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

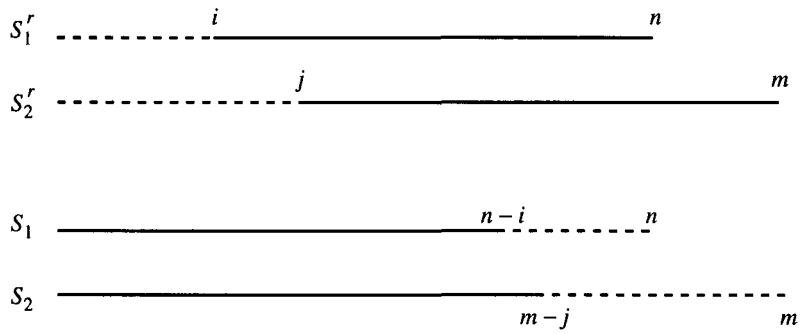


Figure 12.1: The similarity of the first i characters of S_1^r and the first j characters of S_2^r equals the similarity of the last i characters of S_1 and the last j characters of S_2. (The dotted lines denote the substrings being aligned.)

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^r, the best alignment of the first type has value $V(n/2, k')$ and the best alignment of the second type has value $V^r(n/2, m-k')$, so the combined alignment has value $V(n/2, k') + V^r(n/2, m-k') \leq \max_k[V(n/2, k) + V^r(n/2, m-k)] \leq V(n, m)$.

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, $q = V^r(n/2, m-k')$. So $V(n, m) = V(n/2, k') + V^r(n/2, m-k') \leq \max_k[V(n/2, k) + V^r(n/2, m-k)]$.

Having shown both sides of the inequality, we conclude that $V(n, m) = \max_k[V(n/2, k) + V^r(n/2, m-k)]$. □

Definition Let k^* be a position k that maximizes $[V(n/2, k) + V^r(n/2, m-k)]$.

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 $V^r(n/2, m-k)$ is known for every k. Now, for each k, add $V(n/2, k)$ to $V^r(n/2, m-k)$, 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

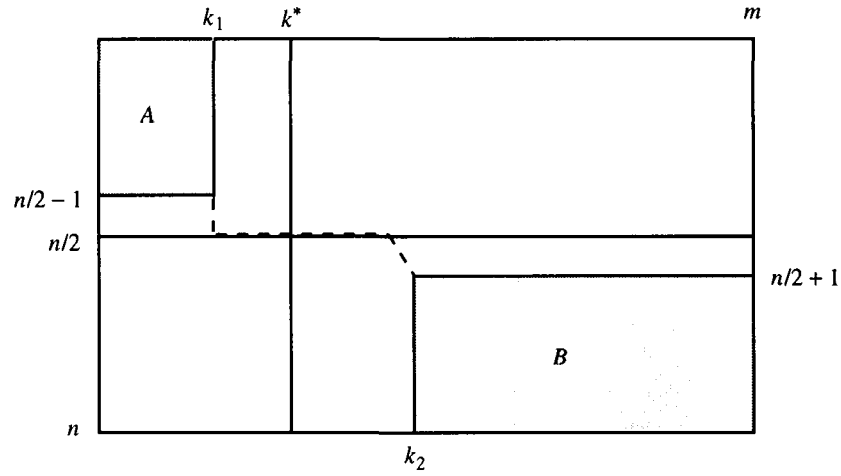


Figure 12.2: After finding k^*, the alignment problem reduces to finding an optimal alignment in section A of the table and another optimal alignment in section B of the table. The total area of subtables A and B is at most $cnm/2$. The subpath $L_{n/2}$ through cell $(n/2, k^*)$ is represented by a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

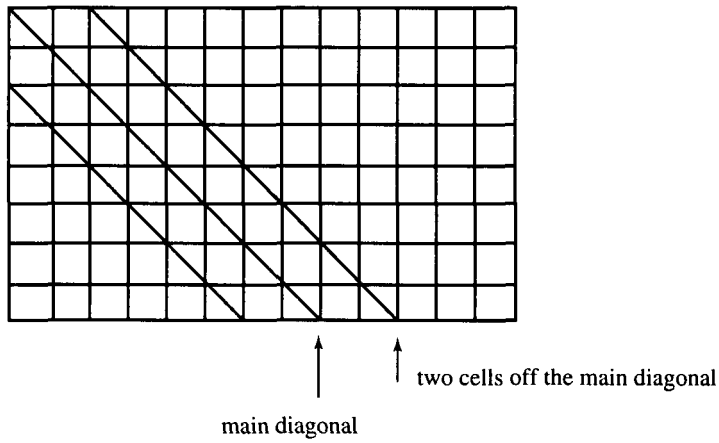


Figure 12.3: The main diagonal and a strip that is $k = 2$ spaces off the main diagonal on each side.

