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

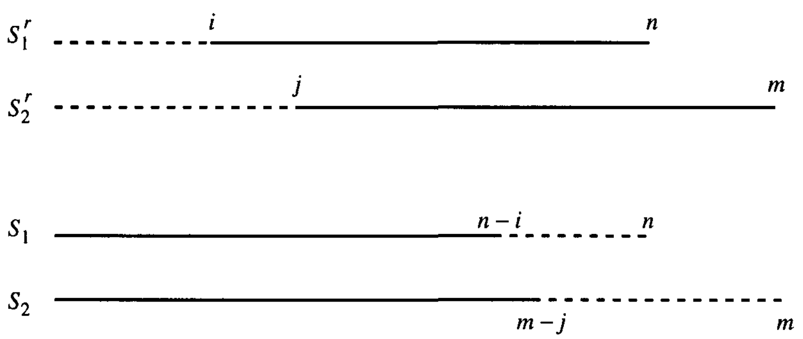

Figure 12.1: The similarity of the first i characters of S_1^r and the first j characters of S_2^r equals the similarity of the last i characters of S_1 and the last j characters of S_2. (The dotted lines denote the substrings being aligned.)

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

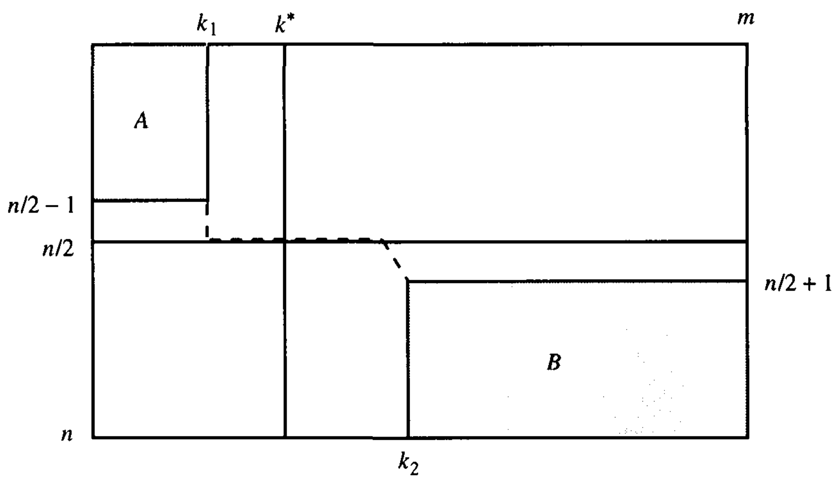

Figure 12.2: After finding k^*, the alignment problem reduces to finding an optimal alignment in section A of the table and another optimal alignment in section B of the table. The total area of subtables A and B is at most $cnm/2$. The subpath $L_{n/2}$ through cell $(n/2, k^*)$ is represented by a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