space-efficient solution to the various applications of it. This will not be explicitly mentioned in every application, and the details will be left to the reader.

9.2. Finding all maximal palindromes in linear time

Definition An even-length substring S' of S is a *maximal palindrome of radius k* if, starting in the middle of S', S' reads the same in both directions for k characters but not for any $k' > k$ characters. An odd-length maximal palindrome S' is similarly defined after excluding the middle character of S'.

For example, if $S = aabactgaaccaat$ then both *aba* and *aaccaa* are maximal palindromes in S of radii one and three, respectively, and each occurrence of *aa* is a maximal palindrome of radius one.

Definition A string is called a *maximal palindrome* if it is a maximal palindrome of radius k for some k.

For example, in the string *cabaabad*, both *aba* and *abaaba* are maximal palindromes.

Any palindrome is contained in some maximal palindrome with the same midpoint, so the maximal palindromes are a compact way to represent the set of all palindromes. Moreover, in most applications, the maximal palindromes are the ones of interest.

Palindrome problem: Given a string S of length n, the palindrome problem is to locate *all* maximal palindromes in S.

9.2.1. Linear-time solution

We will explain how to find all the even-length maximal palindromes in linear, $O(n)$ time – a rather severe goal. The odd-length maximal palindromes can be found by a simple modification of the even-length case.

Let S' be the reverse of string S. Now, suppose there is an even-length maximal palindrome in S whose middle occurs just after character q of S. Let k denote the length of the palindrome. That means there is a string of length k starting at position $q+1$ of S that is identical to a string starting at position $n-q+1$ of S'. Furthermore, because the palindrome is maximal, the next characters (in positions $q+k+1$ and $n-q+k+1$, respectively) are not identical. This implies that k is the length of the longest common extension of position $q+1$ in S and position $n-q+1$ in S'. Hence for any fixed position q, the length of the maximal palindrome (if there is one) with midpoint at q can be computed in constant time.

This leads to the following simple linear-time method to find all the even length maximal palindromes in S:

1. In linear time, create the reverse string S' from S and preprocess the two strings so that any longest common extension query can be solved in constant time.
2. For each q from 1 to $n-1$, solve the longest common extension query for the index pair $(q+1, n-q+1)$ in S and S', respectively. If the extension has nonzero length k, then there is a maximal palindrome of radius k centered at q.

The method takes $O(n)$ time since the suffix tree can be built and preprocessed in that time, and each of the $O(n)$ extension queries is solved in constant time.

In summary, we have

Theorem 9.2.1. *All the maximal even-length palindromes in a string can be identified in linear time.*

9.2.2. Complemented and separated palindromes

Palindromes were briefly discussed in Section 7.11.1, during the general discussion of repetitive structures in biological strings. There, it was mentioned that in DNA (or RNA) the palindromes of interest are *complemented*. That means that the two halves of the substring form a palindrome (in the normal English use of the word) only if the characters in one half are converted to their complement characters, that is, A and T (or U in the case of RNA) are complements, and C and G are complements. For example, $ATTAGCTAAT$ is a complemented palindrome.

The problem of finding all complemented palindromes in a string can also be solved in linear time. Let $c(S')$ be the complement of string S' (i.e., where each A is changed to T, each T to A, each C to G, and each G to C). Then proceed as in the palindrome problem, using $c(S')$ in place of S'.

Another variant of the palindrome problem that comes from biological sequences is to relax the insistence that the two halves of the palindrome (complemented or not) be adjacent. When the two halves are not adjacent, the structure is called a *separated palindrome*, although in the biological literature the distinction between separated and nonseparated palindromes is sometimes blurred. The problem of finding all separated palindromes is really one of finding all inverted repeats (see Section 7.12) and hence is more complex than finding palindromes. However, if there is a fixed bound on the permitted distance of the separation, then all the separated palindromes can again be found in linear time. This is an immediate application of the longest common extension problem, the details of which are left to the reader.

Another variant of the palindrome problem, called the k-mismatch palindrome problem, will be considered below, after we discuss matching with a fixed number of mismatches.

9.3. Exact matching with wild cards

Recall the problem discussed in Sections 4.3.2 and 3.5.2 of finding all occurrences of pattern P in text T when wild cards are allowed in either string. This problem was not easily handled with Knuth-Morris-Pratt or Boyer–Moore–type methods, although the Fast Fourier transform, match-count method could be modified to handle wild cards in both strings. Using the above method to solve the longest common extension problem, when there are k wild cards distributed throughout the two strings, we can find all occurrences of P in T (allowing a wild card to match any *single* character) in $O(km)$ time, where $m \geq n$ is the length of T and n is the length of P.

At the high level, the algorithm increments i from 1 to $m-n+1$ and checks, for each fixed i, whether P occurs in T starting at position i of T. The wild card symbol is treated as an additional character in the alphabet. The idea of the method is to align the left end of P against position i of T and then work left to right through the two strings, successively executing longest common extension queries and checking that every mismatch occurs at a position containing a wild card. After $O(n+m)$ preprocessing time (for longest common extension queries), only $O(k)$ time is used by the algorithm for any fixed i. Thus, $O(km)$ time is used overall. The following detailed algorithm works for a fixed position i of T:

Wild-card match check

Begin

1. Set j to 1 and i' to i.
2. Compute the length l of the longest common extension starting at positions j of P and i' of T.
3. If $j+l=n+1$ then P occurs in T starting at i; stop.
4. Check if a wild card occurs in position $j+l$ of P or position $i'+l$ of T. If so then set j to $j+l+1$, set i' to $i'+l+1$, and go to step 2. Else, P does not occur in T starting at i; stop.

End.

The space needed by this method is $O(n+m)$, since it uses a suffix tree for the two strings. However, as detailed in Theorem 9.1.1, only a suffix tree for P plus the matching statistics for T are needed (although we must still store the original strings). Since $m > n$ we have

Theorem 9.3.1. *The exact matching problem with k wild cards distributed in the two strings can be solved in $O(km)$ time and $O(m)$ space.*

9.4. The k-mismatch problem

The general problem of inexact or approximate matching (matching with some errors allowed) will be considered in detail in Part III of the book (in Section 12.2), where the technique of dynamic programming will be central. But dynamic programming is not always necessary, and we have here all the tools to solve one of the classic "benchmark" problems in approximate matching: the *k-mismatch* problem.

Definition Given a pattern P, a text T, and a fixed number k that is independent of the lengths of P and T, a *k-mismatch* of P is a $|P|$-length substring of T that matches at least $|P| - k$ characters of P. That is, it matches P with at most k mismatches.

Note that the definition of a k-mismatch does not allow any insertions or deletions of characters, just matches and mismatches. Later, in Section 12.2, we will discuss bounded error problems that also allow insertions and deletions of characters.

The **k-mismatch problem** is to find all k-mismatches of P in T.

For example, if $P = bend$, $T = abentbananaend$, and $k = 2$, then T contains three k-matches of P: P matches substring *bent* with one mismatch, substring *bana* with two mismatches, and substring *aend* with one mismatch.

Applications in molecular biology for the k-mismatch problem, along with the more general k-differences problem, will be discussed in Section 12.2.2. The k-mismatch problem is a special case of the match-count problem considered in Section 4.3, and the approaches discussed there apply. But because k is a fixed number unrelated to the lengths of P and T, faster solutions have been obtained. In particular, Landau and Vishkin [287] and Myers [341] were the first to show an $O(km)$-time solution, where P and T have lengths n and $m > n$, respectively. The value of k can never be more than n, but the motivation for the $O(km)$ result comes from applications where k is expected to be very small compared to n.

9.4.1. The solution

The idea is essentially the same as the idea for matching with wild cards (although the meaning of k in these two problems is different). For any position i in T, we determine whether a k-mismatch of P begins at position i in $O(k)$ time by simply executing up to k (constant-time) longest common extension queries. If those extensions reach the end of P, then P matches the substring starting at i with at most k mismatches. If the extensions do not reach the end of P, then more than k mismatches are needed. In either case, at most k extension queries are solved for any i, and $O(k)$ time suffices to determine whether a k-mismatch of P begins at i. Over all positions in T, the method therefore requires at most $O(km)$ time.

k-mismatch check

Begin

1. Set j to 1 and i' to i, and *count* to 0.
2. Compute the length l of the longest common extension starting at positions j of P and i' of T.
3. If $j + l = n + 1$, then a k-mismatch of P occurs in T starting at i (in fact, only *count* mismatches occur); stop.
4. If $count \leq k$, then increment *count* by one, set j to $j + l + 1$, set i' to $i' + l + 1$, and go to step 2.
   If $count = k + 1$, then a k-mismatch of P does not occur starting at i; stop.

End.

Note that the space required for this solution is just $O(n + m)$, and that the method can be implemented using a suffix tree for the small string P alone.

We should note a different practical approach to the k-mismatch problem, based on suffix trees, that is in use in biological database search [320]. The idea is to generate every string P' that can be derived from P by changing up to k characters of P, and then to search for P' in a suffix tree for T. Using a suffix tree, the search for P' takes time just proportional to the length of P' (and can be implemented to be extremely fast), so this approach can be a winner when k and the size of the alphabet are relatively small.

9.5. Approximate palindromes and repeats

We have discussed earlier (Section 7.11.1) the importance of palindromes in molecular biology. That discussion provides most of the motivation for the palindrome problem. But in biological applications, the two parts of the "palindrome" are rarely identical. This motivates the k-mismatch palindrome problem.

Definition A *k-mismatch palindrome* is a substring that becomes a palindrome after k or fewer characters are changed. For example, *axabbcca* is a 2-mismatch palindrome.

With this definition, a palindrome is just a 0-mismatch palindrome. It is now an easy exercise to detail an $O(kn)$-time method to find all k-mismatch palindromes in a string of length n. We leave that to the reader, and we move on to the more difficult problem of finding *tandem repeats*.

Definition A *tandem repeat* α is a string that can be written as $\beta\beta$, where β is a substring.

Each tandem repeat is specified by a starting position of the repeat and the length of the substring β. This definition does not require that β be of maximal length. For example, in the string *xababababy* there are a total of six tandem repeats. Two of these begin at position two: *abab* and *abababab*. In the first case, β is *ab*, and in the second case, β is *abab*.

Using longest common extension queries, it is immediate that all tandem repeats can be found in $O(n^2)$ time – just guess a start position i and a middle position j for the tandem and do a longest common extension query from i and j. If the extension from i reaches j or beyond, then there is a tandem repeat of length $2(j - i + 1)$ starting at position i. There are $\Theta(n^2)$ choices for i and j, yielding the $O(n^2)$ time bound.

Definition A *k-mismatch tandem repeat* is a substring that becomes a tandem repeat after k or fewer characters are changed. For example, *axabaybb* is a 2-mismatch tandem repeat.

Again, all k-mismatch tandem repeats can be found in $O(kn^2)$ time, and the details are left to the reader. Below we will present a method that solves this problem in $O(kn \log(n/k))$ time. To summarize, what we have so far is

Theorem 9.5.1. *All the tandem repeats in S in which the two copies differ by at most k mismatches can be found in $O(kn^2)$ time. Typically, k is a fixed number, and the time bound is reported as $O(n^2)$.*

9.6. Faster methods for tandem repeats

The total number of tandem repeats and k-mismatch tandem repeats (even for fixed k) can grow as fast as $\Theta(n^2)$ (take the case of all n characters being the same). So, no worst-case bound better than $O(n^2)$ is possible for the problem of finding all tandem repeats. But a method whose running time depends on the *number* of tandem repeats contained in the string is possible for both the exact and the k-mismatch versions of the problem. A different approach was explored in Exercises 56, 57, and 58 of Chapter 7, where only maximal primitive tandem arrays were identified.

Landau and Schmidt [288] developed a method to find all k-mismatch tandem repeats in $O(kn\log(\frac{n}{z})+z)$ time, where z is the number of k-mismatch tandem repeats in the string S. Now z can be as large as $\Theta(n^2)$, but in practice z is expected to be small compared to n^2, so the $O(kn\log(\frac{n}{k})+z)$ bound is a significant improvement. Note that we will still find all tandem repeats, but the running time will depend on the actual number of repeats and not on the worst-case possible number of repeats.

We explain the method by first adapting it to find all tandem repeats (with no mismatches) in $O(n\log n+z)$ time, where z is now the total number of tandem repeats in S. That time bound for the case of no mismatches was first obtained in a paper by Main and Lorenz [307], who used a similar idea but did not use suffix trees. Their approach is explored in Exercise 8.

The Landau–Schmidt solution is a recursive, divide-and-conquer algorithm that exploits the ability to compute longest common extension queries in constant time. Let h denote $\lfloor \frac{n}{2} \rfloor$. At the highest level, the Landau–Schmidt method divides the problem of finding all tandem repeats into four subproblems:

1. Find all tandem repeats contained entirely in the first half of S (up to position h).
2. Find all tandem repeats contained entirely in the second half of S (after position h).
3. Find all tandem repeats where the first copy spans (contains) position h of S.
4. Find all tandem repeats where the second copy spans position h of S.

Clearly, no tandem repeat will be found in more than one of these four subproblems. The first two subproblems are solved by recursively applying the Landau–Schmidt solution. The second two problems are symmetric to each other, so we consider only the third subproblem. An algorithm for that subproblem therefore determines the algorithm for finding all tandem repeats.

Algorithm for problem 3

We want to find all the tandem repeats where the first copy *spans* (but does not necessarily begin at) position h. The idea of the algorithm is this: For any fixed number l, one can test in constant time whether there is a tandem repeat of length exactly $2l$ such that the first copy spans position h. Applying this test for all feasible values of l means that in $O(n)$ time we can find all the lengths of tandem repeats whose first copy spans position h. Moreover, for each such length we can enumerate all the starting points of these tandem repeats, in time proportional to the number of them. Here is how to test a number l.

Begin

1. Let $q = h+l$.
2. Compute the longest common extension (in the forward direction) from positions h and q. Let l_1 denote the length of that extension.

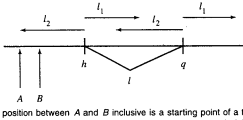


Figure 9.2: Any position between A and B inclusive is a starting point of a tandem repeat of length $2l$. As detailed in Step 4, if l_1 and l_2 are both at least one, then a subinterval of these starting points specify tandem repeats whose first copy spans h.

3. Compute the longest common extension *in the reverse direction* from positions $h-1$ and $q-1$. Let l_2 denote the length of that extension.
4. There is a tandem repeat of length $2l$ whose first copy spans position h if and only if $l_1+l_2 \geq l$ and both l_1 and l_2 are at least one. Moreover, if there is such a tandem repeat of length $2l$, then it can begin at any position from $\text{Max}(h-l_2, h-l+1)$ to $\text{Min}(h+l_1-l, h)$ inclusive. The second copy of the repeat begins l places to the right. Output each of these starting positions along with the length $2l$. (See Figure 9.2.)

End.

To solve an instance of subproblem 3 (finding all tandem repeats whose first copy spans position h), just run the above algorithm for each l from 1 to h.

Lemma 9.6.1. *The above method correctly solves subproblem 3 for a fixed h. That is, it finds all tandem repeats whose first copy spans position h. Further, for fixed h, its running time is $O(n/2)+z_h$, where z_h is the number of such tandem repeats.*

PROOF Assume first that there is a tandem repeat whose first copy spans position h, and it has some length, say $2l$. That means that position $q = h+l$ in the second copy corresponds to position h in the first copy. Hence some substring starting at h must match a substring starting at q, in order to provide the suffix of each copy. This substring can have length at most l_1. Similarly, there must be a substring ending at $h-1$ that matches a substring ending at $q-1$, providing the prefix of each copy. That substring can have length at most l_2. Since all characters between h and q are contained in one of the two copies, l_1+l_2 must be at least l. Conversely, by essentially the same reasoning, if $l_1+l_2 \leq l$ and both l_1 and l_2 are at least one then one can specify a tandem repeat of length $2l$ whose first copy spans h. The necessary and sufficient condition for the existence of such a tandem is therefore proved.

The converse proof that all starting positions fall in the stated range involves similar reasoning and is left to the reader.

For the time analysis, note first that for a fixed choice of h, the method takes constant time per choice of l to execute the common extension queries, and so it takes $O(n/2)$ time for all those queries. For any fixed l, the method takes constant time per tandem that it reports, and it never reports the same tandem twice since it reports a different starting point for each repeat of length $2l$. Since each repeat is reported as a starting point and a length, it follows that over all choices of l, the algorithm never reports any tandem repeat twice. Hence the time spent to report tandem repeats is proportional to z_h, the number of tandem repeats whose first copy spans position h. □

Theorem 9.6.1. *Every tandem repeat in S is found by the execution of subproblems 1 through 4 and is reported exactly once. The time for the algorithm is $O(n\log n+z)$, where z is the total number of tandem repeats in S.*

PROOF That all tandem repeats are found is immediate from the fact that every tandem is of a form considered by one of the subproblems 1 through 4. To show that no tandem repeat is reported twice, recall that for $h = n/2$, no tandem is of the form considered by more than one of the four subproblems. This holds recursively for subproblems 1 and 2. Further, in the proof of Lemma 9.6.1 we established that no execution of subproblem 3 (and also 4) reports the same tandem twice. Hence, over the entire execution of the four subproblems, no tandem repeat is reported twice. It also follows that the total time used to output the tandem repeats is $O(z)$.

To finish the analysis, we consider the time taken by the extension queries. This time is proportional to the number of extension queries executed. Let $T(n)$ denote the number of extension queries executed for a string of length n. Then, $T(n) = 2T(n/2) + 2n$, and $T(n) = O(n \log n)$ as claimed. □

9.6.1. The speedup for k-mismatch tandem repeats

The idea for the $O(kn \log(\frac{n}{k}) + z)$ algorithm of Landau and Schmidt [288] is an immediate extension of the $O(n \log n + z)$ method for finding exact tandem repeats, but the implementation is a bit more involved. The method is again recursive, and again the important part is subproblem 3 (or 4), finding all k-mismatch tandem repeats whose first copy spans position h. The solution to that problem is to run k successive longest common extension queries forward from h and q and to run k successive longest common extension queries backward from $h - 1$ and $q - 1$. Now focus on the interval between h and q. To find all k-mismatch tandem repeats whose first copy spans h, find every position t (if any) in that interval where the number of mismatches from h to t (found during the forward extension) plus the number of mismatches from $t + 1$ to $q - 1$ (found during the backward extension) is at most k. Any such t provides a midpoint of the tandem repeat. We leave the correctness of that claim to the reader.

To achieve the claimed time bound, we must find all the midpoints of the k-mismatch tandem repeats whose first copy spans h in time proportional to the number of them. But unlike the case of exact tandem repeats, the set of correct midpoints need not be contiguous. How are they found? We sketch the idea and leave the details as an exercise. During the k forward extension queries, accumulate an ordered list of the positions in interval $[h, q]$ where a mismatch occurs, and do the same during the backward extension queries. Then merge (in left to right order) those two lists and calculate for each position in the list the total number of mismatches to it from h and to $q - 1$. Since each list is found in order, the time to obtain the merged list and the totals is $O(k)$. The total number of mismatches can change only at a position that is in the merged list; hence an $O(k)$ time scan of that list specifies all subintervals containing permitted midpoints of the k-mismatch tandem. In addition, every point in such a subinterval is a permitted midpoint. Thus, for a fixed h, the total query time for subproblem 3 is $O(k)$ and the total output time is kz_h. Over the entire algorithm, the total output time is $O(kz)$ and the number of queries satisfies $T(n) = 2T(n/2) + 2k$. Thus, at most $O(kn \log n)$ queries are done. In summary, we have

Theorem 9.6.2. *All k-mismatch tandem repeats in a string of length n can be found in $O(kn \log n + z)$ time.*

The bound can be sharpened to $O(kn \log(n/k) + z)$ by the observation that any $l \leq k$ need not be tested in subproblems 3 and 4. We leave the details as an exercise.

We also leave it to the reader to adapt the solutions for the k-mismatch palindrome and tandem repeat problems to allow for string complementation and bounded-distance separation between copies.

9.7. A linear-time solution to the multiple common substring problem

All of the above applications are similar, exploiting the ability to solve longest common extension queries in constant time. Now we examine another use of suffix trees with constant-time *lca* that is not of this form.

The *k-common substring problem* was first discussed in Section 7.6 (the reader should review that discussion before going on). In that section, a generalized suffix tree $\mathcal{T}$ was constructed for the K strings of total length n, and the table of all the $l(k)$ values was obtained by operations on $\mathcal{T}$. That method had a running time of $O(Kn)$. In this section we reduce the time to $O(n)$. The solution was obtained by Lucas Hui [236].[1]

Recall that for any node v in $\mathcal{T}$, $C(v)$ is the number of distinct leaf string identifiers in the subtree of v, and that a table of all the $l(k)$ values can be computed in $O(n)$ time once all the $C(v)$ values are known. Recall also that $S(v)$ is the total number of leaves in the subtree of v and that $S(v)$ can easily be computed in $O(n)$ time for all nodes.

Certainly, $S(v) \geq C(v)$ for any node v, and it will be strictly greater when there are two or more leaves of the same string identifier in v's subtree. Our approach to finding $C(v)$ is to compute both $S(v)$ and a correction factor $U(v)$, which counts how many "duplicate" suffixes from the same string occur in v's subtree. Then $C(v)$ is simply $S(v) - U(v)$.

Definition $n_i(v)$ is the number of leaves with identifier i in the subtree rooted at node v. Let n_i be the total number of leaves with identifier i.

With that definition, we immediately have the following:

Lemma 9.7.1. *$U(v) = \sum_{i:n_i(v)>0}(n_i(v) - 1)$ and $C(v) = S(v) - U(v)$.*

We show below that all the correction factors for all internal nodes can be computed in $O(n)$ total time. That then gives an $O(n)$-time solution to the k-common substring problem.

9.7.1. The method

The algorithm first does a depth-first traversal of $\mathcal{T}$, numbering the leaves in the order that they are encountered. That numbering has the familiar property that for any internal node v, the numbers given to the leaves in the subtree rooted at v are consecutive (i.e., they form a consecutive interval).

For purposes of the exposition, let us focus on the single identifier i and show how to compute $n_i(v) - 1$ for each internal node v. Let L_i be the list of leaves with identifier i, in increasing order of their *dfs* numbers. For example, in Figure 9.3, the leaves with identifier i are shown boxed and the corresponding L_i is 1, 3, 6, 8, 10. By the properties of depth-first numbering, for the subtree rooted at any internal node v, all the $n_i(v)$ leaves with identifier i occur in a consecutive interval of list L_i. Call that interval $L_i(v)$. If x and

[1] In the introduction of an earlier unpublished manuscript [376], Pratt claims a linear-time solution to the problem but the claim doesn't specify whether the problem is for a fixed k or for all values of k. The section where the details were to be presented is not available and was apparently never finished [375].

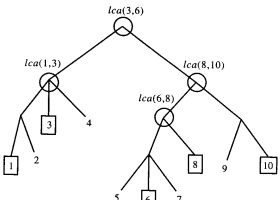


Figure 9.3: The boxed leaves have identifier i. The circled internal nodes are the lowest common ancestors of the four adjacent pairs of leaves from list L_i.

y are any two leaves in $L_i(v)$, then the *lca* of x and y is a node in the subtree of v. So if we compute the *lca* for each consecutive pair of leaves in $L_i(v)$, then all of the $n_i(v) - 1$ computed *lca*s will be found in the subtree of v. Further, if x and y are not both in the subtree of v, then the *lca* of x and y will not be a node in v's subtree. This leads to the following lemma and method.

Lemma 9.7.2. *If we compute the lca for each consecutive pair of leaves in L_i, then for any node v, exactly $n_i(v) - 1$ of the computed lcas will lie in the subtree of v.*

Lemma 9.7.2 is illustrated in Figure 9.3.

Given the lemma, we can compute $n_i(v) - 1$ for each node v as follows: Compute the *lca* of each consecutive pair of leaves in L_i, and accumulate for each node w a count of the number of times that w is the computed *lca*. Let $h(w)$ denote that count for node w. Then for any node v, $n_i(v) - 1$ is exactly $\sum[h(w) : w$ is in the subtree of $v]$. A standard $O(n)$-time bottom-up traversal of $\mathcal{T}$ can therefore be used to find $n_i(v) - 1$ for each node v.

To find $U(v)$, we don't want $n_i(v) - 1$ but rather $\sum_i [n_i(v) - 1]$. However, the algorithm must not do a separate bottom-up traversal for each identifier, since then the time bound would then be $O(Kn)$. Instead, the algorithm should defer the bottom-up traversal until each list L_i has been processed, and it should let $h(w)$ count the total number of times that w is the computed *lca* over all of the lists. Only then is a single bottom-up traversal of $\mathcal{T}$ done. At that point, $U(v) = \sum_{i:n_i>0}[n_i(v) - 1] = \sum[h(w) : w$ is in the subtree of $v]$.

We can now summarize the entire $O(n)$ method for solving the k-common substring problem.

Multiple common substring algorithm

Begin

1. Build a generalized suffix tree $\mathcal{T}$ for the K strings.
2. Number the leaves of $\mathcal{T}$ as they are encountered in a depth-first traversal of $\mathcal{T}$.
3. For each string identifier i, extract the ordered list L_i of leaves with identifier i. (The minor implementation detail needed to do this in $O(n)$ total time is left to the reader.)
4. For each node w in $\mathcal{T}$ set $h(w)$ to zero.

5. For each identifier i, compute the *lca* of each consecutive pair of leaves in L_i, and increment $h(w)$ by one each time that w is the computed *lca*.
6. With a bottom-up traversal of $\mathcal{T}$, compute, for each node v, $S(v)$ and $U(v) = \sum_{i:n_i>0}[n_i(v) - 1] = \sum[h(w) : w$ is in the subtree of $v]$.
7. Set $C(v) = S(v) - U(v)$ for each node v.
8. Accumulate the table of $l(k)$ values as detailed in Section 7.6.

End.

9.7.2. Time analysis

The size of the suffix tree is $O(n)$ and preprocessing of the tree for *lca* computations is done in $O(n)$ time. There are then $\sum_{i=1}^{K} |n_i - 1| < n$ *lca* computations done, each of which takes constant time, so all the *lca* computations take $O(n)$ time in total. Hence only $O(n)$ time is needed to compute all $C(v)$ values. Once these are known, only $O(n)$ additional time is needed to build the output table. That part of the algorithm is the same as in the previously discussed $O(Kn)$-time algorithm of Section 7.6. Therefore, we can state

Theorem 9.7.1. *Let $\mathcal{S}$ be a set of K strings of total length n, and let $l(k)$ denote the length of the longest substring that appears in at least k distinct strings of $\mathcal{S}$. A table of all $l(k)$ values, for k from 2 to K, can be built in $O(n)$ time.*

That so much information about the substrings of $\mathcal{S}$ can be obtained in time proportional to the time needed just to read the strings is very impressive. It would be a good challenge to try to obtain this result without the use of suffix trees (or a similar data structure).

9.7.3. Related uses

The methodology developed for the k-common substring problem can be easily extended to solve related and important problems about sets of strings.

For example, suppose you are given two sets of strings $\mathcal{S}$ and $\mathcal{P}$, and you want to know for each string $A \in \mathcal{P}$, in how many strings of $\mathcal{S}$ does A appear. Let n denote the total size of all the strings in $\mathcal{S}$ and m denote the total size of all the strings in $\mathcal{P}$. The problem can be solved in $O(n + m)$ time, the same time bound attainable via the Aho–Corasick method. Or one could consider another problem: Given a length l, find the string of length at least l that appears in the most strings in a set given of strings. That is, find the most common substring of length at least l. That problem has applications in many multiple alignment methods. See Exercise 26 in Chapter 14.

9.8. Exercises

1. Prove Theorem 9.1.1.
2. Fill in all the details and prove the correctness of the space-efficient method solving the longest common extension problem.
3. Give the details for finding all odd-length maximal palindromes in a string in linear time.
4. Show how to solve all the palindrome problems in linear time using just a suffix tree for the string S rather than for both S and S'.
5. Give the details for searching for complemented palindromes in a linear string.

6. Recall that a *plasmid* is a circular DNA molecule common in bacteria (and elsewhere). Some bacterial plasmids contain relatively long complemented palindromes (whose function is somewhat in question). Give a linear-time algorithm to find all maximal complemented palindromes in a *circular* string.

7. Show how to find all the k-mismatch palindromes in a string of length n in $O(kn)$ time.

8. **Tandem repeats.** In the recursive method discussed in Section 9.6 (page 202) for finding the tandem repeats (no mismatches), problem 3 is solved with a linear number of constant-time common extension queries, exploiting suffix trees and lowest common ancestor computations. An earlier, equally efficient, solution to problem 3 was developed by Main and Lorenz [307], without using suffix trees.

   The idea is that the problem can be solved in an *amortized* linear-time bound without suffix trees. In an instance of problem 3, h is held fixed while $q = h + l - 1$ varies over all appropriate values of l. Each forward common extension query is a problem of finding *the length of the longest substring beginning at position q that matches a prefix of S[h . . . n]*. All those lengths must be found in linear time. But that objective can be achieved by computing Z values (again) from Chapter 1, for the appropriate substring of S. Flesh out the details of this approach and prove the linear amortized time bound.

   Now show how the backward common extensions can also be solved in linear time by computing Z values on the appropriately constructed substring of S. This substring is a bit less direct than the one used for forward extensions.

9. Complete the details for the $O(kn \log n + z)$-time algorithm for the k-mismatch tandem repeat problem. Consider both correctness and time.

10. Complete the details for the $O(kn \log(n/k) + z)$ bound for the k-mismatch tandem repeat method.

11. Try to modify the Main and Lorenz method for finding all the tandem repeats (without errors) to solve the k-mismatch tandem repeat problem in $O(kn \log n + z)$ time. If you are not successful, explain what the difficulties are and how the use of suffix trees and common ancestors solves these problems.

12. The tandem repeat method detailed in Section 9.6 finds all tandem repeats even if they are not maximal. For example, it finds six tandem repeats in the string *xababababy*, even though the left-most tandem repeat *abab* is contained in the longer tandem repeat *abababab*. Depending on the application, that output may not be desirable. Give a definition of *maximality* that would reduce the size of the output and try to give efficient algorithms for the different definitions.

13. Consider the following situation: A long string S is given and remains fixed. Then a sequence of shorter strings $S_1, S_2, \ldots, S_k$ is given. After each string S_i is given (but before S_{i+1} is known), a number of longest common extension queries will be asked about S_i and S. Let r denote the total number of queries and n denote the total length of all the short strings. How can these on-line queries be answered efficiently? The most direct approach is to build a generalized suffix tree for both S and S_i when S_i is presented, preprocess it (do a depth-first traversal assigning *dfs* numbers, setting $l()$ values, etc.) for the constant-time *lca* algorithm, and then answer the queries for S_i. But that would take $\Theta(k|S| + n + r)$ time. The $k|S|$ term comes from two sources: the time to build the k generalized suffix trees and the time to preprocess each of them for *lca* queries.

    Reduce that $k|S|$ term from both sources to $|S|$, obtaining an overall bound of $O(|S| + n + r)$. Reducing the time for building all the generalized suffix trees is easy. Reducing the time for the *lca* preprocessing takes a bit more thought.

    Find a plausible application of the above result.

PART III

Inexact Matching, Sequence Alignment, and Dynamic Programming

At this point we shift from the general area of *exact* matching and exact pattern discovery to the general area of *inexact, approximate* matching, and sequence *alignment*. "Approximate" means that some errors, of various types detailed later, are acceptable in valid matches. "Alignment" will be given a precise meaning later, but generally means lining up characters of strings, allowing mismatches as well as matches, and allowing characters of one string to be placed opposite spaces made in opposing strings.

We also shift from problems primarily concerning *substrings* to problems concerning *subsequences*. A subsequence differs from a substring in that the characters in a substring *must* be contiguous, whereas the characters in a subsequence embedded in a string need not be.[1] For example, the string xyz is a subsequence, but not a substring, in *axayaz*. The shift from substrings to subsequences is a natural corollary of the shift from exact to inexact matching. This shift of focus to inexact matching and subsequence comparison is accompanied by a shift in *technique*. Most of the methods we will discuss in Part III, and many of the methods in Part IV, rely on the tool of *dynamic programming*, a tool that was not needed in Parts I and II.

Much of computational biology concerns sequence alignments

The area of approximate matching and sequence comparison is central in computational molecular biology both because of the presence of errors in molecular data and because of active mutational processes that (sub)sequence comparison methods seek to model and reveal. This will be elaborated in the next chapter and illustrated throughout the book. On the technical side, sequence alignment has become the central tool for sequence comparison in molecular biology. Henikoff and Henikoff [222] write:

> Among the most useful computer-based tools in modern biology are those that involve sequence alignments of proteins, since these alignments often provide important insights into gene and protein function. There are several different types of alignments: global alignments of pairs of proteins related by common ancestry throughout their lengths, local alignments involving related segments of proteins, multiple alignments of members of protein families, and alignments made during data base searches to detect homologies.

This statement provides a framework for much of Part III. We will examine in detail the four types of alignments (and several variants) mentioned above. We will also show how those different alignment models address different kinds of problems in biology. We begin, in Chapter 10, with a more detailed statement of why sequence comparison has become central to current molecular biology. But we won't forget the role of exact matching.

[1] It is a common and confusing practice in the biological literature to refer to a substring as a subsequence. But techniques and results for substring problems can be very different from techniques and results for the analogous subsequence problems, so it is important to maintain a clear distinction. In this book we will never use the term "subsequence" when "substring" is intended.

The role of exact matching

The centrality of *approximate* matching in molecular biology is undisputed. However, it does not follow that exact matching methods have little application there, and several biological applications of exact matching were developed in Parts I and II. As one example, recall from Section 7.15, that suffix trees are now playing a central role in several biological database efforts. Moreover, several exact matching techniques were shown earlier to directly extend or apply to approximate matching problems (the match-count problem, the wild-card problem, the k-mismatch problem, the k-mismatch palindrome problem, and the k-mismatch tandem repeat problem). In Parts III and IV we will develop additional approximate matching techniques that rely in a crucial way on efficient exact matching methods, suffix trees, etc. We will also see exact matching problems that arise as subproblems in multiple sequence comparison, in large-scale sequence comparison, in database searching, and in other biologically important applications.

10

The Importance of (Sub)sequence Comparison in Molecular Biology

Sequence comparison, particularly when combined with the systematic collection, curration, and search of databases containing biomolecular sequences, has become essential in modern molecular biology. Commenting on the (then) near-completion of the effort to sequence the entire yeast genome (now finished), Stephen Oliver says

> In a short time it will be hard to realize how we managed without the sequence data. Biology will never be the same again. [478]

One fact explains the importance of molecular sequence data and sequence comparison in biology.

The first fact of biological sequence analysis

The *first fact* of biological sequence analysis In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.

Evolution reuses, builds on, duplicates, and modifies "successful" structures (proteins, exons, DNA regulatory sequences, morphological features, enzymatic pathways, etc.). Life is based on a repertoire of structured and interrelated molecular building blocks that are shared and passed around. The same and related molecular structures and mechanisms show up repeatedly in the genome of a single species and across a very wide spectrum of divergent species. "Duplication with modification" [127, 128, 129, 130] is the central paradigm of protein evolution, wherein new proteins and/or new biological functions are fashioned from earlier ones. Doolittle emphasizes this point as follows:

> The vast majority of extant proteins are the result of a continuous series of genetic duplications and subsequent modifications. As a result, redundancy is a built-in characteristic of protein sequences, and we should not be surprised that so many new sequences resemble already known sequences. [129]

He adds that

> ... all of biology is based on an enormous redundancy ....[130]

The following quotes reinforce this view and suggest the utility of the "enormous redundancy" in the practice of molecular biology. The first quote is from Eric Wieschaus, cowinner of the 1995 Nobel prize in medicine for work on the genetics of *Drosophila* development. The quote is taken from an Associated Press article of October 9, 1995. Describing the work done years earlier, Wieschaus says

> We didn't know it at the time, but we found out everything in life is so similar, that the same genes that work in flies are the ones that work in humans.

And fruit flies aren't special. The following is from a book review on DNA repair [424]:

> Throughout the present work we see the insights gained through our ability to look for sequence homologies by comparison of the DNA of different species. Studies on yeast are remarkable predictors of the human system!

So "redundancy", and "similarity" are central phenomena in biology. But similarity has its limits – humans and flies do differ in some respects. These differences make *conserved* similarities even more significant, which in turn makes *comparison* and *analogy* very powerful tools in biology. Lesk [297] writes:

> It is characteristic of biological systems that objects that we observe to have a certain form arose by evolution from related objects with similar but not identical from. They must, therefore, be robust, in that they retain the freedom to tolerate some variation. We can take advantage of this robustness in our analysis: By identifying and comparing related objects, we can distinguish variable and conserved features, and thereby determine what is crucial to structure and function.

The important "related objects" to compare include much more than sequence data, because biological universality occurs at many levels of detail. However, it is usually easier to acquire and examine sequences than it is to examine fine details of genetics or cellular biochemistry or morphology. For example, there are vastly more protein sequences known (deduced from underlying DNA sequences) than there are known three-dimensional protein structures. And it isn't just a matter of convenience that makes sequences important. Rather, the biological sequences *encode* and reflect the more complex common molecular structures and mechanisms that appear as features at the cellular or biochemical levels. Moreover, "nowhere in the biological world is the Darwinian notion of 'descent with modification' more apparent than in the sequences of genes and gene products" [130]. Hence a tractable, though partly heuristic, way to search for functional or structural universality in biological systems is to search for similarity and conservation at the *sequence* level. The power of this approach is made clear in the following quotes:

> Today, the most powerful method for inferring the biological function of a gene (or the protein that it encodes) is by sequence similarity searching on protein and DNA sequence databases. With the development of rapid methods for sequence comparison, both with heuristic algorithms and powerful parallel computers, discoveries based solely on sequence homology have become routine. [360]

> Determining function for a sequence is a matter of tremendous complexity, requiring biological experiments of the highest order of creativity. Nevertheless, with only DNA sequence it is possible to execute a computer-based algorithm comparing the sequence to a database of previously characterized genes. In about 50% of the cases, such a mechanical comparison will indicate a sufficient degree of similarity to suggest a putative enzymatic or structural function that might be possessed by the unknown gene. [91]

Thus large-scale sequence comparison, usually organized as database search, is a very powerful tool for biological inference in modern molecular biology. And that tool is almost universally used by molecular biologists. It is now standard practice, whenever a new gene is cloned and sequenced, to translate its DNA sequence into an amino acid sequence and then search for similarities between it and members of the protein databases. No one today would even think of publishing the sequence of a newly cloned gene without doing such database searches.

