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.*

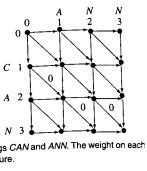


Figure 11.4: Edit graph for the strings *CAN* and *ANN*. The weight on each edge is one, except for the three zero-weight edges marked in the figure.

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

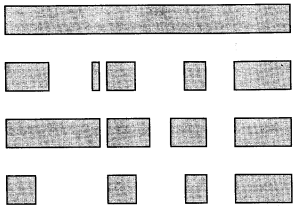


Figure 11.6: Each of the four rows represents part of the RNA sequence of one strain of the HIV-1 virus. The HIV virus mutates rapidly, so that mutations can be observed and traced. The bottom three rows are from virus strains that have each mutated from an ancestral strain represented in the top row. 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].

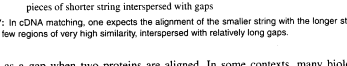


Figure 11.7: In cDNA matching, one expects the alignment of the smaller string with the longer string to consist of a few regions of very high similarity, interspersed with relatively long gaps.

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*.

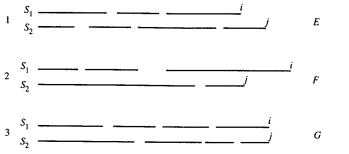


Figure 11.8: The recurrences for alignment with gaps are divided into three types of alignments: 1. those that align $S_1(i)$ to the left of $S_2(j)$, 2. those that align $S_1(i)$ to the right of $S_2(j)$, and 3. those that align them opposite each other.

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, $half(T)$

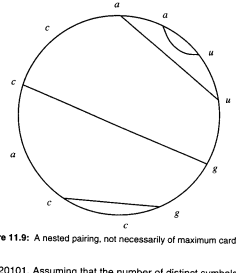


Figure 11.9: A nested pairing, not necessarily of maximum cardinality.

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.

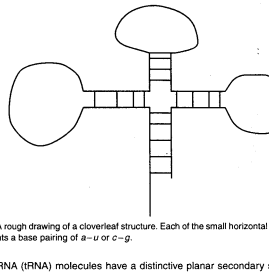


Figure 11.10: A rough drawing of a cloverleaf structure. Each of the small horizontal or vertical lines inside a stem represents a base pairing of $a-u$ or $c-g$.

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

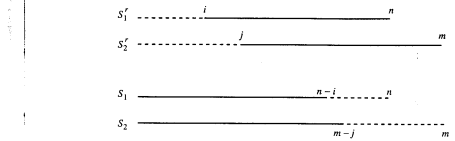


Figure 12.1: The similarity of the first i characters of S_1^r and the first j characters of S_2^r equals the similarity of the last i characters of S_1 and the last j characters of S_2. (The dotted lines denote the substrings being aligned.)

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

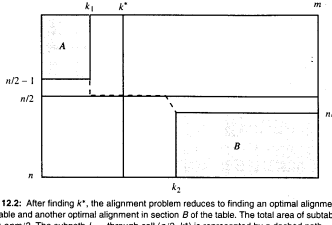


Figure 12.2: After finding k^*, the alignment problem reduces to finding an optimal alignment in section A of the table and another optimal alignment in section B of the table. The total area of subtables A and B is at most $cnm/2$. The subpath $L_{n/2}$ through cell $(n/2, k^*)$ is represented by a 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.

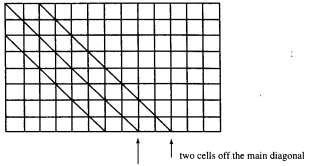


Figure 12.3: The main diagonal and a strip that is $k = 2$ spaces off the main diagonal on each side.

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 $O(k'm + k'm/2 + k'm/4 + \cdots + m) = O(k'm) = O(k^*m)$. □

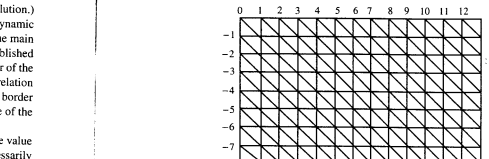


Figure 12.4: The numbered diagonals of the dynamic programming table.

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.

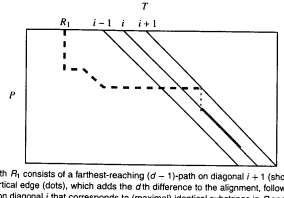


Figure 12.5: Path R_1 consists of a farthest-reaching $(d-1)$-path on diagonal $i+1$ (shown with dashes), followed by a vertical edge (dots), which adds the dth difference to the alignment, followed by a maximal path (solid line) on diagonal i that corresponds to (maximal) identical substrings in P and 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.

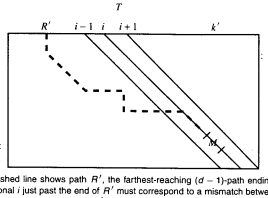


Figure 12.6: The dashed line shows path R', the farthest-reaching $(d-1)$-path ending on diagonal i. The edge M on diagonal i just past the end of R' must correspond to a mismatch between P and T (the characters involved are denoted $P(k)$ and $T(k')$ in the figure).

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

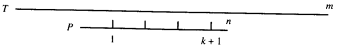


Figure 12.7: The first $k+1$ regions of P are each of length $r = \lfloor \frac{n}{k+1} \rfloor$.

