An O(ND) Difference Algorithm and Its Variations*

EUGENE W. MYERS

Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

ABSTRACT

The problems of finding a longest common subsequence of two sequences A and B and a shortest edit script for transforming A into B have long been known to be dual problems. In this paper, they are shown to be equivalent to finding a shortest/longest path in an edit graph. Using this perspective, a simple O(ND) time and space algorithm is developed where N is the sum of the lengths of A and B and D is the size of the minimum edit script for A and B. The algorithm performs well when differences are small (sequences are similar) and is consequently fast in typical applications. The algorithm is shown to have $O(N+D^2)$ expected-time performance under a basic stochastic model. A refinement of the algorithm requires only O(N) space, and the use of suffix trees leads to an $O(N\lg N+D^2)$ time variation.

KEY WORDS **longest common subsequence** **shortest edit script** **edit graph** **file comparison**

1. Introduction

The problem of determining the differences between two sequences of symbols has been studied extensively [1,8,11,13,16,19,20]. Algorithms for the problem have numerous applications, including spelling correction systems, file comparison tools, and the study of genetic evolution [4,5,17,18]. Formally, the problem statement is to find a longest common subsequence or, equivalently, to find the minimum ‘‘script’’ of symbol deletions and insertions that transform one sequence into the other. One of the earliest algorithms is by Wagner & Fischer [20] and takes $O(N^2)$ time and space to solve a generalization they call the string-to-string correction problem. A later refinement by Hirschberg [7] delivers a longest common subsequence using only linear space. When algorithms are over arbitrary alphabets, use ‘‘equal—unequal’’ comparisons, and are characterized in terms of the size of their input, it has been shown that $\Omega(N^2)$ time is necessary [1]. A ‘‘Four Russians’’ approach leads to slightly better $O(N^2\lg\lg N/\lg N)$ and $O(N^2/\lg N)$ time algorithms for arbitrary and finite alphabets respectively [13]. The existence of faster algorithms using other comparison formats is still open. Indeed, for algorithms that use ‘‘less than—equal—greater than’’ comparisons, $\Omega(N\lg N)$ time is the best lower bound known [9].

* This work was supported in part by the National Science Foundation under Grant MCS82-10096.

Recent work improves upon the basic $O(N^2)$ time arbitrary alphabet algorithm by being sensitive to other problem size parameters. Let the output parameter L be the length of a longest common subsequence and let the dual parameter $D = 2(N-L)$ be the length of a shortest edit script. (It is assumed throughout this introduction that both strings have the same length N.) The two best output-sensitive algorithms are by Hirschberg [8] and take $O(NL+NlgN)$ and $O(DLlgN)$ time. An algorithm by Hunt & Szymanski [11] takes $O((R+N)lgN)$ time where the parameter R is the total number of ordered pairs of positions at which the two input strings match. Note that all these algorithms are $\Omega(N^2)$ or worse in terms of N alone.

In practical situations, it is usually the parameter D that is small. Programmers wish to know how they have altered a text file. Biologists wish to know how one DNA strand has mutated into another. For these situations, an $O(ND)$ time algorithm is superior to Hirschberg's algorithms because L is $O(N)$ when D is small. Furthermore, the approach of Hunt and Szymanski [11] is predicated on the hypothesis that R is small in practice. While this is frequently true, it must be noted that R has no correlation with either the size of the input or the size of the output and can be $O(N^2)$ in many situations. For example, if 10% of all lines in a file are blank and the file is compared against itself, R is greater than $.01N^2$. For DNA molecules, the alphabet size is four implying that R is at least $.25N^2$ when an arbitrary molecule is compared against itself or a very similar molecule.