The final quote reflects the potential total impact on biology of the *first fact* and its exploitation in the form of sequence database searching. It is from an article [179] by Walter Gilbert, Nobel prize winner for the coinvention of a practical DNA sequencing method. Gilbert writes:

> The new paradigm now emerging, is that all the 'genes' will be known (in the sense of being resident in databases available electronically), and that the starting point of biological investigation will be theoretical. An individual scientist will begin with a theoretical conjecture, only then turning to experiment to follow or test that hypothesis.

Already, hundreds (if not thousands) of journal publications appear each year that report biological research where sequence comparison and/or database search is an integral part of the work. Many such examples that support and illustrate the *first fact* are distributed throughout the book. In particular, several in-depth examples are concentrated in Chapters 14 and 15 where multiple string comparison and database search are discussed. But before discussing those examples, we must first develop, in the next several chapters, the techniques used for approximate matching and (sub)sequence comparison.

Caveat

The *first fact of biological sequence analysis* is extremely powerful, and its importance will be further illustrated throughout the book. However, there is not a one-to-one correspondence between sequence and structure or sequence and function, because the converse of the *first fact* is not true. That is, high sequence similarity usually implies significant structural or functional similarity (the first fact), but structural or functional similarity does not necessarily imply sequence similarity. On the topic of protein structure, F. Cohen [106] writes "... similar sequences yield similar structures, but quite distinct sequences can produce remarkably similar structures". This *converse* issue is discussed in greater depth in Chapter 14, which focuses on multiple sequence comparison.

11

Core String Edits, Alignments, and Dynamic Programming

11.1. Introduction

In this chapter we consider the inexact matching and alignment problems that form the core of the field of inexact matching and others that illustrate the most general techniques. Some of those problems and techniques will be further refined and extended in the next chapters. We start with a detailed examination of the most classic inexact matching problem solved by dynamic programming, the *edit distance* problem. The motivation for inexact matching (and, more generally, sequence comparison) in molecular biology will be a recurring theme explored throughout the rest of the book. We will discuss many specific examples of how string comparison and inexact matching are used in current molecular biology. However, to begin, we concentrate on the purely formal and technical aspects of defining and computing inexact matching.

11.2. The edit distance between two strings

Frequently, one wants a measure of the difference or *distance* between two strings (for example, in evolutionary, structural, or functional studies of biological strings; in textual database retrieval; or in spelling correction methods). There are several ways to formalize the notion of distance between strings. One common, and simple, formalization [389, 299], called *edit distance*, focuses on *transforming* (or editing) one string into the other by a series of edit operations on individual characters. The permitted edit operations are *insertion* of a character into the first string, the *deletion* of a character from the first string, or the *substitution* (or *replacement*) of a character in the first string with a character in the second string. For example, letting I denote the insert operation, D denote the delete operation, R the substitute (or replace) operation, and M the nonoperation of "match," then the string "*vintner*" can be edited to become "*writers*" as follows:

```
RIMDMDMMI
v intner
wri t ers
```

That is, v is replaced by w, r is inserted, i matches and is unchanged since it occurs in both strings, n is deleted, t is unchanged, n is deleted, er match and are unchanged, and finally s is inserted. We now more formally define edit transcripts and string transformations.

Definition A string over the alphabet I, D, R, M that describes a transformation of one string to another is called an *edit transcript*, or transcript for short, of the two strings.

In general, given the two input strings S_1 and S_2, and given an edit transcript for S_1 and S_2, the transformation is accomplished by successively applying the specified operation in the transcript to the next character(s) in the appropriate string(s). In particular, let $next_1$ and

next$_2$ be pointers into S_1 and S_2. Both pointers begin with value one. The edit transcript is read and applied left to right. When symbol "I" is encountered, character *next*$_2$ is inserted before character *next*$_1$ in S_1, and pointer *next*$_2$ is incremented one character. When "D" is encountered, character *next*$_1$ is deleted from S_1 and *next*$_1$ is incremented by one character. When either symbol "R" or "M" is encountered, character *next*$_1$ in S_1 is replaced or matched by character *next*$_2$ from S_2, and then both pointers are incremented by one.

Definition The *edit distance* between two strings is defined as the minimum number of edit operations – insertions, deletions, and substitutions – needed to transform the first string into the second. For emphasis, note that matches are not counted.

Edit distance is sometimes referred to as *Levenshtein distance* in recognition of the paper [299] by V. Levenshtein where edit distance was probably first discussed.

We will sometimes refer to an edit transcript that uses the minimum number of edit operations as an *optimal transcript*. Note that there may be more than one optimal transcript. These will be called "cooptimal" transcripts when we want to emphasize the fact that there is more than one optimal.

The **edit distance problem** is to compute the edit distance between two given strings, along with an optimal edit transcript that describes the transformation.

The definition of edit distance implies that all operations are done to one string only. But edit distance is sometimes thought of as the minimum number of operations done on either of the two strings to transform both of them into a common third string. This view is equivalent to the above definition, since an insertion in one string can be viewed as a deletion in the other and vice versa.

11.2.1. String alignment

An edit transcript is a way to *represent* a particular transformation of one string to another. An alternate (and often preferred) way is by displaying an explicit *alignment* of the two strings.

Definition A (global) *alignment* of two strings S_1 and S_2 is obtained by first inserting chosen spaces (or dashes), either into or at the ends of S_1 and S_2, and then placing the two resulting strings one above the other so that every character or space in either string is opposite a unique character or a unique space in the other string.

The term "global" emphasizes the fact that for each string, the entire string is involved in the alignment. This will be contrasted with local alignment to be discussed later. Notice that our use of the word "alignment" is now much more precise than its use in Parts I and II. There, alignment was used in the colloquial sense to indicate how one string is placed relative to the other, and spaces were not then allowed in either string.

As an example of a global alignment, consider the alignment of the strings *qacdbd* and *qawxb* shown below:

```
q  a  c  -  d  b  d
q  a  w  x  -  b  -
```

In this alignment, character c is mismatched with w, both the d's and the x are opposite spaces, and all other characters match their counterparts in the opposite string.

Another example of an alignment is shown on page 215 where *vintner* and *writers* are aligned with each other below their edit transcript. That example also suggests a duality between alignment and edit transcript that will be developed below.

Alignment versus edit transcript

From the mathematical standpoint, an alignment and an edit transcript are equivalent ways to describe a relationship between two strings. An alignment can be easily converted to the equivalent edit transcript and vice versa, as suggested by the *vintner–writers* example. Specifically, two opposing characters that mismatch in an alignment correspond to a substitution in the equivalent edit transcript; a space in an alignment contained in the first string corresponds in the transcript to an insertion of the opposing character into the first string; and a space in the second string corresponds to a deletion of the opposing character from the first string. Thus the edit distance of two strings is given by the alignment minimizing the number of opposing characters that mismatch plus the number of characters opposite spaces.

Although an alignment and an edit transcript are mathematically equivalent, from a modeling standpoint, an edit transcript is quite different from an alignment. An edit transcript emphasizes the putative *mutational events* (point mutations in the model so far) that transform one string to another, whereas an alignment only displays a relationship between the two strings. The distinction is one of *process* versus *product*. Different evolutionary models are formalized via different permitted string operations, and yet these can result in the same alignment. So an alignment alone blurs the mutational model. This is often a pedantic point but proves helpful in some discussions of evolutionary modeling.

We will switch between the language of edit transcript and alignment as is convenient. However, the language of alignment will often be preferred since it is more neutral, making no statement about process. And, the language of alignment will be more natural in the area of multiple sequence comparison.

11.3. Dynamic programming calculation of edit distance

We now turn to the algorithmic question of how to compute, via dynamic programming, the edit distance of two strings along with the accompanying edit transcript or alignment. The general paradigm of dynamic programming is probably well known to the readers of this book. However, because it is such a crucial tool and is used in so many string algorithms, it is worthwhile to explain in detail both the general dynamic programming approach and its specific application to the edit distance problem.

Definition For two strings S_1 and S_2, $D(i, j)$ is defined to be the edit distance of $S_1[1..i]$ and $S_2[1..j]$.

That is, $D(i, j)$ denotes the minimum number of edit operations needed to transform the first i characters of S_1 into the first j characters of S_2. Using this notation, if S_1 has n letters and S_2 has m letters, then the edit distance of S_1 and S_2 is *precisely* the value $D(n, m)$.

We will compute $D(n, m)$ by solving the more general problem of computing $D(i, j)$ for all combinations of i and j, where i ranges from zero to n and j ranges from zero to m. This is the standard *dynamic programming* approach used in a vast number of computational problems. The dynamic programming approach has three essential components – the *recurrence relation*, the *tabular computation*, and the *traceback*. We will explain each one in turn.

11.3.1. The recurrence relation

The recurrence relation establishes a *recursive* relationship between the value of $D(i, j)$, for i and j both positive, and values of D with index pairs smaller than i, j. When there are no smaller indices, the value of $D(i, j)$ must be stated explicitly in what are called the *base conditions* for $D(i, j)$.

For the edit distance problem, the base conditions are

$$D(i, 0) = i$$

and

$$D(0, j) = j.$$

The base condition $D(i, 0) = i$ is clearly correct (that is, it gives the number required by the definition of $D(i, 0)$) because the only way to transform the first i characters of S_1 to zero characters of S_2 is to delete all the i characters of S_1. Similarly, the condition $D(0, j) = j$ is correct because j characters must be inserted to convert zero characters of S_1 to j characters of S_2.

The recurrence relation for $D(i, j)$ when both i and j are strictly positive is

$$D(i, j) = \min[D(i-1, j)+1,\ D(i, j-1)+1,\ D(i-1, j-1)+t(i, j)],$$

where $t(i, j)$ is defined to have value 1 if $S_1(i) \neq S_2(j)$, and $t(i, j)$ has value 0 if $S_1(i) = S_2(j)$.

Correctness of the general recurrence

We establish correctness in the next two lemmas using the concept of an edit transcript.

Lemma 11.3.1. *The value of $D(i, j)$ must be $D(i, j-1)+1$, $D(i-1, j)+1$, or $D(i-1, j-1)+t(i, j)$. There are no other possibilities.*

PROOF Consider an edit transcript for the transformation of $S_1[1..i]$ to $S_2[1..j]$ using the minimum number of edit operations, and focus on the last symbol in that transcript. That last symbol must either be I, D, R, or M. If the last symbol is an I then the last edit operation is the insertion of character $S_2(j)$ onto the end of the (transformed) first string. It follows that the symbols in the transcript before that I must specify the minimum number of edit operations to transform $S_1[1..i]$ to $S_2[1..j-1]$ (if they didn't, then the specified transformation of $S_1[1..i]$ to $S_2[1..j]$ would use more than the minimum number of operations). By definition, that latter transformation takes $D(i, j-1)$ edit operations. Hence if the last symbol in the transcript is I, then $D(i, j) = D(i, j-1) + 1$.

Similarly, if the last symbol in the transcript is a D, then the last edit operation is the deletion of $S_1(i)$, and the symbols in the transcript to the left of that D must specify the minimum number of edit operations to transform $S_1[1..i-1]$ to $S_2[1..j]$. By definition, that latter transformation takes $D(i-1, j)$ edit operations. So if the last symbol in the transcript is D, then $D(i, j) = D(i-1, j)+1$.

If the last symbol in the transcript is an R, then the last edit operation replaces $S_1(i)$ with $S_2(j)$, and the symbols to the left of R specify the minimum number of edit operations to transform $S_1[1..i-1]$ to $S_2[1..j-1]$. In that case $D(i, j) = D(i-1, j-1)+1$. Finally, and by similar reasoning, if the last symbol in the transcript is an M, then $S_1(i) = S_2(j)$ and $D(i, j) = D(i-1, j-1)$. Using the variable $t(i, j)$ introduced earlier [i.e., that $t(i, j) = 0$ if $S_1(i) = S_2(j)$; otherwise $t(i, j) = 1$] we can combine these last two cases as one: If the last transcript symbol is R or M, then $D(i, j) = D(i-1, j-1)+t(i, j)$.

Since the last transcript symbol must either be I, D, R, or M, we have covered all cases and established the lemma. □

Now we look at the other side.

Lemma 11.3.2. $D(i, j) \leq \min[D(i-1, j)+1,\ D(i, j-1)+1,\ D(i-1, j-1)+t(i, j)]$.

PROOF The reasoning is very similar to that used in the previous lemma, but it achieves a somewhat different goal. The objective here is to demonstrate constructively the existence of transformations achieving each of the three values specified in the inequality. Then since all three values are feasible, their minimum is certainly feasible.

First, it *is* possible to transform $S_1[1..i]$ into $S_2[1..j]$ with exactly $D(i, j-1)+1$ edit operations. Simply transform $S_1[1..i]$ to $S_2[1..j-1]$ with the minimum number of edit operations, and then use one more to insert character $S_2(j)$ at the end. By definition, the number of edit operations in that particular way to transform S_1 to S_2 is exactly $D(i, j-1)+1$. Second, it *is* possible to transform $S_1[1..i]$ to $S_2[1..j]$ with exactly $D(i-1, j)+1$ edit operations. Transform $S_1[1..i-1]$ to $S_2[1..j]$ with the fewest operations, and then delete character $S_1(i)$. The number of edit operations in that particular transformation is exactly $D(i-1, j)+1$. Third, it *is* possible to do the transformation with exactly $D(i-1, j-1)+t(i, j)$ edit operations, using the same argument. □

Lemmas 11.3.1 and 11.3.2 immediately imply the correctness of the general recurrence relation for $D(i, j)$.

Theorem 11.3.1. *When both i and j are strictly positive,* $D(i, j) = \min[D(i-1, j)+1,\ D(i, j-1)+1,\ D(i-1, j-1)+t(i, j)]$.

PROOF Lemma 11.3.1 says that $D(i, j)$ must be equal to one of the three values $D(i-1, j)+1$, $D(i, j-1)+1$, or $D(i-1, j-1)+t(i, j)$. Lemma 11.3.2 says that $D(i, j)$ must be *less than or equal* to the smallest of those three values. It follows that $D(i, j)$ must therefore be equal to the smallest of those three values, and we have proven the theorem. □

This completes the first component of the dynamic programming method for edit distance, the recurrence relation.

11.3.2. Tabular computation of edit distance

The second essential component of any dynamic program is to use the recurrence relations to efficiently compute the value $D(n, m)$. We could easily code the recurrence relations and base conditions for $D(i, j)$ as a recursive computer procedure using any programming language that allows recursion. Then we could call that procedure with input m, n and sit back and wait for the answer.[1] This *top-down* recursive approach to evaluating $D(n, m)$ is simple to program but extremely inefficient for large values of n and m.

The problem is that the number of recursive calls grows exponentially with n and m (an easy exercise to establish). But there are only $(n+1) \times (m+1)$ combinations of i and j, so there are only $(n+1) \times (m+1)$ *distinct* recursive calls possible. Hence the inefficiency of the top-down approach is due to a massive number of redundant recursive calls to the procedure. A nice discussion of this phenomenon is contained in [112]. The key to a (vastly) more efficient computation of $D(n, m)$ is to abandon the simplicity of top-down computation and instead compute *bottom-up*.

[1] and wait, and wait, ...

$D(i,j)$		w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1							
i	2	2							
n	3	3							
t	4	4							
n	5	5							
e	6	6							
r	7	7							

Figure 11.1: Table to be used to compute the edit distance between *vintner* and *writers*. The values in row zero and column zero are already included. They are given directly by the base conditions.

Bottom-up computation

In the bottom-up approach, we first compute $D(i, j)$ for the smallest possible values for i and j, and then compute values of $D(i, j)$ for increasing values of i and j. Typically, this bottom-up computation is organized with a dynamic programming table of size $(n+1) \times (m+1)$. The table holds the values of $D(i, j)$ for all the choices of i and j (see Figure 11.1). Note that string S_1 corresponds to the vertical axis of the table, while string S_2 corresponds to the horizontal axis. Because the ranges of i and j begin at zero, the table has a zero row and a zero column. The values in row zero and column zero are filled in directly from the base conditions for $D(i, j)$. After that, the remaining $n \times m$ subtable is filled in one row at time, in order of increasing i. Within each row, the cells are filled in order of increasing j.

To see how to fill in the subtable, note that by the general recurrence relation for $D(i, j)$, all the values needed for the computation of $D(1, 1)$ are known once $D(0, 0)$, $D(1, 0)$, and $D(0, 1)$ have been computed. Hence $D(1, 1)$ can be computed after the zero row and zero column have been filled in. Then, again by the recurrence relations, after $D(1, 1)$ has been computed, all the values needed for the computation of $D(1, 2)$ are known. Following this idea, we see that the values for row one can be computed in order of increasing index j. After that, all the values needed to compute the values in row two are known, and that row can be filled in, in order of increasing j. By extension, the entire table can be filled in one row at a time, in order of increasing i, and in each row the values can be computed in order of increasing j (see Figure 11.2).

Time analysis

How much work is done by this approach? When computing the value for a specific cell (i, j), only cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$ are examined, along with the two characters $S_1(i)$ and $S_2(j)$. Hence, to fill in one cell takes a constant number of cell examinations, arithmetic operations, and comparisons. There are $O(nm)$ cells in the table, so we obtain the following theorem.

Theorem 11.3.2. *The dynamic programming table for computing the edit distance between a string of length n and a string of length m can be filled in with $O(nm)$ work. Hence, using dynamic programming, the edit distance $D(n, m)$ can be computed in $O(nm)$ time.*

$D(i,j)$		w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4	4	4	*			
n	5	5							
e	6	6							
r	7	7							

Figure 11.2: Edit distances are filled in one row at a time, and in each row they are filled in from left to right. The example shows the edit distances $D(i, j)$ to column 3 of row 4. The next value to be computed is $D(4, 4)$, where an asterisk appears. The value for cell (4, 4) is 3, since $S_1(4) = S_2(4) = t$ and $D(3, 3) = 3$.

The reader should be able to establish that the table could also be filled in *columnwise* instead of rowwise, after row zero and column zero have been computed. That is, column one could be first filled in, followed by column two, etc. Similarly, it is possible to fill in the table by filling in successive anti-diagonals. We leave the details as an exercise.

11.3.3. The traceback

Once the value of the edit distance has been computed, how is the associated optimal edit transcript extracted? The easiest way (conceptually) is to establish *pointers* in the table as the table values are computed.

In particular, when the value of cell (i, j) is computed, set a pointer from cell (i, j) to cell $(i, j-1)$ if $D(i, j) = D(i, j-1) + 1$; set a pointer from (i, j) to $(i-1, j)$ if $D(i, j) = D(i-1, j) + 1$; and set a pointer from (i, j) to $(i-1, j-1)$ if $D(i, j) = D(i-1, j-1) + t(i, j)$. This rule applies to cells in row zero and column zero as well. Hence, for most objective functions, each cell in row zero points to the cell to its left, and each cell in column zero points to the cell just above it. For other cells, it is possible (and common) that more than one pointer is set from (i, j). Figure 11.3 shows an example.

The pointers allow easy recovery of an optimal edit transcript: Simply follow *any* path of pointers from cell (n, m) to cell $(0, 0)$. The edit transcript is recovered from the path by interpreting each *horizontal* edge in the path, from cell (i, j) to cell $(i, j-1)$, as an *insertion* (I) of character $S_2(j)$ into S_1; interpreting each *vertical* edge, from (i, j) to $(i-1, j)$, as a *deletion* (D) of $S_1(i)$ from S_1; and interpreting each *diagonal* edge, from (i, j) to $(i-1, j-1)$, as a *match* (M) if $S_1(i) = S_2(j)$ and as a *substitution* (R) if $S_1(i) \neq S_2(j)$. That this traceback path specifies an optimal edit transcript can be proved in a manner similar to the way that the recurrences for edit distances were established. We leave this as an exercise.

Alternatively, in terms of *aligning* S_1 and S_2, each horizontal edge in the path specifies a space inserted into S_1, each vertical edge specifies a space inserted into S_2, and each diagonal edge specifies either a match or a mismatch, depending on the specific characters.

For example, there are three traceback paths from cell (7, 7) to cell (0, 0) in the example given in Figure 11.3. The paths are identical from cell (7, 7) to cell (3, 3), at which point

$D(i,j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6	$\leftarrow$ 7
v	1	$\uparrow$ 1	$\nwarrow$ 1	$\nwarrow\leftarrow$ 2	$\nwarrow\leftarrow$ 3	$\nwarrow\leftarrow$ 4	$\nwarrow\leftarrow$ 5	$\nwarrow\leftarrow$ 6	$\nwarrow\leftarrow$ 7
i	2	$\uparrow$ 2	$\nwarrow\uparrow$ 2	$\nwarrow$ 2	$\nwarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6
n	3	$\uparrow$ 3	$\nwarrow\uparrow$ 3	$\nwarrow\uparrow$ 3	$\nwarrow\uparrow$ 3	$\nwarrow$ 3	$\nwarrow\leftarrow$ 4	$\nwarrow\leftarrow$ 5	$\nwarrow\leftarrow$ 6
t	4	$\uparrow$ 4	$\nwarrow\uparrow$ 4	$\nwarrow\uparrow$ 4	$\nwarrow\uparrow$ 4	$\nwarrow$ 3	$\nwarrow\leftarrow$ 4	$\nwarrow\leftarrow$ 5	$\nwarrow\leftarrow$ 6
n	5	$\uparrow$ 5	$\nwarrow\uparrow$ 5	$\nwarrow\uparrow$ 5	$\nwarrow\uparrow$ 5	$\uparrow$ 4	$\nwarrow$ 4	$\nwarrow\leftarrow$ 5	$\nwarrow\leftarrow$ 6
e	6	$\uparrow$ 6	$\nwarrow\uparrow$ 6	$\nwarrow\uparrow$ 6	$\nwarrow\uparrow$ 6	$\uparrow$ 5	$\nwarrow$ 4	$\nwarrow\leftarrow$ 5	$\nwarrow\leftarrow$ 6
r	7	$\uparrow$ 7	$\nwarrow\uparrow$ 7	$\nwarrow$ 6	$\nwarrow\uparrow$ 7	$\uparrow$ 6	$\uparrow$ 5	$\nwarrow$ 4	$\leftarrow$ 5

Figure 11.3: The complete dynamic programming table with pointers included. The arrow $\leftarrow$ in cell (i, j) points to cell $(i, j-1)$, the arrow $\uparrow$ points to cell $(i-1, j)$, and the arrow $\nwarrow$ points to cell $(i-1, j-1)$.

it is possible to either go up or to go diagonally. The three optimal alignments are:

```
w r i t _ e r s
v i n t n e r _

w r i _ t _ e r s
v _ i n t n e r _
```

and

```
w r i _ t _ e r s
_ v i n t n e r _
```

If there is more than one pointer from cell (n, m), then a path from (n, m) to $(0, 0)$ can start with either of those pointers. Each of them is on a path from (n, m) to $(0, 0)$. This property is repeated from any cell encountered. Hence a traceback path from (n, m) to $(0, 0)$ can start simply by following any pointer out of (n, m); it can then be extended by following any pointer out of any cell encountered. Moreover, every cell except $(0, 0)$ has a pointer out of it, so no path from (n, m) can get stuck. Since any path of pointers from (n, m) to $(0, 0)$ specifies an optimal edit transcript or alignment, we have the following:

Theorem 11.3.3. *Once the dynamic programming table with pointers has been computed, an optimal edit transcript can be found in $O(n + m)$ time.*

We have now completely described the three crucial components of the general dynamic programming paradigm, as illustrated by the edit distance problem. We will later consider ways to increase the speed of the solution and decrease its needed space.

The pointers represent all optimal edit transcripts

The pointers that are built up while computing the values of the table do more than allow one optimal transcript (or optimal alignment) to be retrieved. They allow *all* optimal transcripts to be retrieved.

Theorem 11.3.4. *Any path from (n, m) to $(0, 0)$ following pointers established during the computation of $D(i, j)$ specifies an edit transcript with the minimum number of edit operations. Conversely, any optimal edit transcript is specified by such a path. Moreover, since a path describes only one transcript, the correspondence between paths and optimal transcripts is one-to-one.*

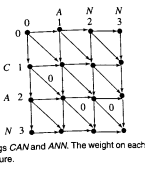


Figure 11.4: Edit graph for the strings *CAN* and *ANN*. The weight on each edge is one, except for the three zero-weight edges marked in the figure.

The theorem can be proven by essentially the same reasoning that established the correctness of the recurrence relations for $D(i, j)$, and this is left to the reader. An alternative way to find the optimal edit transcript(s), without using pointers, is discussed in Exercise 9. Once the pointers have been established, all the cooptimal edit transcripts can be enumerated in $O(n + m)$ time per transcript. That is the focus of Exercise 12.

11.4. Edit graphs

It is often useful to represent dynamic programming solutions of string problems in terms of a *weighted edit graph*.

Definition Given two strings S_1 and S_2 of lengths n and m, respectively, a *weighted edit graph* has $(n + 1) \times (m + 1)$ nodes, each labeled with a distinct pair (i, j) $(0 \leq i \leq n, 0 \leq j \leq m)$. The specific edges and their edge weights depend on the specific string problem.

In the case of the edit distance problem, the edit graph contains a directed edge from each node (i, j) to each of the nodes $(i, j + 1)$, $(i + 1, j)$, and $(i + 1, j + 1)$, provided those nodes exist. The weight on the first two of these edges is one; the weight on the third (diagonal) edge is $t(i + 1, j + 1)$. Figure 11.4 shows the edit graph for strings *CAN* and *ANN*.

The central property of an edit graph is that any *shortest path* (one whose total weight is minimum) from start node $(0, 0)$ to destination node (n, m) specifies an edit transcript with the minimum number of edit operations. Equivalently, any shortest path specifies a global alignment of minimum total weight. Moreover, the following theorem and corollary can be stated.

Theorem 11.4.1. *An edit transcript for S_1, S_2 has the minimum number of edit operations if and only if it corresponds to a shortest path from $(0, 0)$ to (n, m) in the edit graph.*

Corollary 11.4.1. The set of all shortest paths from $(0, 0)$ to (n, m) in the edit graph exactly specifies the set of all optimal edit transcripts of S_1 to S_2. Equivalently, it specifies all the optimal (minimum weight) alignments of S_1 and S_2.

Viewing dynamic programming as a shortest path problem is often useful because there

are many tools for investigating and compactly representing shortest paths in graphs. This view will be exploited in Section 13.2 when suboptimal solutions are discussed.

11.5. Weighted edit distance

11.5.1. Operation weights

An easy, yet crucial, generalization of edit distance is to allow an arbitrary *weight* or *cost* or *score*[2] to be associated with every edit operation, as well as with a match. Thus, any insertion or deletion has a weight denoted d, a substitution has a weight r, and a match has a weight e (which is usually small compared to the other weights and is often zero). Equivalently, an *operation-weight alignment* is one where each mismatch costs r, each match costs e, and each space costs d.

Definition With arbitrary operation weights, the *operation-weight edit distance problem* is to find an edit transcript that transforms string S_1 into S_2 with the minimum total operation weight.

In these terms, the edit distance problem we have considered so far is just the problem of finding the minimum operation-weight edit transcript when $d = 1, r = 1$, and $e = 0$. But, for example, if each mismatch has a weight of 2, each space has a weight of 4, and each match a weight of 1, then the alignment

$$\begin{array}{cccccccc} w & r & i & t & - & e & r & s \\ v & i & n & t & n & e & r & - \end{array}$$

has a total weight of 17 and is an optimal alignment.

Because the objective function is to minimize total weight and because a substitution can be achieved by a deletion followed by an insertion, if substitutions are to be allowed then a substitution weight should be less than the sum of the weights for a deletion plus an insertion.

Computing operation-weight edit distance

The operation-weight edit distance problem for two strings of length n and m can be solved in $O(nm)$ time by a minor extension of the recurrences for edit distance. $D(i, j)$ now denotes the minimum total weight for edit operations transforming $S_1[1..i]$ to $S_2[1..j]$. We again use $t(i, j)$ to handle both substitution and equality, where now $t(i, j) = e$ if $S_1(i) = S_2(j)$; otherwise $t(i, j) = r$. Then the base conditions are

$$D(i, 0) = i \times d$$

and

$$D(0, j) = j \times d.$$

The general recurrence is

$$D(i, j) = \min[D(i, j-1) + d, D(i-1, j) + d, D(i-1, j-1) + t(i, j)].$$

[2] The terms "weight" or "cost" are heavily used in the computer science literature, while the term "score" is used in the biological literature. We will use these terms more or less interchangeably in discussing algorithms, but the term "score" will be used when talking about specific biological applications.

The operation-weight edit distance problem can also be represented and solved as a shortest path problem on a weighted edit graph, where the edge weights correspond in the natural way to the weights of the edit operations. The details are straightforward and are thus left to the reader.

11.5.2. Alphabet-weight edit distance

Another critical, yet simple, generalization of edit distance is to allow the weight or score of a substitution to depend on exactly which character in the alphabet is being removed and which is being added. For example, it may be more costly to replace an A with a T than with a G. Similarly, we may want the weight of a deletion or insertion to depend on exactly which character in the alphabet is being deleted or inserted. We call this form of edit distance the *alphabet-weight edit distance* to distinguish it from the operation-weight edit distance problem.

The operation-weight edit distance problem is a special case of the alphabet-weight problem, and it is trivial to modify the previous recurrence relations (for operation-weight edit distance) to compute alphabet-weight edit distance. We leave that as an exercise. We will usually use the simple term *weighted edit distance* when we mean the alphabet-weight version. Notice that in weighted edit distance, the weight of an operation depends on what characters are involved in an operation but not on *where* those characters appear in the string.

When comparing proteins, "the edit distance" almost always means the alphabet-weight edit distance over the alphabet of amino acids. There is an extensive literature (and continuing research) on what scores should be used for operations on amino acid characters and how they should be determined. The dominant amino acid scoring schemes are now the PAM matrices of Dayhoff [122] and the newer BLOSUM scoring matrices of the Henikoffs [222], although these matrices are actually defined in terms of a maximization problem (similarity) rather than edit distance.[3] Recently, a mathematical theory has been developed [16, 262] concerning the way scores should be interpreted and how a scoring scheme should relate both to the data it is obtained from and to the types of searches it is designed for. We will briefly discuss this issue again in Section 15.11.2.

When comparing DNA strings, unweighted or operation-weight edit distance is more often computed. For example, the popular database searching program, BLAST, scores identities as $+5$ and mismatches as -4. However, alphabet-weighted edit distance is also of interest and alphabet-based scoring schemes for DNA have been suggested (for example see [252]).

11.6. String similarity

Edit distance is one of the ways that the relatedness of two strings has been formalized. An alternate, and often preferred, way of formalizing the relatedness of two strings is to measure their *similarity* rather than their distance. This approach is chosen in most biological applications for technical reasons that should be clear later. When focusing on

[3] In a pure computer science or mathematical discussion of alphabet-weight edit distance, we would prefer to use the general term "weight matrix" for the matrix holding the alphabet-dependent substitution scores. However, molecular biologists use the terms "amino acid substitution matrix" or "nucleotide substitution matrix" for those matrices, and they use the term "weight matrix" for a very different object (See Section 14.3.1). Therefore, to maintain generality, and yet to keep in some harmony with the molecular biology literature, we will use the general term "scoring matrix".

similarity, the language of alignment is usually more convenient than the language of edit transcript. We now begin to develop a precise definition of similarity.

Definition Let Σ be the alphabet used for strings S_1 and S_2, and let Σ' be Σ with the added character "–" denoting a space. Then, for any two characters x, y in Σ', $s(x, y)$ denotes the value (or *score*) obtained by aligning character x against character y.

Definition For a given alignment $\mathcal{A}$ of S_1 and S_2, let S_1' and S_2' denote the strings after the chosen insertion of spaces, and let l denote the (equal) length of the two strings S_1' and S_2' in $\mathcal{A}$. The *value* of alignment $\mathcal{A}$ is defined as $\sum_{i=1}^{l} s(S_1'(i), S_2'(i))$.

That is, every position i in $\mathcal{A}$ specifies a pair of opposing characters in the alphabet Σ', and the value of $\mathcal{A}$ is obtained by summing the value contributed by each pair.

For example, let $\Sigma = \{a, b, c, d\}$ and let the pairwise scores be defined in the following matrix:

s	a	b	c	d	–
a	1	−1	−2	0	−1
b		3	−2	−1	0
c			0	−4	−2
d				3	−1
–					0

Then the alignment

c	a	c	–	d	b	d
c	a	b	b	d	b	–

has a total value of $0 + 1 - 2 + 0 + 3 + 3 - 1 = 4$.

In string similarity problems, scoring matrices usually set $s(x, y)$ to be greater than or equal to zero if characters x, y of Σ' match and less than zero if they mismatch. With such a scoring scheme, one seeks an alignment with as *large* a value as possible. That alignment will emphasize matches (or similarities) between the two strings while penalizing mismatches or inserted spaces. Of course, the meaningfulness of the resulting alignment may depend heavily on the scoring scheme used and how match scores compare to mismatch and space scores. Numerous character-pair scoring matrices have been suggested for proteins and for DNA [81, 122, 127, 222, 252, 400], and no single scheme is right for all applications. We will return to this issue in Sections 13.1, 15.7, and 15.10.

Definition Given a pairwise scoring matrix over the alphabet Σ', the *similarity* of two strings S_1 and S_2 is defined as the value of the alignment $\mathcal{A}$ of S_1 and S_2 that maximizes total alignment value. This is also called the *optimal alignment value* of S_1 and S_2.

String similarity is clearly related to alphabet-weight edit distance, and depending on the specific scoring matrix involved, one can often transform one problem into the other. An important difference between similarity and weighted edit distance will become clear in Section 11.7, after we discuss local alignment.

11.6.1. Computing similarity

The similarity of two strings S_1 and S_2, and the associated optimal alignment, can be computed by dynamic programming with recurrences that should by now be very intuitive.

Definition $V(i, j)$ is defined as the *value* of the optimal alignment of prefixes $S_1[1..i]$ and $S_2[1..j]$.

Recall that a dash ("–") is used to represent a space inserted into a string. The base conditions are

$$V(0, j) = \sum_{1 \le k \le j} s(\_, S_2(k))$$

and

$$V(i, 0) = \sum_{1 \le k \le i} s(S_1(k), \_).$$

For i and j both strictly positive, the general recurrence is

$$V(i, j) = \max[V(i-1, j-1) + s(S_1(i), S_2(j)), V(i-1, j) + s(S_1(i), \_),$$
$$V(i, j-1) + s(\_, S_2(j))].$$

The correctness of this recurrence is established by arguments similar to those used for edit distance. In particular, in any alignment $\mathcal{A}$, there are three possibilities: characters $S_1(i)$ and $S_2(j)$ are in the same position (opposite each other), $S_1(i)$ is in a position after $S_2(j)$, or $S_1(i)$ is in a position before $S_2(j)$. The correctness of the recurrence is based on that case analysis. Details are left to the reader.

If S_1 and S_2 are of length n and m, respectively, then the value of their optimal alignment is given by $V(n, m)$. That value, and the entire dynamic programming table, can be obtained in $O(nm)$ time, since only three comparisons and arithmetic operations are needed per cell. By leaving pointers while filling in the table, as was done with edit distance, an optimal alignment can be constructed by following any path of pointers from cell (n, m) to cell $(0, 0)$. So the optimal (global) alignment problem can be solved in $O(nm)$ time, the same time as for edit distance.

11.6.2. Special cases of similarity

By choosing an appropriate scoring scheme, many problems can be modeled as special cases of optimal alignment or similarity. One important example is the *longest common subsequence problem*.

Definition In a string S, a *subsequence* is defined as a subset of the characters of S arranged in their original "relative" order. More formally, a subsequence of a string S of length n is specified by a list of indices $i_1 < i_2 < i_3 < \ldots < i_k$, for some $k \le n$. The subsequence specified by this list of indices is the string $S(i_1)S(i_2)S(i_3)\ldots S(i_k)$.

To emphasize again, a *subsequence* need not consist of contiguous characters in S, whereas the characters of a *substring* must be contiguous.[4] Of course, a substring satisfies the definition for a subsequence. For example, "its" is a subsequence of "winters" but not a substring, whereas "inter" is both a substring and a subsequence.

Definition Given two strings S_1 and S_2, a *common subsequence* is a subsequence that appears both in S_1 and S_2. The *longest common subsequence problem* is to find a longest common subsequence (*lcs*) of S_1 and S_2.

[4] The distinction between subsequence and substring is often lost in the biological literature. But algorithms for substrings are usually quite different in spirit and efficiency than algorithms for subsequences, so the distinction is an important one.

The *lcs* problem is important in its own right, and we will discuss some of its uses and some ideas for improving its computation in Section 12.5. For now we show that it can be modeled and solved as an optimal alignment problem.

Theorem 11.6.1. *With a scoring scheme that scores a one for each match and a zero for each mismatch or space, the matched characters in an alignment of maximum value form a longest common subsequence.*

The proof is immediate and is left to the reader. It follows that the longest common subsequence of strings of lengths n and m, respectively, can be computed in $O(nm)$ time.