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

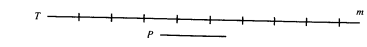


Figure 12.8: Each full region in T has length $r = n/2$. This assures that no matter how P is aligned with T, P spans one full region.

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).

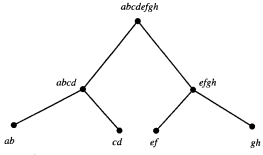


Figure 12.10: Binary tree B defining the successive divisions of P and its partition into regions of length $\log_c m$ (equal to two in this figure).

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.

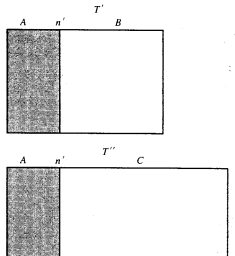


Figure 12.11: A cartoon of the dynamic programming tables for computing the edit distance between P and substring T' (top) and between P and substring T'' (bottom). The two tables share the subtable for P and substring A (shown as a shaded rectangle). This shaded subtable only needs to be computed once.

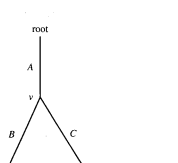


Figure 12.12: A piece of the suffix tree for T. The traversal from the root to node v is accompanied by the computation of subtable A (from the previous figure). At that point, the last row and column of subtable A are stored at node v. Computing the subtable B corresponds to the traversal from v to 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.

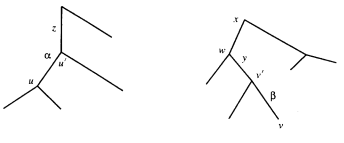
But the full all-against-all problem calls for an amount of output that is often excessive, and the output can be reduced by choosing a meaningful threshold. Or the criteria for reporting a substring pair might be a function of both length and edit distance. Whatever the specific reporting criteria, 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

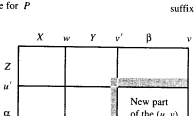


Figure 12.13: The dynamic programming table for (u,v) is shown below the suffix trees for P and T. The string on the path to node u is $Z\alpha$ and the string to node v is $XY\beta$. Every cell in the (u,v) table, except any in the lower right rectangle, is also in the (u,v'), (u',v), or (u',v') tables. 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.

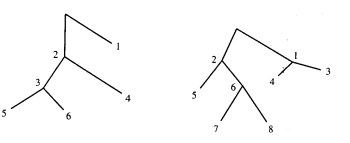
The threshold all-against-all problem could be solved (ignoring time) by computing the dynamic programming table for each pair of leaves, one from each tree, and then examining every entry in each of those tables. Hence it certainly would be solved by computing the dynamic programming table for each 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.

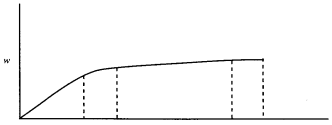


Figure 12.16: A convex function w.

[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.

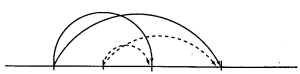


Figure 12.17: Graphical illustration of the *key observation*. Winning candidates are shown with a solid curve and losers with a dashed curve. If the candidate from j loses to the candidate from k at cell j', then the candidate from j will lose to the candidate from k at every cell j'' to the right of 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

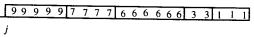


Figure 12.18: Partition of the cells $j+1$ through m into maximal blocks of consecutive cells such that all the cells in any block have the same b value. The common b value in any block is less than the common b value in the preceding block.

$k < j'$ 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.

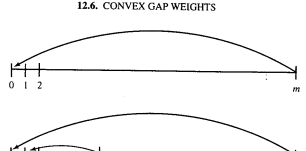


Figure 12.19: The three possible ways that the block partition changes after $E(1)$ is set. The curves with arrows represent the common pointer for the block and leave from the last entry in the block.

- 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 $Cand(j, p_i)$

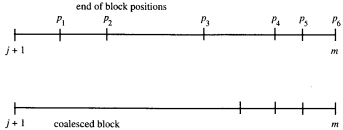


Figure 12.20: To update the block-partition the algorithm successively examines cell p_i to find the first index s where $\overline{E}(p_s) \geq Cand(j, p_s)$. In this figure, s is 4. Blocks 1 through $s-1 = 3$ coalesce into a single block with some initial part of block $s = 4$. Blocks to the right 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.

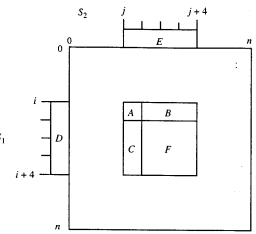


Figure 12.21: A single block with $t = 4$ drawn inside the full dynamic programming table. The distance values in the part of the block labeled F are determined by the values in the parts labeled A, B, and C together with the substrings of S_1 and S_2 in D and E. Note that A is the intersection of the 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.

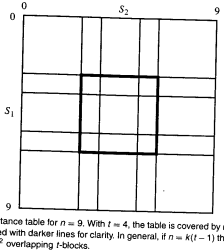


Figure 12.22: An edit distance table for $n = 9$. With $t = 4$, the table is covered by nine overlapping blocks. The center block is outlined with darker lines for clarity. In general, if $n = k(t-1)$ then the $(n+1)$ by $(n+1)$ table will be covered by k^2 overlapping t-blocks.

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.