In this paper an O(ND) time algorithm is presented. Our algorithm is simple and based on an intuitive edit graph formalism. Unlike others it employs the ‘‘greedy’’ design paradigm and exposes the relationship of the longest common subsequence problem to the single-source shortest path problem. Another O(ND) algorithm has been presented elsewhere [16]. However, it uses a different design paradigm and does not share the following features. The algorithm can be refined to use only linear space, and its expected-case time behavior is shown to be $O(N+D^2)$. Moreover, the method admits an $O(NlgN+D^2)$ time worst-case variation. This is asymptotically superior to previous algorithms [8,16,20] when D is o(N).

With the exception of the $O(NlgN+D^2)$ worst-case variation, the algorithms presented in this paper are practical. The basic O(ND) algorithm served as the basis for a new implementation of the UNIX *diff* program [15]. This version usually runs two to four times faster than the System 5 implementation based on the Hunt and Szymanski algorithm [10]. However, there are cases when D is large where their algorithm is superior (e.g. for files that are completely different, R=0 and D=2N). The linear space refinment is roughly twice as slow as the basic O(ND) algorithm but still competitive because it can perform extremely large compares that are out of the range of other algorithms. For instance, two 1.5 million byte sequences were compared in less than two minutes (on a VAX 785 running 4.2BSD UNIX) even though the difference was greater than 500.

2. Edit Graphs

Let $A = a_1 a_2 \cdots a_N$ and $B = b_1 b_2 \cdots b_M$ be sequences of length N and M respectively. The *edit graph* for A and B has a vertex at each point in the grid (x,y), $x \in [0,N]$ and $y \in [0,M]$. The vertices of the edit graph are connected by horizontal, vertical, and diagonal directed edges to form a directed acyclic graph. *Horizontal edges* connect each vertex to its right neighbor, i.e. $(x-1,y) \rightarrow (x,y)$ for $x \in [1,N]$ and $y \in [0,M]$. *Vertical edges* connect each vertex to the neighbor below it, i.e. $(x,y-1) \rightarrow (x,y)$ for $x \in [0,N]$ and $y \in [1,M]$. If $a_x = b_y$ then there is a *diagonal* *edge* connecting vertex (x–1,y–1) to vertex (x,y). The points (x,y) for which $a_x = b_y$ are called *match points*. The total number of match points between A and B is the parameter R characterizing the Hunt & Szymanski algorithm [11]. It is also the number of diagonal edges in the edit graph as diagonal edges are in one-to-one correspondence with match points. Figure 1 depicts the edit graph for the sequences A = *abcabba* and B = *cbabac*.