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.

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

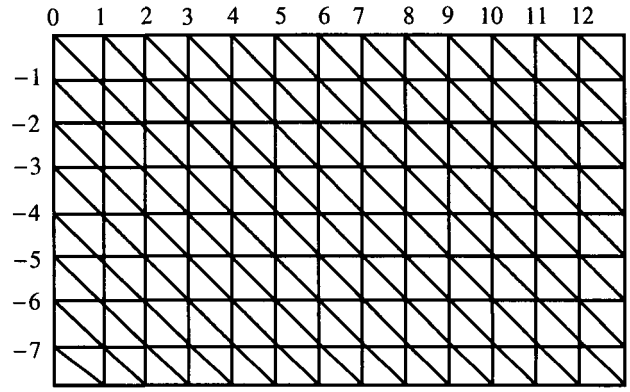


Figure 12.4: The numbered diagonals of the dynamic programming table.

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.

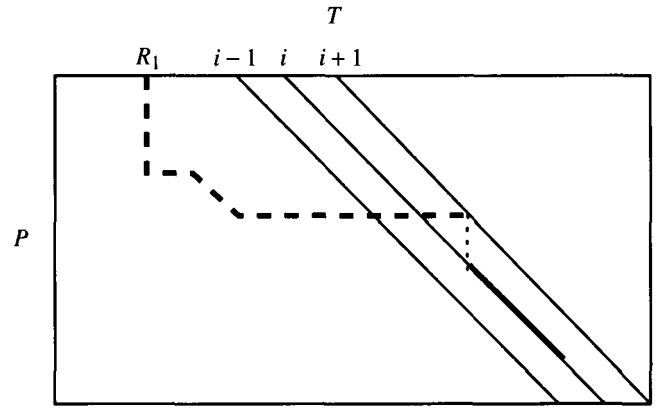


Figure 12.5: Path R_1 consists of a farthest-reaching $(d-1)$-path on diagonal $i+1$ (shown with dashes), followed by a vertical edge (dots), which adds the dth difference to the alignment, followed by a maximal path (solid line) on diagonal i that corresponds to (maximal) identical substrings in P and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

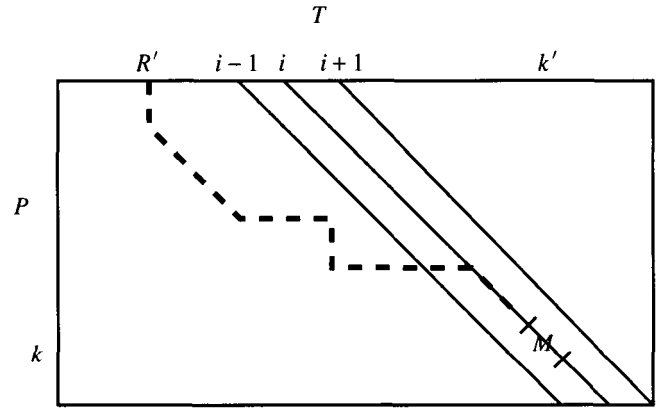


Figure 12.6: The dashed line shows path R', the farthest-reaching $(d-1)$-path ending on diagonal i. The edge M on diagonal i just past the end of R' must correspond to a mismatch between P and T (the characters involved are denoted $P(k)$ and $T(k')$ in the figure).

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 R_1, R_2, and R_3. 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

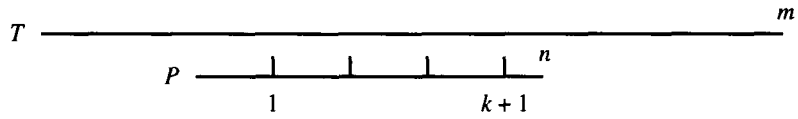


Figure 12.7: The first $k+1$ regions of P are each of length $r = \lfloor \frac{n}{k+1} \rfloor$.