At this point we see the first of many differences between substring and subsequence problems and why it is important to clearly distinguish between them. In Section 7.4 we established that the longest common substring could be found in $O(n+m)$ time, whereas here the bound established for finding longest common subsequence is $O(n \times m)$ (although this bound can be reduced somewhat). This is typical – substring and subsequence problems are generally solved by different methods and have different time and space complexities.

11.6.3. Alignment graphs for similarity

As was the case for edit distance, the computation of similarity can be viewed as a path problem on a directed acyclic graph called an *alignment graph*. The graph is the same as the edit graph considered earlier, but the weights on the edges are the specific values for aligning a specific pair of characters or a character against a space. The start node of the alignment graph is again the node associated with cell $(0, 0)$, and the destination node is associated with cell (n, m) of the dynamic programming table, but the optimal alignment comes from the longest start to destination path rather than from the shortest path. It is again true that the longest paths in the alignment graph are in one-to-one correspondence with the optimal (maximum value) alignments. In general, computing longest paths in graphs is difficult, but for directed acyclic graphs the longest path is found in time proportional to the number of edges in the graph, using a variant of dynamic programming (which should come as no surprise). Hence for alignment graphs, the longest path can be found in $O(nm)$ time.

11.6.4. End-space free variant

There is a commonly used variant of string alignment called *end-space free alignment*. In this variant, any spaces at the end or the beginning of the alignment contribute a weight of zero, no matter what weight other spaces contribute. For example, in the alignment

```
-  -  c  a  c  -  d  b  d
l  t  c  a  b  b  d  b  -
```

the two spaces at the left end of the alignment are free, as is the single space at the right end.

Making end spaces free in the objective function encourages one string to align in the interior of the other, or the suffix of one string to align with a prefix of the other. This is desirable when one believes that those kinds of alignments reflect the "true" relationship of the two strings. Without a mechanism to encourage such alignments, the optimal alignment might have quite a different shape and not capture the desired relationship.

One example where end-spaces should be free is in the *shotgun sequence assembly* (see Sections 16.14 and 16.15). In this problem, one has a large set of partially overlapping substrings that come from many copies of one original but unknown string; the problem is to use comparisons of pairs of substrings to infer the correct original string. Two random substrings from the set are unlikely to be neighbors in the original string, and this is reflected by a low end-space free alignment score for those two substrings. But if two substrings do overlap in the original string, then a "good-sized" suffix of one should align to a "good-sized" prefix of the other with only a small number of spaces and mismatches (reflecting a small percentage of sequencing errors). This overlap is detected by an end-space free weighted alignment with high score. Similarly the case when one substring contains another can be detected in this way. The procedure for deducing candidate neighbor pairs is thus to compute the end-space free alignment between every pair of substrings; those pairs with high scores are then the best candidates. We will return to shotgun sequencing and extend this discussion in Part IV, Section 16.14.

To implement free end spaces in computing similarity, use the recurrences for global alignment (where all spaces count) detailed on page 227, but change the base conditions to $V(i, 0) = V(0, j) = 0$, for every i and j. That takes care of any spaces on the left end of the alignment. Then fill in the table as in the case of global alignment. However, unlike global alignment, the value of the optimal alignment is not necessarily found in cell (n, m). Rather, the value of the optimal alignment with free ends is the maximum value over all cells in row n or column m. Cells in row n correspond to alignments where the last character of string S_1 contributes to the value of the alignment, but characters of S_2 to its right do not. Those characters are opposite end spaces, which are free. Cells in column m have a similar characterization. Clearly, optimal alignment with free end spaces is solved in $O(nm)$ time, the same time as for global alignment.

11.6.5. Approximate occurrences of P in T

We now examine another important variant of global alignment.

Definition Given a parameter δ, a substring T' of T is said to be an *approximate occurrence* of P if and only if the optimal alignment of P to T' has value at least δ.

The problem of determining if there is an approximate occurrence of P in T is an important and natural generalization of the exact matching problem. It can be solved as follows: Use the same recurrences (given on page 227) as for global alignment between P and T and change only the base condition for $V(0, j)$ to $V(0, j) = 0$ for all j. Then fill in the table (leaving the standard backpointers). Using this variant of global alignment, the following theorem can be proved.

Theorem 11.6.2. *There is an approximate occurrence of P in T ending at position j of T if and only if $V(n, j) \geq \delta$. Moreover, $T[k..j]$ is an approximate occurrence of P in T if and only if $V(n, j) \geq \delta$ and there is a path of backpointers from cell (n, j) to cell $(0, k)$.*

Clearly, the table can be filled in using $O(nm)$ time, but if all approximate occurrence of P in T are to be explicitly output, then $\Theta(nm)$ time may not be sufficient. A sensible compromise is to identify every position j in T such that $V(n, j) \geq \delta$, and then for each such j, explicitly output only the *shortest* approximate occurrence of P that ends at position j. That substring T' is found by traversing the backpointers from (n, j) until a

cell in row zero is reached, breaking ties by choosing a vertical pointer over a diagonal one and a diagonal one over a horizontal one.

11.7. Local alignment: finding substrings of high similarity

In many applications, two strings may not be highly similar in their entirety but may contain regions that are highly similar. The task is to find and extract a pair of regions, one from each of the two given strings, that exhibit high similarity. This is called the *local alignment* or *local similarity problem* and is defined formally below.

Local alignment problem Given two strings S_1 and S_2, find substrings α and β of S_1 and S_2, respectively, whose similarity (optimal global alignment value) is maximum over all pairs of substrings from S_1 and S_2. We use v^* to denote the value of an optimal solution to the local alignment problem.

For example, consider the strings $S_1 = pqraxabcstvq$ and $S_2 = xyaxbacsll$. If we give each match a value of 2, each mismatch a value of -2, and each space a value of -1, then the two substrings $\alpha = axabcs$ and $\beta = axbacs$ of S_1 and S_2, respectively, have the following optimal (globat) alignment

$$\begin{array}{cccccccc} a & x & a & b & - & c & s \\ a & x & - & b & a & c & s \end{array}$$

which has a value of 8. Furthermore, over all choices of pairs of substrings, one from each of the two strings, those two substrings have maximum similarity (for the chosen scoring scheme). Hence, for that scoring scheme, the optimal local alignment of S_1 and S_2 has value 8 and is defined by substrings $axabcs$ and $axbacs$.

It should be clear why local alignment is defined in terms of similarity, which maximizes an objective function, rather than in terms of edit distance, which minimizes an objective. When one seeks a pair of substrings to minimize distance, the optimal pairs would be exactly matching substrings under most natural scoring schemes. But the matching substrings might be just a single character long and would not identify a region of high similarity. A formulation such as local alignment, where matches contribute positively and mismatches and spaces contribute negatively, is more likely to find more meaningful regions of high similarity.

Why local alignment?

Global alignment of protein sequences is often meaningful when the two strings are members of the same protein family. For example, the protein *cytochrome c* has almost the same length in most organisms that produce it, and one expects to see a relationship between two cytochromes from any two different species over the entire length of the two strings. The same is true of proteins in the *globin* family, such as *myoglobin* and *hemoglobin*. In these cases, global alignment is meaningful. When trying to deduce evolutionary history by examining protein sequence similarities and differences, one usually compares proteins in the same sequence family, and so global alignment is typically meaningful and effective in those applications.

However, in many biological applications, local similarity (local alignment) is far more meaningful than global similarity (global alignment). This is particularly true when long stretches of anonymous DNA are compared, since only *some* internal sections of those

strings may be related. When comparing protein sequences, local alignment is also critical because proteins from very different families are often made up of the same structural or functional subunits (motifs or domains), and local alignment is appropriate in searching for these (unknown) subunits. Similarly, different proteins are often made from related motifs that form the inner core of the protein, but the motifs are separated by outside surface looping regions that can be quite different in different proteins.

A very interesting example of conserved domains comes from the proteins encoded by *homeobox* genes. Homeobox genes [319, 381] show up in a wide variety of species, from fruit flies to frogs to humans. These genes regulate high-level embryonic development, and a single mutation in these genes can transform one body part into another (one of the original mutation experiments causes fruit fly antenna to develop as legs, but it doesn't seem to bother the fly very much). The protein sequences that these genes encode are very different in each species, except in one region called the *homeodomain*. The homeodomain consists of about sixty amino acids that form the part of the regulatory protein that binds to DNA. Oddly, homeodomains made by certain insect and mammalian genes are particularly similar, showing about 50 to 95% identity in alignments without spaces. Protein-to-DNA binding is central in how those proteins regulate embryo development and cell differentiation. So the amino acid sequence in the most biologically critical part of those proteins is highly conserved, whereas the other parts of the protein sequences show very little similarity. In cases such as these, local alignment is certainly a more appropriate way to compare protein sequences than is global alignment.

Local alignment in protein is additionally important because particular isolated characters of related proteins may be more highly conserved than the rest of the protein (for example, the amino acids at the *active site* of an enzyme or the amino acids in the *hydrophobic core* of a globular protein are the most highly conserved). Local alignment will more likely detect these conserved characters than will global alignment. A good example is the family of *serine proteases* where a few isolated, conserved amino acids characterize the family. Another example comes from the Helix-Turn-Helix motif, which occurs frequently in proteins that regulate DNA transcription by binding to DNA. The tenth position of the Helix-Turn-Helix motif is very frequently occupied by the amino acid glycine, but the rest of the motif is more variable.

The following quote from C. Chothia [101] further emphasizes the biological importance of protein domains and hence of local string comparison.

> Extant proteins have been produced from the original set not just by point mutations, insertions and deletions but also by combinations of genes to give chimeric proteins. This is particularly true of the very large proteins produced in the recent stages of evolution. Many of these are built of different combinations of protein domains that have been selected from a relatively small repertoire.

Doolittle [129] summarizes the point: "The underlying message is that one must be alert to regions of similarity even when they occur embedded in an overall background of dissimilarity."

Thus, the dominant viewpoint today is that local alignment is the most appropriate type of alignment for comparing proteins from different protein families. However, it has also been pointed out [359, 360] that one often sees extensive global similarity in pairs of protein strings that are first recognized as being related by strong local similarity. There are also suggestions [316] that in some situations global alignment is more effective than local alignment in exposing important biological commonalities.

11.7.1. Computing local alignment

Why not look for regions of high similarity in two strings by first globaly aligning those strings? A global alignment between two long strings will certainly be influenced by regions of high similarity, and an optimal global alignment might well align those corresponding regions with each other. But more often, local regions of high local similarity would get lost in the overall optimal global alignment. Therefore, to identify high local similarity it is more effective to search explicitly for local similarity.

We will show that if the lengths of strings S_1 and S_2 are n and m, respectively, then the local alignment problem can be solved in $O(nm)$ time, the same time as for global alignment. This efficiency is surprising because there are $\Theta(n^2m^2)$ pairs of substrings, so even if a global alignment could be computed in constant time for each chosen pair, the time bound would be $\Theta(n^2m^2)$. In fact, if we naively use $O(kl)$ for the bound on the time to align strings of lengths k and l, then the resulting time bound for the local alignment problem would be $O(n^3m^3)$, instead of the $O(nm)$ bound that we will establish. The $O(nm)$ time bound was obtained by Temple Smith and Michael Waterman [411] using the algorithm we will describe below.

In the definition of local alignment given earlier, any scoring scheme was permitted for the global alignment of two chosen substrings. One slight restriction will help in computing local alignment. We assume that the global alignment of two empty strings has value zero. That assumption is used to allow the local alignment algorithm to choose two empty substrings for α and β. Before describing the solution to the local alignment problem, it will be helpful to consider first a more restricted version of the problem.

Definition Given a pair of indices $i \leq n$ and $j \leq m$, the *local suffix alignment problem* is to find a (possibly empty) suffix α of $S_1[1..i]$ and a (possibly empty) suffix β of $S_2[1..j]$ such that $V(\alpha, \beta)$ is the maximum over all pairs of suffixes of $S_1[1..i]$ and $S_2[1..j]$. We use $v(i, j)$ to denote the value of the optimal local suffix alignment for the given index pair i, j.

For example, suppose the objective function counts 2 for each match and -1 for each mismatch or space. If $S_1 = abcxdex$ and $S_2 = xxxcde$, then $v(3, 4) = 2$ (the two cs match), $v(4, 5) = 1$ (cx aligns with cd), $v(5, 5) = 3$ (x_d aligns with xcd), and $v(6, 6) = 5$ (x_de aligns with $xcde$).

Since the definition allows either or both of the suffixes to be empty, $v(i, j)$ is always greater than or equal to zero.

The following theorem shows the relationship between the local alignment problem and the local suffix alignment problem. Recall that v^* is the value of the optimal local alignment for two strings of length n and m.

Theorem 11.7.1. $v^* = \max[v(i, j) : i \leq n, j \leq m]$.

PROOF Certainly $v^* \geq \max[v(i, j) : i \leq n, j \leq m]$, because the optimal solution to the local suffix alignment problem for any i, j is a feasible solution to the local alignment problem. Conversely, let α, β be the substrings in an optimal solution to the local alignment problem and suppose α ends at position i^* and β ends at j^*. Then α, β also defines a local suffix alignment for index pair i^*, j^*, and so $v^* \leq v(i^*, j^*) \leq \max[v(i, j) : i \leq n, j \leq m]$, and both directions of the lemma are established. □

Theorem 11.7.1 only specifies the value v^*, but its proof makes clear how to find substrings whose alignment have that value. In particular,

Theorem 11.7.2. *If i', j' is an index pair maximizing $v(i, j)$ over all i, j pairs, then a pair of substrings solving the local* suffix *alignment problem for i', j' also solves the local alignment problem.*

Thus a solution to the local suffix alignment problem solves the local alignment problem. We now turn our attention to the problem of finding $\max[v(i, j) : i \leq n, j \leq m]$ and a pair of strings whose alignment has maximum value.

11.7.2. How to solve the local suffix alignment problem

First, $v(i, 0) = 0$ and $v(0, j) = 0$ for all i, j, since we can always choose an empty suffix.

Theorem 11.7.3. *For $i > 0$ and $j > 0$, the proper recurrence for $v(i, j)$ is*

$$v(i, j) = \max[0, v(i-1, j-1) + s(S_1(i), S_2(j)),$$
$$v(i-1, j) + s(S_1(i), \_), v(i, j-1) + s(\_, S_2(j))].$$

PROOF The argument is similar to the justifications of previous recurrence relations. Let α and β be the substrings of S_1 and S_2 whose global alignment establishes the optimal local alignment. Since α and β are permitted to be empty suffixes of $S_1[1..i]$ and $S_2[1..j]$, it is correct to include 0 as a *candidate* value for $v(i, j)$. However, if the optimal α is not empty, then character $S_1(i)$ must either be aligned with a space or with character $S_2(j)$. Similarly, if the optimal β is not empty, then $S_2(j)$ is aligned with a space or with $S_1(i)$. So we justify the recurrence based on the way characters $S_1(i)$ and $S_2(j)$ may be aligned in the optimal local suffix alignment for i, j.

If $S_1(i)$ is aligned with $S_2(j)$ in the optimal local i, j suffix alignment, then those two characters contribute $s(S_1(i), S_2(j))$ to $v(i, j)$, and the remainder of $v(i, j)$ is determined by the local suffix alignment for indices $i-1, j-1$. That local suffix alignment must be optimal and so has value $v(i-1, j-1)$. Therefore, if $S_1(i)$ and $S_2(j)$ are aligned with each other, $v(i, j) = v(i-1, j-1) + s(S_1(i), S_2(j))$.

If $S_1(i)$ is aligned with a space, then by similar reasoning $v(i, j) = v(i-1, j) + s(S_1(i), \_)$, and if $S_2(j)$ is aligned with a space then $v(i, j) = v(i, j-1) + s(\_, S_2(j))$. Since all cases are exhausted, we have proven that $v(i, j)$ must either be zero or be equal to one of the three other terms in the recurrence.

On the other hand, for each of the four terms in the recurrence, there *is* a way to choose suffixes of $S_1[1..i]$ and $S_2[1..j]$ so that an alignment of those two suffixes has the value given by the associated term. Hence the optimal suffix alignment value is at least the maximum of the four terms in the recurrence. Having proved that $v(i, j)$ must be one of the four terms, and that it must be greater than or equal to the maximum of the four terms, it follows that $v(i, j)$ must be equal to the maximum which proves the theorem. □

The recurrences for local suffix alignment are almost identical to those for global alignment. The only difference is the inclusion of zero in the case of local suffix alignment. This makes intuitive sense. In both global alignment and local suffix alignment of prefixes $S_1[1..i]$ and $S_2[1..j]$ the end characters of any alignment are specified, but in the case of local suffix alignment, any number of initial characters can be ignored. The zero in the recurrence implements this, acting to "restart" the recurrence.

Given Theorem 11.7.2, the method to compute v^* is to compute the dynamic programming table for $v(i, j)$ and then find the largest value in *any* cell in the table, say in cell (i^*, j^*). As usual, pointers are created while filling in the values of the table. After cell

(i^*, j^*) is found, the substrings α and β giving the optimal local alignment of S_1 and S_2 are found by tracing back the pointers from cell (i^*, j^*) until an entry (i', j') is reached that has value zero. Then the optimal local alignment substrings are $\alpha = S_1[i'..i^*]$ and $\beta = S_2[j'..j^*]$.

Time analysis

Since it takes only four comparisons and three arithmetic operations per cell to compute $v(i, j)$, it takes only $O(nm)$ time to fill in the entire table. The search for v^* and the traceback clearly require only $O(nm)$ time as well, so we have established the following desired theorem:

Theorem 11.7.4. *For two strings S_1 and S_2 of lengths n and m, the local alignment problem can be solved in $O(nm)$ time, the same time as for global alignment.*

Recall that the pointers in the dynamic programming table for edit distance, global alignment, and similarity encode all the optimal alignments. Similarly, the pointers in the dynamic programming table for local alignment encode the optimal local alignments as follows.

Theorem 11.7.5. *All optimal local alignments of two strings are represented in the dynamic programming table for $v(i, j)$ and can be found by tracing any pointers back from any cell with value v^*.*

We leave the proof as an exercise.

11.7.3. Three final comments on local alignment

Terminology for local and global alignment

In the biological literature, global alignment (similarity) is often referred to as a Needleman–Wunsch [347] alignment after the authors who first discussed global similarity. Local alignment is often referred to as a Smith–Waterman [411] alignment after the authors who introduced local alignment. There is, however, some confusion in the literature between Needleman–Wunsch and Smith–Waterman as *problem statements* and as *solution methods*. The original solution given by Needleman–Wunsch runs in cubic time and is rarely used. Hence "Needleman–Wunsch" usually refers to the global alignment *problem*. The Smith–Waterman method runs in quadratic time and is commonly used, so "Smith–Waterman" often refers to their specific solution as well as to the problem statement. But there are solution methods to the (Smith–Waterman) local alignment problem that differ from the Smith–Waterman solution and yet are sometimes also referred to as "Smith–Waterman".

Using Smith–Waterman to find several regions of high similarity

Very often in biological applications it is not sufficient to find just a single pair of substrings of input strings of S_1 and S_2 with the optimal local alignment. Rather, what is required is to find all or "many" pairs of substrings that have similarity above some threshold. A specific application of this kind will be discussed in Section 18.2, and the general problem will be studied much more deeply in Section 13.2. Here we simply point out that, in practice, the dynamic programming table used to solve the local suffix alignment problem is often used to find additional pairs of substrings with "high" similarity. The key observation is that for any cell (i, j) in the table, one can find a pair of substrings of S_1 and S_2 (by traceback)


```
c t t t a a c - - a - a c
c - - - c a c c c a t - c
```

Figure 11.5: An alignment with seven spaces distributed into four gaps.

with similarity (global alignment value) of $v(i, j)$. Thus, an easy way to look for a set of highly similar substrings is to find a set of cells in the table with a value above some set threshold. Not all similar substrings will be identified in this way, but this approach is common in practice.

The need for good scoring schemes

The utility of optimal local alignment is affected by the scoring scheme used. For example, if matches are scored as one, and mismatches and spaces as zero, then the optimal local alignment will be determined by the longest common *subsequence*. Conversely, if mismatches and spaces are given large negative scores, and each match is given a score of one, then the optimal local alignment will be the longest common *substring*. In most cases, neither of these is the local alignment of interest and some care is required to find an application-dependent scoring scheme that yields meaningful local alignments. For local alignment, the entries in the scoring matrix must have an average score that is negative. Otherwise the resulting "local" optimal alignment tends to be a global alignment. Recently, several authors have developed a rather elegant theory of what scoring schemes for local alignment mean in the context of database search and how they should be derived. We will briefly discuss this theory in Section 15.11.2.

11.8. Gaps

11.8.1. Introduction to Gaps

Until now the central constructs used to measure the value of an alignment (and to define similarity) have been *matches*, *mismatches*, and *spaces*. Now we introduce another important construct, *gaps*. Gaps help create alignments that better conform to underlying biological models and more closely fit patterns that one expects to find in meaningful alignments.

Definition A gap is any *maximal, consecutive run* of spaces in a *single* string of a given alignment.[5]

A gap may begin before the start of S, in which case it is bordered on the right by the first character of S, or it may begin after the end of S, in which case it is bordered on the left by the last character of S. Otherwise, a gap must be bordered on both sides by characters of S. A gap may be as small as a single space. As an example of gaps, consider the alignment in Figure 11.5, which has four gaps containing a total of seven spaces. That alignment would be described as having five matches, one mismatch, four gaps, and seven spaces. Notice that the last space in the first string is followed by a space in the second string, but those two spaces are in two gaps and do not form a single gap.

By including a term in the objective function that reflects the gaps in the alignment one has some influence on the *distribution* of spaces in an alignment and hence on the overall shape of the alignment. In the simplest objective function that includes gaps,

[5] Sometimes in the biology literature the term "space" (as we use it) is not used. Rather, the term "gap" is used both for "space" and for "gap" (as we have defined it here). This can cause much confusion, and in this book the terms "gap" and "space" have distinct meanings.

each gap contributes a constant weight W_g, independent of how long the gap is. That is, each individual space is free, so that $s(x, \_) = s(\_, x) = 0$ for every character x. Using the notation established in Section 11.6, (page 226), we write the value of an alignment containing k gaps as

$$\sum_{i=1}^{l} s(S_1'(i), S_2'(i)) - kW_g.$$

Changing the value of W_g relative to the other weights in the objective function can change how spaces are distributed in the optimal alignment. A large W_g encourages the alignment to have few gaps, and the aligned portions of the two strings will fall into a few substrings. A smaller W_g allows more fragmented alignments. The influence of W_g on the alignment will be discussed more deeply in Section 13.1.

11.8.2. Why gaps?

Most of the biological justifications given for the importance of local alignment (see Section 11.7) apply as well to justify the gap as an explicit concept in string alignment.

Just as a space in an alignment corresponds to an insertion or deletion of a single character in the edit transcript, a gap in string S_1 opposite substring α in string S_2 corresponds to either a deletion of α from S_1 or to an insertion of α into S_2. The concept of a gap in an alignment is therefore important in many biological applications because the insertion or deletion of an entire substring (particularly in DNA) often occurs as single mutational event. Moreover, many of these single mutational events can create gaps of quite varying sizes with almost equal likelihood (within a wide, but bounded, range of sizes). Much of the repetitive DNA discussed in Section 7.11.1 is caused by single mutational events that copy and insert long pieces of DNA. Other mutational mechanisms that make long insertions or deletions in DNA include: *unequal crossing-over* in meiosis (causing an insertion in one string and a reciprocal deletion in the other); DNA *slippage* during replication (where a portion of the DNA is repeated on the replicated copy because the replication machinery loses its place on the template, slipping backwards and repeating a section); insertion of *transposable* elements (jumping genes) into a DNA string; insertions of DNA by *retroviruses*; and *translocations* of DNA between chromosomes [301, 317]. See Figure 11.6 for an example of gaps in genomic sequence data.

When computing alignments for the purpose of deducing evolutionary history over a long period of time, it is often the gaps that are the most informative part of the alignments. In DNA strings, single character substitutions due to point mutations occur continuously and usually at a much faster rate than (nonfatal) mutational events causing gaps. The analogous gene (specifying the "same" protein) in two species can thus be very different at the DNA sequence level, making it difficult to sort out evolutionary relationships on the basis of string similarity (without gaps). But large insertions and deletions in molecules that show up as gaps in alignments occur less frequently than substitutions. Therefore, common gaps in pairs of aligned strings can sometimes be the key features used to deduce the overall evolutionary history of a set of strings [45, 405]. Later, in Section 17.3.2, we will see that such gaps can be considered as evolutionary characters in certain approaches to building evolutionary trees.

At the protein level, recall that many proteins are "built of different combinations of protein domains that have been selected from a relatively small repertoire"[101]. Hence two protein strings might be relatively similar over several intervals but differ in intervals where one contains a protein domain that the other does not. Such an interval most naturally

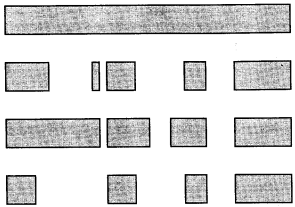


Figure 11.6: Each of the four rows represents part of the RNA sequence of one strain of the HIV-1 virus. The HIV virus mutates rapidly, so that mutations can be observed and traced. The bottom three rows are from virus strains that have each mutated from an ancestral strain represented in the top row. Each of the bottom sequences is shown aligned to the top sequence. A dark box represents a substring that matches the corresponding substring in the top sequence, while each white space represents a gap resulting from a known sequence deletion. This figure is adapted from one in [123].

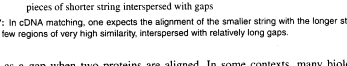


Figure 11.7: In cDNA matching, one expects the alignment of the smaller string with the longer string to consist of a few regions of very high similarity, interspersed with relatively long gaps.

shows up as a gap when two proteins are aligned. In some contexts, many biologists consider the proper identification of the major (long) gaps as *the* essential problem of protein alignment. If the long (major) gaps have been selected correctly, the rest of the alignment – reflecting point mutations – is then relatively easy to obtain.

An alignment of two strings is intended to reflect the cost (or likelihood) of mutational events needed to transform one string to another. Since a gap of more than one space can be created by a single mutational event, the alignment model should reflect the true distribution of spaces into gaps, not merely the number of spaces in the alignment. It follows that the model must specify how to weight gaps so as to reflect their biological meaning. In this chapter we will discuss different proposed schemes for weighting gaps, and in later chapters we will discuss additional issues in scoring gaps. First we consider a concrete example illustrating the utility of the gap concept.

11.8.3. cDNA matching: a concrete illustration

One concrete illustration of the use of gaps in the alignment model comes from the problem of *cDNA matching*. In this problem, one string is much longer than the other, and the alignment best reflecting their relationship should consist of a few regions of very high similarity interspersed with "long" gaps in the shorter string (see Figure 11.7). Note that the matching regions can have mismatches and spaces, but these should be a small percentage of the region.

Biological setting of the problem

In eukaryotes, a gene that codes for a protein is typically made up of alternating *exons* (expressed sequences), which contribute to the code for the protein, and *introns* (intervening sequences), which do not. The number of exons (and hence also introns) is generally modest (four to twenty say), but the lengths of the introns can be huge compared to the lengths of the exons.

At a very coarse level, the protein specified by a eukaryotic gene is made in the following steps. First, an RNA molecule is transcribed from the DNA of the gene. That RNA transcript is a complement of the DNA in the gene in that each A in the gene is replaced by U (uracil) in the RNA, each T is replaced by A, each C by G, and each G by C. Moreover, the RNA transcript covers the entire gene, introns as well as exons. Then, in a process that is not completely understood, each intron-exon boundary in the transcript is located, the RNA corresponding to the introns is spliced out (or *snurped* out by a molecular complex called a *snrp* [420]), and the RNA regions corresponding to exons are concatenated. Additional processing occurs that we will not describe. The resulting RNA molecule is called the *messenger RNA* (*mRNA*); it leaves the cell nucleus and is used to create the protein it encodes.

Each cell (usually) contains a copy of all the chromosomes and hence of all the genes of the entire individual, yet in each specialized cell (a liver cell for example) only a small fraction of the genes are expressed. That is, only a small fraction of the proteins encoded in the genome are actually produced in that specialized cell. A standard method to determine which proteins are expressed in the specialized cell line, and to hunt for the location of the encoding genes, involves capturing the mRNA in that cell after it leaves the cell nucleus. That mRNA is then used to create a DNA string complementary to it. This string is called *cDNA* (complementary DNA). Compared to the original gene, the cDNA string consists only of the concatenation of exons in the gene.

It is routine to capture mRNA and make cDNA libraries (complete collections of a cell's mRNA) for specific cell lines of interest. As more libraries are built up, one collects a reflection of all the genes in the genome and a taxonomy of the cells that the genes are expressed in. In fact, a major component of the Human Genome Project [111], [399] is to obtain cDNAs reflecting most of the genes in the human genome. This effort is also being conducted by several private companies and has led to some interesting disputes over patenting cDNA sequences.

After cDNA is obtained, the problem is to determine where the gene associated with that cDNA resides. Presently, this problem is most often addressed with laboratory methods. However, if the cDNA is sequenced or partially sequenced (and in the Human Genome Project, for example, the intent is to sequence parts of each of the obtained cDNAs), and if one has sequenced the part of the genome containing the gene associated with that cDNA (as, for example, one would have after sequencing the entire genome), then the problem of finding the gene site given a cDNA sequence becomes a string problem. It becomes one of aligning the cDNA string against the longer string of sequenced DNA in a way that reveals the exons. It becomes the cDNA matching problem discussed above.

Why gaps are needed in the objective function

If the objective function includes terms only for matches, mismatches, and spaces, there seems no way to encourage the optimal alignment to be of the desired form. It's worth a moment's effort to explain why.

Certainly, you don't want to set a large penalty for spaces, since that would align all the cDNA string close together, rather than allowing gaps in the alignment corresponding to the long introns. You would also want a rather high penalty for mismatches. Although there may be a few sequencing errors in the data, so that some mismatches will occur even when the cDNA is properly cut up to match the exons, there should not be a large percentage of mismatches. In summary, you want small penalties for spaces, relatively large penalties for mismatches, and positive values for matches.

What kind of alignment would likely result using an objective function that has low space penalty, high mismatch penalty, positive match value of course, and no term for gaps? Remember that the long string contains more than one gene, that the exons are separated by long introns, and that DNA has an alphabet of only four letters present in roughly equal amounts. Under these conditions, the optimal alignment would probably be the *longest common subsequence* between the short cDNA string and the long anonymous DNA string. And because the introns are long and DNA has only four characters, that common subsequence would likely match *all* of the characters in the cDNA. Moreover, because of small but real sequencing errors, the true alignment of the cDNA to its exons would not match all the characters. Hence the longest common subsequence would likely have a higher score than the correct alignment of the cDNA to exons. But the longest common subsequence would fragment the cDNA string over the longer DNA and not give an alignment of the desired form – it would not pick out its exons.

Putting a term for gaps in the objective function rectifies the problem. By adding a constant gap weight W_g for each gap in the alignment, and setting W_g appropriately (by experimenting with different values of W_g), the optimal alignment can be induced to cut up the cDNA to match its exons in the longer string.[6] As before, the space penalty is set to zero, the match value is positive, and the mismatch penalty is set high.

Processed pseudogenes

A more difficult version of cDNA matching arises in searching anonymous DNA for *processed pseudogenes*. A pseudogene is a near copy of a working gene that has mutated sufficiently from the original copy so that it can no longer function. Pseudogenes are very common in eukaryotic organisms and may play an important evolutionary role, providing a ready pool of diverse "near genes". Following the view that new genes are created by the process of *duplication with modification* of existing genes [127, 128, 130], pseudogenes either represent trial genes that failed or future genes that will function after additional mutations.

A pseudogene may be located very far from the gene it corresponds to, even on a different chromosome entirely, but it will usually contain both the introns and the exons derived from its working relative. The problem of finding pseudogenes in anonymous sequenced DNA is therefore related to that of finding repeated substrings in a very long string.

A more interesting type of pseudogene, the *processed pseudogene*, contains only the exon substrings from its originating gene. Like cDNA, the introns have been removed and the exons concatenated. It is thought that a processed pseudogene originates as an mRNA that is retranscribed back into DNA (by the enzyme Reverse Transcriptase) and inserted into the genome at a random location.

Now, given a long string of anonymous DNA that might contain both a processed pseudogene and its working ancestor, how could the processed pseudogenes be located?

[6] This really works, and it is a very instructive exercise to try it out empirically.

The problem is similar to cDNA matching but more difficult because one does not have the cDNA in hand. We leave it to the reader to explore the use of repeat finding methods, local alignment, and gap weight selection in tackling this problem.

Caveat

The problems of cDNA and pseudogene matching illustrate the utility of including gaps in the alignment objective function and the importance of weighting the gaps appropriately. It should be noted, however, that in practice one can approach these matching problems by a judicious use of local alignment without gaps. The idea is that in computing local alignment, one can find not only the most similar pair of substrings but many other highly similar pairs of substrings (see Sections 13.2.4, and 11.7.3). In the context of cDNA or pseudogene matching, these pairs will likely be the exons, and so the needed match of cDNA to exons can be pieced together from a number of nonoverlapping local alignments. This is the more typical approach in practice.

11.8.4. Choices for gap weights

As illustrated by the example of cDNA matching, the appropriate use of gaps in the objective function aids in the discovery of alignments that satisfy an expected shape. But clearly, the way gaps are weighted critically influences the effectiveness of the gap concept. We will examine in detail four general types of gap weights: *constant, affine, convex,* and *arbitrary*.

The simplest choice is the *constant* gap weight introduced earlier, where each individual space is free, and each gap is given a weight of W_g independent of the number of spaces in the gap. Letting W_m and W_{ms} denote weights for matches and mismatches, respectively, the *operator-weight* version of the problem is:

Find an alignment $\mathcal{A}$ to maximize $[W_m(\#\ matches) - W_{ms}(\#\ mismatches) - W_g(\#\ gaps)]$.

More generally, if we adopt the alphabet-dependent weights for matches and mismatches, the objective in the constant gap weight model is:

Find an alignment $\mathcal{A}$ to maximize $\left(\sum_{i=1}^{l}[s(S_1'(i), S_2'(i))] - W_g(\#\,gaps)\right)$,

where $s(x, -) = s(-, x) = 0$ for every character x, and S_1' and S_2' represent the strings S_1 and S_2 after insertion of spaces.

A generalization of the constant gap weight model is to add a weight W_s for each space in the gap. In this case, W_g is called the *gap initiation weight* because it can represent the cost of starting a gap, and W_s is called the *gap extension weight* because it can represent the cost of extending the gap by one space. Then the operator-weight version of the problem is:

Find an alignment to maximize $[W_m(\#\ matches) - W_{ms}(\#\ mismatches) - W_g(\#\ gaps) - W_s(\#\ spaces)]$.

This is called the *affine gap weight* model[7] because the weight contributed by a single gap of length q is given by the affine function $W_g + qW_s$. The constant gap weight model is simply the affine model with $W_s = 0$.

[7] The affine gap model is sometimes called the *linear* weight model, and I prefer that term. However, "affine" has become the dominant term in the biological literature, and "linear" there usually refers to an affine function with $W_g = 0$.

The alphabet-weight version of the affine gap weight model again sets $s(x, -) = s(-, x) = 0$ and has the objective of finding an alignment to

$$\text{maximize}\left(\sum_{i=1}^{l}[s(S_1'(i), S_2'(i))] - W_g(\#\,gaps) - W_s(\#\,spaces)\right).$$

The affine gap weight model is probably the most commonly used gap model in the molecular biology literature, although there is considerable disagreement about what W_g and W_s should be [161] (in addition to questions about W_m and W_{ms}). For aligning amino acid strings, the widely used search program FASTA [359] has chosen the default settings of $W_g = 10$ and $W_s = 2$. We will return to the question of the choice of these settings in Section 13.1.

It has been suggested [57, 183, 466] that some biological phenomena are better modeled by a gap weight function where each additional space in a gap contributes less to the gap weight than the preceding space (a function with negative second derivative). In other words, a gap weight that is a *convex*,[8] but not affine, function of its length. An example is the function $W_g + \log_e q$, where q is the length of the gap. Some biologists have suggested that a gap function that initially increases to a maximum value and then decreases to near zero would reflect a *combination* of different biological phenomena that insert or delete DNA.

Finally, the most general gap weight we will consider is the *arbitrary gap weight*, where the weight of a gap is an arbitrary function $w(q)$ of its length q. The constant, affine, and convex weight models are of course subcases of the arbitrary weight model.

Time bounds for gap choices

As might be expected, the time needed to optimally solve the alignment problem with arbitrary gap weights is greater than for the other models. In the case that $w(q)$ is a totally arbitrary function of gap length, the optimal alignment can be found in $O(nm^2 + n^2m)$ time, where n and $m \geq n$ are the lengths of the two strings. In the case that $w(q)$ is convex, we will show that the time can be reduced to $O(nm \log m)$ (a further reduction is possible, but the algorithm is much too complex for our interests). In the affine (and hence constant) case the time bound is $O(nm)$, which is the same time bound established for the alignment model without the concept of gaps. In the next sections we will first discuss alignment for arbitrary gap weights and then show how to reduce the running time for the case of affine weight functions. The $O(nm \log m)$-time algorithm for convex weights is more complex than the others and is deferred until Chapter 13.

11.8.5. Arbitrary gap weights

This case was first introduced and solved in the classic paper of Needleman and Wunsch [347], although with somewhat different detail and terminology than used here.

For arbitrary gap weights, we will develop recurrences that are similar to (but more detailed than) the ones used in Section 11.6.1 for optimal alignment without gaps. There is, however, a subtle question about whether these recurrences correctly model the biologist's view of gaps. We will examine that issue in Exercise 45.

To align strings S_1 and S_2, consider, as usual, the prefixes $S_1[1..i]$ of S_1 and $S_2[1..j]$ of S_2. Any alignment of those two prefixes is one of the following three types (see Figure 11.8):

[8] Some call this *concave*.