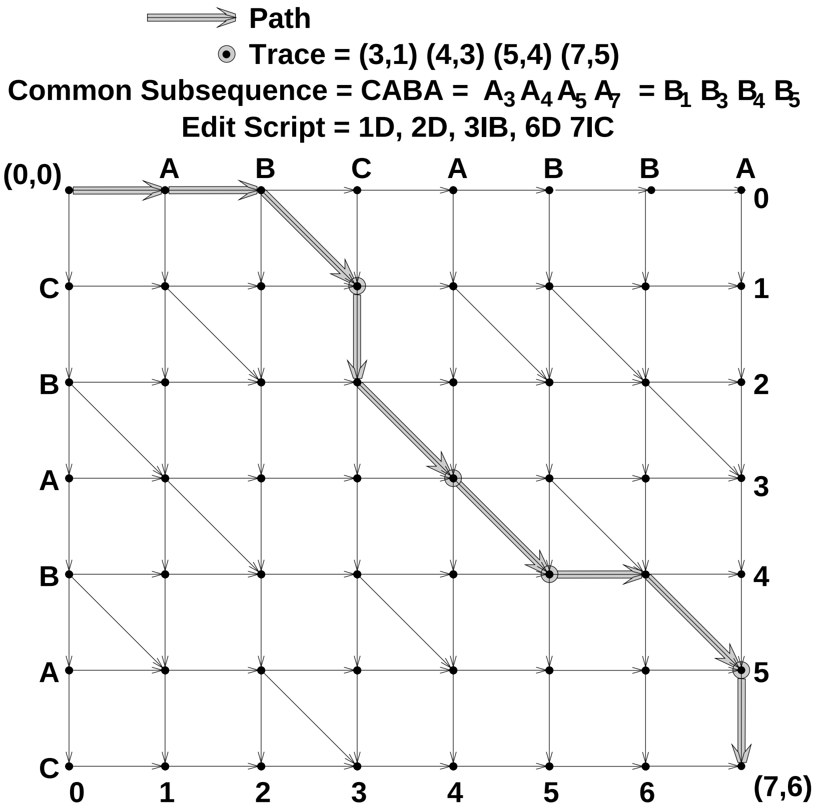

Fig. 1. An edit graph

A *trace* of length L is a sequence of L match points, $(x_1,y_1)(x_2,y_2) \cdots (x_L,y_L)$, such that $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for successive points (x_i,y_i) and (x_{i+1},y_{i+1}), $i \in [1,L-1]$. Every trace is in exact correspondence with the diagonal edges of a path in the edit graph from (0,0) to (N,M). The sequence of match points visited in traversing a path from start to finish is easily verified to be a trace. Note that L is the number of diagonal edges in the corresponding path. To construct a path from a trace, take the sequence of diagonal edges corresponding to the match points of the trace and connect successive diagonals with a series of horizontal and vertical edges. This can always be done as $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for successive match points. Note that several paths differing only in their non-diagonal edges can correspond to a given trace. Figure 1 illustrates this relation between paths and traces.

A *subsequence* of a string is any string obtained by deleting zero or more symbols from the given string. A *common subsequence* of two strings, A and B, is a subsequence of both. Each trace gives rise to a common subsequence of A and B and vice versa. Specifically, $a_{x_1} a_{x_2} \cdots a_{x_L} = b_{y_1} b_{y_2} \cdots b_{y_L}$ is a common subsequence of A and B if and only if $(x_1, y_1)(x_2, y_2) \cdots (x_L, y_L)$ is a trace of A and B.

An *edit script* for A and B is a set of insertion and deletion commands that transform A into B. The *delete command* ''x*D*'' deletes the symbol a_x from A. The *insert command* ''x I $b_1, b_2, \cdots b_t$'' inserts the sequence of symbols $b_1 \cdots b_t$ immediately after a_x. Script commands refer to symbol positions within A before any commands have been performed. One must think of the set of commands in a script as being executed simultaneously. The length of a script is the number of symbols inserted and deleted.

Every trace corresponds uniquely to an edit script. Let $(x_1, y_1)(x_2, y_2) \cdots (x_L, y_L)$ be a trace. Let $y_0 = 0$ and $y_{L+1} = M + 1$. The associated script consists of the commands: ''x*D*'' for $x \notin \{x_1, x_2, \cdots, x_L\}$, and ''$x_k$ I $b_{y_k+1}, \cdots, b_{y_{k+1}-1}$'' for k such that $y_k + 1 < y_{k+1}$. The script deletes $N - L$ symbols and inserts $M - L$ symbols. So for every trace of length L there is a corresponding script of length D = N+M–2L. To map an edit script to a trace, simply perform all delete commands on A, observe that the result is a common subsequence of A and B, and then map the subsequence to its unique trace. Note that inverting the action of the insert commands gives a set of delete commands that map B to the same common subsequence.

Common subsequences, edit scripts, traces, and paths from (0,0) to (N,M) in the edit graph are all isomorphic formalisms. The edges of every path have the following direct interpretations in terms of the corresponding common subsequence and edit script. Each diagonal edge ending at (x,y) gives a symbol, a_x (= b_y), in the common subsequence; each horizontal edge to point (x,y) corresponds to the delete command ''x D''; and a sequence of vertical edges from (x,y) to (x,z) corresponds to the insert command, ''x I $b_{y+1}, \cdots, b_z$''. Thus the number of vertical and horizontal edges in the path is the length of its corresponding script, the number of diagonal edges is the length of its corresponding subsequence, and the total number of edges is N+M–L. Figure 1 illustrates these observations.