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 $k = O\left(\frac{n}{\log n}\right)$.*

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

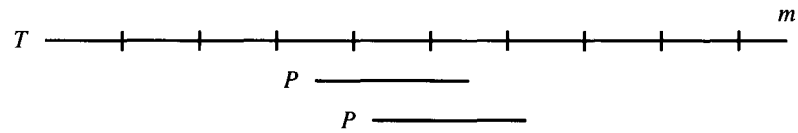


Figure 12.8: Each full region in T has length $r = n/2$. This assures that no matter how P is aligned with T, P spans one full region.

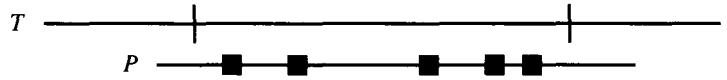


Figure 12.9: Blowup of one region in T aligned with one copy of P. Each black box shows a mismatch between a character in P and its counterpart in T.

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

$$E(M) < E(X) < \log_\sigma n + \sum_{l=\log_\sigma n}^{\infty} \frac{l}{\sigma^l} = \log_\sigma n + 2.$$

□

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).

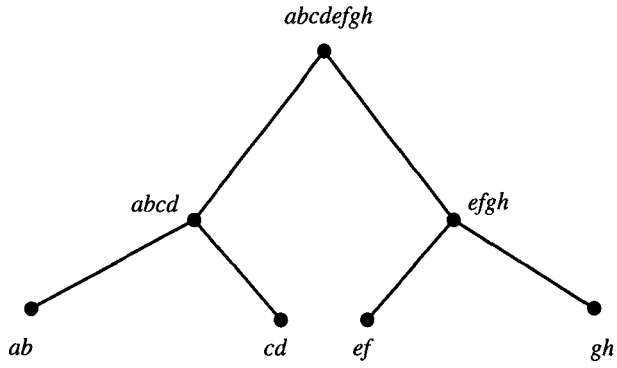


Figure 12.10: Binary tree B defining the successive divisions of P and its partition into regions of length $\log_\sigma m$ (equal to two in this figure).

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.

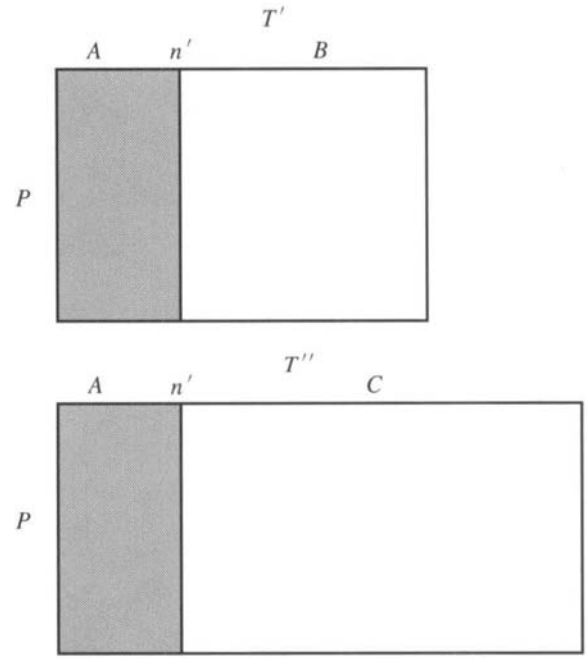


Figure 12.11: A cartoon of the dynamic programming tables for computing the edit distance between P and substring T' (top) and between P and substring T'' (bottom). The two tables share the subtable for P and substring A (shown as a shaded rectangle). This shaded subtable only needs to be computed once.

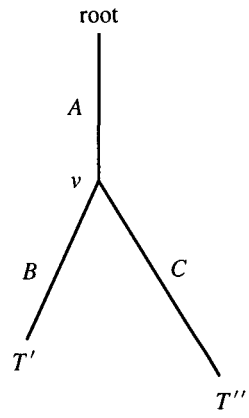


Figure 12.12: A piece of the suffix tree for T. The traversal from the root to node v is accompanied by the computation of subtable A (from the previous figure). At that point, the last row and column of subtable A are stored at node v. Computing the subtable B corresponds to the traversal from v to 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 $\mathcal{T}$. 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* $\Theta(n \times (\textit{length of } \mathcal{T}))$.