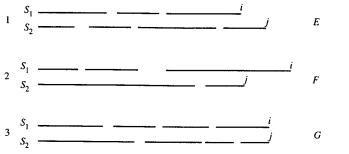


Figure 11.8: The recurrences for alignment with gaps are divided into three types of alignments: 1. those that align $S_1(i)$ to the left of $S_2(j)$, 2. those that align $S_1(i)$ to the right of $S_2(j)$, and 3. those that align them opposite each other.

1. Alignments of $S_1[1..i]$ and $S_2[1..j]$ where character $S_1(i)$ is aligned to a character strictly to the left of character $S_2(j)$. Therefore, the alignment ends with a gap in S_1.
2. Alignments of the two prefixes where $S_1(i)$ is aligned strictly to the right of $S_2(j)$. Therefore, the alignment ends with a gap in S_2.
3. Alignments of the two prefixes where characters $S_1(i)$ and $S_2(j)$ are aligned opposite each other. This includes both the case that $S_1(i) = S_2(j)$ and that $S_1(i) \neq S_2(j)$.

Clearly, these three types of alignments cover all the possibilities.

Definition Define $E(i, j)$ as the maximum value of any alignment of type 1; define $F(i, j)$ as the maximum value of any alignment of type 2; define $G(i, j)$ as the maximum value of any alignment of type 3; and finally define $V(i, j)$ as the maximum value of the three terms $E(i, j)$, $F(i, j)$, $G(i, j)$.

Recurrences for the case of arbitrary gap weights

By dividing the types of alignments into three cases, as above, we can write the following recurrences that establish $V(i, j)$:

$$V(i, j) = \max[E(i, j), F(i, j), G(i, j)],$$

$$G(i, j) = V(i-1, j-1) + s(S_1(i), S_2(j)),$$

$$E(i, j) = \max_{0 \le k \le j-1} [V(i, k) - w(j-k)],$$

$$F(i, j) = \max_{0 \le l \le i-1} [V(l, j) - w(i-l)].$$

To complete the recurrences, we need to specify the base cases and where the optimal alignment value is found. If all spaces are included in the objective function, even spaces that begin or end an alignment, then the optimal value for the alignment is found in cell (n, m), and the base case is

$$V(i, 0) = -w(i),$$

$$V(0, j) = -w(j),$$

$$E(i, 0) = -w(i),$$

$$F(0, j) = -w(j),$$

where $G(0, 0) = 0$, but $G(i, j)$ is undefined when exactly one of i or j is zero. Note that $V(0, 0) = w(0)$, which will most naturally be assigned to be zero.

When end spaces, and hence end gaps, are free, then the optimal alignment value is the maximum value over any cell in row n or column m, and the base cases are

$$V(i, 0) = 0,$$

$$V(0, j) = 0.$$

Time analysis

Theorem 11.8.1. *Assuming that $|S_1| = n$ and $|S_2| = m$, the recurrences can be evaluated in $O(nm^2 + n^2m)$ time.*

PROOF We evaluate the recurrences by the usual approach of filling in an $(n+1) \times (m+1)$ size table one row at time, where each row is filled from left to right. For any cell (i, j), the algorithm examines one other cell to evaluate $G(i, j)$, j cells of row i to evaluate $E(i, j)$, and i cells of column j to evaluate $F(i, j)$. Therefore, for any fixed row, $m(m+1)/2 = \Theta(m^2)$ cells are examined to evaluate all the E values in that row, and for any fixed column, $\Theta(n^2)$ cells are examined to evaluate all the F values of that column. The theorem then follows since there are n rows and m columns. □

The increase in running time over the previous case ($O(nm)$ time when gaps are not in the model) is caused by the need to look j cells to the left and i cells above to determine $V(i, j)$. Before gaps were included in the model, $V(i, j)$ depended only on the three cells adjacent to (i, j), and so each $V(i, j)$ value was computed in constant time. We will show next how to reduce the number of cell examinations for the case of affine gap weights; later we will show a more complex reduction for the case of convex gap weights.

11.8.6. Affine (and constant) gap weights

Here we examine in detail the simplest affine gap weight model and show that optimal alignments in that model can be computed in $O(nm)$ time. That bound is the same as for the alignment model without a gap term in the objective function. So although an explicit gap term in the objective function makes the alignment model much richer, it does not increase the running time used (in an asymptotic, worst-case sense) to find an optimal alignment. This important result was derived by several different authors (e.g., [18], [166], [186]). The same result then holds immediately for constant gap weights.

Recall that the objective is to find an alignment to

$$\text{maximize}[W_m(\#\,matches) - W_{ms}(\#\,mismatches) - W_g(\#\,gaps) - W_s(\#\,spaces)].$$

We will use the same variables $V(i, j)$, $E(i, j)$, $F(i, j)$, and $G(i, j)$ used in the recurrences for arbitrary gap weights. The definition and meanings of these variables remain unchanged, but the recurrence relations will be modified for the case of affine gap weights.

The key insight leading to greater efficiency in the affine gap case is that the increase in the total weight of a gap contributed by each additional space is a constant W_s independent of the size of the gap to that point. In other words, in the affine gap weight model $w(q+1) - w(q) = W_s$ for any gap length q greater than zero. This is in contrast to the arbitrary weight case where there is no predictable relationship between $w(q)$ and $w(q+1)$. Because the gap weight increases by the same W_s for each space after the first one, when evaluating $E(i, j)$ or $F(i, j)$ we need not be concerned with exactly where a gap begins, but only whether it

has already begun or whether a new gap is being started (either opposite character i of S_1 or opposite character j of S_2). This insight, as usual, is formalized in a set of recurrences.

The recurrences

For the case where end gaps are included in the alignment value, the base case is easily seen to be

$$V(i,0) = E(i,0) = -W_g - iW_s,$$
$$V(0,j) = F(0,j) = -W_g - jW_s,$$

so that the zero row and columns of the table for V can be filled in easily. When end gaps are free, then $V(i,0) = V(0,j) = 0$.

The general recurrences are

$$V(i,j) = \max[E(i,j), F(i,j), G(i,j)],$$
$$G(i,j) = \begin{cases} V(i-1,j-1) + W_m, \text{ if } S_1(i) = S_2(j) \\ V(i-1,j-1) - W_{ms}, \text{ if } S_1(i) \neq S_2(j), \end{cases}$$
$$E(i,j) = \max[E(i,j-1), V(i,j-1) - W_g] - W_s,$$
$$F(i,j) = \max[F(i-1,j), V(i-1,j) - W_g] - W_s.$$

To better understand these recurrences, consider the recurrence for $E(i,j)$. By definition, $S_1(i)$ will be aligned to the left of $S_2(j)$. The recurrence says that either 1. $S_1(i)$ is exactly one place to the left of $S_2(j)$, in which case a gap begins in S_1 opposite character $S_2(j)$, and $E(i,j) = V(i,j-1) - W_g - W_s$ or 2. $S_1(i)$ is to the left of $S_2(j-1)$, in which case the same gap in S_1 is opposite both $S_2(j-1)$ and $S_2(j)$, and $E(i,j) = E(i,j-1) - W_s$. An explanation for $F(i,j)$ is similar, and $G(i,j)$ is the simple case of aligning $S_1(i)$ opposite $S_2(j)$.

As before, the value of the optimal alignment is found in cell (n,m) if right end spaces contribute to the objective function. Otherwise the value of the optimal alignment is the maximum value in the nth row or mth column.

The reader should be able to verify that these recurrences are correct but might wonder why $V(i,j-1)$ and not $G(i,j-1)$ is used in the recurrence for $E(i,j)$. That is, why is $E(i,j)$ not $\max[E(i,j-1), G(i,j-1) - W_g] - W_s$? This recurrence would be incorrect because it would not consider alignments that have a gap in S_2 bordered on the left by character $j-1$ of S_2 and ending opposite character i of S_1, followed immediately by a gap in S_1. The expanded recurrence $E(i,j) = \max[E(i,j-1), G(i,j-1) - W_g, V(i,j-1) - W_g] - W_s$ would allow for all alignments and would be correct, but the inclusion of the middle term $(G(i,j-1) - W_g)$ is redundant because the last term $(V(i,j-1) - W_g)$ includes it.

Time analysis

Theorem 11.8.2. *The optimal alignment with affine gap weights can be computed in $O(nm)$ time, the same time as for optimal alignment without a gap term.*

PROOF Examination of the recurrences shows that for any pair (i,j), each of the terms $V(i,j)$, $E(i,j)$, $F(i,j)$, and $G(i,j)$ is evaluated by a constant number of references to previously computed values, arithmetic operations, and comparisons. Hence $O(nm)$ time suffices to fill in all the $(n+1) \times (m+1)$ cells in the dynamic programming table. □

11.9. Exercises

1. Write down the edit transcript for the alignment example on page 226.
2. The definition given in this book for string transformation and edit distance allows at most one operation per position in each string. But part of the motivation for string transformation and edit distance comes from an attempt to model evolution, where there is no restriction on the number of mutations that could occur at the same position. A deletion followed by an insertion and then a replacement could all happen at the same position. However, even though multiple operations at the same position are allowed, they will not occur in the transformation that uses the fewest number of operations. Prove this.
3. In the discussion of edit distance, all transforming operations were assumed to be done to one string only, and a "hand-waiving" argument was given to show that no greater generality is gained by allowing operations on both strings. Explain in detail why there is no loss in generality in restricting operations to one string only.
4. Give the details for how the dynamic programming table for edit distance or alignment can be filled in columnwise or by successive antidiagonals. The antidiagonal case is useful in the context of practical parallel computation. Explain this.
5. In Section 11.3.3, we described how to create an edit transcript from the traceback path through the dynamic programming table for edit distance. Prove that the edit transcript created in this way is an optimal edit transcript.
6. In Part I we discussed the exact matching problem when don't-care symbols are allowed. Formalize the edit distance problem when don't-care symbols are allowed in both strings, and show how to handle them in the dynamic programming solution.
7. Prove Theorem 11.3.4 showing that the pointers in the dynamic programming table completely capture all the optimal alignments.
8. Show how to use the optimal (global) alignment value to compute the edit distance of two strings and vice versa. Discuss in general the formal relationship between edit distance and string similarity. Under what circumstances are these concepts essentially equivalent, and when are they different?
9. The method discussed in this chapter to construct an optimal alignment left back-pointers while filling in the dynamic programming (DP) table, and then used those pointers to trace back a path from cell (n,m) to cell $(0,0)$. However, there is an alternate approach that works even if no pointers are available. If given the full DP table without pointers, one can construct an alignment with an algorithm that "works through" the table in a single pass from cell (n,m) to cell $(0,0)$. Make this precise and show it can be done as fast as the algorithm that fills in the table.
10. For most kinds of alignments (for example, global alignment without arbitrary gap weights), the traceback using pointers (as detailed in Section 11.3.3) runs in $O(n+m)$ time, which is less than the time needed to fill in the table. Determine which kinds of alignments allow this speedup.
11. Since the traceback paths in a dynamic programming table correspond one-to-one with the optimal alignments, the number of distinct cooptimal alignments can be obtained by computing the number of distinct traceback paths. Give an algorithm to compute this number in $O(nm)$ time.

    Hint: Use dynamic programming.
12. As discussed in the previous problem, the cooptimal alignments can be found by enumerating all the traceback paths in the dynamic programming table. Give a backtracking method to find each path, and each cooptimal alignment, in $O(n+m)$ time per path.
13. In a dynamic programming table for edit distance, must the entries along a row be

nondecreasing? What about down a column or down a diagonal of the table? Now discuss the same questions for optimal global alignment.

14. Give a complete argument that the formula in Theorem 11.6.1 is correct. Then provide the details for how to find the longest common subsequence, not just its length, using the algorithm for weighted edit distance.

15. As shown in the text, the longest common subsequence problem can be solved as an optimal alignment or similarity problem. It can also be solved as an operation-weight edit distance problem.

    Let u represent the length of the longest common subsequence of two strings of lengths n and m. Using the operation weights of $d = 1$, $r = 2$, and $e = 0$, we claim that $D(n, m) = m+n-2u$ or $u = (m+n-D(n, m))/2$. So, $D(n, m)$ is minimized by maximizing u. Prove this claim and explain in detail how to find a longest common subsequence using a program for operation-weight edit distance.

16. Write recurrences for the longest common subsequence problem that do not use weights. That is, solve the *lcs* problem more directly, rather than expressing it as a special case of similarity or operation-weighted edit distance.

17. Explain the correctness of the recurrences for similarity given in Section 11.6.1.

18. Explain how to compute edit distance (as opposed to similarity) when end spaces are free.

19. Prove the one-to-one correspondence between shortest paths in the edit graph and minimum weight global alignments.

20. Show in detail that the end-space free variant of the similarity problem is correctly solved using the method suggested in Section 11.6.4.

21. Prove Theorem 11.6.2, and show in detail the correctness of the method presented for finding the shortest approximate occurrence of P in T ending at position j.

22. Explain how to use the dynamic programming table and traceback to find all the optimal solutions (pairs of substrings) to the local alignment problem for two strings S_1 and S_2.

23. In Section 11.7.3, we mentioned that the dynamic programming table is often used to identify pairs of substrings of high similarity, which may not be optimal solutions to the local alignment problem. Given similarity threshold t, that method seeks to find pairs of substrings with similarity value t or greater. Give an example showing that the method might miss some qualifying pairs of substrings.

24. Show how to solve the alphabet-weight alignment problem with affine gap weights in $O(nm)$ time.

25. The discussions for alignment with gap weights focused on how to compute the values in the dynamic programming table and did not detail how to construct an optimal alignment. Show how to augment the algorithm so that it constructs an optimal alignment. Try to limit the amount of additional space required.

26. Explain in detail why the recurrence $E(i, j) = \max[E(i, j-1), G(i, j-1) - W_g, V(i, j-1) - W_s]$ is correct for the affine gap model, but is redundant, and that the middle term $(G(i, j-1) - W_g)$ can be removed.

27. The recurrences relations we developed for the affine gap model follow the logic of paying $W_g + W_s$ when a gap is "initiated" and then paying W_s for each additional space used in that gap. An alternative logic is to pay $W_g + W_s$ at the point when the gap is "completed." Write recurrences relations for the affine gap model that follow that logic. The recurrences should compute the alignment in $O(nm)$ time. Recurrences of this type are developed in [166].

28. In the *end-gap free* version of alignment, spaces and gaps at either end of the alignment do not contribute to the cost of the alignment. Show how to use the affine gap recurrences developed in the text to solve the end-gap free version of the affine gap model of alignment. Then consider using the alternate recurrences developed in the previous exercise. Both should run in $O(nm)$ time. Is there any advantage to using one over the other of these recurrences?

29. Show how to extend the *agrep* method of Section 4.2.3 to allow character insertions and deletions.

30. Give a *simple* algorithm to solve the local alignment problem in $O(nm)$ time if no spaces are allowed in the local alignment.

31. **Repeated subatrings.** Local alignment between two different strings finds pairs of substrings from the two strings that have high similarity. It is also important to find substrings of a single string that have high similarity. Those substrings represent *inexact repeated substrings*. This suggests that to find inexact repeats in a single string one should locally align of a string against itself. But there is a problem with this approach. If we do local alignment of a string against itself, the best substring will be the entire string. Even using all the values in the table, the best path to a cell (i, j) for $i \neq j$ may be strongly influenced by the main diagonal. There is a simple fix to this problem. Find it. Can your method produce two substrings that overlap? Is that desirable? Later in Exercise 17 of Chapter 13, we will examine the problem of finding the most similar *nonoverlapping* substrings in a single string.

32. **Tandem repeats.** Let P be a pattern of length n and T a text of length m. Let P^m be the concatenation of P with itself m times, so P^m has length mn. We want to compute a local alignment between P^m and T. That will find an interval in T that has the best global alignment (according to standard alignment criteria) with some tandem repeat of P. This problem differs from the problem considered in Exercise 4 of Chapter 1, because errors (mismatches and insertions and deletions) are now allowed. The particular problem arises in studying the secondary structure of proteins that form what is called a *coiled-coil* [158]. In that context, P represents a *motif* or *domain* (a pattern for our purposes) that can repeat in the protein an unknown number of times, and T represents the protein. Local alignment between P^m and T picks out an interval of T that "optimally" consists of tandem repeats of the motif (with errors allowed). If P^m is explicitly created, then standard local alignment will solve the problem in $O(nm^2)$ time. But because P^m consists of identical copies of P, an $O(nm)$-time solution is possible. The method essentially simulates what the dynamic programming algorithm for local alignment would do if it were executed with P^m and T explicitly. Below we outline the method.

    The dynamic programming algorithm will fill in an $m + 1$ by $n + 1$ table V, whose rows are numbered 0 to n, and whose columns are numbered 0 to m. Row 0 and column 0 are initialized to all 0 entries. Then in each row i, from 1 to m, the algorithm does the following: It executes the standard local alignment recurrences in row i; it sets $V(i, 0)$ to $V(i, n)$; and then it executes the standard local alignment recurrences in row i again. After completely filling in each row, the algorithm selects the cell with largest V value, as in the standard solution to the local alignment problem.

    Clearly, this algorithm only takes $O(nm)$ time. Prove that it correctly finds the value of the optimal local alignment between P^m and T. Then give the details of the traceback to construct the optimal local alignment. Discuss why P was (conceptually) expanded to P^m and not a longer or shorter string.

33. **a.** Given two strings S_1 and S_2 (of lengths n and m) and a parameter δ, show how to construct the following matrix in $O(nm)$ time: $\mathcal{M}(i, j) = 1$ if and only if there is an alignment of S_1 and S_2 in which characters $S_1(i)$ and $S_2(j)$ are aligned with each other and the value of the

alignment is within δ of the maximum value alignment of S_1 and S_2. That is, if $V(S_1, S_2)$ is the value of the optimal alignment, then the best alignment that puts $S_1(i)$ opposite $S_2(j)$ should have value at least $V(S_1, S_2) - \delta$. This matrix $\mathcal{M}$ is used [490] to provide *some* information, such as common or uncommon features, about the set of *suboptimal* alignments of S_1 and S_2. Since the biological significance of the *optimal* alignment is sometimes uncertain, and optimality depends on the choice of (often disputed) weights, it is useful to efficiently produce or study a set of suboptimal (but close) alignments in addition to the optimal one. How can the matrix $\mathcal{M}$ be used to produce or study these alignments?

b. Show how to modify matrix $\mathcal{M}$ so that $\mathcal{M}(i, j) = 1$ if and only if $S_1(i)$ and $S_2(j)$ are aligned in every alignment of S_1 and S_2 that has value at least $V(S_1, S_2) - \delta$. How efficiently can this matrix be computed? The motivation for this matrix is essentially the same as for the matrix described in the preceding problem and is used in [443] and [445].

34. Implement the dynamic programming solution for alignment with a gap term in the objective function, and then experiment with the program to find the right weights to solve the cDNA matching problem.

35. The process by which intron-exon boundaries (called *splice sites*) are found in mRNA is not well understood. The simplest hope – that splice sites are marked by patterns that always occur there and never occur elsewhere – is false. However, it is true that certain short patterns very frequently occur at the splice sites of introns. In particular, most introns start with the dinucleotide *GT* and end with *AG*. Modify the dynamic programming recurrences used in the cDNA matching problem to enforce this fact.

There are additional pattern features that are known about introns. Search a library to find information about those conserved features – you'll find a lot of interesting things while doing the search.

36. Sequence to structure deduction via alignment

An important application for aligning protein strings is to deduce unknown secondary structure of one protein from known secondary structure of another protein. From that secondary structure, one can then try to determine the three-dimensional structure of the protein by model building methods. Before describing the alignment exercise, we need some background on protein structure.

A string of a typical globular protein (a typical enzyme) consists of substrings that form the tightly wrapped *core* of the protein, interspersed by substrings that form *loops* on the exterior of the protein. There are essentially three types of secondary structures that appear in globular proteins: α-helixes and β-sheets, which make up the core of the protein, and loops on the exterior of the protein. There are also turns, which are smaller than loops. The structure of the core of the protein is highly conserved over time, so that any large insertions or deletions are much more likely to occur in the loops than in the core.

Now suppose one knows the secondary (or three-dimensional) structure of a protein from one species, and one has the *sequence* of the homologous protein from another species, but not its two- or three-dimensional structure. Let S_1 denote the string for the first protein and S_2 the second. Determining two- or three-dimensional structure by crystallography or NMR is very complex and expensive. Instead, one would like to use sequence alignment of S_1 and S_2 to identify the α and β structures in S_2. The hope is that with the proper alignment model, scoring matrix, and gap penalties, the substrings of the α and β structures in the two strings will align with each other. Since the locations of the α and β regions are known in S_1, a "successful" alignment will identify the α and β regions in S_2. Now, insertions and deletions in core regions are rare, so an alignment that successfully identifies the α and β regions in S_2 should not have large gaps in the α and β regions in S_1. Similarly, the alignment should not have large gaps in the substrings of S_2 that align to the known α and β regions of S_1.

Usually a scoring matrix is used to score matches and mismatches, and a affine (or linear) gap penalty model is also used. Experiments [51, 447] have shown that the success of this approach is very sensitive to the exact choice of the scoring matrix and penalties. Moreover, it has been suggested that the gap penalty must be made higher in the substrings forming the α and β regions than in the rest of the string (for example, see [51] and [296]). That is, no *fixed* choice for gap penalty and space penalty (gap initiation and gap extension penalties in the vernacular of computational biology) will work. Or at least, having a higher gap penalty in the secondary regions will more likely result in a better alignment. High gap penalties tend to keep the α and β regions unbroken. However, since insertions and deletions do definitely occur in the loops, gaps in the alignment of regions outside the core should be allowed.

This leads to the following alignment problem: How do you modify the alignment model and penalty structure to achieve the requirements outlined above? And, how do you find the optimal alignment within those new constraints?

Technically, this problem is not very hard. However, the application to deducing secondary structure is very important. Orders of magnitude more protein sequence data are available than are protein structure data. Much of what is "known" about protein structure is actually obtained by deductions from protein sequence data. Consequently, deducing structure from sequence is a central goal.

A multiple alignment version of this structure prediction problem is discussed in the first part of Section 14.10.2.

37. Given two strings S_1 and S_2 and a text T, you want to find whether there is an occurrence of S_1 and S_2 *interwoven* (without spaces) in T. For example, the strings *abac* and *bbc* occur interwoven in *cabbabccdw*. Give an efficient algorithm for this problem. (It may have a relationship to the longest common subsequence problem.)

38. As discussed earlier in the exercises of Chapter 1, bacterial DNA is often organized into circular molecules. This motivates the following problem: Given two linear strings of lengths n and m, there are n circular shifts of the first string and m circular shifts of the second string, and so there are nm pairs of circular shifts. We want to compute the global alignment for each of these nm pairs of strings. Can that be done more efficiently than by solving the alignment problem from scratch for each pair? Consider both worst-case analysis and "typical" running time for "naturally occurring" input.

Examine the same problem for local alignment.

39. The stuttering subsequence problem [328]. Let P and T be strings of n and m characters each. Give an $O(m)$-time algorithm to determine if P occurs as a *subsequence* of T.

Now let P^i denote the string P where each character is repeated i times. For example, if P = *abc* then P^3 is *aaabbbccc*. Certainly, for any fixed i, one can test in $O(m)$ time whether P^i occurs as a subsequence of T. Give an algorithm that runs in $O(m \log m)$ time to determine the largest i such that P^i is a subsequence of T. Let $Maxi(P, T)$ denote the value of that largest i.

Now we will outline an approach to this problem that reduces the running time from $O(m \log m)$ to $O(m)$. You will fill in the details.

For a string T, let d be the number of distinct characters that occur in T. For string T and character x in T, define $odd(x)$ to be the positions of the odd occurrences of x in T, that is, the positions of the first, third, fifth, etc. occurrence of x in T. Since there are d distinct characters in T, there are d such odd sets. For example, if T = 01200021120222201100 01 then $odd(1)$ is 2,9,18. Now define $half(T)$ as the subsequence of T that remains after *removing* all the characters in positions specified by the d odd sets. For example, $half(T)$

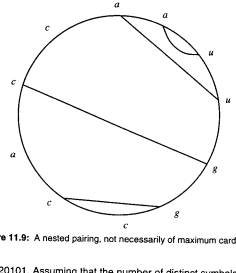


Figure 11.9: A nested pairing, not necessarily of maximum cardinality.

above is 0021220101. Assuming that the number of distinct symbols, d, is fixed ahead of time, give an $O(m)$-time algorithm to find *half*(T). Now argue that the length of *half*(T) is at most $m/2$. This will be used later in the time analysis.

Now prove that $|Max\,i(P, T) - 2Max\,i(P, half(T))| \leq 1$.

This fact is the critical one in the method.

The above facts allow us to find *Maxi*(P, T) in $O(m)$ time by a divide-and-conquer recursion. Give the details of the method: Specify the termination conditions of the divide and conquer, prove correctness of the method, set up a recurrence relation to analyze the running time, and then solve the relation to obtain an $O(m)$ time bound.

Harder problem: What is a realistic application for the stuttering subsequence problem?

40. As seen in the previous problem, it is easy to determine if a single pattern P occurs as a subsequence in a text T. This takes $O(m)$ time. Now consider the problem of determining if *any* pattern in a *set* of patterns occurs in a text. If n is the length of all the patterns in the set, then $O(nm)$ time is obtained by solving the problem for each pattern separately. Try for a time bound that is significantly better than $O(nm)$. Recall that the analogous substring set problem can be solved in $O(n + m)$ time by Aho–Corasik or suffix tree methods.

41. The tRNA folding problem. The following is an extremely crude version of a problem that arises in predicting the secondary (planar) structure of transfer RNA molecules. Let S be a string of n characters over the RNA alphabet a, c, u, g. We define a *pairing* as set of disjoint pairs of characters in S. A pairing is called *proper* if it only contains (a, u) pairs or (c, g) pairs. This constraint arises because in RNA a and u are complementary nucleotides, as are c and g. If we draw S as a circular string, we define a *nested pairing* as a proper pairing where each pair in the pairing is connected by a line inside the circle, and where the lines do not cross each other. (See Figure 11.9). The problem is to find a nested pairing of largest cardinality. Often one has the additional constraint that a character may not be in a pair with either of its two immediate neighbors. Show how to solve this version of the tRNA folding problem in $O(n^3)$ time using dynamic programming.

Now modify the problem by adding weights to the objective function so that the weight of an $a-u$ pair is different than the weight of a $c-g$ pair. The goal now is to find a nested pairing of maximum total weight. Give an efficient algorithm for this weighted problem.

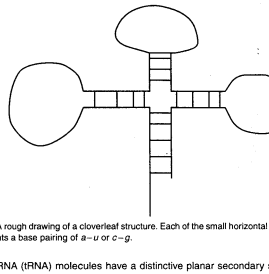


Figure 11.10: A rough drawing of a cloverleaf structure. Each of the small horizontal or vertical lines inside a stem represents a base pairing of $a-u$ or $c-g$.

42. Transfer RNA (tRNA) molecules have a distinctive planar secondary structure called the *cloverleaf* structure. In a cloverleaf, the string is divided into alternating *stems* and *loops* (see Figure 11.10). Each stem consists of two parallel substrings that have the property that any pair of opposing characters in the stem must be complements (a with u; c with g). Chemically, each complementary stem pair forms a bond that contributes to the overall stability of the molecule. A $c-g$ bond is stronger than an $a-u$ bond.

Relate this (very superficial) description of tRNA secondary structure to the weighted nested pairing problem discussed above.

43. The true bonding pattern of complementary bases (in the stems) of tRNA molecules mostly conforms to the noncrossing condition in the definition of a nested pairing. However, there are exceptions, so that when the secondary structure of known tRNA molecules is represented by lines through the circle, a few lines may cross. These violations of the noncrossing condition are called *psuedoknots*.

Consider the problem of finding a maximum cardinality proper pairing where a fixed number of psuedoknots are allowed. Give an efficient algorithm for this problem, where the complexity is a function of the permitted number of crossings.

44. RNA sequence and structure alignment. Because of the nested pairing structure of RNA, it is easy to incorporate some structural considerations when aligning RNA strings. Here we examine alignments of this kind.

Let P be an RNA pattern string with a known pairing structure, and let T be a larger RNA text string with a known pairing structure. To represent pairing structure in P, let $O_P(i)$ be the *offset* (positive or negative) of the mate of the character at position i, if any. For example, if the character at position 17 is mated to the character at position 46, then $O_P(17) = 29$ and $O(46) = -29$. If the character at position i has no mate, then $O_P(i)$ is zero. The structure of T is similarly represented by an offset vector O_T. Then P exactly occurs in T starting at position j if and only if $P(i) = T(j+i-1)$ and $O_P(i) = O_T(j+i-1)$, for each position i in P.

a. Assuming the lengths of P and T are n and m, respectively, give an $O(n+m)$-time algorithm to find every place that P exactly occurs in T.

b. Now consider a more liberal criteria for deciding that P occurs in T starting at position j. We again require that $P(i) = T(j+i-1)$ for each position i in P, but now only require that $O_P(i) = O_T(j+i-1)$ when $O_P(i)$ is not zero.

Give an efficient algorithm to find all locations where P occurs in T under the more liberal definition of occurrence. The naive, $O(nm)$-time solution of explicitly aligning P to every starting position j and then checking for a match is not efficient. An efficient solution can be obtained using only methods in Part I and II of the book.

c. Discuss when the more liberal definition is reasonable and when it may not be.

45. A gap modeling question

The recurrences given in Section 11.8.5 for the case of arbitrary gap weights raise a subtle question about the proper gap model when the gap penalty w is arbitrary. With those recurrences, any single gap can be considered as two or more gaps that just happen to be adjacent. Suppose, for example, $w(q) = 1$ for $q \leq 5$, and $w(q) = 10^6$ for $i > 5$. Then, a gap of length 10 would have weight 10^6 if considered as a single gap, but would only have weight 2 if considered as two adjacent gaps of length five each. The recurrences from Section 11.8.5 would treat those ten spaces as two adjacent gaps with total weight 2. Is this the proper gap model?

There are two viewpoints on this question. In one view, the goal is to model the most likely set of mutation events transforming one string into another, and the alignment is just an aid in *displaying* this transformation. The primitive mutational events allowed are the transformation of single characters (mismatches in the alignment) and insertion and deletion of blocks of characters of arbitrary lengths (each of which causes a gap in the alignment). With this view, it is perfectly proper to have two adjacent gaps on the same string. These are just two block insertions or deletions that happen to have occurred next to each other. If the gap weights correctly model the costs of such block operations, and the cost is a concave increasing function of length as in the above example, then it is much more likely that a long gap will be created by several insertion or deletion events than by a single such event. With this view, one should insist that the dynamic program allow adjacent gaps when they are advantageous.

In the other view, one is just interested in how "similar" two strings are, and there may be no explicit mutational model. Then, a given alignment of two strings is simply one way to demonstrate the similarity of the two strings. In that view, a gap is a maximal set of adjacent spaces and so should not be broken into smaller gaps.

With arbitrary gap weights, the dynamic programming recurrences presented correctly model the first view, but not the second. Also, in the case of convex (and hence affine or constant) gap weights, the given recurrences correctly model both views, since there is no incentive to break up a gap into shorter gaps. However, if gap weights with concave increasing sections are thought proper, then different recurrences are required to correctly model the second view. The recurrences below correctly implement the second view:

$$V(i, j) = \max[E(i, j), F(i, j), G(i, j)].$$

$$G(i, j) = V(i-1, j-1) + s(S_1(i), S_2(j)).$$

$$E(i, j) = \max[G(i, k) - w(j-k)] \text{ (over } 0 \leq k \leq j-1).$$

$$F(i, j) = \max[G(l, j) - w(i-l)] \text{ (over } 0 \leq l \leq i-1).$$

These equations differ from the recurrences of Section 11.8.5 by the change of $V(i, k)$ and $V(l, j)$ to $G(i, k)$ and $G(l, j)$ in the equations for $E(i, j)$ and $F(i, j)$, respectively. The effect is that every gap except the left-most one must be preceded by two aligned characters; hence there cannot be two adjacent gaps in the same string. However, this also prohibits

two adjacent gaps where each is in a different string. For example, the alignment

```
x x a b c _ _ _ y y
x x _ _ _ i d e y y
```

would never be found by these modified recurrences.

There seems no modeling justification to prohibit adjacent gaps in opposite strings. In fact some mutations, such as *substring inversions* (which are common in DNA), would be best represented in an alignment as adjacent gaps of this type, unless the model of alignment has an explicit notion of inversion (we will consider such a model in Chapter 19). Another example where adjacent spaces would be natural occurs when comparing two mRNA strings that arise from alternative intron splicing. In eukaryotes, genes are often comprised of alternating regions of exons and introns. In the normal mode of transcription, every intron is eventually spliced out, so that the mRNA molecule reflects a concatenation of the exons. But it can also happen, in what is called *alternative splicing*, that exons can be spliced out as well as introns. Consider then the situation where all the introns plus exon i are spliced out, and the situation where all the introns plus exon $i+1$ are spliced out. When these two mRNA strings are compared, the best alignment may very well put exon i against a gap in the second string, and then put exon $i+1$ against a gap in the first string. In other words, the informative alignment *would* have two adjacent gaps in alternate strings. In that case, the recurrences above do not correctly implement the second viewpoint.

Write recurrences for arbitrary gap weights to allow adjacent gaps in the two opposite strings and yet prohibit adjacent gaps in a single string.

12

Refining Core String Edits and Alignments

In this chapter we look at a number of important refinements that have been developed for certain core string edit and alignment problems. These refinements either speed up a dynamic programming solution, reduce its space requirements, or extend its utility.

12.1. Computing alignments in only linear space

One of the defects of dynamic programming for all the problems we have discussed is that the dynamic programming tables use $\Theta(nm)$ space when the input strings have length n and m. (When we talk about the space used by a method, we refer to the maximum space ever in use simultaneously. Reused space does not add to the count of space use.) It is quite common that the limiting resource in string alignment problems is not time but space. That limit makes it difficult to handle large strings, no matter how long we may be willing to wait for the computation to finish. Therefore, it is very valuable to have methods that reduce the use of space without dramatically increasing the time requirements.

Hirschberg [224] developed an elegant and practical space-reduction method that works for many dynamic programming problems. For several string alignment problems, this method reduces the required space from $\Theta(nm)$ to $O(n)$ (for $n < m$) while only doubling the worst-case time bound. Miller and Myers expanded on the idea and brought it to the attention of the computational biology community [344]. The method has since been extended and applied to many more problems [97]. We illustrate the method using the dynamic programming solution to the problem of computing the optimal weighted global alignment of two strings.

12.1.1. Space reduction for computing similarity

Recall that the *similarity* of two strings is a *number*, and that under the similarity objective function there is an optimal alignment whose value equals that number. Now if we only require the similarity $V(n, m)$, and not an actual alignment with that value, then the maximum space needed (in addition to the space for the strings) can be reduced to $2m$. The idea is that when computing V values for row i, the only values needed from previous rows are from row $i-1$; any rows before $i-1$ can be discarded. This observation is clear from the recurrences for similarity. Thus, we can implement the dynamic programming solution using only two rows, one called row C for *current*, and one called row P for *previous*. In each iteration, row C is computed using row P, the recurrences, and the two strings. When that row C is completely filled in, the values in row P are no longer needed and C gets copied to P to prepare for the next iteration. After n iterations, row C holds the values for row n of the full table and hence $V(n, m)$ is located in the last cell of that row. In this way, $V(n, m)$ can be computed in $O(m)$ space and $O(nm)$ time. In fact, any

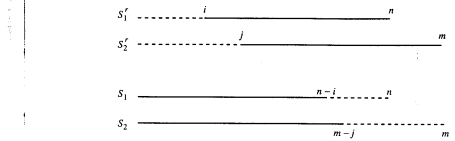


Figure 12.1: The similarity of the first i characters of S_1^r and the first j characters of S_2^r equals the similarity of the last i characters of S_1 and the last j characters of S_2. (The dotted lines denote the substrings being aligned.)

single row of the full table can be found and stored in those same time and space bounds. This ability will be critical in the method to come.

As a further refinement of this idea, the space needed can be reduced to one row plus one additional cell (in addition to the space for the strings). Thus $m+1$ space is all that is needed. And, if $n < m$ then space use can be further reduced to $n+1$. We leave the details as an exercise.

12.1.2. How to find the optimal alignment in linear space

The above idea is fine *if* we only want the similarity $V(n, m)$ or just want to store one preselected row of the dynamic programming table. But what can we do if we actually want an *alignment* that achieves value $V(n, m)$? In most cases it is such an alignment that is sought, not just its value. In the basic algorithm, the alignment would be found by traversing the pointers set while computing the full dynamic programming table for similarity. However, the above linear space method does not store the whole table and linear space is insufficient to store the pointers.

Hirschberg's high-level scheme for finding the optimal alignment in only linear space performs several smaller alignment computations, each using only linear space and each determining a bit more about an actual optimal alignment. The net result of these computations is a full description of an optimal alignment. We first describe how the initial piece of the full alignment is found using only linear space.

Definition For any string α, let α^r denote the reverse of string α.

Definition Given strings S_1 and S_2, define $V^r(i, j)$ as the similarity of the string consisting of the first i characters of S_1^r, and the string consisting of the first j characters of S_2^r. Equivalently, $V^r(i, j)$ is the similarity of the last i characters of S_1 and the last j characters of S_2 (see Figure 12.1).

Clearly, the table of $V^r(i, j)$ values can be computed in $O(nm)$ time, and any single preselected row of that table can be computed and stored in $O(nm)$ time using only $O(m)$ space.

The initial piece of the full alignment is computed in linear space by computing $V(n, m)$ in two parts. The first part uses the original strings; the second part uses the reverse strings. The details of this two-part computation are suggested in the following lemma.

Lemma 12.1.1. $V(n, m) = \max_{0 \le k \le m}[V(n/2, k) + V^r(n/2, m-k)]$.