The problem of finding a *longest common subsequence* (LCS) is equivalent to finding a path from (0,0) to (N,M) with the maximum number of diagonal edges. The problem of finding a *shortest edit script* (SES) is equivalent to finding a path from (0,0) to (N,M) with the minimum number of non-diagonal edges. These are dual problems as a path with the maximum number of diagonal edges has the minimal number of non-diagonal edges (D+2L = M+N). Consider adding a weight or cost to every edge. Give diagonal edges weight 0 and non-diagonal edges weight 1. The LCS/SES problem is equivalent to finding a minimum-cost path from (0,0) to (N,M) in the weighted edit graph and is thus a special instance of the single-source shortest path problem.

3. An O((M+N)D) Greedy Algorithm

The problem of finding a shortest edit script reduces to finding a path from (0,0) to (N,M) with the fewest number of horizontal and vertical edges. Let a D-path be a path starting at (0,0) that has exactly D non-diagonal edges. A 0-path must consist solely of diagonal edges. By a simple induction, it follows that a D-path must consist of a $(D - 1)$-path followed by a non-diagonal edge and then a possibly empty sequence of diagonal edges called a *snake*.

Number the diagonals in the grid of edit graph vertices so that diagonal k consists of the points (x,y) for which $x - y = k$. With this definition the diagonals are numbered from –M to N. Note that a vertical (horizontal) edge with start point on diagonal k has end point on diagonal $k-1$ $(k+1)$ and a snake remains on the diagonal in which it starts.

Lemma 1: A D-path must end on diagonal $k \in \{ -D, -D+2, \cdots D-2, D \}$.

Proof:

A 0-path consists solely of diagonal edges and starts on diagonal 0. Hence it must end on diagonal 0. Assume inductively that a D-path must end on diagonal k in $\{ -D, -D+2, \cdots D-2, D \}$. Every (D+1)-path consists of a prefix D-path, ending on say diagonal k, a non-diagonal edge ending on diagonal k+1 or k–1, and a snake that must also end on diagonal k+1 or k–1. It then follows that every (D+1)-path must end on a diagonal in $\{ (-D)\pm1, (-D+2)\pm1, \cdots (D-2)\pm1, (D)\pm1 \} = \{ -D-1, -D+1, \cdots D-1, D+1 \}$. Thus the result holds by induction. □

The lemma implies that D-paths end solely on odd diagonals when D is odd and even diagonals when D is even.

A D-path is *furthest reaching* in diagonal k if and only if it is one of the D-paths ending on diagonal k whose end point has the greatest possible row (column) number of all such paths. Informally, of all D-paths ending in diagonal k, it ends furthest from the origin, (0,0). The following lemma gives an inductive characterization of furthest reaching D-paths and embodies a greedy principle: furthest reaching D-paths are obtained by greedily extending furthest reaching $(D-1)$-paths.

Lemma 2: A furthest reaching 0-path ends at (x,x), where x is $\min(z-1 \parallel a_z \neq b_z$ or $z>M$ or $z>N)$. A furthest reaching D-path on diagonal k can without loss of generality be decomposed into a furthest reaching $(D-1)$-path on diagonal $k-1$, followed by a horizontal edge, followed by the longest possible snake or it may be decomposed into a furthest reaching $(D-1)$-path on diagonal k+1, followed by a vertical edge, followed by the longest possible snake.