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* $\Theta(n \times (\textit{length of } \mathcal{T}) + m^2)$, *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* $\Theta[n(\textit{length of } \mathcal{T}) + m^2]$ *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.

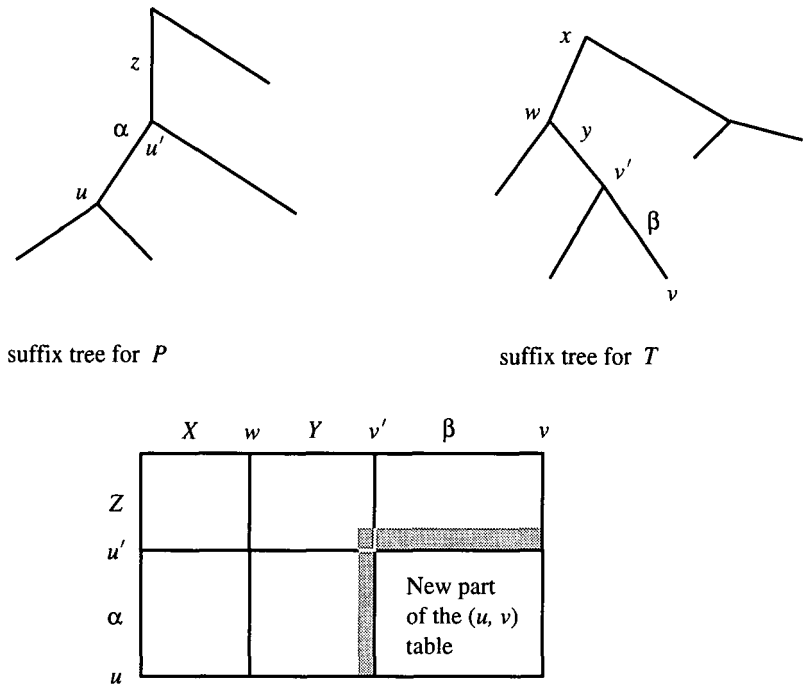


Figure 12.13: The dynamic programming table for (u,v) is shown below the suffix trees for P and T. The string on the path to node u is $Z\alpha$ and the string to node v is $XY\beta$. Every cell in the (u,v) table, except any in the lower right rectangle, is also in the (u,v'), (u',v), or (u',v') tables.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

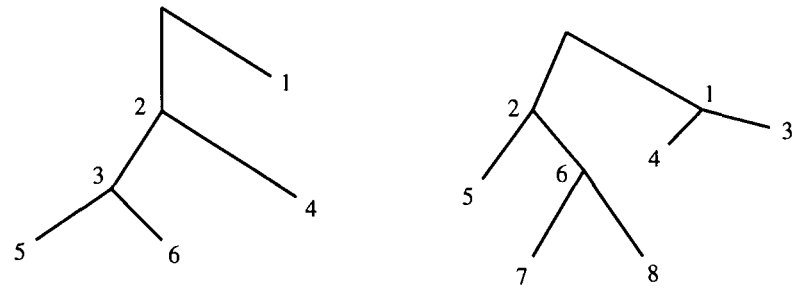
The threshold all-against-all problem could be solved (ignoring time) by computing the dynamic programming table for each pair of leaves, one from each tree, and then examining every entry in each of those tables. Hence it certainly would be solved by computing the dynamic programming table for each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

the pairs of substrings specified by that cell using a traversal to a subset of leaves in the two suffix trees. A formal proof of correctness is left to the reader as an exercise.

For the time analysis, recall that the length of $\mathcal{T}_P$ is the sum of lengths of all the edge labels in $\mathcal{T}_P$. 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 $\log_\sigma m$ 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

[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.

clearer description of it we cannot define precisely what specific all-against-all problem was solved.

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.

5	4	9	8	10
3		6	7	
2				
1				

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 $S_1 =$

[6] This is one of the few places in the book where we deviate from the standard assumption that the alphabet is fixed.

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.

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.