PROOF This result is almost obvious, and yet it requires a proof. Recall that $S_1[1..i]$ is the prefix of string S_1 consisting of the first i characters and that $S_1^r[1..i]$ is the reverse of the suffix of S_1 consisting of the last i characters of S_1. Similar definitions hold for S_2 and S_2^r.

For any fixed position k' in S_2, there is an alignment of S_1 and S_2 consisting of an alignment of $S_1[1..n/2]$ and $S_2[1..k']$ followed by a disjoint alignment of $S_1[n/2+1..n]$ and $S_2[k'+1..m]$. By definition of V and V', the best alignment of the first type has value $V(n/2, k')$ and the best alignment of the second type has value $V'(n/2, m-k')$, so the combined alignment has value $V(n/2, k') + V'(n/2, m-k') \leq \max_k[V(n/2, k) + V'(n/2, m-k)] \leq V(n, m)$.

Conversely, consider an optimal alignment of S_1 and S_2. Let k' be the right-most position in S_2 that is aligned with a character at or before position $n/2$ in S_1. Then the optimal alignment of S_1 and S_2 consists of an alignment of $S_1[1..n/2]$ and $S_2[1..k']$ followed by an alignment of $S_1[n/2+1..n]$ and $S_2[k'+1..m]$. Let the value of the first alignment be denoted p and the value of the second alignment be denoted q. Then p must be equal to $V(n/2, k')$, for if $p < V(n/2, k')$ we could replace the alignment of $S_1[1..n/2]$ and $S_2[1..k']$ with the alignment of $S_1[1..n/2]$ and $S_2[1..k']$ that has value $V(n/2, k')$. That would create an alignment of S_1 and S_2 whose value is larger than the claimed optimal. Hence $p = V(n/2, k')$. By similar reasoning, $q = V'(n/2, m-k')$. So $V(n, m) = V(n/2, k') + V'(n/2, m-k') \leq \max_k[V(n/2, k) + V'(n/2, m-k)]$.

Having shown both sides of the inequality, we conclude that $V(n, m) = \max_k[V(n/2, k) + V'(n/2, m-k)]$. □

Definition Let k^* be a position k that maximizes $[V(n/2, k) + V'(n/2, m-k)]$.

By Lemma 12.1.1, there is an optimal alignment whose traceback path in the full dynamic programming table (if one had filled in the full n by m table) goes through cell $(n/2, k^*)$. Another way to say this is that there is an optimal (longest) path L from node $(0, 0)$ to node (n, m) in the alignment graph that goes through node $(n/2, k^*)$. That is the key feature of k^*.

Definition Let $L_{n/2}$ be the subpath of L that starts with the last node of L in row $n/2-1$ and ends with the first node of L in row $n/2+1$.

Lemma 12.1.2. *A position k^* in row $n/2$ can be found in $O(nm)$ time and $O(m)$ space. Moreover, a subpath $L_{n/2}$ can be found and stored in those time and space bounds.*

PROOF First, execute dynamic programming to compute the optimal alignment of S_1 and S_2, but stop after iteration $n/2$ (i.e., after the values in row $n/2$ have been computed). Moreover, when filling in row $n/2$, establish and save the normal traceback pointers for the cells in that row. At this point, $V(n/2, k)$ is known for every $0 \leq k \leq m$. Following the earlier discussion, only $O(m)$ space is needed to obtain the values and pointers in rows $n/2$. Second, begin computing the optimal alignment of S_1^r and S_2^r but stop after iteration $n/2$. Save both the values for cells in row $n/2$ along with the traceback pointers for those cells. Again, $O(m)$ space suffices and value $V'(n/2, m-k)$ is known for every k. Now, for each k, add $V(n/2, k)$ to $V'(n/2, m-k)$, and let k^* be an index k that gives the largest sum. These additions and comparisons take $O(m)$ time.

Using the first set of saved pointers, follow any traceback path from cell $(n/2, k^*)$ to a cell k_1 in row $n/2-1$. This identifies a subpath that is on an optimal path from cell $(0, 0)$ to cell $(n/2, k^*)$. Similarly, using the second set of traceback pointers, follow any traceback

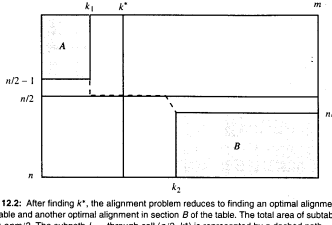


Figure 12.2: After finding k^*, the alignment problem reduces to finding an optimal alignment in section A of the table and another optimal alignment in section B of the table. The total area of subtables A and B is at most $cnm/2$. The subpath $L_{n/2}$ through cell $(n/2, k^*)$ is represented by a dashed path.

path from cell $(n/2, k^*)$ to a cell k_2 in row $n/2+1$. That path identifies a subpath of an optimal path from $(n/2, k^*)$ to (n, m). These two subpaths taken together form the subpath $L_{n/2}$ that is part of an optimal path L from $(0, 0)$ to (n, m). Moreover, that optimal path goes through cell $(n/2, k^*)$. Overall, $O(nm)$ time and $O(m)$ space is used to find k^*, k_1, k_2, and $L_{n/2}$. □

To analyze the full method to come, we will express the time needed to fill in the dynamic programming table of size p by q as cpq, for some unspecified constant c, rather than as $O(pq)$. In that view, the $n/2$ row of the first dynamic program computation is found in $cnm/2$ time, as is the $n/2$ row of the second computation. Thus, a total of cnm time is needed to obtain and store both rows.

The key point to note is that with a cnm-time and $O(m)$-space computation, the algorithm learns k^*, k_1, k_2, and $L_{n/2}$. This specifies part of an optimal *alignment* of S_1 and S_2, and not just the value $V(n, m)$. By Lemma 12.1.1 it learns that there is an optimal alignment of S_1 and S_2 consisting of an optimal alignment of the first $n/2$ characters of S_1 with the first k^* characters of S_2, followed by an optimal alignment of the last $n/2$ characters of S_1 with the last $m-k^*$ characters of S_2. In fact, since the algorithm has also learned the subpath (subalignment) $L_{n/2}$, the problem of aligning S_1 and S_2 reduces to two smaller alignment problems, one for the strings $S_1[1..n/2-1]$ and $S_2[1..k_1]$, and one for the strings $S_1[n/2+1..n]$ and $S_2[k_2..m]$. We call the first of the two problems the *top* problem and the second the *bottom* problem. Note that the top problem is an alignment problem on strings of lengths at most $n/2$ and k^*, while the bottom problem is on strings of lengths at most $n/2$ and $m-k^*$.

In terms of the dynamic programming table, the top problem is computed in section A of the original n by m table shown in Figure 12.2, and the bottom problem is computed in section B of the table. The rest of the table can be ignored. Again, we can determine the values in the middle row of A (or B) in time proportional to the total size of A (or B). Hence the middle row of the top problem can be determined at most $ck^*n/2$ time, and the middle row in the bottom problem can be determined in at most $c(m-k^*)n/2$ time. These two times add to $cnm/2$. This leads to the full idea for computing the optimal alignment of S_1 and S_2.

12.1.3. The full idea: use recursion

Having reduced the original n by m alignment problem (for S_1 and S_2) to two smaller alignment problems (the top and bottom problems) using $O(nm)$ time and $O(m)$ space, we now solve the top and bottom problems by a recursive application of this reduction. (For now, we ignore the space needed to save the subpaths of L.) Applying exactly the same idea as was used to find k^* in the n by m problem, the algorithm uses $O(m)$ space to find the best column in row $n/4$ to break up the top $n/2$ by k_1 alignment problem. Then it reuses $O(m)$ space to find the best column to break up the bottom $n/2$ by $m - k_2$ alignment problem. Stated another way, we have two alignment problems, one on a table of size at most $n/2$ by k^* and another on a table of size at most $n/2$ by $m - k^*$. We can therefore find the best column in the middle row of each of the two subproblems in at most $cnk^*/2 + cn(m - k^*)/2 = cnm/2$ time, and recurse from there with four subproblems.

Continuing in this recursive way, we can find an optimal alignment of the two original strings with $\log_2 n$ levels of recursion, and at no time do we ever use more than $O(m)$ space. For convenience, assume that n is a power of two so that each successive halving gives a whole number. At each recursive call, we also find and store a subpath of an optimal path L, but these subpaths are edge disjoint, and so their total length is $O(n + m)$. In summary, the recursive algorithm we need is:

Hirschberg's linear-space optimal alignment algorithm

Procedure OPTA(l, l', r, r');
  begin
  $h := (l' - l)/2$;
  In $O(l' - l) = O(m)$ space, find an index k^* between l and l', inclusively, such that there is an optimal alignment of $S_1[l..l']$ and $S_2[r..r']$ consisting of an optimal alignment of $S_1[l..h]$ and $S_2[r..k^*]$ followed by an optimal alignment of $S_1[h+1..l']$ and $S_2[k^*+1..r']$. Also find and store the subpath L_h that is part of an optimal (longest) path L' from cell (l, r) to cell (l', r') and that begins with the last cell k_1 on L' in row $h - 1$ and ends with the first cell k_2 on L' in row $h + 1$. This is done as described earlier.
  Call *OPTA*$(l, h - 1, r, k_1)$; {new top problem}
  Output subpath L_h;
  Call *OPTA*$(h + 1, l', k_2, r')$; {new bottom problem}
  end.

The call that begins the computation is to *OPTA*$(1, n, 1, m)$. Note that the subpath L_h is output between the two *OPTA* calls and that the top problem is called before the bottom problem. The effect is that the subpaths are output in order of increasing h value, so that their concatenation describes an optimal path L from $(0, 0)$ to (n, m), and hence an optimal alignment of S_1 and S_2.

12.1.4. Time analysis

We have seen that the first level of recursion uses cnm time and the second level uses at most $cnm/2$ time. At the ith level of recursion, we have 2^{i-1} subproblems, each of which has $n/2^{i-1}$ rows but a variable number of columns. However, the columns in these subproblems are distinct so the total size of all the problems is at most the total number of columns, m, times $n/2^{i-1}$. Hence the total time used at the ith level of recursion is at most $cnm/2^{i-1}$. The final dynamic programming pass to describe the optimal alignment takes cnm time. Therefore, we have the following theorem:

Theorem 12.1.1. *Using Hirschberg's procedure OPTA, an optimal alignment of two strings of length n and m can be found in* $\sum_{i=1}^{\log n} cnm/2^{i-1} \leq 2cnm$ *time and* $O(m)$ *space.*

For comparison, recall that cnm time is used by the original method of filling in the full n by m dynamic programming table. Hirschberg's method reduces the space use from $\Theta(nm)$ to $\Theta(m)$ while only doubling the worst-case time needed for the computation.

12.1.5. Extension to local alignment

It is easy to apply Hirschberg's linear-space method for (global) alignment to solve the local alignment problem for strings S_1 and S_2. Recall that the optimal local alignment of S_1 and S_2 identifies substrings α and β whose global alignment has maximum value over all pairs of substrings. Hence, if substrings α and β can be found using only linear space, then their actual alignment can be found in linear space, using Hirschberg's method for global alignment.

From Theorem 11.7.1, the value of the optimal local alignment is found in the cell (i^*, j^*) containing the maximum v value. The indices i^* and j^* specify the *ends* of strings α and β whose global alignment has a maximum similarity value. The v values can be computed rowwise, and the algorithm must store values for only two rows at a time. Hence the end positions i^* and j^* can be found in linear space. To find the starting positions of the two strings, the algorithm can execute a reverse dynamic program using linear space (we leave this to the reader to detail). Alternatively, the dynamic programming algorithm for v can be extended to set a pointer $h(i, j)$ for each cell (i, j), as follows: If $v(i, j)$ is set to zero, then set the pointer $h(i, j)$ to (i, j); if $v(i, j)$ is set greater than zero, and if the normal traceback pointer would point to cell (p, q), then set $h(i, j)$ to $h(p, q)$. In this way, $h(i^*, j^*)$ specifies the starting positions of substrings α and β, respectively. Since α and β can be found in linear space, the local alignment problem can be solved in $O(nm)$ time and $O(m)$ space. More on this topic can be found in [232] and [97].

12.2. Faster algorithms when the number of differences is bounded

In Sections 9.4 and 9.5 we considered several alignment and matching problems where the number of allowed mismatches was bounded by a parameter k, and we obtained algorithms that run faster than without the imposed bound. One particular problem was the *k-mismatch problem*, finding all places in a text T where a pattern P occurs with at most k mismatches. A direct dynamic programming solution to this problem runs in $O(nm)$ time for a pattern of length n and a text of length m. But in Section 9.4 we developed an $O(km)$-time solution based on the use of a suffix tree, without any need for dynamic programming.

The $O(km)$-time result for the k-mismatch problem is useful because many applications seek only exact or nearly exact occurrences of P in T. Motivated by the same kinds of applications (and additional ones to be discussed in Section 12.2.1), we now extend the k-mismatch result to allow both mismatches and spaces (insertions and deletions from the viewpoint of edit distance). We use the term "differences" to refer to both mismatches and spaces.

Two specific bounded difference problems

We study two specific problems: *the k-difference global alignment problem* and the more involved *k-difference inexact matching problem*. This material was developed originally in the papers of Ukkonen [439], Fickett [155], Myers [341], and Landau and Vishkin [289]. The latter paper was expanded and illustrated with biological applications by Landau, Vishkin, and Nussinov [290]. There is much additional algorithmic work exploiting the assumption that the number of differences may be small [341, 345, 342, 337, 483, 94, 93, 95, 373, 440, 482, 413, 414, 415]. A related topic, algorithms whose expected running time is fast, is studied in Section 12.3.

Definition Given strings S_1 and S_2 and a fixed number k, the *k-difference global alignment problem* is to find the best global alignment of S_1 and S_2 containing at most k mismatches and spaces (if one exists).

The k-difference global alignment problem is a special case of edit distance and is useful when S_1 and S_2 are believed to be fairly similar. It also arises as a subproblem in more complex string processing problems, such as the approximate PCR primer problem considered in Section 12.2.5. The solution to the k-difference global alignment problem will also be used to speed up global alignment when no bound k is specified.

Definition Given strings P and T, the *k-difference inexact matching problem* is to find all ways (if any) to match P in T using at most k character substitutions, insertions, and deletions. That is, find all occurrences of P in T using at most k mismatches and spaces. (End spaces in T but not P are free.)

The inclusion of spaces, in addition to mismatches, allows a more robust version of the k-mismatch problem discussed in Section 9.4, but it complicates the problem. Unlike our solution to the k-mismatch problem, the k-differences problem seems to require the use of dynamic programming. The approach we take is to speed up the basic $O(nm)$-time dynamic programming solution, making use of the assumption that only alignments with at most k differences are of interest.

12.2.1. Where do bounded difference problems arise?

There is a large (and growing) computer science literature on algorithms whose efficiency is based on assuming a bounded number of differences. (See [93] for a survey and comparison of some of these, along with an additional method.) It is therefore appropriate, before discussing specific algorithmic results, to ask whether bounded difference problems arise frequently enough to justify the extensive research effort.

Bounded difference problems arise naturally in situations where a text is repeatedly modified (edited). Alignment of the text before and after modification can highlight the places where changes were made. A related application [345] concerns updating a graphics screen after incremental changes have been made to the displayed text. The assumption behind incremental screen update is that the text has changed by only a small amount, and that changing the text on the screen is slow enough to be seen by the user. The alignment of the old and new text then specifies the fewest changes to the existing screen needed to display the new text. Graphic displays with random access can exploit this information to very rapidly update the screen. This approach has been taken by a number of text editors. The effects of the speedup are easily seen and are often quite dramatic.

12.2.2. Illustrations from molecular biology

In biological applications of alignment, it may be less apparent that a bound on the number of allowed (or expected) differences between strings is ever justified. It has been explicitly stated by some computer scientists that bounded difference alignment methods have no relevance in biology. Certainly, the *major* open problems in aligning and comparing biological sequences arise from strings (usually protein) that have very *little* overall similarity. There is no argument on that point. Still, there are many sequence problems in molecular biology (particularly problems that come from genomics and handling DNA sequences rather than proteins) where it is appropriate to restrict the number of allowed (or expected) differences. A few hours of skimming biology journals will turn up many such examples.[1] We have already discussed one application, that of searching for STSs and ESTs in newly sequenced DNA (see Section 7.8.3). We have also mentioned the approximate PCR primer problem, which will be discussed in detail in Section 12.2.5. We mention here a few additional examples of alignment problems in biology where setting a bound on the number of differences is appropriate.

Chang and Lawler [94] point out that present DNA sequence assembly methods (see Sections 16.14 and 16.15.1) solve a massive number of instances of the approximate suffix-prefix matching problem. These methods compute, for every pair of strings S_1, S_2 in a large set of strings, the best match of a suffix of S_1 with a prefix of S_2, where the match is permitted to contain a "modest" percentage of differences. Using standard dynamic programming methods, those suffix-prefix computations have accounted for over 90% of the computation time used in past sequence assembly projects [363]. But in this application, the only suffix-prefix matches of interest are those with a modest number of differences. Accordingly, it is appropriate to use a faster algorithm that explicitly exploits that assumption. A related problem occurs in the "BAC-PAC" sequencing method involving hundreds of thousands of sequence alignments (see Section 16.13.1).

Another example arises in approaches to locating genes whose mutation causes or contributes to certain genetic diseases. The basic idea is to first identify (through genetic linkage analysis, functional analysis, or other means) a gene, or a region containing a gene, that is believed to cause or contribute to the disease of interest. Copies of that gene or region are then obtained and sequenced from people who are affected by the disease and people (usually relatives) who are not. The sequenced DNA from the affected and unaffected individuals is compared to find any consistent differences. Since many genetic diseases are caused by very small changes in a gene (possibly a single base change, deletion, or inversion), the problem involves comparing strings that have a very small number of differences. Systematic investigation of gene *polymorphisms* (differences) is an active area of research, and there are databases holding all the different sequences that have been found for certain specific genes. These sequences generally will be very similar to one another, so alignment and string manipulation tools that assume a bounded number of differences between strings are useful in handling those sequences.

A similar situation arises in the emerging field of "molecular epidemiology" where one tries to trace the transmission history of a pathogen (usually a virus) whose genome is mutating rapidly. This fine-scale analysis of the changing viral DNA or RNA gives rise to string comparisons between very similar strings. Aligning pairs of these strings to reveal

[1] I recently attended a meeting concerning the Human Genome Project, where numerous examples were presented in talks. I stopped taking notes after the tenth one.

their similarities and differences is a first step in sorting out their history and the constraints on how they can mutate. The history of their mutations is then represented in the form of an evolutionary tree (see Chapter 17). Collections of HIV viruses have been studied in this way. Another good example of molecular epidemiology [348] arises in tracing the history of *Hantavirus* infections in the southwest United States that appeared during the early 1990s.

The final two examples come from the milestone paper [162] reporting the first complete DNA sequencing of a free-living organism, the bacteria *Haemophilus influenzae Rd*. The genome of this bacteria consists of 1,830,137 base pairs and its full sequence was determined by pure shotgun sequencing without initial mapping (see Section 16.14). Before the large-scale sequencing project, many small, disparate pieces of the bacterial genome had been sequenced by different groups, and these sequences were in the DNA databases. One of the ways the sequencers checked the quality of their large-scale sequencing was to compare, when possible, their newly obtained sequence to the previously determined sequence. If they could not match the appropriate new sequences to the old ones with only a small number of differences, then additional steps were taken to assure that the new sequences were correct. Quoting from [162], "The results of such a comparison show that our sequence is 99.67 percent identical overall to those GenBank sequences annotated as *H. influenzae Rd*".

From the standpoint of alignment, the problem discussed above is to determine whether or not the new sequences match the old ones with few differences. This application illustrates both kinds of bounded difference alignment problems introduced earlier. When the location in the genome of the database sequence is known, the corresponding string in the full sequence can be extracted for comparison. The resulting comparison problem is then an instance of the *k-difference global alignment problem* that will be discussed next, in Section 12.2.3. When the genome location of the database sequence P is *not* known (and this is common), the comparison problem is to find all the places in the full sequence where P occurs with a very small number of allowed differences. That is then an instance of the k-difference inexact matching problem, which will be considered in Section 12.2.4.

The above story of *H. influenzae* sequencing will be repeated frequently as systematic large-scale DNA sequencing of various organisms becomes more common. Each full sequence will be checked against the shorter sequences for that organism already in the databases. This will be done not only for quality control of the large-scale sequencing, but also to correct entries in the databases, since it is generally believed that large-scale sequencing is more accurate.

The second application from [162] concerns building a *nonredundant* database of bacterial proteins (NRBP). For a number of reasons (for example, to speed up the search or to better evaluate the statistical significance of matches that are found), it is helpful to reduce the number of entries in a sequence database (in this case, bacterial protein sequences) by culling out, or combining in some way, highly similar, "redundant" sequences. This was done in the work presented in [162], and a "nonredundant" version of GenBank is regularly compiled at The National Center for Biotechnology Information. Fleischmann et al. [162] write:

> Redundancy was removed from NRBP at two stages. All DNA coding sequences were extracted from GenBank ... and sequences from the same species were searched against each other. Sequences having more than 97 percent identity over regions longer than 100 nucleotides were combined. In addition, the sequences were translated and used in protein comparisons with all sequences in SwissProt ... Sequences belonging to the same species and having more than 98 percent similarity over 33 amino acids were combined.

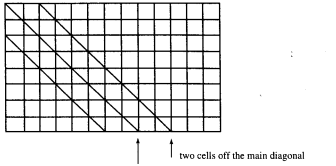


Figure 12.3: The main diagonal and a strip that is $k = 2$ spaces off the main diagonal on each side.

A similar example is discussed in [399] where roughly 170,000 DNA sequences "were subjected to an optimal alignment procedure to identify sequence pairs with at least 97% identity". In these alignment problems, one can impose a bound on the number of allowed differences. Alignments that exceed that bound are not of interest – the computation only needs to determine whether two sequences are "sufficiently similar" or not. Moreover, because these applications involve a large number of alignments (all database entries against themselves), efficiency of the method is important.

Admittedly, not every bounded-difference alignment problem in biology requires a sophisticated algorithm. But applications are so common, the sizes of some of the applications are so large, and the speedsups so great, that it seems unproductive to completely dismiss the potential utility to molecular biology of bounded-difference and bounded-mismatch methods. With this motivation, we now discuss specific techniques that efficiently solve bounded-difference alignment problems.

12.2.3. k-difference global alignment

The problem is to find the best global alignment subject to the added condition that the alignment contains at most k mismatches and spaces, for a given value k. The goal is to reduce the time bound for the solution from $O(nm)$ (based on standard dynamic programming) to $O(km)$. The basic approach is to compute the *edit distance* of S_1 and S_2 using dynamic programming but fill in only an $O(km)$-size portion of the full table.

The key observation is the following: If we define the *main diagonal* of the dynamic programming table as the cells (i, i) for $i \leq n \leq m$, then any path in the dynamic programming table that defines a k-difference global alignment must not contain any cell $(i, i + l)$ or $(i, i - l)$ where l is greater than k (see Figure 12.3). To understand this, note that any path specifying a global alignment begins on the main diagonal (in cell (0, 0)) and ends on, or to the right of, the main diagonal (in cell (n, m)). Therefore, the path must introduce one space in the alignment for every horizontal move that the path makes off the main diagonal. Thus, only those paths that are never more than k horizontal cells from the main diagonal are candidates for specifying a k-difference global alignment. (Note

that this implies that $m - n \leq k$ is a necessary condition for there to be any solution.) Therefore, to find any k-difference global alignment, it suffices to fill in the dynamic programming table in a strip consisting of $2k + 1$ cells in each row, centered on the main diagonal. When assigning values to cells in that strip, the algorithm follows the established recurrence relations for edit distance except for cells on the upper and lower border of the strip. Any cell on the upper border of the strip ignores the term in the recurrence relation for the cell above it (since it is out of the strip); similarly, any cell on the lower border ignores the term in the recurrence relation for the cell to its left. If $m = n$, the size of the strip can be reduced by half (Exercise 4).

If there is no global alignment of S_1 and S_2 with k or fewer differences, then the value obtained for cell (n, m) will be greater than k. That value, greater than k, is not necessarily the correct edit distance of S_1 and S_2, but it will indicate that the correct value for (n, m) is greater than k. Conversely, if there is a global alignment with $d \leq k$ differences, then the corresponding path is contained inside the strip and so the value in cell (n, m) will be correctly set to d. The total area of the strip is $O(kn)$ which is $O(km)$, because n and m can differ by at most k. In summary, we have

Theorem 12.2.1. *There is a global alignment of S_1 and S_2 with at most k differences if and only if the above algorithm assigns a value of k or less to cell (n, m). Hence the k-difference global alignment problem can be solved in $O(km)$ time and $O(km)$ space.*

What if k is not specified?

The solution presented above can be used in somewhat different context. Suppose the edit distance of S_1 and S_2 is k^*, but we don't know k^* or any bound on it ahead of time. The straightforward dynamic programming solution to compute the edit distance, k^*, takes $\Theta(nm)$ time and space. We will reduce those bounds to $\Theta(k^*m)$. So when the edit distance is small, the method runs fast and uses little space. When the edit distance is large, the method only uses $O(nm)$-time and space, the same as for the standard dynamic programming solution.

The idea is to successively guess a bound k on k^* and use Theorem 12.2.1 to determine if the guessed bound is big enough. In detail, the method starts with $k = 1$ and checks if there is a global alignment with at most one difference. If so, then the best global alignment (with zero or one difference) has been found. If not, then the method doubles k and again checks if there is a k-difference global alignment. At each successive iteration the method doubles k and checks whether the current k is sufficient. The process continues until a global alignment is found that has at most k differences, for the current value of k. When the method stops, the best alignment in the present strip (of width k on either side of the main diagonal) must have value k^*. The reason is that the alignment paths are divided into two types: those contained entirely in the present strip and those that go out of the strip. The alignment in hand is the best alignment of the first type, and any path that goes out of the strip specifies an alignment with more than k spaces. It follows that the current value of cell (n, m) must be k^*.

Theorem 12.2.2. *By successively doubling k until there is a k-difference global alignment, the edit distance k^* and its associated alignment are computed in $O(k^*m)$ time and space.*

PROOF Let k' be the largest value of k used in the method. Clearly, $k' \leq 2k^*$. So the total work in the method is $O(k'm + k'm/2 + k'm/4 + \cdots + m) = O(k'm) = O(k^*m)$. □

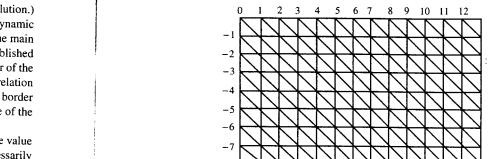


Figure 12.4: The numbered diagonals of the dynamic programming table.

12.2.4. The return of the suffix tree: k-difference inexact matching

We now consider the problem of inexactly matching a pattern P to a text T, when the number of differences is required to be at most k. This is an extension of the k-mismatch problem but is more difficult because it allows spaces in addition to mismatches. The k-mismatch problem was solved using suffix trees alone, but suffix trees are not well structured to handle insertion and deletion errors. The k-difference inexact matching problem is also more difficult than the k-difference global alignment problem because we seek an alignment of P and T in which the end spaces occurring in T are not counted. Therefore, the sizes of P and T can be very different, and we cannot restrict attention to paths that stay within k cells of the main diagonal.

Even so, we will again obtain an $O(km)$ time and space method, combining dynamic programming with the ability to solve longest common extension queries in constant time (see Section 9.1). The resulting solution will be the first of several examples of *hybrid dynamic programming*, where suffix trees are used to solve subproblems within the framework of a dynamic programming computation. The $O(km)$-time result was first obtained by Landau and Vishkin [287] and Myers [341] and extended in a number of papers. Good surveys of many methods for this problem appear in [93] and [421].

Definition As before, the *main diagonal* of the n by m dynamic programming table consists of cells (i, i) for $0 \leq i \leq n \leq m$. The diagonals above the main diagonal are numbered 1 through m; the diagonal starting in cell $(0, i)$ is diagonal i. The diagonals below the main diagonal are numbered -1 through $-n$; the diagonal starting in cell $(i, 0)$ is diagonal $-i$. (See Figure 12.4.)

Since end spaces in the text T are free, row zero of the dynamic programming table is initialized with all zero entries. That allows a left end of T to be opposite a gap without incurring any penalty.

Definition A d-path in the dynamic programming table is a path that starts in row zero and specifies a total of exactly d mismatches and spaces.

Definition A d-path is *farthest-reaching in diagonal i* if it is a d-path that ends in diagonal i, and the index of it's ending column c (along diagonal i) is greater than or equal to the ending column of any other d-path ending in diagonal i.

Graphically, a d-path is farthest reaching in diagonal i if no other d-path reaches a cell further along diagonal i.

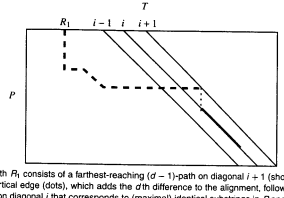


Figure 12.5: Path R_1 consists of a farthest-reaching $(d-1)$-path on diagonal $i+1$ (shown with dashes), followed by a vertical edge (dots), which adds the dth difference to the alignment, followed by a maximal path (solid line) on diagonal i that corresponds to (maximal) identical substrings in P and T.

Hybrid dynamic programming: the high-level idea

At the high level, the $O(km)$ method will run in k iterations, each taking $O(m)$ time. In every iteration $d \leq k$, the method finds the end of the farthest-reaching d-path on diagonal i, for each i from $-n$ to m. The farthest-reaching d-path on diagonal i is found from the farthest-reaching $(d-1)$-paths on diagonals $i-1$, i, and $i+1$. This will be explained in detail below. Any farthest-reaching d-path that reaches row n specifies the end location (in T) of an occurrence of P with exactly d differences. We will implement each iteration in $O(n+m)$ time, yielding the desired $O(km)$-time bound. Space will be similarly bounded.

Details

To begin, when $d = 0$, the farthest-reaching 0-path ending on diagonal i corresponds to the longest common extension of $T[i..m]$ and $P[1..n]$, since a 0-path allows no mismatches or spaces. Therefore, the farthest-reaching 0-path ending on diagonal i can be found in constant time, as detailed in Section 9.1.

For $d > 0$, the farthest-reaching d-path on diagonal i can be found by considering the following three particular paths that end on diagonal i.

- Path R_1 consists of the farthest-reaching $(d-1)$-path on diagonal $i+1$, followed by a vertical edge (a space in text T) to diagonal i, followed by the maximal extension along diagonal i that corresponds to identical substrings in P and T. (See Figure 12.5). Since R_1 begins with a $(d-1)$-path and adds one more space for the vertical edge, R_1 is a d-path.
- Path R_2 consists of the farthest-reaching $(d-1)$-path on diagonal $i-1$, followed by a horizontal edge (a space in pattern P) to diagonal i, followed by the maximal extension along diagonal i that corresponds to identical substrings in P and T. Path R_2 is a d-path.
- Path R_3 consists of the farthest-reaching $(d-1)$-path on diagonal i, followed by a diagonal edge corresponding to a mismatch between a character of P and a character of T, followed by a maximal extension along diagonal i that corresponds to identical substrings from P and T. Path R_3 is a d-path. (See Figure 12.6.)

Each of the paths R_1, R_2, and R_3 ends with a maximal extension corresponding to identical substrings of P and T. In the case of R_1 (or R_2), the starting positions of the two substrings are given by the last entry point of R_1 (or R_2) into diagonal i. In the case of R_3, the starting position is the position just past the last mismatch on R_3.

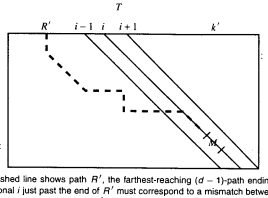


Figure 12.6: The dashed line shows path R', the farthest-reaching $(d-1)$-path ending on diagonal i. The edge M on diagonal i just past the end of R' must correspond to a mismatch between P and T (the characters involved are denoted $P(k)$ and $T(k')$ in the figure).

Theorem 12.2.3. *Each of the three paths R_1, R_2, and R_3 are d-paths ending on diagonal i. The farthest-reaching d-path on diagonal i is the path R_1, R_2, or R_3 that extends the farthest along diagonal i.*

PROOF Each of the three paths is an extension of a $(d-1)$-path, and each extension adds either one more space or one more mismatch. Hence each is a d-path, and each ends on diagonal i by definition. So the farthest-reaching d-path on diagonal i must either be the farthest-reaching of R_1, R_2, and R_3, or it must reach farther on diagonal i than any of those three paths.

Let R' be the farthest-reaching $(d-1)$-path on diagonal i. The edge of the alignment graph along diagonal i that immediately follows R' must correspond to a mismatch, otherwise R' would not be the farthest-reaching $(d-1)$-path on i. Let M denote that edge (see Figure 12.6).

Let R^* denote the farthest-reaching d-path on diagonal i. Since R^* ends on diagonal i, there is a point where R^* enters diagonal i for the last time and then never leaves diagonal i. If R^* enters diagonal i for the last time above edge M, then R^* must traverse edge M, otherwise R^* would not reach as far as R_3. When R^* reaches M (which marks the end of R'), it must also have $(d-1)$ differences; if that portion of R^* had less than a total of $(d-1)$ differences, then it could traverse M creating a $(d-1)$-path on diagonal i that reached farther on diagonal i than R', contradicting the definition of R'. It follows that if R^* enters diagonal i above M, then it will have d differences after it traverses M, and so it will end exactly where R_3 ends. So if R^* is not R_3, then R^* must enter diagonal i *below* edge M.

Suppose R^* enters diagonal i for the last time below edge M. Then R^* must have d differences, at that point of entry; if it had fewer differences then R' would again fail to be the farthest-reaching $(d-1)$-path on diagonal i. Now R^* enters diagonal i for the last time either from diagonal $i-1$ or diagonal $i+1$, say $i+1$ (the case of $i-1$ is symmetric). So R^* traverses a vertical edge from diagonal $i+1$ to diagonal i, which adds a space to R^*. That means that the point where R^* ends on diagonal $i+1$ defines a $(d-1)$-path on diagonal $i+1$. Hence R^* leaves diagonal $i+1$ at or above the point where the path R_1 does. Then R_1 and R^* each have d spaces or mismatches at the points where they enter diagonal i for the last time, and then they each run along diagonal i until reaching an edge corresponding to a mismatch. It follows that R^* cannot reach farther along diagonal i then R_1 does. So in this case, R^* ends exactly where R_1 ends.

The case that R^* enters diagonal i for the last time from diagonal $i-1$ is symmetric, and R^* ends exactly where R_2 ends. In each case we have shown that R^*, the assumed farthest-reaching d-path on diagonal i, ends at the ending point of either R_1, R_2, or R_3. Hence the farthest-reaching d-path on diagonal i is the farthest-reaching of R_1, R_2, and R_3. □

Theorem 12.2.3 is the key to the $O(km)$-time method.

Hybrid dynamic programming: k-differences algorithm

```
begin
d := 0
for i := 0 to m do
find the longest common extension between P[1..n] and T[i..m]. This specifies the
end column of the farthest-reaching 0-path on diagonal i.

For d = 0 to k do
    begin
    For i = −n to m do
        begin
        using the farthest-reaching (d − 1)-paths on diagonals i, i − 1, and i + 1,
        find the end, on diagonal i, of paths R1, R2, and R3. The farthest-reaching
        of these three paths is the farthest-reaching d-path on diagonal i;
        end;
    end;
    Any path that reaches row n in column c say, defines an inexact match of P in
    T that ends at character c of T and that contains at most k differences.
end.
```

Implementation and time analysis

For each value of d and each diagonal i, we record the column in diagonal i where the farthest-reaching d-path ends. Since d ranges from 0 to k and there are only $O(n+m)$ diagonals, all of these values can be stored in $O(km)$ space. In iteration d, the algorithm only needs to retrieve the values computed in iteration $(d-1)$. The entire set of stored values can be used to reconstruct any alignment of P in T with at most k differences. We leave the details of that reconstruction as an exercise.

Now we proceed with the time analysis. For each d and each i, the end of three particular $(d-1)$-paths must be retrieved. For a fixed d and i, this takes constant time, so these retrievals take $O(km)$-time over the entire algorithm. There are also $O(km)$ path extensions, each along a diagonal, that must be computed. But each path extension corresponds to a maximal identical substring in P and T starting at particular known positions in P and T. Hence each path extension requires finding the longest substring starting at a given location in T that matches a substring starting at a given location of P. In other words, each path extension requires a *longest common extension* computation. In Section 9.1 on page 196 we showed that any longest common extension computation can be done in constant time, after linear preprocessing of the strings. Hence the $O(km)$ extensions can all be computed in $O(n+m+km) = O(km)$ total time. Furthermore, as shown in Section 9.1.2, these extensions can be implemented using only a copy of the two strings and a suffix tree for the smaller of the two strings. In summary, we have

Theorem 12.2.4. *All locations in T where pattern P occurs with at most k differences can be found in $O(km)$-time and $O(km)$ space. Moreover, the actual alignment of P and T for each of these locations can be reconstructed in $O(km)$ total time.*

Sometimes this k differences result is reported in a somewhat simpler but less useful form, requiring less space. If one is only interested in the *end locations* in T where P inexactly matches in T with at most k differences, then the $O(km)$ space bound can be reduced to $O(n+m)$. The idea is that the ends of the farthest-reaching $(d-1)$-paths in each diagonal would then not be needed after iteration d and could be discarded. Thus only $O(n+m)$ space is needed to solve the simpler problem.

Theorem 12.2.5. *In $O(km)$-time and $O(n+m)$ space, the algorithm can find all the end locations in T where P matches T with at most k differences.*

12.2.5. The primer (and probe) selection problem revisited – An application of bounded difference matching

In Exercise 61 of Chapter 7, we introduced an exact matching version of the *primer (and probe) selection problem*. The simplest version of that problem starts with two strings α and β. The exact matching version is:

Exact matching primer (and probe) problem For each index j past some starting point, find the shortest substring γ of α (if any) that begins at position j and that does *not* appear as a substring of β.

That problem can be solved in time proportional to the sum of the lengths, of α and β.

The exact matching version of the primer selection problem may not fully model the real primer selection problem (although as noted earlier, the exact matching version may be realistic for probe selection). Recall that primers are short substrings of DNA that *hybridize* to the desired part of string α and that ideally should not hybridize to any parts of another string β. Exact matching is not an adequate model of practical hybridization because a substring of DNA can hybridize, under the right conditions, to another string of DNA even without exact matching; inexact matching of the right type may be enough to allow hybridization. A more realistic version of the primer selection problem moves from exact matching to inexact matching as follows:

Inexact matching primer problem Given a parameter p, find for each index j (past some starting point), the shortest substring γ of α (if any) that begins at position j and that has *edit distance* at least $|\gamma|/p$ from any substring in β.

We solve the above problem efficiently by solving the following-k-difference problem:

k-difference primer problem Given a parameter k, find for each index j (past some starting point), the shortest substring γ of α (if any) that begins at position j and that has edit distance at least k from any substring in β.

Changing $|\gamma|/p$ to k in the problem statement (converting the Inexact matching primer problem to the k-difference primer problem) makes the solution easier but does not reduce the utility of the solution. The reason is that the length of a practical primer must be within a fixed and fairly narrow range, so for fixed p, $|\gamma|/p$ also falls in a small range. Hence for

a specified p, the k-difference primer problem can be solved for a small range of choices for k and still be expected to pick out useful primer candidates.

How to solve the k-difference primer problem

We follow the approach introduced in [243]. The method examines each position j in α separately. For any position j, the k-difference primer problem becomes:

> Find the shortest prefix of string $\alpha[j..n]$ (if it exists) that has edit distance at least k from *every* substring in β.

The problem for a fixed j is essentially the "reverse" of the k-differences inexact matching problem. In the k-difference inexact matching problem we want to find the substrings of T that P matches, with *at most* k differences. But now, we want to *reject* any prefix of $\alpha[j..n]$ that matches a substring of β with less than k differences. The viewpoint is reversed, but the same machinery works.

The solution is to run the k-differences algorithm with string $\alpha[j..n]$ playing the role of P and β playing the role of T. The algorithm computes the farthest-reaching d-paths, for $d = k$, in each diagonal. If row n is reached by any d-path for $d \leq k-1$, then the entire string $\alpha[j..n]$ matches a substring of β with less than k differences, so no acceptable primer can start at j. But, if none of the farthest-reaching $(k-1)$-paths reach row n, then there is an acceptable primer starting at position j. In detail, if none of the farthest-reaching of the d-paths for $d = k-1$ reach row $r < n$, then the substring $\gamma = \alpha[j..r]$ has edit distance at least k from every substring in β. Moreover, if r is the smallest row with that property, then $\alpha[j..r]$ is the shortest substring starting at j that has edit distance at least k from every substring in β.

The above algorithm is applied to each potential starting position j in α, yielding the following theorem:

Theorem 12.2.6. *If α has length n and β has length m, then the k-differences primer selection problem can be solved in $O(knm)$ total time.*

12.3. Exclusion methods: fast expected running time

The k-mismatch and k-difference methods we have presented so far all have worst-case running times of $\Theta(km)$. For $k \ll n$, these speedups are significant improvements over the $\Theta(nm)$ bound for straight dynamic programming. Still, even greater efficiency is desired when m (the size of the text T) is large. The typical situation is that T represents a large database of sequences, and the problem is to find an approximate match of a pattern P in T. The goal is to obtain methods that are significantly faster than $\Theta(km)$ not in worst case, but in *expected* running time. This is reminiscent of the way that the Boyer–Moore method, which typically skips over a large fraction of the text, has an expected running time that is sublinear in the size of the text.

Several methods have been devised for approximate matching problems whose expected running times are faster than $\Theta(km)$. In fact, some of the methods have an expected running time that is *sublinear* in m, for a reasonable range of k. These methods artfully mix *exact* matching with dynamic programming and explicitly use many of the ideas in Parts I and II of the book. Although the details differ considerably, all the methods we will discuss have a similar high-level flavor. We focus on methods due to Baeza-Yates and Perleberg [36], Chang and Lawler [94], and Myers [342], although only the first method will be explained and analyzed in full detail. Two other methods (Wu-Manber [482] and Pevzner-Waterman [373]) will also be mentioned. These methods do not completely achieve the goal of *provable* linear and sublinear expected running times for all practical ranges of errors (and this remains a superb open problem), but they do achieve the goal when the error rate k/n is "modest".

Let σ be the size of the alphabet used in P and T. As usual, n is the length of P and m is the length of T. For the general discussion, an occurrence of P in T with at most k errors (mismatches or differences depending on the particular problem) will be called an *approximate occurrence* of P. The high-level outline of most of the methods is the following:

Partition approach to approximate matching

a. Partition T or P into consecutive regions of a given length r (to be specified later).

b. Search phase Using various exact matching methods, search T to find length-r intervals of T (or regions, if T was partitioned) that could be contained in an approximate occurrence of P. These are called *surviving* intervals. The nonsurviving intervals are definitely not contained in any approximate occurrence of P, and the goal of this phase is to eliminate as many intervals as possible.

c. Check phase For each surviving interval R of T, use some approximate matching method to explicitly check if there is an approximate occurrence of P in a larger interval around R.

The methods differ primarily in the choice of r, in the choice of string to partition, and in the exact matching methods used in the search phase. The methods also differ in the definition of a region but are not generally affected by the specific choice of checking algorithm. The point of the partition approach is to exclude a large amount of T, using only (sub)linear expected time in the search phase, so that only (sub)linear expected time is needed to check the few surviving intervals. A balance is needed between searching and checking because a reduction in the time used in one phase causes an increase in the time used in the other phase.

12.3.1. The BYP method

The first specific method we will look at is due to R. Baeza-Yates and C. Perleberg [36]. Its expected running time is $O(m)$ for modest error rates (made precise below).

Let $r = \lfloor \frac{n}{k+1} \rfloor$, and partition P into consecutive r-length regions (the last region may be of length less than r). By the choice of r, there are $k+1$ regions that have the full length r. The utility of this partition is suggested in the following lemma.

Lemma 12.3.1. *Suppose P matches a substring T' of T with at most k differences. Then T' must contain at least one interval of length r that exactly matches one of the r-length regions of the partition of P.*

PROOF In the alignment of P to T', each region of P aligns to some part of T' (see Figure 12.7), defining $k+1$ subalignments. If each of those $k+1$ subalignments were to contain at least one error (mismatch or space), then there would be more than k differences in total, a contradiction. Therefore, one of the first $k+1$ regions of P must be aligned to an interval of T' without any errors. □

Note that the lemma also holds even for the k-mismatch problem (i.e., when no space

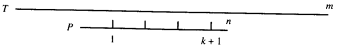


Figure 12.7: The first $k+1$ regions of P are each of length $r = \lfloor \frac{n}{k+1} \rfloor$.

insertions are allowed). Lemma 12.3.1 leads to the following approximate matching algorithm:

Algorithm BYP

a. Let $\mathcal{P}$ be the set of $k+1$ substrings of P taken from the first $k+1$ regions of P's partition.

b. Build a keyword tree (Section 3.4) for the set of "patterns" $\mathcal{P}$.

c. Using the Aho–Corasik algorithm (Section 3.4), find $\mathcal{I}$, the set of all starting locations in T where any pattern in $\mathcal{P}$ occurs exactly.

d. For each index $i \in \mathcal{I}$ use an approximate matching algorithm (usually based on dynamic programming) to locate the end points of all approximate occurrences of P in the substring $T[i-n-k..i+n+k]$ (i.e., in an appropriate-length interval around i).

By Lemma 12.3.1, it is easy to establish that the algorithm correctly finds all approximate occurrences of P in T. The point is that the interval around each i is "large enough" to align with any approximate occurrence of P that spans i, and there can be no approximate occurrence of P outside such an interval. A formal proof is left as an exercise. Now we focus on specific implementation details and time analysis.

Building the keyword tree takes $O(n)$ time, and the Aho-Corasik algorithm takes $O(m)$ (worst-case) time (Section 3.4). So steps b and c take $O(n+m)$ time. There are a number of alternate implementations for steps b and c. One is to build a suffix tree for T, and then use it to find every occurrence in T of a pattern in $\mathcal{P}$ (see Section 7.1). However, that would be very space intensive. A space-efficient version of this approach is to construct a generalized suffix tree for only $\mathcal{P}$, and then match T to it (in the way that matching statistics are computed in Section 7.8.1). Both approaches take $\Theta(n+m)$ worst-case time, but are no faster in expected time because every character in T is examined. A faster approach in practice is to use the Boyer–Moore *set matching* method based on suffix trees, which was developed in Section 7.16. That algorithm will skip over parts of T, and hence it breaks the $\Theta(m)$ bottleneck. A different variation was developed by Wu and Manber [482] who implement steps b and c using the *Shift-And* method (Section 4.2) on a set of patterns. Another approach, found in the paper of Pevzner and Waterman [373] and elsewhere, uses *hashing* to identify long exact matching substrings of P and T. Of course, one can use suffix trees to find long common substrings, and one could develop a Karp–Rabin type method as well. Hashing, or approaches based on suffix trees, that look directly for long common substrings between P and T, seem a bit more robust than BYP because there is no string partition involved. But the only stated time bounds in [373] are the same as those for BYP.

In the checking phase, step d, the algorithm executes some approximate matching algorithm between P and an interval of T of length $O(n)$, for each index in $\mathcal{I}$. Naively, each of these checks can be done in $O(n^2)$ time by dynamic programming (global alignment). Even this time bound will be adequate to establish an expected $O(m)$ overall running time for the range of error rates that will be detailed below. Alternately, the Landau–Vishkin method (Section 12.2) based on suffix trees could be used, so that each check

takes only $O(kn)$ worst-case time. If no spaces are allowed in the alignment of P to T' (only matches and mismatches) then the simpler $O(kn)$-time approach based on longest common extension (Section 9.1) can be used, or if attention is paid to exactly where in P any match is found, then $O(n)$ time suffices for each check.

12.3.2. Expected time analysis of algorithm BYP

Since steps b and c run in $O(m)$ worst-case time, we only need to analyze step d. The key is to estimate the expected size of set $\mathcal{I}$.

In the following analysis, we assume that each character of T is drawn uniformly (i.e., with equal probability) from an alphabet of size σ. However, P can be an arbitrary string. Consider any pattern $p \in \mathcal{P}$. Since p has length r, and T contains roughly m substrings of length r, the expected number of exact occurrences of p in T is m/σ^r. Therefore, the expected total number of occurrences in T of patterns from $\mathcal{P}$ (i.e., the expected size of $\mathcal{I}$) is $m(k+1)/\sigma^r$.

For each $i \in \mathcal{I}$, the algorithm spends $O(n^2)$ time (or less if faster methods are used) in the checking phase. So the expected checking time is $mn^2(k+1)/\sigma^r$. The goal is to make the expected checking time linear in m for modest k, so we must determine what values of k make

$$\frac{mn^2(k+1)}{\sigma^r} < cm,$$

for some constant c.

To simplify the analysis, replace k by $n-1$, and solve for r in

$$\frac{mn^3}{\sigma^r} = cm.$$

This gives $\sigma^r = \frac{n^3}{c}$, so $r = \log_\sigma n^3 - \log_\sigma c$. But $r = \lfloor \frac{n}{k+1} \rfloor$, so

Theorem 12.3.1. *Algorithm BYP runs in $O(m)$ time for $k = O\left(\frac{n}{\log_\sigma n}\right)$.*

Stated another way, as long as the error rate is less than one in $\log_\sigma n$ characters, algorithm BYP will run in linear time as a function of m.

The bottleneck in the BYP method is the $\Theta(m)$ time required to run the Aho–Corasik algorithm. Using the Boyer–Moore set matching method should reduce that time in practice, but we cannot present a time analysis for that approach. However, the Chang–Lawler method has an expected time bound that is provably sublinear for $k = O\left(\frac{n}{\log n}\right)$.

12.3.3. The Chang–Lawler method

For ease of exposition, we will explain the Chang–Lawler (CL) method [94] for the k-mismatches problem; we leave the extension to k-differences as an exercise.

In CL, it is string T, not P, that is partitioned into consecutive fixed regions of length $r = n/2$. These regions are large compared to the regions in BYP. The purpose of the length $n/2$ is to assure that no matter how P is aligned to T (without inserted spaces), at least one of the fixed regions in T's partition is completely contained in the interval spanned by P (see Figure 12.8). Therefore, if P occurs in T with at most k mismatches, there must be one region of T that is spanned by that occurrence of P and, of course, that region matches its counterpart in P with at most k mismatches. Based on this observation, the search phase of CL examines each region in the partition of T to find regions that cannot match

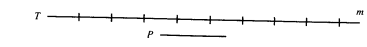


Figure 12.8: Each full region in T has length $r = n/2$. This assures that no matter how P is aligned with T, P spans one full region.

Figure 12.9: Blowup of one region in T aligned with one copy of P. Each black box shows a mismatch between a character in P and its counterpart in T.

any substring of P with at most k mismatches. These regions are excluded, and then an interval around each surviving region is checked using an approximate matching method, as in BYP. The search phase of CL relies heavily on the *matching statistics* discussed in Section 7.8.1.

Recall that the value of matching statistic $ms(i)$ is the length of the longest substring starting at position i of T that matches a substring *somewhere* (an unspecified location) in P. Recall also, that for any string S, all the matching statistics for the positions in S can be computed in $O(|S|)$ total time. This is true even when S is a substring of a larger string T.

Now let T' be the substring of one of the regions of T's partition that matches a substring P' of P with at most k mismatches (see Figure 12.9). The alignment of P' and T' can be divided into at most $k+1$ intervals where no mismatches occur, alternating with intervals containing only mismatches. Let i be the starting position of any one of those matching intervals, and let l be its length. Then clearly, $ms(i) \geq l$. The CL search phase exploits this observation. It executes the following algorithm for each region R in the partition of T:

The CL search in region R

Set j to the starting position j^* of region R in T.
$cn := 0$;
Repeat
compute $ms(j)$;
$j := j + ms(j) + 1$;
$cn := cn + 1$;
Until $cn = k$ or $j - j^* > n/2$.
If $j - j^* > n/2$ then region R survives, otherwise it is excluded.

If R is a surviving region, then in the checking phase CL executes an approximate matching algorithm for P against a neighborhood of T that starts $n/2$ positions to the left of R and ends $n/2$ positions to its right. This neighborhood is of size $3n/2$, and so each check can be executed in $O(kn)$ time.

The correctness of the CL method comes from the following lemma, and the fact that the neighborhoods are "large enough".

Lemma 12.3.2. *When the CL search declares a region R excluded, then there is no occurrence of P in T with at most k mismatches that completely contains region R.*

The proof is easy and is left to the reader, as is its use in a formal proof of the correctness of CL. Now we consider the time analysis.

The CL search is executed on $2m/n$ regions of T. For any region R let j' be the last value of j (i.e., the value of j when cn reaches k or when $j - j^*$ exceeds $n/2$). Thus, in R, matching statistics are computed for the interval of length $j' - j^* \leq n/2$. With the matching statistics algorithm in Section 7.8.1, the time used to compute those matching statistics is $O(j' - j^*)$. Now the expected value of $j' - j^*$ is less than or equal to k times the expected value of $ms(i)$, for any i. Let $E(M)$ denote the expected value of a matching statistic, and let e denote the expected number of regions that survive the search phase. Then the expected time for the search phase is $O(2mkE(M)/n)$, and the expected time for the checking phase is $O(kne)$.

In the following analysis, we assume that P is a random string where each character is chosen uniformly from an alphabet of size σ.

Lemma 12.3.3. *$E(M)$, the expected value of a matching statistic, is $O(\log_\sigma n)$.*

PROOF For fixed length d, there are roughly n substrings of length d in P, and there are σ^d substrings of length d that can be constructed. So, for any specific string α of length d, the probability that α is found somewhere in P is *less than* n/σ^d. This is true for any d, but vacuously true until $\sigma^d = n$ (i.e., when $d = \log_\sigma n$).

Let X be the random variable that has value $\log_\sigma n$ for $ms(i) \leq \log_\sigma n$; otherwise it has value $ms(i)$. Then

$$E(M) < E(X) < \log_\sigma n + \sum_{l=\log_\sigma n}^{\infty} \frac{l}{\sigma^l} = \log_\sigma n + 2. \qquad \square$$

Corollary 12.3.1. The expected time that CL spends in the search phase is $O(2mk \log_\sigma n/n)$, which is sublinear in m for $k < n/\log_\sigma n$.

The analysis for e, the expected number of surviving regions is too difficult to present here. It is shown in [94] that when $k = O(n/\log_\sigma n)$, then $e = m/n^4$, so the expected time that CL spends in the checking phase is $O(km/n^3) = o(m)$. The search phase of CL is so effective in excluding regions of T that the checking phase has very small expected running time.

12.3.4. Multiple filtration for k-mismatches

Both the BYP and the CL methods use fairly simple combinatorial criteria in their search phases to exclude intervals of T. One can devise more stringent conditions that are *necessary* for an interval of T to be contained in an approximate occurrence of P. In the context of the k-mismatches problem, conditions of this type (called filtration conditions) were developed and studied by Pevzner and Waterman [373]. These conditions are used together with substring hashing to obtain another linear expected-time method for the k-mismatch problem. Empirical results are given in [373] that show faster running times in practice than other methods for the k-mismatch problem.

12.3.5. Myers's sublinear-time method

Gene Myers [342, 337] developed an exclusion method that is more sophisticated than the ones we have discussed so far and that runs in sublinear time for a wider range of error rates. The method handles approximate matching with insertions and deletions as well as mismatches. The full algorithm and its analysis are too complex for detailed discussion

here, but we can introduce some of the ideas it uses to address deficiencies in the other exclusion methods.

There are two basic problems with the Baeza-Yates-Perlberg and the Chang–Lawler methods (and the other exclusion methods we have mentioned). First, the exclusion criteria they use permit a large expected number of surviving regions compared to the expected number of true approximate matches. That is, not every initial surviving region is actually contained in an approximate match, and the ratio of expected survivors to expected matches is fairly high (for random patterns and text). Further, the higher the permitted error rate, the more severe is the problem. Second, when a surviving region is first located, the methods move directly to full dynamic programming computations (or some other relatively expensive operations) to check for an approximate match in a large interval around the surviving region. Hence the methods are required to do a large amount of computation for a large number of intervals that don't contain any approximate match.

Compared to the other exclusion methods, Myers's method contains two different ideas to make it both more selective (finding fewer initial surviving regions) and less expensive to test the ones that are found. Myers's algorithm begins in a manner similar to the other exclusion methods. It partitions P into short substrings (to be specified later) and then finds all locations in T where these substrings appear with a small number of allowed differences. The details of the search are quite different from the other methods, but the intent (to exclude a large portion of T from further consideration) is the same. Each of these initial alignments of a substring of P that is found (approximately) in T is called a *surviving match*. A surviving match roughly plays the role of a surviving *region* in the other exclusion methods, but it specifies two substrings (one in P and one in T) rather than just a single substring, as a surviving region does. Another way to think of a surviving region is as a roughly diagonal subpath in the alignment graph for P and T.

Having found the initial surviving matches (or surviving regions), all the other exclusion methods we have mentioned would next check a full interval of length roughly $2n$ around each surviving region in T to see if it contains an approximate match to P. In contrast, Myers's method will *incrementally extend* and check a growing interval around each initial surviving match to create longer surviving matches or to exclude a surviving match from further consideration. This is done in about $O(\log n)$ iterations. (Recall that n is the length of the pattern and m is the length of the text.)

Definition For a given error rate ϵ, a string S ϵ-matches a substring of T if S matches the substring using at most $\epsilon|S|$ insertions, deletions, and mismatches.

For example, let $S = aba$ and $\epsilon = 2/3$. Then ac ϵ-matches S using one mismatch and one deletion operation.

In the first iteration, the pattern P is partitioned into consecutive, nonoverlapping subpatterns of length $\log_\sigma m$ (assumed to be an integer), and the algorithm finds all substrings in T that ϵ-match one of these short subpatterns (discussed in more detail below). The length of these subpatterns is short enough that all the ϵ-matches can be found in sublinear expected time for a wide range of ϵ values. These ϵ-matches are the initial surviving matches.

The algorithm next tries to extend each initial surviving match to become an ϵ-match between substrings (in P and T) that are roughly twice as long as those in the current surviving match. This is done by dynamic programming in an appropriate interval around the surviving match. In each successive iteration, the method applies a more selective and expensive filter, trying to double the length of the ϵ-match around each surviving match.

Since the intervals of interest double in length, the time used per interval grows four fold in each successive iteration. However, the number of surviving matches is expected to fall hyper-exponentially in each successive iteration, more than offsetting the increase in computation time per interval.

With this iterative expansion, the effort expended to check any initial surviving match is doled out incrementally throughout the $O\left(\log \frac{n}{\log m}\right)$ iterations, and is not continued for any surviving match past an iteration where it is excluded. We now describe in a bit more detail how the initial surviving matches are found and how they are incrementally extended in successive iterations.

The first iteration

Definition For a string S and value of ϵ, let $d = \epsilon|S|$. The *d-neighborhood* of S is the set of all strings that ϵ-match S.

For example, over the two-letter alphabet $\{a,b\}$, if $S = aba$ and $d = 1$, then the *1-neighborhood* of S is $\{bba, aaa, abb, aaba, abaa, baba, abba, abab, ba, aa, ab\}$. It is created from S by the operations of mismatch, insertion and deletion respectively. The *condensed d-neighborhood* of S is created from the d-neighborhood of S by removing any substring that is a *prefix* of another string in the d-neighborhood. The condensed 1-neighborhood S is $\{bba, aaa, aaba, abaa, baba, abba, abab\}$.

Recall that pattern P is initially partitioned into subpatterns of length $\log_\sigma m$ (assumed to be an integer). Let $\mathcal{P}$ be the set of these subpatterns. In the first iteration, the algorithm (conceptually) constructs the condensed d-neighborhood for each subpattern in $\mathcal{P}$, and then finds all locations of substrings in text T that *exactly* match one of the substrings in one of the condensed d-neighborhoods. In this way, the method finds all substrings of T that ϵ-match one of the subpatterns in $\mathcal{P}$. These ϵ-matches form the initial surviving matches.

In actuality, the tasks of generating the substrings in the condensed d-neighborhoods and of searching for their exact occurrences in T are intertwined and require text T to have been preprocessed into some index structure. This structure could be a suffix tree, a suffix array or a hash table holding short substrings of T. Details are found in [342].

Myers [342] shows that when the length of the subpatterns is $O(\log_\sigma m)$, then the first iteration can be implemented to run in $O(km^{p(\epsilon)} \log m)$ expected time. The function $p(\epsilon)$ is complicated, but it is convex (negative second derivative) increasing, and increases more slowly as the alphabet size grows. For DNA, it has value less than one for $\epsilon \leq \frac{1}{3}$, and for proteins it has value less than one for $\epsilon \leq 0.56$.

Successive iterations

To explain the central idea, let $\alpha = \alpha_0\alpha_1$, where $|\alpha_0|$ is assumed equal to $|\alpha_1|$.

Lemma 12.3.4. *Suppose α ϵ-matches β. Then β can be divided into two substrings β_0 and β_1 such that $\beta = \beta_0\beta_1$, and either α_0 ϵ-matches β_0 or α_1 ϵ-matches β_1.*

This lemma (used in reverse) is the key to determining how to expand the intervals around the surviving matches in each iteration. For simplicity, assume that n is a power of two and that $\log_\sigma m$ is also a power of two. Let B be a binary tree representing successive divisions of P into two equal size parts, until each part has length $\log_\sigma m$ (see Figure 12.10). The substrings written at the leaves are the subpatterns used in the first iteration of Myers's algorithm. Iteration i of the algorithm examines substrings of P that label (some) nodes of B i levels above the leaves (counting the leaves as level 1).

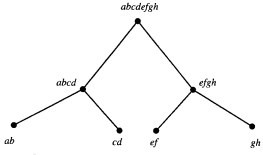


Figure 12.10: Binary tree B defining the successive divisions of P and its partition into regions of length $\log_c m$ (equal to two in this figure).

Suppose at iteration $i-1$ that substrings P' and T' in the query and text, respectively, form a surviving match (i.e., are found to align to form an ϵ-match). Let P'' be the parent of P' in tree B. If P' is a left child of P'', then in iteration i, the algorithm tries to ϵ-match P'' to a substring of T in an interval that extends T' to the right. Conversely, if P' is a right child of P'', then the algorithm tries to ϵ-match P'' with a substring in an interval that extends T' to its left. By Lemma 12.3.4, if the ϵ-match of P' to T' is part of an ϵ-match of P to a substring of T, then P'' will ϵ-match the appropriate substring of T. Moreover, the specified interval in T that must be compared against P'' is just twice as long as the interval for T'. The end result, as detailed in [342], is that all of the checking, and hence the entire algorithm, runs in $O(km^{p(\epsilon)}\log m)$ expected time.

Final comments on Myers's method

There are several points to emphasize. First, the exposition given above is only intended to be an outline of Myers's method, without any analysis. The full details of the algorithm and analysis are found in [342]; [337] provides an overview, in relation to other exclusion methods. Second, unlike the BYP and CL methods, the error rates that establish sublinear (or linear) running times do not depend on the length of P. In BYP and CL, the permitted error rate *decreases* as the length of P increases. In Myers's method, the permitted error rate depends only on the alphabet size. Third, although the expected running times for both CL and for Myers's method are sublinear (for the proper range of error rates), there is an important difference in the nature of these sublinearities. In the CL method, the sublinearity is due to a multiplicative factor that is less than one. But in Myers's method, the sublinearity is due to an *exponent* that is less than one. So as a function of m, the CL bound increases linearly (although for any fixed value of m the expected running time is less than m), while the bound for Myers's method increases sublinearly in m. This is an important distinction since many databases are rapidly increasing in size.

However, Myers's method assumes that the text T has already been preprocessed into some index structure, and the time for that preprocessing (while linear in m) is not included in the above time bounds. In contrast, the running times of the BYP and CL methods include all the work needed for those methods. Finally, Myers has shown that in experiments on problems of meaningful size in molecular biology (patterns of length 80 on texts of length 3 million), the k-difference algorithms of Sections 12.2.4 and 12.2.3 run 100 to 500 times slower than his expected sublinear method.

12.3.6. Final comment on exclusion methods

The fast expected-time exclusion methods have all been developed with the motivation of searching large DNA and protein databases for approximate occurrences of query strings. But the proven results are a bit weak for the case of protein database search, because error rates as high as 85% (the so-called twilight zone) are of great interest when comparing protein sequences [127, 360]. In the twilight zone, evidence of common ancestry may still remain, but it takes some skill to determine if a given match is meaningful or not. Another problem with the exclusion methods presented here is that not all of the methods or analyses extend nicely to the case of weighted or local alignment.

Nonetheless, these results are promising, and the open problem of finding sublinear expected-time algorithms for higher error rates is very inviting. Moreover, we will see in Chapter 15 on database searching that the most effective practical database search methods in use today (BLAST, FASTA, and variants) can be considered as exclusion methods and are based on ideas similar to some of the more formal methods presented here.

12.4. Yet more suffix trees and more hybrid dynamic programming

Although the suffix tree was initially designed and employed to handle complex problems of exact matching, it can be used to great advantage in various problems of *inexact matching*. This has already been demonstrated in Sections 9.4 and 12.2 where the k-mismatch and k-difference problems were discussed. The suffix tree in the latter application was used in combination with dynamic programming to produce a *hybrid dynamic programming* method that is faster than dynamic programming alone. One deficiency of that approach is that it does not generalize nicely to problems of *weighted* alignment. In this section, we introduce a different way to combine suffix trees with dynamic programming for problems of weighted alignment. These ideas have been claimed to be very effective in practice, particularly for large computational projects. However, the methods do not always lend themselves to greatly improved *provable, worst-case* time bounds. The ideas presented here loosely follow the published work of Ukkonen [437] and an unpublished note of Gonnet and Baeza-Yates [34]. The thesis by Bieganski [63] discusses a related idea for using suffix trees in regular expression pattern matching (with errors) and its large-scale application in managing genomic databases. The method of Gonnet and Baeza-Yates has been implemented and extensively used for large-scale protein comparisons [57], [183].

Two problems

We assume the existence of a scoring matrix used to compute the value of any alignment, and hence "edit distance" here refers to *weighted* edit distance. We will discuss two problems in the text and introduce two more related problems in the exercises.

1. **The P-against-all problem** Given strings P and T, compute the edit distance between P and every substring T' of T.
2. **The threshold all-against-all problem** Given strings P and T and a threshold d, find every pair of substrings P' of P and T' of T such that the edit distance between P' and T' is less than d.

The threshold all-against-all problem is similar to problems mentioned in Section 12.2.1 concerning the construction of nonredundant sequence databases. However, the threshold all-against-all problem is harder, because it asks for the alignment of all pairs of *substrings*,

not just the alignment of all pairs of strings. This critical distinction has been the source of some confusion in the literature [50], [56].

12.4.1. The P-against-all problem

The P-against-all problem is an example of a *large-scale alignment* problem that asks for a great amount of related alignment information. If not done carefully, its solution will involve a large amount of redundant computation.

Assume that P has length n and T has length $m > n$. The most naive solution to the P-against-all problem is to enumerate all $\binom{m}{2}$ substrings of T, and then separately compute the edit distance between P and each substring of T. This takes $\Theta(nm^3)$ total time. A moment's thought leads to an improvement. Instead of choosing all substrings of T, we need only choose each *suffix* S of T and compute the dynamic programming edit distance table for strings P and S. If S begins at position i of T, then the last row of that table gives the edit distance between P and every substring of T that begins at position i. That is, the edit distance between P and $T[i..j]$ is found in cell $(n, j-i+1)$ of the table. This approach takes $\Theta(nm^2)$ total time.

We are interested in the P-against-all problem when T is very long. In that case, the introduction of a suffix tree may greatly speed up the dynamic programming computation, depending on how much repetition is contained in string T.[2] (See also Section 7.11.1.) To get the basic idea of the method, consider two substrings T' and T'' of T that are identical for their first n' characters. In the dynamic programming approach above, the edit distances between P and T' and between P and T'' would be computed separately. But if we compute edit distance *columnwise* (instead of in the usual rowwise manner), then we can combine the two edit distance computations for the first n' columns, since the first n' characters of T' and T'' are the same (see Figure 12.11). It would be redundant to compute the first n by n' subtable separately for the two edit distances. This idea of using the commonality of T' and T'' can be formalized and fully exploited through the use of a suffix tree for string T.

Consider a suffix tree $\mathcal{T}$ for string T and recall that any path from the root of $\mathcal{T}$ specifies some substring S of T. If we traverse a path from the root of $\mathcal{T}$, and we let S denote the growing substring corresponding to that path, then during the traversal we can build up (columnwise) the dynamic programming table for the edit distance between P and the growing substring S of T. The full idea then is to traverse $\mathcal{T}$ in a depth-first manner, computing the appropriate dynamic programming column (from the column to its left) for every substring S specified by the current path. When the traversal reaches a node v of $\mathcal{T}$, it stores there the last (most recently generated) column and last subrow of the current subtable (the last row will always be row n). That is, if S is the substring specified by the path to a node v, then what will be stored at v is the last row and column of the dynamic programming table for the edit distance between P and S. When the depth-first traversal visits a child v' of v, it adds columns (one for each character on the (v, v') edge) to this table to correspond to the extension of substring S. When the depth-first traversal reaches a leaf of $\mathcal{T}$ corresponding to the suffix starting at a position i (say) of T, it can then output the values in the last row of the current table. Those values specify the edit distances

[2] Recent estimates put the amount of repeated human DNA at 50 to 60%. That is, 50 to 60% of all human DNA is contained in *nontrivial length*, structured substrings that show up repeatedly throughout the genome. Similar levels of redundancy appear in many other organisms.

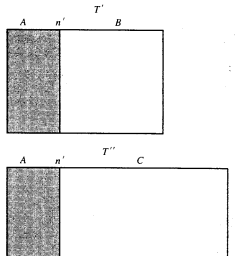


Figure 12.11: A cartoon of the dynamic programming tables for computing the edit distance between P and substring T' (top) and between P and substring T'' (bottom). The two tables share the subtable for P and substring A (shown as a shaded rectangle). This shaded subtable only needs to be computed once.

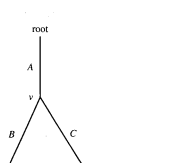


Figure 12.12: A piece of the suffix tree for T. The traversal from the root to node v is accompanied by the computation of subtable A (from the previous figure). At that point, the last row and column of subtable A are stored at node v. Computing the subtable B corresponds to the traversal from v to the leaf representing substring T'. After the traversal reaches the leaf for T', it backs up to node v, retrieves the row and column stored there, and uses them to compute the subtable C needed to compute the edit distance between P and T''.

between P and every substring beginning at position i of T. When the depth-first traversal backs up to a node v, and v has an unvisited child v', the row and column stored at v are retrieved and extended as the traversal follows a new (v, v') edge (see Figure 12.12).

It should be clear that this suffix-tree approach does correctly compute the edit distance between P and every substring of T, and it does exploit repeated substrings (small or large) that may occur in T. But how effective is it compared to the $\Theta(nm^2)$-time dynamic programming approach?

Definition The *string-length* of an edge label in a suffix tree is the length of the string labeling that edge (even though the label is compactly represented by a constant number of characters). The *length of a suffix tree* is the sum of the string-lengths for all of its edges.

The length for a suffix tree $\mathcal{T}$ for a string T of length m can be anywhere between $\Theta(m)$ and $\Theta(m^2)$, depending on how much repetition exists in T. In computational experiments using long substrings of mammalian DNA (length around one million), the string-lengths of the resulting suffix trees have been around $m^2/10$. Now the number of dynamic programming columns that are generated during the depth-first traversal of $\mathcal{T}$ is exactly the length of T. Each column takes $\Theta(n)$ time to generate, and so we can state

Lemma 12.4.1. *The time used to generate the needed columns in the depth-first traversal is* $\Theta(n \times (\text{length of } T))$.

We must also account for the time and space used to write the rows and columns stored at each node of $\mathcal{T}$. In a suffix tree with m leaves there are $\Theta(m)$ internal nodes and a single row and column take at most $O(m+n)$ time and space to write. Therefore, the time and space needed for the row and column stores is $\Theta(m^2+nm) = \Theta(m^2)$. Hence, we have

Theorem 12.4.1. *The total time for the suffix-tree approach is* $\Theta(n \times (\text{length of } T) + m^2)$, *and the maximum space used is* $\Theta(m^2)$.

Reducing space

The size of the required output is $\Theta(m^2)$, since the problem calls for the edit distance between P and *each* of $\Theta(m^2)$ substrings of T, making the $\Theta(m^2)$ term in the time bound acceptable. On the other hand, the space used seems excessive since the space needed by the dynamic programming solution without using a suffix tree is just $\Theta(nm)$ and can be reduced to $O(m)$. We now modify the suffix-tree approach to also use only $O(n+m)$ space and the same time bounds as before.

First, there is no need to store the current column at each node v. When backing up from a child v' of v, we can use the current column at v' and the string labeling edge (v, v') to recompute the column for node v. This does, however, double the total time for computing the columns. There is also no need to keep the current row n at each node v. Instead, only $O(n)$ space is needed for row entries. The key idea is that the current table is expanded columnwise, so if the string-depth of v' is j and the string-depth of v' is $j+d$, then the row n stored at v and v' would be identical for the first j entries. We leave it as an exercise to work out the details. In summary, we have

Theorem 12.4.2. *The hybrid suffix-tree/dynamic programming approach to the P-against-all problem can be implemented to run in* $\Theta[n(\text{length of } \mathcal{T}) + m^2]$ *time and* $O(n+m)$ *space.*

The above time and space bounds should be compared to the $\Theta(nm^2)$ time and $O(n+m)$ space bounds that result from a straightforward application of dynamic programming. The effectiveness in practice of this method depends on the length of $\mathcal{T}$ for realistic strings. It is known that for random strings, the length of $\mathcal{T}$ is $\Theta(m^2)$, making the method unattractive. (For random strings, the suffix tree is bushy for string-depths of $\log_\sigma m$ or less, where σ is the size of the alphabet. But beyond that depth, the suffix tree becomes very sparse, since the probability is very low that a substring of length greater than $\log_\sigma m$ occurs more than once in the string.) However, strings with more structured repetitions (as occur in DNA) should give rise to suffix trees with lengths that are small enough to make this method useful. We examined this question empirically for DNA strings up to one million characters, and the lengths of the resulting suffix trees were around $m^2/10$.

12.4.2. The (threshold) all-against-all problem

Now we consider a more ambitious problem: Given strings P and T, find every pair of substrings where the edit distance is below a fixed threshold d. Computations of this type have been conducted when P and T are both equal to the combined set of protein strings in the database Swiss-Prot [183]. The importance of this kind of large-scale computation and the way in which its results are used are discussed in [57]. The way suffix trees are used to accelerate the computation is discussed in [34].

Since P and T have respective lengths of n and m, the full all-against-all problem (with threshold ∞) calls for the computation of n^2m^2 pieces of output. Hence no method for this problem can run faster than $\Theta(n^2m^2)$ time. Moreover, that time bound is easily achieved: Pick a pair of starting positions in P and T (in nm possible ways), and for each choice of starting positions i, j fill in the dynamic programming table for the edit distance of $P[i..n]$ and $T[j..m]$ (in $O(nm)$-time). For any choice of i and j, the entries in the corresponding table give the edit distance for every pair of substrings that begin at position i in P and at position j in T. Thus, achieving the $O(n^2m^2)$ bound for the full all-against-all problem does not require suffix trees.