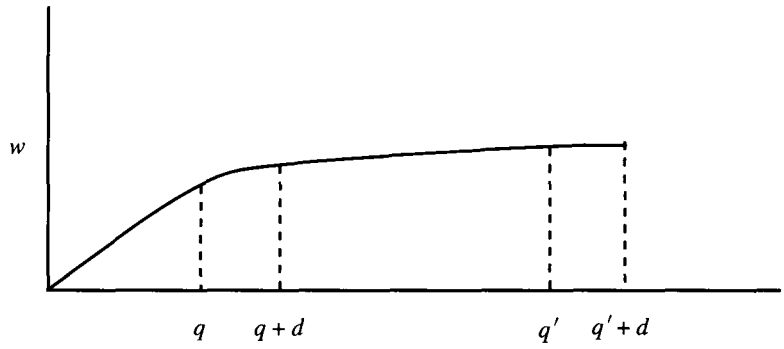


Figure 12.16: A convex function w.

[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.

$$V(i, j) = \max[E(i, j), F(i, j), G(i, j)],$$

$$G(i, j) = V(i - 1, j - 1) + s(S_1(i), S_2(j)),$$

$$E(i, j) = \max_{0 \le k \le j-1} [V(i, k) - w(j - k)],$$

$$F(i, j) = \max_{0 \le l \le i-1} [V(l, j) - w(i - l)],$$

$$V(i, 0) = -w(i),$$

$$V(0, j) = -w(j),$$

$$E(i, 0) = -w(i),$$

$$F(0, j) = -w(j).$$

$G(i, j)$ 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

$$E(j) = \max_{0 \le k \le j-1} [V(k) - w(j - k)].$$

Further, we introduce the following notation to simplify the recurrence:

$$Cand(k, j) = V(k) - w(j - k);$$

therefore,

$$E(j) = \max_{0 \le k \le j-1} Cand(k, j).$$

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.

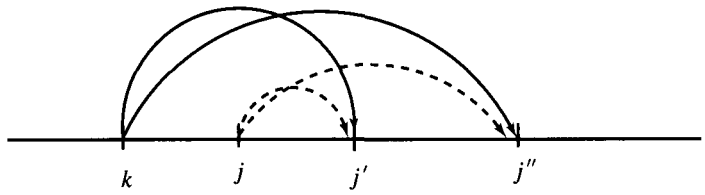


Figure 12.17: Graphical illustration of the *key observation*. Winning candidates are shown with a solid curve and losers with a dashed curve. If the candidate from j loses to the candidate from k at cell j', then the candidate from j will lose to the candidate from k at every cell j'' to the right of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

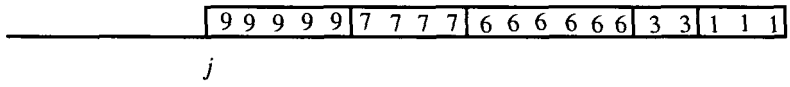


Figure 12.18: Partition of the cells $j+1$ through m into maximal blocks of consecutive cells such that all the cells in any block have the same b value. The common b value in any block is less than the common b value in the preceding block.

$k < j'$ 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 $j, j+1, j+2, \ldots, m$ 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.

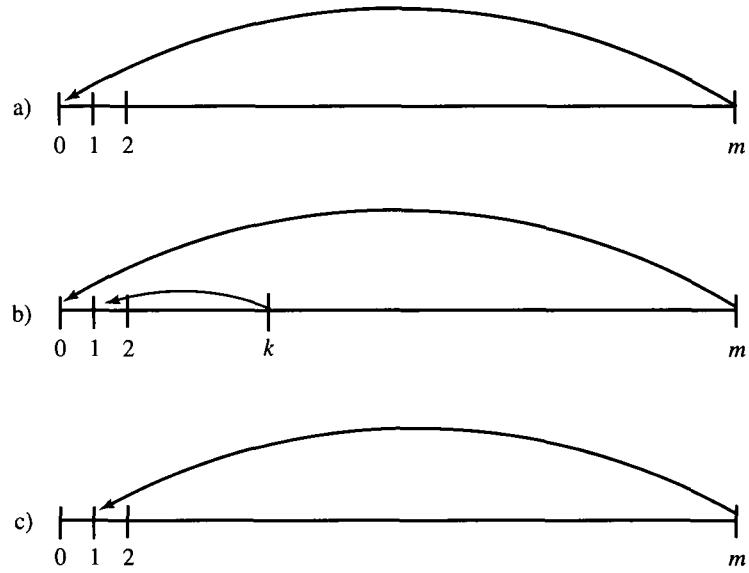


Figure 12.19: The three possible ways that the block partition changes after $E(1)$ is set. The curves with arrows represent the common pointer for the block and leave from the last entry in the block.