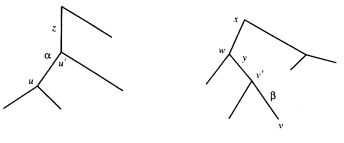
But the full all-against-all problem calls for an amount of output that is often excessive, and the output can be reduced by choosing a meaningful threshold. Or the criteria for reporting a substring pair might be a function of both length and edit distance. Whatever the specific reporting criteria, if it is no longer necessary to report the edit distance of every pair, it is no longer certain that $\Theta(n^2m^2)$ time is required. Here we develop a method whose worst-case running time is expressed as $O(C+R)$, where C is a computation time that may be less than $\Theta(n^2m^2)$ and R is the output size (i.e., the number of reported pairs of substrings). In this setting, the use of suffix trees may be quite valuable depending on the size of the output and the amount of repetition in the two strings.

An $O(C+R)$-time method

The method uses a suffix tree $\mathcal{T}_P$ for string P and a suffix tree $\mathcal{T}_T$ for string T. The worst-case time for the method will be shown to be $O(C+R)$, where C is the length of $\mathcal{T}_P$ times the length of $\mathcal{T}_T$ *independent of whatever the output criteria are*, and R is the size of the output. (The definition of the length of a suffix tree is found in Section 12.4.1.) That is, the method will compute certain dynamic programming cell values, which will be the same no matter what the output criteria are, and then when a cell value satisfies the particular output criteria, the algorithm will collect the relevant substrings associated with that cell. Hence our description of the method holds for the full all-against-all problem, the threshold version of the problem, or any other version with different reporting criteria.

To start, recall that each node in $\mathcal{T}_P$ represents a substring of P and that every substring of P is a prefix of a substring represented by a node of $\mathcal{T}_P$. In particular, each suffix of P is represented by a leaf of $\mathcal{T}_P$. The same is true of T and $\mathcal{T}_T$.

Definition The dynamic programming table for a pair of nodes (u, v), from $\mathcal{T}_P$ and $\mathcal{T}_T$, respectively, is defined as the dynamic programming table for the edit distance between the string represented by node u and the string represented by node v.



suffix tree for P suffix tree for T

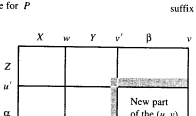


Figure 12.13: The dynamic programming table for (u,v) is shown below the suffix trees for P and T. The string on the path to node u is $Z\alpha$ and the string to node v is $XY\beta$. Every cell in the (u,v) table, except any in the lower right rectangle, is also in the (u,v'), (u',v), or (u',v') tables. The new part of the (u,v) table can be computed from the shaded entries and substrings α and β. The shaded entries contain exactly one entry from the (u',v') table; $|\alpha|$ entries from the last column in the (u,v') table; and $|\beta|$ entries from the last row in the (u',v) table.

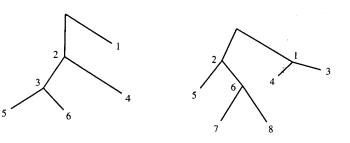
The threshold all-against-all problem could be solved (ignoring time) by computing the dynamic programming table for each pair of leaves, one from each tree, and then examining every entry in each of those tables. Hence it certainly would be solved by computing the dynamic programming table for each pair of nodes and then examining each entry in those tables. This is essentially what we will do, but we proceed in a way that avoids redundant computation and examination. The following lemma gives the key observation.

Lemma 12.4.2. *Let u' be the parent of node u in $\mathcal{T}_P$ and let α be the string labeling the edge between them. Similarly, let v' be the parent of v in $\mathcal{T}_T$ and let β be the string labeling the edge between them. Then, all but the bottom right $|\alpha||\beta|$ entries in the dynamic programming table for the pair (u,v) appear in one of the tables for (u',v'), (u',v), or (u,v'). Moreover, that bottom right part of the (u,v) table can be obtained from the other three tables in $O(|\alpha||\beta|)$ time. (See Figure 12.13.)*

The proof of this lemma is immediate from the definitions and the edit distance recurrences.

The computation for the new part of the (u,v) table produces an $|\alpha|$ by $|\beta|$ rectangular subtable that forms the lower right section of the (u,v) table. In the algorithm to be developed below, we will store and associate with each node pair (u,v) the last column and the last row of this $|\alpha|$ by $|\beta|$ subtable.

We can now fully describe the algorithm.



suffix tree for P suffix tree for T

Figure 12.14: The suffix trees for P and T with nodes numbered by string-depth. Note that these numbers are not the standard suffix position numbers that label the leaves. The ordered list of node pairs begins (1,1),(1,2),(1,3) . . . and ends with (6,8).

Details of the algorithm

First, number the nonroot nodes of $\mathcal{T}_P$ according to string-depth, with smaller string-depth first.[3] Separately, number the nodes of $\mathcal{T}_T$ according to string-depth. Then form a list L of all pairs of node numbers, one from each tree, in lexicographic order. Hence, pair (u,v) appears before pair (p,q) in the list if and only if u is less than p, or if u is equal to p and v is less than q. (See Figure 12.14). It follows that if u' is the parent of u in $\mathcal{T}_P$ and v' is the parent of v in $\mathcal{T}_T$, then (u',v') appears before (u,v).

Next, process each pair of nodes (u,v) in the order that it appears in L. Assume again that u' is the parent of u, that v' is the parent of v, and that the labels on the respective edges are α and β. To process a node pair (u,v), retrieve the value in the single lower right cell from the stored part of the (u',v') table; retrieve the column stored with the pair (u,v'), and retrieve the row stored with the pair (u',v). These three pairs of nodes have already been processed, due to the lexicographic ordering of the list. From those retrieved values, and from the substrings α and β, compute the new $|\alpha|$ by $|\beta|$ subtable completing the (u,v) table. Store with pair (u,v) the last row and column of newly computed subtable.

Now suppose cell (i,j) is in the new $|\alpha|$ by $|\beta|$ subtable, and its value satisfies the output criteria. The algorithm must find and output all locations of the two substrings specified by (i,j). As usual, a depth-first traversal to the leaves below u and v will then find all the starting positions of those strings. The length of the strings is determined by i and j. Hence, when it is required to output pairs of substrings that satisfy the reporting criteria, the time to collect the pairs is just proportional to the number of them.

Correctness and time analysis

The correctness of the method follows from the fact that at the highest level of description, the method computes the edit distance for every pair of substrings, one from each string. It does this by generating and examining every cell in the dynamic programming table for every pair of substrings (although it avoids redundant examinations). The only subtle point is that the method generates and examines the cells in each table in an incremental manner to exploit the commonalities between substrings, and hence it avoids regenerating and reexamining any cell that is part of more than one table. Further, when the method finds a cell satisfying the reporting criteria (a function of value and length), it can find all

[3] Actually, any topological numbering will do, but string-depth has some advantages when heuristic accelerations are added.

of substrings specified by that cell using a traversal to a subset of leaves in the trees. A formal proof of correctness is left to the reader as an exercise.

ime analysis, recall that the length of $\mathcal{T}_P$ is the sum of lengths of all the edge $\mathcal{T}_P$. If P has length n, then the length of $\mathcal{T}_P$ ranges between n and $n^2/2$, depending on how repetitive P is. The length of $\mathcal{T}_T$ is similarly defined and ranges between m and $m^2/2$, where m is the length of T.

Lemma 12.4.3. *The time used by the algorithm for all the needed dynamic programming computations and cell examinations is proportional to the product of the length of $\mathcal{T}_P$ and the length of $\mathcal{T}_T$. Hence that time, defined as C, ranges between nm and n^2m^2.*

PROOF In the algorithm, each pair of nodes is processed exactly once. At the point a pair (u, v) is processed, the algorithm spends $O(|\alpha||\beta|)$ time to compute a subtable and examine it, where α and β are the labels on the edges into u and v, respectively. Each edge-label in $\mathcal{T}_P$ therefore forms exactly one dynamic programming table with each of the edge-labels in $\mathcal{T}_T$. The time to build those tables is $|\alpha|$(length of $\mathcal{T}_T$). Summing over all edges in $\mathcal{T}_P$ gives the claimed time bound. □

The above lemma counts all the time used in the algorithm except the time used to collect and report pairs of substrings (by their starting position, length, and edit distance). But since the algorithm collects substrings when it sees a cell value that satisfies the reporting criteria, the time devoted to output is just the time needed to traverse the tree to collect output pairs. We have already seen that this time is proportional to the number of pairs collected, R. Hence, we have

Theorem 12.4.3. *The complete time for the algorithm is $O(C + R)$.*

How effective is the suffix tree approach?

As in the P-against-all problem, the effectiveness of this method in practice depends on the lengths of $\mathcal{T}_P$ and $\mathcal{T}_T$. Clearly, the product of those lengths, C, falls as P and T increase in repetitiveness. We have built a suffix tree for DNA strings of total length around one million bases and have observed that the tree length is around one tenth of the maximum possible. In that case, C is around $n^2m^2/100$, so all else being equal (which is unrealistic), standard dynamic programming for the all-against-all problem should run about one hundred times slower than the hybrid dynamic programming approach.

A vastly larger "all-against-all" computation on amino acid strings was reported in [183]. Although their description is very vague, they essentially used the suffix tree approach described here, computing similarity instead of edit distance. But, rather than a hundred-fold speedup, they claim to have achieved nearly a million-fold speedup over standard dynamic programming.[4] That level of speedup is not supported by theoretical considerations (recall that for a random string S of length m, a substring of length greater than $\log_4 m$ is very unlikely to occur in S more than once). Nor is it supported by the experiments we have done. The explanation may be the incorporation of an early stopping rule described in [183] only by the vague statement "Time is saved because the matching of patricia[5] subtrees is aborted when the score falls below a liberally chosen similarity limit". That rule is apparently very effective in reducing running time, but without a clearer description of it we cannot define precisely what specific all-against-all problem was solved.

[4] They finish a computation in 405 cpu days that they claim would otherwise have taken more than a million cpu years without the use of suffix trees.

[5] A patricia tree is a variant of a suffix tree.

12.5. A faster (combinatorial) algorithm for longest common subsequence

The longest common subsequence problem (*lcs*) is a special case of general weighted alignment or edit distance, and it can be solved in $\Theta(nm)$ time either by applying those general methods or with more direct recurrences (Exercise 16 of Chapter 11). However, the *lcs* problem plays a special role in the field of string algorithms and merits additional discussion. This is partly for historical reasons (many string and alignment ideas were first worked out for the special case of *lcs*) and partly because *lcs* often seems to capture the desired relationship between the strings of interest.

In this section we present an alternative (combinatorial) method for *lcs* that is *not* based on dynamic programming. For two strings of lengths n and $m > n$, the method runs in $O(r \log n)$ worst-case time, where r is a parameter that is typically small enough to make this bound attractive compared to $\Theta(nm)$. The main idea is to reduce the *lcs* problem to a simpler sounding problem, the *longest increasing subsequence problem* (*lis*). The method can also be adapted to compute the length of the *lcs* in $O(r \log n)$ time, using only linear space, without the need for Hirschberg's method. That will be considered in Exercise 23.

12.5.1. Longest increasing subsequence

Definition Let Π be a list of n integers, not necessarily distinct. An *increasing subsequence* of Π is a subsequence of Π whose values *strictly* increase from left to right.

For example, if $\Pi = 5, 3, 4, 9, 6, 2, 1, 8, 7, 10$ then {3, 4, 6, 8, 10} and {5, 9, 10} are both increasing subsequences in Π. (Recall the distinction between subsequences and substrings.) We are interested in the problem of computing a *longest increasing subsequence* in Π. The method we develop here will later be used to solve the problem of finding the *longest common subsequence* of two (or more) strings.

Definition A *decreasing subsequence* of Π is a subsequence of Π where the numbers are *nonincreasing* from left to right.

For example, under this definition, {8, 5, 5, 3, 1, 1} is a decreasing subsequence in the sequence 4, 8, 3, 9, 5, 2, 5, 3, 10, 1, 9, 1, 6. Note the asymmetry in the definitions of *increasing* and *decreasing* subsequences. The term "decreasing" is slightly misleading. Although "nonincreasing" is more precise, it is too clumsy a term to use in high repetition.

Definition A *cover* of Π is a set of decreasing subsequences of Π that contain all the numbers of Π.

For example, {5, 3, 2, 1}; {4}; {9, 6}; {8, 7}; {10} is a cover of $\Pi = 5, 3, 4, 9, 6, 2, 1, 8, 7, 10$. It consists of five decreasing subsequences, two of which contain only a single number.

Definition The *size* of the cover is the number of decreasing subsequences in it, and a *smallest* cover is a cover with minimum size among all covers.

We will develop an $O(n \log n)$-time method that simultaneously constructs a longest increasing subsequence (*lis*) and a smallest cover of Π. The following lemma is the key.

Figure 12.15: Decreasing cover of {5, 3, 4, 9, 6, 2, 1, 8, 7, 10}

Lemma 12.5.1. *If I is an increasing subsequence of Π with length equal to the size of a cover of Π, call it C, then I is a longest increasing subsequence of Π and C is a smallest cover of Π.*

PROOF No increasing subsequence of Π can contain more than one number contained in any decreasing subsequence of Π, since the numbers in an increasing subsequence strictly increase left to right, whereas the numbers in a decreasing subsequence are nonincreasing left to right. Hence no increasing subsequence of Π can have length greater than the size of any cover of Π.

Now assume that the length of I is equal to the size of C. This implies that I is a longest increasing subsequence of Π because no other increasing subsequence can be longer than the size of C. Conversely, C must be a smallest cover of Π, for if there were a smaller cover C' then I would be longer than the size of C', which is impossible. Hence, if the length of I equals the size of C, then I is a longest increasing subsequence and C is a smallest cover. □

Lemma 12.5.1 is the basis of a method to find a longest increasing subsequence and a smallest cover of Π. The idea is to decompose Π into a cover C such that there is an increasing subsequence I containing exactly one number from each decreasing subsequence in C. Without concern for efficiency, a cover of Π can be built in the following straightforward way:

Naive cover algorithm Starting from the left of Π, examine each successive number in Π and place it at the end of the first (left-most) decreasing subsequence that it can extend. If there are no decreasing subsequences it can extend, then start a new (decreasing) subsequence to the right of all the existing decreasing subsequences.

To elaborate, if x denotes the current number from Π being examined, then x extends a subsequence i if x is smaller than or equal to the current number at the end of subsequence i, and if x is strictly larger than the last number of each subsequence to the left of i.

For example, with Π as before the first two numbers examined are put into a decreasing subsequence {5, 3}. Then the number 4 is examined, which is in position 3 of Π. Number 4 cannot be placed at the end of the first subsequence because 4 is larger than 3. So 4 begins a new subsequence of its own to the right of the first subsequence. Next, the number 9 is considered and since it cannot be added to the end of either subsequence {5,3} or 4, it begins a third subsequence. Next, 6 is considered; it can be added to 9 but not to the end of any of the two subsequences to the left of 9. The final cover of Π produced by the algorithm is shown in Figure 12.15, where each subsequence runs vertically.

Clearly, this algorithm produces a cover of Π, which we call the *greedy cover*. To see whether a number x can be added to any particular decreasing subsequence, we only have to compare x to the number, say y, currently at the end of the subsequence – x can be added if and only if $x \leq y$. Hence if there are k subsequences at the time x is considered, then the time to add x to the correct subsequence is $O(k)$. Since $k \leq n$, we have the following:

Lemma 12.5.2. *The greedy cover of Π can be built in $O(n^2)$ time.*

We will shortly see how to reduce the time needed to find the greedy cover to $O(n \log n)$, but we first show that the greedy cover is a smallest cover of Π and that a longest increasing subsequence can easily be extracted from it.

Lemma 12.5.3. *There is an increasing subsequence I of Π containing exactly one number from each decreasing subsequence in the greedy cover C. Hence I is the longest possible, and C is the smallest possible.*

PROOF Let x be an arbitrary number placed into decreasing subsequence $i > 1$ (counting from the left) by the greedy algorithm. At the time x was considered, the last number y of subsequence $i - 1$ must have been smaller than x. Also, since y was placed before x was, y appears before x in Π, and $\{y, x\}$ forms an increasing subsequence in Π. Since x was arbitrary, the same argument applies to y, and if $i - 1 > 1$ then there must be a number z in subsequence $i - 2$ such that $z < y$ and z appears before y in Π. Repeating this argument until the first subsequence is reached, we conclude that there is an increasing subsequence in Π containing one number from each of the first i subsequences in the greedy cover and ending with x. Choosing x to be any number in the last decreasing subsequence proves the lemma. □

Algorithmically, we can find a longest increasing subsequence *given* the greedy cover as follows:

Longest increasing subsequence algorithm

begin

0. Set i to be the number of subsequences in the greedy cover. Set I to the empty list; pick any number x in subsequence i and place it on the front of list I.

1. While $i > 1$ do
begin

2. Scanning down from the *top* of subsequence $i - 1$, find the first number y that is smaller than x.

3. Set x to y and i to $i - 1$.

4. Place x on the front of list I.
end

end.

Since no number is examined twice during this algorithm, a longest increasing subsequence can be found in $O(n)$ time given the greedy cover.

An alternate approach is to use pointers. As the greedy cover is being constructed, whenever a number x is added to subsequence i, connect a pointer from x to the number at the current end of subsequence $i - 1$. After the greedy algorithm finishes, pick any number in the last decreasing subsequence and follow the unique path of pointers starting from it and ending at the first subsequence.

Faster construction of the greedy cover

Now we reduce the time to construct a greedy cover to $O(n \log n)$, reducing the overall running time to find a longest increasing subsequence to $O(n \log n)$ as well.

At any point during the running of the greedy cover algorithm, let L be the ordered list containing the last number of each of the decreasing subsequences built so far. That

is, the last number from any subsequence $i-1$ appears in L before the last number from subsequence i.

Lemma 12.5.4. *At any point in the execution of the algorithm, the list L is sorted in increasing order.*

PROOF Assume inductively that the lemma holds through iteration $k-1$. When examining the kth number in Π, call it x, suppose x is to be placed at the end of subsequence i. Let w be the current number at the end of subsequence $i-1$, let y be the current number at the end of subsequence i (if any), and let z be the number at the end of subsequence $i+1$ (if it exists). Then $w < x \leq y$ by the workings of the algorithm, and since $y < z$ by the inductive assumption, $x < z$ also. In summary, $w < x < z$, so the new subsequence L remains sorted. □

Note that L itself need not be (and generally will not be) an increasing subsequence of Π. Although $x < z$, x appears to the right of z in Π. Despite this, the fact that L is in sorted order means that we can use *binary search* to implement each iteration of the algorithm building the greedy cover. Each iteration k considers the kth number x in Π and the current list L to find the left-most number in L larger than x. Since L is in sorted order, this can be done in $O(\log n)$ time by binary search. The list Π has n numbers, so we have

Theorem 12.5.1. *The greedy cover can be constructed in $O(n \log n)$ time. A longest increasing subsequence and a smallest cover of Π can therefore be found in $O(n \log n)$ time.*

In fact, if p is the length of the *lis*, then it can be found in $O(n \log p)$ time.

12.5.2. Longest common subsequence reduces to longest increasing subsequence

We will now solve the *longest common subsequence problem* for a pair of strings, using the method for finding a longest increasing subsequence in a list of integers.

Definition Given strings S_1 and S_2 (of length m and n, respectively) over an alphabet Σ, let $r(i)$ be the number of times that the ith character of string S_1 appears in string S_2.

Definition Let r denote the sum $\sum_{i=1}^{m} r(i)$.

For example, suppose we are using the normal English alphabet; when $S_1 = abacx$ and $S_2 = baabca$ then $r(1) = 3$, $r(2) = 2$, $r(3) = 3$, $r(4) = 1$, and $r(5) = 0$, so $r = 9$. Clearly, for any two strings, r will fall in the range 0 to nm. We will solve the *lcs* problem in $O(r \log n)$ time (where $n \leq m$), which is inferior to $O(nm)$ when the r is large. However, r is often substantially smaller than nm, depending on the alphabet Σ. We will discuss this more fully later.

The reduction

For each alphabet character x that occurs at least once in S_1, create a list of the positions where character x occurs in string S_2; write this list in *decreasing* order. Two distinct alphabet characters will have totally disjoint lists. In the above example ($S_1 = abacx$ and $S_2 = baabca$) the list for character a is 6, 3, 2 and the list for b is 4, 1.

Now create a list called $\Pi(S_1, S_2)$ of length r, in which each character *instance* in S_1 is replaced with the associated list for that character. That is, for each position i in S_1, insert the list associated with the character $S_1(i)$. For example, list $\Pi(S_1, S_2)$ for the above two strings is 6, 3, 2, 4, 1, 6, 3, 2, 5.

To understand the importance of $\Pi(S_1, S_2)$, we examine what an increasing subsequence in that list means in terms of the original strings.

Theorem 12.5.2. *Every increasing subsequence I in $\Pi(S_1, S_2)$ specifies an equal length common subsequence of S_1 and S_2 and vice versa. Thus a longest common subsequence of S_1 and S_2 corresponds to a longest increasing subsequence in the list $\Pi(S_1, S_2)$.*

PROOF First, given an increasing subsequence I of $\Pi(S_1, S_2)$, we can create a string S and show that S is a subsequence of both S_1 and S_2. String S is successively built up during a left-to-right scan of I. During this scan, also construct two lists of indices specifying a subsequence of S_1 and a subsequence of S_2. In detail, if number j is encountered in I during the scan, and number j is contained in the sublist contributed by character i of S_1, then add character $S_1(i)$ to the right end of S, add number i to the right end of the first index list, and add j to the right end of the other index list.

For example, consider $I = 3, 4, 5$ in the running example. The number 3 comes from the sublist for character 1 of S_1, the number 4 comes from the sublist for character 2, and the number 5 comes from the sublist for character 4. So the string S is *abc*. That string is a subsequence of S_1 found in positions 1, 2, 4 and is a subsequence of S_2 found in positions 3, 4, 5.

The list $\Pi(S_1, S_2)$ contains one sublist for every position in S_1, and each such sublist in $\Pi(S_1, S_2)$ is in decreasing order. So at most one number from any sublist is in I and any position in S_1 contributes at most one character to S. Further, the m lists are arranged left to right corresponding to the order of the characters in S_1, so S is certainly a subsequence of S_1. The numbers in I strictly increase and correspond to positions in S_2, so S is also a subsequence of S_2.

In summary, we have proven that every increasing subsequence in $\Pi(S_1, S_2)$ can be used to create an equal length common subsequence in S_1 and S_2. The converse argument, that a common subsequence yields an increasing subsequence, is very similar and is left as an exercise. □

$\Pi(S_1, S_2)$ is a list of r integers, and the longest increasing subsequence problem can be solved in $O(r \log l)$ time on an r-length list when the longest increasing subsequence is of length l. If $n \leq m$ then $l \leq n$, yielding the following theorem:

Theorem 12.5.3. *The longest common subsequence problem can be solved in $O(r \log n)$ time.*

The $O(r \log n)$ result for *lcs* was first obtained by Hunt and Szymanski [238]. Their algorithm is superficially very different than the one above, but in retrospect one can see similar ideas embodied in it. The relationship between the *lcs* and *lis* problems was partly identified by Apostolico and Guerra [25, 27] and made explicit by Jacobson and Vo [244] and independently by Pevzner and Waterman [370].

The *lcs* method based on *lis* is an example of what is called *sparse dynamic programming*, where the input is a relatively sparse set of pairs that are permitted to align. This approach, and in fact the solution technique discussed here, has been very extensively generalized by a number of people and appears in detail in [137] and [138].

12.5.3. How good is the method

How good is the *lcs* method based on the *lis* compared to the original $\Theta(nm)$-time dynamic programming approach? It depends on the size of r. Let σ denote the size of the alphabet Σ. A very naive analysis would say that r can be expected to be about nm/σ. This assumes that each character in Σ appears with equal probability and hence is expected to appear n/σ times in the short string. That means that $r_i = n/\sigma$ for each i. The long string has length m, so r is expected to be nm/σ. But of course, equal distribution of characters is not really typical, and the value of r is then highly dependent on the specific strings.

For the Roman alphabet with capital letters, digits, and punctuation marks added, σ is around 100, but the assumption of equal distribution is clearly flawed. Still, one can ask whether $(nm/100)\log n$ looks attractive compared to nm. For such alphabets, the speedup doesn't look so compelling, although the method retains its simplicity and space efficiency. Thus for typical English text, the *lis*-based approach may not be much superior to the dynamic programming approach. However, in many applications, the "alphabet" size is quite large and grows with the size of the text.[6] This is true, for example, in the unix utility *diff* where each line in the text is considered as a character in the "alphabet" used for the *lcs* computation. In certain applications in molecular biology the alphabet consists of patterns or substrings, rather than the four-character alphabet of DNA or the twenty-character alphabet of protein. These substrings might be genes, exons, or restriction enzyme recognition sequences. In those cases, the alphabet size is large compared to the string size, so r is small and $r \log n$ is quite attractive compared to nm.

Constrained *lcs*

The *lcs* method based on *lis* has another advantage over the standard dynamic programming approach. In some applications there are additional constraints imposed on which pairs of positions are permitted to align in the *lcs*. That is, in addition to the constraint that position i in S_1 can align with position j in S_2 only if $S_1(i) = S_2(j)$, some additional constraints may apply. The reduction of *lcs* to *lis* can be easily modified to incorporate these additional constraints, and we leave the details to the reader. The effect is to reduce the size of r and consequently to speed up the entire *lcs* computation. This is another example and variant of sparse dynamic programming.

12.5.4. The *lcs* of more than two strings

One of the nice features of the *lcs* method based on *lis* is that it easily generalizes to the *lcs* problem for more than two strings. That problem is a special case of *multiple sequence alignment*, a crucial problem in computational molecular biology that we will more fully discuss in Chapter 14. The generalization from two to many strings will be presented here for three strings, S_1, S_2, and S_3.

The idea is to again reduce the *lcs* problem to the *lis* problem. As before, we start by creating a list for each character x in S_1. In particular, the list for x will contain pairs of integers, each pair containing a position in S_2 where x occurs and a position in S_3 where x occurs. Further, the list for character x will be ordered so that the pairs in the list are in *lexically decreasing* order. That is, if pair (i, j) appears before pair (i', j') in the list for x, then either $i > i'$ or $i = i'$ and $j > j'$. For example, if $S_1 =$ *abacx* and $S_2 =$ *baabca* (as above) and $S_3 =$ *babbac*, then the list for character a is (6, 5), (6, 2), (3, 5). (3, 2), (2, 5), (2, 2).

The lists for each character are again concatenated in the order that the characters appear in string S_1, forming the sequence of pairs $\Pi(S_1, S_2, S_3)$. We define an increasing subsequence in $\Pi(S_1, S_2, S_3)$ to be a subsequence of pairs such that the first numbers in each pair form an increasing subsequence of integers, and the second numbers in each pair also form an increasing subsequence of integers. We can easily modify the greedy cover algorithm to find a longest increasing subsequence of pairs under this definition. This increasing subsequence is used as follows.

Theorem 12.5.4. *Every increasing subsequence in $\Pi(S_1, S_2, S_3)$ specifies an equal length common subsequence of S_1, S_2, S_3 and vice versa. Therefore, a longest common subsequence of S_1, S_2, S_3 corresponds to a longest increasing subsequence in $\Pi(S_1, S_2, S_3)$.*

The proof of this theorem is similar to the case of two strings and is left as an exercise. Adaptation of the greedy cover algorithm and its time analysis for the case of three strings is also left to the reader. Extension to more than three strings is immediate. The combinatorial approach to computing *lcs* also has a nice space-efficiency feature that we will explore in the exercises.

[6] This is one of the few places in the book where we deviate from the standard assumption that the alphabet is fixed.

12.6. Convex gap weights

Overwhelmingly, the affine gap weight model is the model most commonly used by molecular biologists today. This is particularly true for aligning amino acid sequences. However, a richer gap model, the *convex* gap weight, was proposed and studied by Waterman in 1984 [466], and has been more extensively examined since then. In discussing the common use of the affine gap weight, Benner, Cohen and Gonnet state "There is no justification either theoretical or empirical for this treatment" [183] and forcefully argue that "a non-linear gap penalty is the only one that is grounded in empirical data" [57]. They propose [57] that to align two protein sequences that are d PAM units diverged (see Section 15.7.2), a gap of length q should be given the weight:

$$35.03 - 6.88 \log_{10} d + 17.02 \log_{10} q$$

Under this weighting model, the cost to initiate a gap is at most 35.03, and declines with increasing evolutionary (PAM) distance between the two sequences. In addition to this initiation weight, the function adds $17.02 \log_{10} q$ for the actual length, q, of the gap.

It is hard to believe that a function this precise could be correct, but the key point is that, for a fixed PAM distance, the proposed gap weight is a *convex* function of its length.[7]

The alignment problem with convex gap weights is more difficult to solve than with affine gap weights, but it is not as difficult as the problem with arbitrary gap weights. In this section we develop a practical algorithm to optimally align two strings of lengths n and $m > n$, when the gap weights are specified by a *convex* function of the gap length. The algorithm runs in $O(nm \log m)$ time, in contrast to the $O(nm)$-time bound for affine gap weights and the $O(nm^2)$ time for arbitrary gap weights. The speedup for the convex case was established by Miller and Myers [322] and independently by Galil and Giancarlo

[7] Unfortunately, there is no standard agreement on terminology, and some of the papers refer to the model as the "convex" gap weight model, while others call it the "concave" gap model. In this book, a convex function is one with a negative or zero second derivative, and a concave function is one with a positive second derivative.

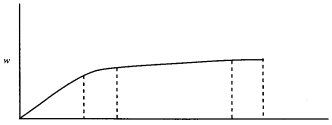


Figure 12.16: A convex function w.

[170]. However, the solution in the second paper is given in terms of edit distance rather than similarity. Similarity is often more useful than edit distance because it can be used to handle the extremely important case of local comparison. Hence we will discuss convex gap weights in terms of similarity (maximum weighted alignment) and leave it to the reader to derive the analogous algorithms for computing edit distance with convex gap weights. More advanced results on alignment with convex or concave gap weights appear in [136], [138], and [276].

Recall from the discussion of arbitrary gap weights that $w(q)$ is the weight given to a gap of length q. That gap then contributes a penalty of $-w(q)$ to the total weight of the alignment.

Definition Assume that $w(q)$ is a nonnegative function of q. Then $w(q)$ is *convex* if and only if $w(q+1) - w(q) \leq w(q) - w(q-1)$ for every q.

That is, as a gap length increases, the additional penalty contributed by the gap decreases for each additional unit of the gap. It follows that $w(q+d) - w(q) \geq w(q'+d) - w(q')$ for $q < q'$ and any fixed d (see Figure 12.16). Note that the function w can have regions of both positive and negative slope, although any region of positive slope must be to the left of the region of negative slope. Note that the definition allows $w(q)$ to become negative for large enough n and m. At that point, $-w(q)$ becomes positive, which is probably not desirable. Hence, gap weight functions with negative slope must be used with care.

The convex gap weight was introduced in [466] with the suggestion that mutational events that insert or delete varying length blocks of DNA can be more meaningfully modeled by convex gap weights, compared to affine or constant gap weights. A convex gap penalty allows the modeler more specificity in reflecting the cost or probability of different gap lengths, and yet it can be more efficiently handled than arbitrary gap weights. One particular convex function that is appealing in this context is the *log* function, although it is not clear which base of the logarithm might be most meaningful.

The argument for or against convex gap weights is still open, and the affine gap model remains dominant in practice. Still, even if the convex gap model never becomes popular in molecular biology it could well find application elsewhere. Furthermore, the algorithm for alignment with convex gaps is of interest in itself, as a representative of a number of related algorithms in the general area of "sparse dynamic programming".

Speeding up the general recurrences

To solve the convex gap weight case we use the same dynamic programming recurrences developed for arbitrary gap weights (page 242), but reduce the time needed to evaluate

those recurrences. For convenience, we restate the general recurrences for arbitrary gap weights.

$$V(i,j) = \max[E(i,j), F(i,j), G(i,j)],$$
$$G(i,j) = V(i-1, j-1) + s(S_1(i), S_2(j)),$$
$$E(i,j) = \max_{0 \leq k \leq j-1} [V(i,k) - w(j-k)],$$
$$F(i,j) = \max_{0 \leq l \leq i-1} [V(l,j) - w(i-l)],$$
$$V(i,0) = -w(i),$$
$$V(0,j) = -w(j),$$
$$E(i,0) = -w(i),$$
$$F(0,j) = -w(j).$$

$G(i,j)$ is undefined when i or j is zero.

Even with arbitrary gap weights, the work required by the first and second recurrences is $O(m)$ per row, which is within our desired time bound. It is the recurrences for $E(i,j)$ and $F(i,j)$ that respectively require $\Theta(m^2)$ time per *row* and $\Theta(n^2)$ time per *column* when the function w is arbitrary. Hence, it is the evaluation of E and F for any given row or column that will be improved in the case where w is convex. We will focus on the computation of E for a single row. The computation of F and the associated time analysis for a single column is symmetric, with one caveat to be discussed later.

Simplifying notation

The value $E(i,j)$ depends on i only through the values $V(i,k)$ for $k < i$. Hence, in any fixed row, we can drop the reference to the row index i, simplifying the recurrence for E. That is, in any fixed row we define

$$E(j) = \max_{0 \leq k \leq j-1} [V(k) - w(j-k)].$$

Further, we introduce the following notation to simplify the recurrence:

$$Cand(k,j) = V(k) - w(j-k);$$

therefore,

$$E(j) = \max_{0 \leq k \leq j-1} Cand(k,j).$$

The term *Cand* stands for "candidate"; the meaning of this will become clear later.

12.6.1. Forward dynamic programming

It will be useful in the exposition to change the way we normally implement dynamic programming. Normally when setting the value $E(j)$, we would look *backwards* in the row to compare all the $Cand(k,j)$ values for $k < j$, taking the largest one to be the value $E(j)$. But an alternative *forward-looking* implementation is also possible and is more helpful in this exposition.[8]

[8] Gene Lawler pointed out that in some circles forward and backward implementations are referred to as "*push you – pull me*" dynamic programming. The reader may determine which term denotes forwards and which denotes backwards.

In the forward implementation, we first initialize a variable $\overline{E}(j')$ to $Cand(0, j')$ for each cell $j' > 0$ in the row. The E values are set left to right in the row, as in backward dynamic programming. However, to set the value of $E(j)$ (for any $j > 0$) the algorithm merely sets $E(j)$ to the current value of $\overline{E}(j)$, since every cell to the left of j will have contributed a candidate value to cell j. Then, before setting the value of $E(j+1)$, the algorithm traverses *forwards* in the row to set $\overline{E}(j')$ (for each $j' > j$) to be the maximum of the current $\overline{E}(j')$ and $Cand(j, j')$. To summarize, the forward implementation for a fixed row is:

Forward dynamic programming for a fixed row

```
For j := 1 to m do
begin
Ē(j) := Cand(0, j);
b(j) := 0
end;

For j := 1 to m do
begin
E(j) := Ē(j);
V(j) := max{G(j), E(j), F(j)};
{We assume, but do not show that F(j) and G(j)
have been computed for cell j in the row.}

For j' := j + 1 to m do {Loop 1}
    if Ē(j') < Cand(j, j') then
        begin
        Ē(j') := Cand(j, j');
        b(j') := j; {This sets a pointer from j' to j to be explained later.}
        end
end;
```

An alternative way to think about forward dynamic programming is to consider the weighted edit graph for the alignment problem (see Section 11.4). In that (acyclic) graph, the optimal path (shortest or longest distance, depending on the type of alignment being computed) from cell $(0, 0)$ to cell (n, m) specifies an optimal alignment. Hence algorithms that compute optimal distances in (acyclic) graphs can be used to compute optimal alignments, and distance algorithms (such as Dijkstra's algorithm for shortest distance) can be described as forward looking. When the correct distance $d(v)$ to a node v has been computed, and there is an edge from v to a node w whose correct distance is still unknown, the algorithm adds $d(v)$ to the distance on the edge (v, w) to obtain a candidate value for the correct distance to w. When the correct distances have been computed to all nodes with a direct edge to w, and each has contributed a candidate value for v, the correct distance to v is the best of those candidate values.

It should be clear that exactly the same arithmetic operations and comparisons are done in both backward and forward dynamic programming – the only difference is the order in which the operations take place. It follows that the forward algorithm correctly sets all the E values in a fixed row and still requires $\Theta(m^2)$ time per row. Thus forward dynamic programming is no faster than backwards dynamic programming, but the concept will help explain the speedup to come.

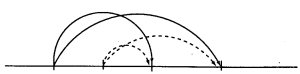


Figure 12.17: Graphical illustration of the *key observation*. Winning candidates are shown with a solid curve and losers with a dashed curve. If the candidate from j loses to the candidate from k at cell j', then the candidate from j will lose to the candidate from k at every cell j'' to the right of j'.

12.6.2. The basis of the speedup

At the point when $E(j)$ is set, call cell j the *current cell*. We interpret $Cand(j, j')$ as the "candidate value" for $E(j')$ that cell j "sends forward" to cell j'. When j is the current cell, it "sends forward" $m - j$ candidate values, one to each cell $j' > j$. Each such $Cand(j, j')$ value is compared to the current $\overline{E}(j')$; it either *wins* (when $Cand(j, j')$ is greater than $\overline{E}(j')$) or *loses* the comparison. The speedup works by identifying and eliminating large numbers of candidate values that have no chance of winning any comparison. In this way, the algorithm avoids a large number of useless comparisons. This approach is sometimes called a *candidate list* approach. The following is the key observation used to identify "loser" candidates:

Key observation Let j be the current cell. If $Cand(j, j') \leq \overline{E}(j')$ for *some* $j' > j$, then $Cand(j, j'') \leq \overline{E}(j'')$ for *every* $j'' > j'$. That is, "one strike and you're out".

Hence the current cell j need not send forward any candidate values to the right of the first cell $j' > j$ where $Cand(j, j')$ is less than or equal to $\overline{E}(j')$. This suggests the obvious practical speedup of stopping the loop labeled {Loop 1} in the Forward dynamic programming algorithm as soon as j's candidate loses. But this improvement does not lead directly to a better (worst-case) time bound. For that, we will have to use one more trick. But first, we prove the key observation with the following more precise lemma.

Lemma 12.6.1. *Let $k < j < j' < j''$ be any four cells in the same row. If $Cand(j, j') \leq Cand(k, j')$ then $Cand(j, j'') \leq Cand(k, j'')$. See Figure 12.17 for reference.*

PROOF $Cand(k, j') \geq Cand(j, j')$ implies that $V(k) - w(j' - k) \geq V(j) - w(j' - j)$, so $V(k) - V(j) \geq w(j' - k) - w(j' - j)$.

Trivially, $(j' - k) = (j' - j) + (j - k)$. Similarly, $(j'' - k) = (j'' - j) + (j - k)$. For future use, note that $(j' - k) < (j'' - k)$.

Now let q denote $(j' - j)$, let q' denote $(j'' - j)$, and let d denote $(j - k)$. Since $j' < j''$, then $q < q'$. By convexity, $w(q + d) - w(q) \geq w(q' + d) - w(q')$ (see Figure 12.16). Translating back, we have $w(j' - k) - w(j' - j) \geq w(j'' - k) - w(j'' - j)$. Combining this with the result in the first paragraph gives $V(k) - V(j) \geq w(j'' - k) - w(j'' - j)$, and rewriting gives $V(k) - w(j'' - k) \geq V(j) - w(j'' - j)$, i.e., $Cand(k, j'') \geq Cand(j, j'')$, as claimed. □

Lemma 12.6.1 immediately implies the key observation.

12.6.3. Cell pointers and row partition

Recall from the details of the forward dynamic programming algorithm that the algorithm maintains a variable $b(j')$ for each cell j'. This variable is a pointer to the left-most cell

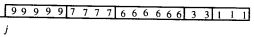


Figure 12.18: Partition of the cells $j+1$ through m into maximal blocks of consecutive cells such that all the cells in any block have the same b value. The common b value in any block is less than the common b value in the preceding block.

$k < j'$ that has contributed the best candidate yet seen for cell j'. Pointer $b(j')$ is updated every time the value of $\overline{E}(j')$ changes. The use of these pointers combined with the next lemma leads ultimately to the desired speedup.

Lemma 12.6.2. *Consider the point when j is the current cell, but before j sends forward any candidate values. At that point, $b(j') \geq b(j'+1)$ for every cell j' from $j+1$ to $m-1$.*

PROOF For notational simplicity, let $b(j') = k$ and $b(j'+1) = k'$. Then, by the selection of k, $Cand(k, j') \geq Cand(k', j')$. Now suppose $k < k'$. Then, by Lemma 12.6.1, $Cand(k, j'+1) \geq Cand(k', j'+1)$, in which case $b(j'+1)$ should be set to k, not k'. Hence $k \geq k'$ and the lemma is proved. □

The following corollary restates Lemma 12.6.2 in a more useful way.

Corollary 12.6.1. At the point that j is the current cell but before j sends forward any candidates, the values of the b pointers form a nonincreasing sequence from left to right. Therefore, cells j, $j+1$, $j+2$, ..., m are partitioned into maximal blocks of consecutive cells such that all b pointers in the block have the same value, and the pointer values decline in successive blocks.

Definition The partition of cells j through m referred to in Corollary 12.6.1 is called the current *block-partition*. See Figure 12.18.

Given Corollary 12.6.1, the algorithm doesn't need to explicitly maintain a b pointer for every cell but only record the common b pointer for each block. This fact will next be exploited to achieve the desired speedup.

Preparation for the speedup

Our goal is to reduce the time per row used in computing the E values from $\Theta(m^2)$ to $O(m \log m)$. The main work done in a row is to update the $\overline{E}$ values and to update the current block-partition with its associated pointers. We first focus on updating the block-partition and the b pointers; after that, the treatment of the $\overline{E}$ values will be easy. So for now, assume that all the $\overline{E}$ values are maintained for free.

Consider the point where j is the current cell, but before it sends forward any candidate values. After $E(j)$ (and $F(j)$ and then $V(j)$) have been computed, the algorithm must update the block-partition and the needed b pointers. To see the new idea, take the case of $j = 1$. At this point, there is only one block (containing cells 1 through m), with common b pointer set to cell zero (i.e., $b(j') = 0$ for each cell j' in the block). After $E(1)$ is set to $\overline{E}(1) = Cand(0, 1)$, any $\overline{E}(j')$ value that then changes will cause the block-partition to change as well. In particular, if $\overline{E}(j')$ changes, then $b(j')$ changes from zero to one. But since the b values in the new block-partition must be nonincreasing from left to right, there are only three possibilities for the new block-partition:[9]

- Cells 2 through m might remain in a single block with common pointer $b = 0$. By Lemma 12.6.1, this happens if and only if $Cand(1, 2) \leq \overline{E}(2)$.

[9] The $\overline{E}$ values in these three cases are the values before any $\overline{E}$ changes.

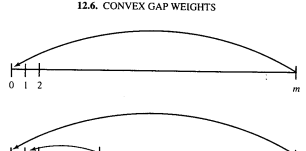


Figure 12.19: The three possible ways that the block partition changes after $E(1)$ is set. The curves with arrows represent the common pointer for the block and leave from the last entry in the block.

- Cells 2 through m might get divided into two blocks, where the common pointer for the first block is $b = 1$, and the common pointer for the second is $b = 0$. This happens (again by Lemma 12.6.1) if and only if for some $k < m$ $Cand(1, j') > \overline{E}(j')$ for j' from 2 to k and $Cand(1, j') \leq \overline{E}(j')$ for j' from $k+1$ to m.
- Cells 2 through m might remain in a single block, but now the common pointer b is set to 1. This happens if and only if $Cand(1, j') > \overline{E}(j')$ for j' from 2 to m.

Figure 12.19 illustrates the three possibilities.

Therefore, before making any changes to the $\overline{E}$ values, the new partition of the cells from 2 to m can be efficiently computed as follows: The algorithm first compares $\overline{E}(2)$ and $Cand(1, 2)$. If $\overline{E}(2) \geq Cand(1, 2)$ then all the cells to the right of 2 remain in a single block with common b pointer set to zero. However, if $\overline{E}(2) < Cand(1, 2)$ then the algorithm searches for the left-most cell $j' > 2$ such that $\overline{E}(j') \geq Cand(1, j')$. If j' is found, then cells 2 through $j'-1$ form a new block with common pointer to cell one, and the remaining cells form another block with common pointer to cell zero. If no j' is found, then all cells 2 through m remain in a single block, but the common pointer is changed to one.

Now for the punch line: By Corollary 12.6.1, this search for j' can be done by binary search. Hence only $O(\log m)$ *comparisons* are used in searching for j'. And, since we only record one b pointer per block, at most one pointer update is needed.

Now consider the general case of $j > 1$. Suppose that $E(j)$ has just been set and that the cells $j+1, \ldots, m$ are presently partitioned into r maximal blocks ending at cells $p_1 < p_2 < \cdots < p_r = m$. The block ending at p_i will be called the ith block. We use b_i to denote the common pointer for cells in block i. We assume that the algorithm has a list of the *end-of-block* positions $p_1 < p_2 < \cdots < p_r$ and a parallel list of common pointers $b_1 > b_2 > \cdots > b_r$.

After $E(j)$ is set, the new partition of cells $j+1$ through m is found in the following way: First, if $\overline{E}(j+1) \geq Cand(j, j+1)$ then, by Lemma 12.6.1, $\overline{E}(j') \geq Cand(j, j')$ for all $j' > j$, so the partition of cells greater than j remains unchanged. Otherwise (if $\overline{E}(j+1) < Cand(j, j+1)$), the algorithm successively compares $\overline{E}(p_i)$ to $Cand(j, p_i)$

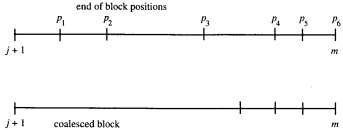


Figure 12.20: To update the block-partition the algorithm successively examines cell p_i to find the first index s where $\overline{E}(p_s) \geq Cand(j, p_s)$. In this figure, s is 4. Blocks 1 through $s-1 = 3$ coalesce into a single block with some initial part of block $s = 4$. Blocks to the right of s remain unchanged.

for i from 1 to r, until either the end-of-block list is exhausted, or until it finds the first index s with $\overline{E}(p_s) \geq Cand(j, p_s)$. In the first case, the cells $j+1, \ldots, m$ fall into a single block with common pointer to cell j. In the second case, the blocks $s+1$ through r remain unchanged, but all the blocks 1 through $s-1$ *coalesce* with some initial part (possibly all) of block s, forming one block with common pointer to cell j (see Figure 12.20). Note that every comparison but the last one results in two neighboring blocks coalescing into one.

Having found block s, the algorithm finds the proper place to split block s by doing binary search over the cells in the block. This is exactly as in the case already discussed for $j = 1$.

12.6.4. Final implementation details and time analysis

We have described above how to update the block-partition and the common b pointers, but that exposition uses $\overline{E}$ values that we assumed could be maintained for free. We now deal with that problem.

The key observation is that the algorithm retrieves $\overline{E}(j)$ only when j is the current cell and retrieves $\overline{E}(j')$ only when examining cell j' in the process of updating the block-partition. But the current cell j is always in the first block of the current block-partition (whose endpoint is denoted p_1), so $b(j) = b_1$, and $\overline{E}(j)$ equals $Cand(b_1, j)$, which can be computed in constant time when needed. In addition, when examining a cell j' in the process of updating the block-partition, the algorithm knows the block that j' falls into, say block i, and hence it knows b_i. Therefore, it can compute $\overline{E}(j')$ in constant time by computing $Cand(b_i, j')$. The result is that *no* explicit $\overline{E}$ values ever need to be stored. They are simply computed when needed. In a sense, they are only an expositional device. Moreover, the number of $\overline{E}$ values that need to be computed on the fly is proportional to the number of comparisons that the algorithm does to maintain the block-partition. These observations are summarized in the following:

Revised forward dynamic programming for a fixed row

Initialize the end-of-block list to contain the single number m.
Initialize the associated pointer list to contain the single number 0.

For $j := 1$ to m do
begin
  Set k to be the first pointer on the b-pointer list.
  $E(j) := Cand(k, j)$;
  $V(j) := \max\{G(j), E(j), F(j)\}$;
  {As before we assume that the needed F and G values have been computed.}

  {Now see how j's candidates change the block-partition.}
  Set j' equal to the first entry on the end-of-block list.

  {look for the first index s in the end-of-block list where j loses}
  If $Cand(b(j'), j+1) < Cand(j, j+1)$ then {j's candidate wins one}
  begin
    While
    The end-of-block list is not empty and $Cand(b(j'), j') < Cand(j, j')$ do
      begin
      remove the first entry on the end-of-block list,
      and remove the corresponding b-pointer
      If the end-of-block list is not empty then
      set j' to the new first entry on the end-of-block list.
      end;
    end {while};
  If the end-of-block list is empty then
  place m at the head of that list;
  Else {when the end-of-block list is not empty}
    begin
    Let p_s denote the first end-of-block entry.
    Using binary search over the cells in block s, find the
    right-most point p in that block such that $Cand(j, p) > Cand(b_s, p)$.
    Add p to the head of the end-of-block list;
    end;

  Add j to the head of the b pointer list.

  end;
end.

Time analysis

An $\overline{E}$ value is computed for the current cell, or when the algorithm does a comparison involved in maintaining the current block-partition. Hence the total time for the algorithm is proportional to the number of those comparisons. In iteration j, when j is the current cell, the comparisons are divided into those used to find block s and those used in the binary search to split block s. If the algorithm does $l > 2$ comparisons to find s in iteration j, then at least $l-1$ full blocks coalesce into a single block. The binary search then splits at most one block into two. Hence if, in iteration j, the algorithm does $l > 2$ comparisons to find s, then the total number of blocks decreases by at least $l-2$. If it does one or two comparisons, then the total number of blocks at most increases by one. Since the algorithm begins with a single block and there are m iterations, it follows that over the entire algorithm there can be at most $O(m)$ comparisons done to find every s, excluding the comparisons done during the binary searches. Clearly, the total number of comparisons used in the m binary searches is $O(m \log m)$. Hence we have

Theorem 12.6.1. *For any fixed row, all the $E(j)$ values can be computed in $O(m \log m)$ total time.*

The case of F values is essentially symmetric

A similar algorithm and analysis is used to compute the F values, except that for $F(i, j)$ the lists partition column j from cell i through n. There is, however, one point that might cause confusion: Although the analysis for F focuses on the work in a single column and is symmetric to the analysis for E in a single row, the computations of E and F are actually *interleaved* since, by the recurrences, each $V(i, j)$ value depends on both $E(i, j)$ and $F(i, j)$. Even though both the E values and the F values are computed rowwise (since V is computed rowwise), one row after another, $E(i, j)$ is computed just prior to the computation of $E(i, j + 1)$, while between the computation of $F(i, j)$ and $F(i + 1, j)$, $m - 1$ other F values will be computed ($m - j$ in row i and $j - 1$ in row $i + 1$). So although the analysis treats the work in a column as if it is done in one contiguous time interval, the algorithm actually breaks up the work in any given column.

Only $O(nm)$ total time is needed to compute the G values and to compute every $V(i, j)$ once $E(i, j)$ and $F(i, j)$ is known. In summary we have

Theorem 12.6.2. *When the gap weight w is a convex function of the gap length, an optimal alignment can be computed in $O(nm \log m)$ time, where $m > n$ are the lengths of the two strings.*

12.7. The Four-Russians speedup

In this section we will discuss an approach that leads both to a theoretical and to a practical speedup of many dynamic programming algorithms. The idea, comes from a paper [28] by four authors, Arlazarov, Dinic, Kronrod, and Faradzev, concerning boolean matrix multiplication. The general idea taken from this paper has come to be known in the West as the Four-Russians technique, even though only one of the authors is Russian.[10] The applications in the string domain are quite different from matrix multiplication, but the general idea suggested in [28] applies. We illustrate the idea with the specific problem of computing (unweighted) *edit distance*. This application was first worked out by Masek and Paterson [313] and was further discussed by those authors in [312]; many additional applications of the Four-Russians idea have been developed since then (for example [340]).

12.7.1. t-blocks

Definition A *t-block* is a t by t square in the dynamic programming table.

The rough idea of the Four-Russians method is to partition the dynamic programming table into *t-blocks* and compute the essential values in the table one t-block at a time, rather than one cell at a time. The goal is to spend only $O(t)$ time per block (rather than $\Theta(t^2)$ time), achieving a factor of t speedup over the standard dynamic programming solution. In the exposition given below, the partition will not be exactly achieved, since neighboring t-blocks will overlap somewhat. Still, the rough idea given here does capture the basic flavor and advantage of the method presented below. That method will compute the edit distance in $O(n^2/\log n)$ time, for two strings of length n (again assuming a fixed alphabet).

[10] This reflects our general level of ignorance about ethnicities in the then Soviet Union.

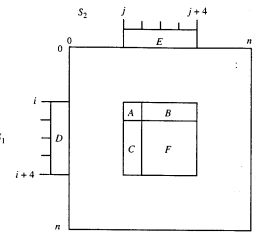


Figure 12.21: A single block with $t = 4$ drawn inside the full dynamic programming table. The distance values in the part of the block labeled F are determined by the values in the parts labeled A, B, and C together with the substrings of S_1 and S_2 in D and E. Note that A is the intersection of the first row and column of the block.

Consider the standard dynamic programming approach to computing the edit distance of two strings S_1 and S_2. The value $D(i, j)$ given to any cell (i, j), when i and j are both greater than 0, is determined by the values in its three neighboring cells, $(i - 1, j - 1)$, $(i - 1, j)$, and $(i, j - 1)$, and by the characters in positions i and j of the two strings. By extension, the values given to the cells in an entire t-block, with upper left-hand corner at position (i, j) say, are determined by the values in the first row and column of the t-block together with the substrings $S_1[i..i + t - 1]$ and $S_2[j..j + t - 1]$ (see Figure 12.21). Another way to state this observation is the following:

Lemma 12.7.1. *The distance values in a t-block starting in position (i, j) are a function of the values in its first row and column and the substrings $S_1[i..i+t-1]$ and $S_2[j..j+t-1]$.*

Definition Given Lemma 12.7.1, and using the notation shown in Figure 12.21, we define the *block function* as the function from the five inputs (A, B, C, D, E) to the output F.

It follows that the values in the last row and column of a t-block are also a function of the inputs (A, B, C, D, E). We call the function from those inputs to the values in the last row and column of a t-block, the *restricted block function*.

Notice that the total size of the input and the size of the output of the restricted block function is $O(t)$.

Computing edit distance with the restricted block function

By Lemma 12.7.1, the edit distance between S_1 and S_2 can be computed using the restricted block function. For simplicity, suppose that S_1 and S_2 are both of length $n = k(t - 1)$, for some k.

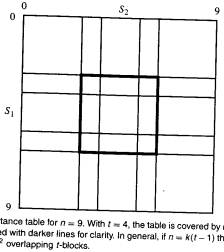


Figure 12.22: An edit distance table for $n = 9$. With $t = 4$, the table is covered by nine overlapping blocks. The center block is outlined with darker lines for clarity. In general, if $n = k(t-1)$ then the $(n+1)$ by $(n+1)$ table will be covered by k^2 overlapping t-blocks.

Block edit distance algorithm

Begin

1. Cover the $(n+1)$ by $(n+1)$ dynamic programming table with t-blocks, where the last column of every t-block is shared with the first column of the t-block to its right (if any), and the last row of every t-block is shared with the first row of the t-block below it (if any). (See Figure 12.22). In this way, and since $n = k(t-1)$, the table will consist of k rows and k columns of partially overlapping t-blocks.
2. Initialize the values in the first row and column of the full table according to the base conditions of the recurrence.
3. In a rowwise manner, use the *restricted* block function to successively determine the values in the last row and last column of each block. By the overlapping nature of the blocks, the values in the last column (or row) of a block are the values in the first column (or row) of the block to its right (or below it).
4. The value in cell (n, n) is the edit distance of S_1 and S_2.

end.

Of course, the heart of the algorithm is step 3, where specific instances of the restricted block function must be computed. Any instance of the restricted block function can be computed $O(t^2)$ time, but that gains us nothing. So how is the restricted block function computed?

12.7.2. The Four-Russians idea for the restricted block function

The general Four-Russians observation is that a speedup can often be obtained by *precomputing* and storing information about all possible instances of a subproblem that might arise in solving a problem. Then, when solving an instance of the full problem and specific subproblems are encountered, the computation can be accelerated by looking up the answers to precomputed subproblems, instead of recomputing those answers. If the subproblems are chosen correctly, the total time taken by this method (including the time for the precomputations) will be less than the time taken by the standard computation.

In the case of edit distance, the precomputation suggested by the Four-Russians idea is to enumerate all possible inputs to the restricted block function (the proper size of the block will be determined later), compute the resulting output values (a t-length row and a t-length column) for each input, and store the outputs indexed by the inputs. Every time a specific restricted block function must be computed in step 3 of, the *block edit distance algorithm*, the value of the function is then retrieved from the precomputed values and need not be computed. This clearly works to compute the edit distance $D(n, n)$, but is it any faster than the original $O(n^2)$ method? Astute readers should be skeptical, so please suspend disbelief for now.

Accounting detail

Assume first that all the precomputation has been done. What time is needed to execute the *block edit distance algorithm*? Recall that the sizes of the input and the output of the restricted block function are both $O(t)$. It is not difficult to organize the input-output values of the (precomputed) restricted block function so that the correct output for any specific input can be retrieved in $O(t)$ time. Details are left to the reader. There are $\Theta(n^2/t^2)$ blocks, hence the total time used by the *block edit distance algorithm* is $O(n^2/t)$. Setting t to $\Theta(\log n)$, the time is $O(n^2/\log n)$. However, in the unit-cost RAM model of computation, each output value can be retrieved in constant time since $t = O(\log n)$. In that case, the time for the method is reduced to $O(n^2/(\log n)^2)$.

But what about the precomputation time? The key issue involves the number of input choices to the restricted block function. By definition, every cell has an integer from zero to n, so there are $(n+1)^t$ possible values for any t-length row or column. If the alphabet has size σ, then there are σ^t possible substrings of length t. Hence the number of distinct input combinations to the restricted block function is $(n+1)^{2t}\sigma^{2t}$. For each input, it takes $\Theta(t^2)$ time to evaluate the last row and column of the resulting t-block (by running the standard dynamic program). Thus the overall time used in this way to precompute the function outputs to all possible input choices is $\Theta((n+1)^{2t}\sigma^{2t}t^2)$. But t must be at least one, so $\Omega(n^2)$ time is used in this way. No progress yet! The idea is right, but we need another trick to make it work.

12.7.3. The trick: offset encoding

The dominant term in the precomputation time is $(n+1)^{2t}$, since σ is assumed to be fixed. That term comes from the number of distinct choices there are for two t-length subrows and subcolumns. But $(n+1)^t$ overcounts the number of different t-length subrows (or subcolumns) that could appear in a real table, since the value in a cell is not independent of the values of its neighbors. We next make this precise.

Lemma 12.7.2. *In any row, column, or diagonal of the dynamic programming table for edit distance, two adjacent cells can have a value that differs by at most one.*

PROOF Certainly, $D(i, j) \leq D(i, j-1) + 1$. Conversely, if the optimal alignment of $S_1[1..i]$ and $S_2[1..j]$ matches $S_2(j)$ to some character of S_1, then by simply omitting $S_2(j)$ and aligning its mate against a space, the distance increases by at most one. If $S_2(j)$ is not matched then its omission reduces the distance by one. Hence $D(i, j-1) \leq D(i, j) + 1$, and the lemma is proved for adjacent row cells. Similar reasoning holds along a column.

In the case of adjacent cells in a diagonal, it is easy to see that $D(i, j) \leq D(i-1, j-1) + 1$. Conversely, if the optimal alignment of $S_1[1..i]$ and $S_2[1..j]$ aligns i against j,

then $D(i-1, j-1) \leq D(i, j)+1$. If the optimal alignment doesn't align i against j, then at least one of the characters, $S_1(i)$ or $S_2(j)$, must align against a space, and $D(i-1, j-1) \leq D(i, j)$. □

Given Lemma 12.7.2, we can *encode* the values in a row of a t-block by a t-length vector specifying the value of the first entry in the row, and then specifying the difference (offset) of each successive cell value to its left neighbor: A zero indicates equality, a one indicates an increase by one, and a minus one indicates a decrease by one. For example, the row of distances 5, 4, 4, 5 would be encoded by the row of offsets 5, -1, 0, $+1$. Similarly, we can encode the values in any column by such offset encoding. Since there are only $(n+1)3^{t-1}$ distinct vectors of this type, a change to offset encoding is surely a move in the right direction. We can, however, reduce the number of possible vectors even further.

Definition The *offset vector* is a t-length vector of values from $\{-1, 0, 1\}$, where the first entry must be zero.

The key to making the Four-Russians method efficient is to compute edit distance using only offset vectors rather than actual distance values. Because the number of possible offset vectors is much less than the number of possible vectors of distance values, much less precomputation will be needed. We next show that edit distance can be computed using offset vectors.

Theorem 12.7.1. *Consider a t-block with upper left corner in position (i, j). The two offset vectors for the last row and last column of the block can be determined from the two offset vectors for the first row and column of the block and from substrings $S_1[1..i]$ and $S_2[1..j]$. That is, no D value is needed in the input in order to determine the offset vectors in the last row and column of the block.*

PROOF The proof is essentially a close examination of the dynamic programming recurrences for edit distance. Denote the unknown value of $D(i, j)$ by C. Then for column q in the block, $D(i, q)$ equals C plus the total of the offset values in row i from column $j+1$ to column q. Hence even if the algorithm doesn't know the value of C, it can express $D(i, q)$ as C plus an integer that it can determine. Each $D(q, j)$ can be similarly expressed. Let $D(i, j+1)$ be $C+J$ and let $D(i+1, j)$ be $C+I$, where the algorithm can know I and J. Now consider cell $(i+1, j+1)$. $D(i+1, j+1)$ is equal to $D(i, j) = C$ if character $S_1(i)$ matches $S_2(j)$. Otherwise $D(i+1, j+1)$ equals the minimum of $D(i, j+1)+1$, $D(i+1, j)+1$, and $D(i, j)+1$, i.e., the minimum of $C+I+1$, $C+J+1$, and $C+1$. The algorithm can make this comparison by comparing I and J (which it knows) to the number zero. So the algorithm can correctly express $D(i+1, j+1)$ as C, $C+I+1$, $C+J+1$, or $C+1$. Continuing in this way, the algorithm can correctly express each D value in the block as an unknown C plus some integer that it can determine. Since every term involves the same unknown constant C, the offset vectors can be correctly determined by the algorithm. □

Definition The function that determines the two offset vectors for the last row and last column from the two offset vectors for the first row and column of a block together with substrings $S_1[1..i]$ and $S_2[1..j]$ is called the *offset function*.

We now have all the pieces of the Four-Russians–type algorithm to compute edit distance. We again assume, for simplicity, that each string has length $n = k(t-1)$ for some k.

Four-Russians edit distance algorithm

1. Cover the n by n dynamic programming table with t-blocks, where the last column of every t-block is shared with the first column of the t-block to its right (if any), and the last row of every t-block is shared with the first row of the t-block below it (if any).
2. Initialize the values in the first row and column of the full table according to the base conditions of the recurrence. Compute the offset values in the first row and column.
3. In a rowwise manner, use the *offset* block function to successively determine the offset vectors of the last row and column of each block. By the overlapping nature of the blocks, the offset vector in the last column (or row) of a block provides the next offset vector in the first column (or row) of the block to its right (or below it). Simply change the first entry in the next vector to zero.
4. Let Q be the total of the offset values computed for cells in row n. $D(n, n) = D(n, 0)+Q = n+Q$.

Time analysis

As in the analysis of the *block edit distance algorithm*, the execution of the *four-Russians edit distance algorithm* takes $O(n^2/\log n)$ time (or $O[n^2/(\log n)^2]$ time in the unit-cost RAM model) by setting t to $\Theta(\log n)$. So again, the key issue is the time needed to precompute the block offset function. Recall that the first entry of an offset vector must be zero, so there are $3^{2(t-1)}$ possible offset vectors. There are σ^t ways to specify a substring over an alphabet with σ characters, and so there are $3^{2(t-1)}\sigma^{2t}$ ways to specify the input to the offset function. For any specific input choice, the output is computed in $O(t^2)$ time (via dynamic programming), hence the entire precomputation takes $O(3^{2t}\sigma^{2t}t^2)$ time. Setting t equal to $(\log_{3\sigma} n)/2$, the precomputation time is just $O(n(\log n)^2)$. In summary, we have

Theorem 12.7.2. *The edit distance of two strings of length n can be computed in $O\left(\frac{n^2}{\log n}\right)$ time or $O\left(\frac{n^2}{(\log n)^2}\right)$ time in the unit-cost RAM model.*

Extension to strings of unequal lengths is easy and is left as an exercise.

12.7.4. Practical approaches

The theoretical result that edit distance can be computed in $O\left(\frac{n^2}{\log n}\right)$ time has been extended and applied to a number of different alignment problems. For truly large strings, these theoretical results are worth using. But the Four-Russians method is primarily a theoretical contribution and is not used in its full detail. Instead, the basic idea of precomputing either the restricted block function or the offset function is used, but only for *fixed* size blocks. Generally, t is set to a fixed value independent of n and often a rectangular 2 by t block is used in place of a square block. The point is to pick t so that the restricted block or offset function can be determined in constant time on practical machines. For example, t could be picked so that the offset vector fits into a single computer word. Or, depending on the alphabet and the amount of space available, one might hash the input choices for rapid function retrieval. This should lead to a computing time of $O\left(\frac{n^2}{t}\right)$, although practical programming issues become important at this level of detail. A detailed experimental analysis of these ideas [339] has shown that this approach is one of the most effective ways to speed up the practical computation of edit distance, providing a factor of t speedup over the standard dynamic programming solution.

12.8. Exercises

1. Show how to compute the *value* $V(n,m)$ of the optimal alignment using only $\min(n,m)+1$ space in addition to the space needed to represent the two input strings.
2. Modify Hirschberg's method to work for alignment with a gap penalty (affine and general) in the objective function. It may be helpful to use both the affine gap recurrences developed in the text, and the alternative recurrences that pay for a gap when terminated. The latter recurrences were developed in the exercise 27 of Chapter 11.
3. Hirschberg's method computes one optimal alignment. Try to find ways to modify the method to produce more (all?) optimal alignments while still achieving substantial space reduction and maintaining a good time bound compared to the $O(nm)$-time and space method? I believe this is an open area.
4. Show how to reduce the size of the strip needed in the method of Section 12.2.3, when $|m-n| < k$.
5. Fill in the details of how to find the actual alignments of P in T that occur with at most k differences. The method uses the $O(km)$ values stored during the k differences algorithm. The solution is somewhat simpler if the k differences algorithm also stores a sparse set of pointers recording how each farthest-reaching d-path extends a farthest-reaching $(d-1)$-path. These pointers only take $O(km)$ space and are a sparse version of the standard dynamic programming pointers. Fill in the details for this approach as well.
6. The k differences problem is an unweighted (or unit weighted) alignment problem defined in terms of the number of mismatches and spaces. Can the $O(km)$ result be extended to operator- or alphabet-weighted versions of alignment? The answer is: not completely. Explain why not. Then find special cases of weighted alignment, and plausible uses for these cases, where the result does extend.
7. Prove Lemma 12.3.2 from page 274.
8. Prove Lemma 12.3.4 from page 277.
9. Prove Theorem 12.4.2 that concerns space use in the P-against-all problem.
10. **The threshold *P*-against-all problem**

    The P-against-all problem was introduced first because it most directly illustrates one general approach to using suffix trees to speed up dynamic programming computations. And, it has been proposed that such a massive study of how P relates to substrings of T can be important in certain problems [183]. Nonetheless, for most applications the output of the P-against-all problem is excessive and a more focused computation is desirable. The *threshold P-against-all problem* is of this type: Given strings P and T and a threshold d, find every substring T' of T such that the edit distance between P and T' is less than d. Of course, it would be cheating to first solve the P-against-all problem and then filter out the substrings of T whose edit distance to P is d or greater. We want a method whose speed is related to d. The computation should increase in speed as d falls.

    The idea is to follow the solution to the P-against-all problem, doing a depth-first traversal of suffix tree T, but recognize subtrees that need not be traversed. The following lemma is the key.

    Lemma 12.8.1. *In the P-against-all problem, suppose that the current path in the suffix tree specifies a substring S of T and that the current dynamic programming column (including the zero row) contains no values below d. Then the column representing an extension of S will also contain no values below d. Hence no columns need be computed for any extensions of S.*

    Prove the lemma and then show how to exploit it in the solution to the threshold P-against-all problem. Try to estimate how effective the lemma is in practice. Be sure to consider how the output is efficiently collected when the dynamic programming ends high in the tree, before a leaf is reached.
11. Give a complete proof of the correctness of the all-against-all suffix tree algorithm.
12. Another, faster, alternative to the P-against-all problem is to change the problem slightly as follows: For each position i in T such that there is a substring starting at i with edit distance less than d from P, report only the *smallest* such substring starting at position i. This is the (P-against-all) *starting location problem*, and it can be solved by modifying the approach discussed for the threshold P-against-all problem. The starting location problem (actually the equivalent ending location problem) is the subject of a paper by Ukkonen [437]. In that paper, Ukkonen develops three hybrid dynamic programming methods in the same spirit as those presented in this chapter, but with additional technical observations. The main result of that paper was later improved by Cobbs [105].

    Detail a solution to the starting location problem, using a hybrid dynamic programming approach.
13. Show that the suffix tree methods and time bounds for the P-against-all and the all-against-all problems extend to the problem of computing similarity instead of edit distance.
14. Let R be a regular expression. Show how to modify the P-against-all method to solve the R-against-all problem. That is, show how to use a suffix tree to efficiently search for a substring in a large text T that matches the regular expression R. (This problem is from [63].)

    Now extend the method to allow for a bounded number of errors in the match.
15. Finish the proof of Theorem 12.5.2.
16. Show that in any permutation of n integers from 1 to n, there is either an increasing subsequence of length at least $\sqrt{n}$ or a decreasing subsequence of length at least $\sqrt{n}$. Show that, averaged over all the $n!$ permutations, the average length of the longest increasing subsequence is at least $\sqrt{n}/2$. Show that the lower bound of $\sqrt{n}/2$ cannot be tight.
17. What do the results from the previous problem imply for the *lcs* problem?
18. If S is a subsequence of another string S', then S' is said to be a *supersequence* of S. If two strings S_1 and S_2 are subsequences of S', then S' is a *common supersequence* of S_1 and S_2. That leads to the following natural question: Given two strings S_1 and S_2, what is the *shortest* supersequence common to both S_1 and S_2. This problem is clearly related to the longest common subsequence problem. Develop an explicit relationship between the two problems, and the lengths of their solutions. Then develop efficient methods to find a shortest common supersequence of two strings. For additional results on subsequences and supersequences see [240] and [241].
19. Can the results in the previous problem be generalized to the case of more than two strings? For instance, is there a natural relationship between the longest common subsequence and the shortest common supersequence of three strings?
20. Let T be a string whose characters come from an alphabet Σ with σ characters. A subsequence S of T is *nondecreasing* if each successive character in S is lexically greater than or equal to the preceding character. For example, using the English alphabet let T = *characterstring*; then S = *aacrst* is a nondecreasing subsequence of T. Give an algorithm that finds the longest nondecreasing subsequence of a string T in time $O(n\sigma)$, where n is the length of T. How does this bound compare to the $O(n \log n)$ bound given for the longest increasing subsequence problem over integers.
21. Recall the definition of r given for two strings in Section 12.5.2 on page 290. Extend the

definition for r to the longest common subsequence problem for more than two strings, and use r to express the time for finding an *lcs* in this case.

22. Show how to model and solve the *lis* problem as a shortest path problem in a directed, acyclic graph. Are there any advantages to viewing the problem in this way?

23. Suppose we only want to learn the length of the *lcs* of two strings S_1 and S_2. That can be done, as before, in $O(r \log n)$ time, but now only using linear space. The key is to keep only the last element in each list of the cover (when computing the *lis*), and not to generate all of $\Pi(S_1, S_2)$ at once, but to generate (in linear space) parts of $\Pi(S_1, S_2)$ on the fly. Fill in the details of these ideas and show that the length of the *lcs* can be computed as quickly as before in only linear space.

Open problem: Extend the above combinatorial ideas, to show how to compute the actual *lcs* of two strings using only linear space, without increasing the needed time. Then extend to more than two strings.

24. (This problem requires a knowledge of systolic arrays.) Show how to implement the longest increasing subsequence algorithm to run in $O(n)$ time on an $O(n)$-element systolic array (remember that each array element has only constant memory). To make the problem simpler, first consider how to compute the length of the *lis*, and then work out how to compute the actual increasing subsequence.

25. Work out how to compute the *lcs* in $O(n)$ time on an $O(n)$-element systolic array.

26. We have reduced the *lcs* problem to the *lis* problem. Show how to do the reduction in the opposite direction.

27. Suppose each character in S_1 and S_2 is given an individual weight. Give an algorithm to find an increasing subsequence of maximum total weight.

28. Derive an $O(nm \log m)$-time method to compute edit distance for the convex gap weight model.

29. The idea of forward dynamic programming can be used to speed up (in practice) the (global) alignment of two strings, even when gaps are not included in the objective function. We will explain this in terms of computing unweighted edit distance between strings S_1 and S_2 (of lengths n and m respectively), but the basic idea works for computing similarity as well. Suppose a cell (i, j) is reached during the (forward) dynamic programming computation of edit distance and the value there is $D(i, j)$. Suppose also that there is a fast way to compute a *lower bound*, $L(i, j)$, on the distance between substrings $S_1[i+1, \ldots, n]$ and $S_2[j+1, \ldots, m]$. If $D(i, j) + L(i, j)$ is greater than or equal to a known distance between S_1 and S_2 obtained from some particular alignment, then there is no need to propogate candidate values forward from cell (i, j). The question now is to find efficient methods to compute "effective" values of $L(i, j)$. One simple one is $|n - m + j - i|$. Explain this. Try it out in practice to see how effective it is. Come up with other simple lower bounds that are much more effective.

Hint: Use the count of the number of times each character appears in each string.

30. As detailed in the text, the Four-Russians method precomputes the offset function for $3^{2(t-1)}\sigma^{2t}$ specifications of input values. However, the problem statement and time bound allow the precomputation of the offset function to be done *after* strings S_1 and S_2 are known. Can that observation be used to reduce the running time?

An alternative encoding of strings allows the σ^{2t} term to be changed to $(t+2)^t$ even in problem settings where S_1 and S_2 are not known when the precomputation is done. Discover and explain the encoding and how edit distance is computed when using it.

31. Consider the situation when the edit distance must be computed for each pair of strings from a large set of strings. In that situation, the precomputation needed by the Four-Russians

method seems more justified. In fact, why not pick a "reasonable" value for t, do the precomputation of the offset function once for that t, and then embed the offset function in an edit distance algorithm to be used for all future edit distance computations. Discuss the merits and demerits of this proposal.

32. The Four-Russians method presented in the text only computes the edit distance. How can it be modified to compute the edit transcript as well?

33. Show how to apply the Four-Russians method to strings of unequal length.

34. What problems arise in trying to extend the Four-Russians method and the improved time bound to the *weighted* edit distance problem? Are there restrictions on weights (other than equality) that make the extension easier?

35. Following the lines of the previous question, show in detail how the Four-Russians approach can be used to solve the longest common subsequence problem between two strings of length n, in $O(n^2 / \log n)$ time.