- 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 $Cand(j, p_i)$

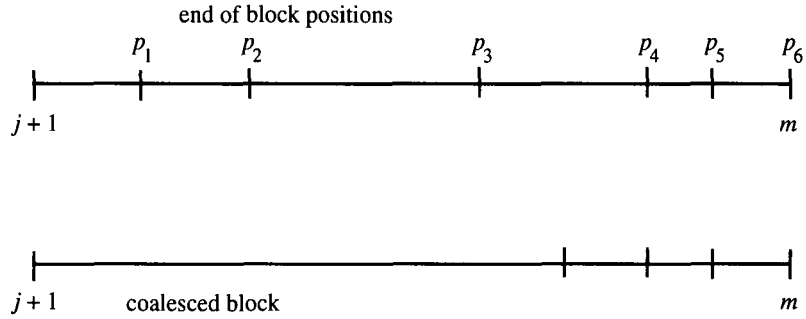


Figure 12.20: To update the block-partition the algorithm successively examines cell p_i to find the first index s where $\overline{E}(p_s) \geq Cand(j, p_s)$. In this figure, s is 4. Blocks 1 through $s-1=3$ coalesce into a single block with some initial part of block $s=4$. Blocks to the right of s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

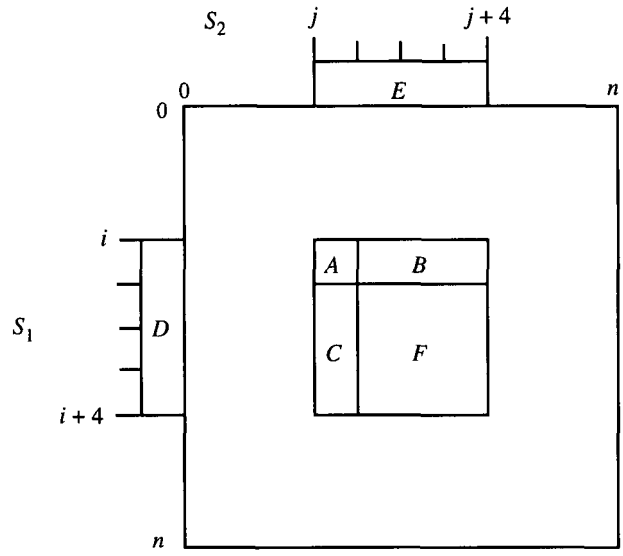


Figure 12.21: A single block with $t = 4$ drawn inside the full dynamic programming table. The distance values in the part of the block labeled F are determined by the values in the parts labeled A, B, and C together with the substrings of S_1 and S_2 in D and E. Note that A is the intersection of the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

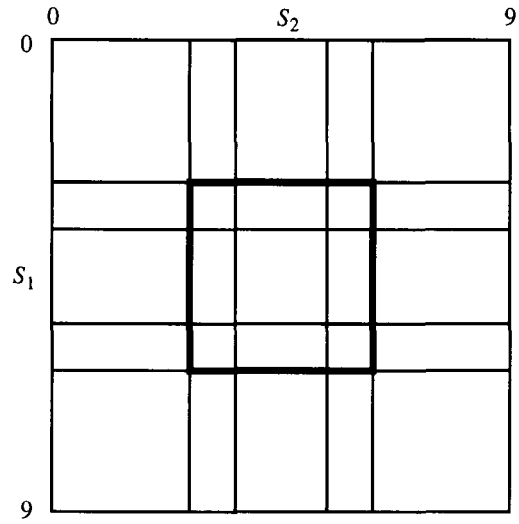


Figure 12.22: An edit distance table for $n = 9$. With $t = 4$, the table is covered by nine overlapping blocks. The center block is outlined with darker lines for clarity. In general, if $n = k(t-1)$ then the $(n+1)$ by $(n+1)$ table will be covered by k^2 overlapping t-blocks.

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.