Proof:

The basis for 0-paths is straightforward. As noted before, a D-path consists of a $(D-1)$-path, a non-diagonal edge, and a snake. If the D-path ends on diagonal k, it follows that the $(D-1)$-path must end on diagonal $k\pm1$ depending on whether a vertical or horizontal edge precedes the final snake. The final snake must be maximal, as the D-path would not be furthest reaching if the snake could be extended. Suppose that the $(D-1)$-path is not furthest reaching in its diagonal. But then a further reaching $(D-1)$-path can be connected to the final snake of the D-path with an appropriate non-diagonal move. Thus the D-path can always be decomposed as desired. □

Given the endpoints of the furthest reaching $(D-1)$-paths in diagonal k+1 and k–1, say (x’,y’) and (x",y") respectively, Lemma 2 gives a procedure for computing the endpoint of the furthest reaching D-path in diagonal k. Namely, take the further reaching of (x’,y’+1) and (x"+1,y") in diagonal k and then follow diagonal edges until it is no longer possible to do so or until the boundary of the edit graph is reached. Furthermore, by Lemma 1 there are only D+1 diagonals in which a D-path can end. This suggests computing the endpoints of D-paths in the relevant D+1 diagonals for successively increasing values of D until the furthest reaching path in diagonal $N-M$ reaches (N,M).

For D $\leftarrow$ 0 **to** M+N **Do**
  For k $\leftarrow$ −D **to** D **in steps of** 2 **Do**
    Find the endpoint of the furthest reaching D-path in diagonal k.
    If (N,M) is the endpoint **Then**
      The D-path is an optimal solution.
      Stop

The outline above stops when the smallest D is encountered for which there is a furthest reaching D-path to (N,M). This must happen before the outer loop terminates because D must be less than or equal to M+N. By construction this path must be minimal with respect to the number of non-diagonal edges within it. Hence it is a solution to the LCS/SES problem.

In presenting the detailed algorithm in Figure 2 below, a number of simple optimizations are employed. An array, V, contains the endpoints of the furthest reaching D-paths in elements $V[-D]$, $V[-D+2]$, $\cdots$, V[D-2], V[D]. By Lemma 1 this set of elements is disjoint from those where the endpoints of the (D+1)-paths will be stored in the next iteration of the outer loop. Thus the array V can simultaneously hold the endpoints of the D-paths while the (D+1)-path endpoints are being computed from them. Furthermore, to record an endpoint (x,y) in diagonal k it suffices to retain just x because y is known to be $x-k$. Consequently, V is an array of integers where V[k] contains the row index of the endpoint of a furthest reaching path in diagonal k.

Constant MAX $\in$ [0,M+N]

Var V: **Array** [$-$MAX .. MAX] **of Integer**

V[1] $\leftarrow$ 0
For D $\leftarrow$ 0 **to** MAX **Do**
  For k $\leftarrow$ −D **to** D **in steps of** 2 **Do**
    If k = −D **or** k $\neq$ D **and** $V[k-1] < V[k+1]$ **Then**
      x $\leftarrow$ $V[k+1]$
    Else
      x $\leftarrow$ $V[k-1]+1$
    y $\leftarrow$ $x-k$
    While x < N **and** y < M **and** $a_{x+1} = b_{y+1}$ **Do (x,y) $\leftarrow$ (x+1,y+1)**
    $V[k]$ $\leftarrow$ **x**
    If x $\geq$ N and y $\geq$ M Then
      Length of an SES is D
      Stop
Length of an SES is greater than MAX

FIGURE 2: The Greedy LCS/SES Algorithm

As a practical matter the algorithm searches D-paths where D≤MAX and if no such path reaches (N,M) then it reports that any edit script for A and B must be longer than MAX in Line 14. By setting the constant MAX to M+N as in the outline above, the algorithm is guaranteed to find the length of the LCS/SES. Figure 3 illustrates the D-paths searched when the algorithm is applied to the example of Figure 1. Note that a fictitious endpoint, $(0,-1)$, set up in Line 1 of the algorithm is used to find the endpoint of the furthest reaching 0-path. Also note that D-paths extend off the left and lower boundaries of the edit graph proper as the algorithm progresses. This boundary situation is correctly handled by assuming that there are no diagonal edges in this region.

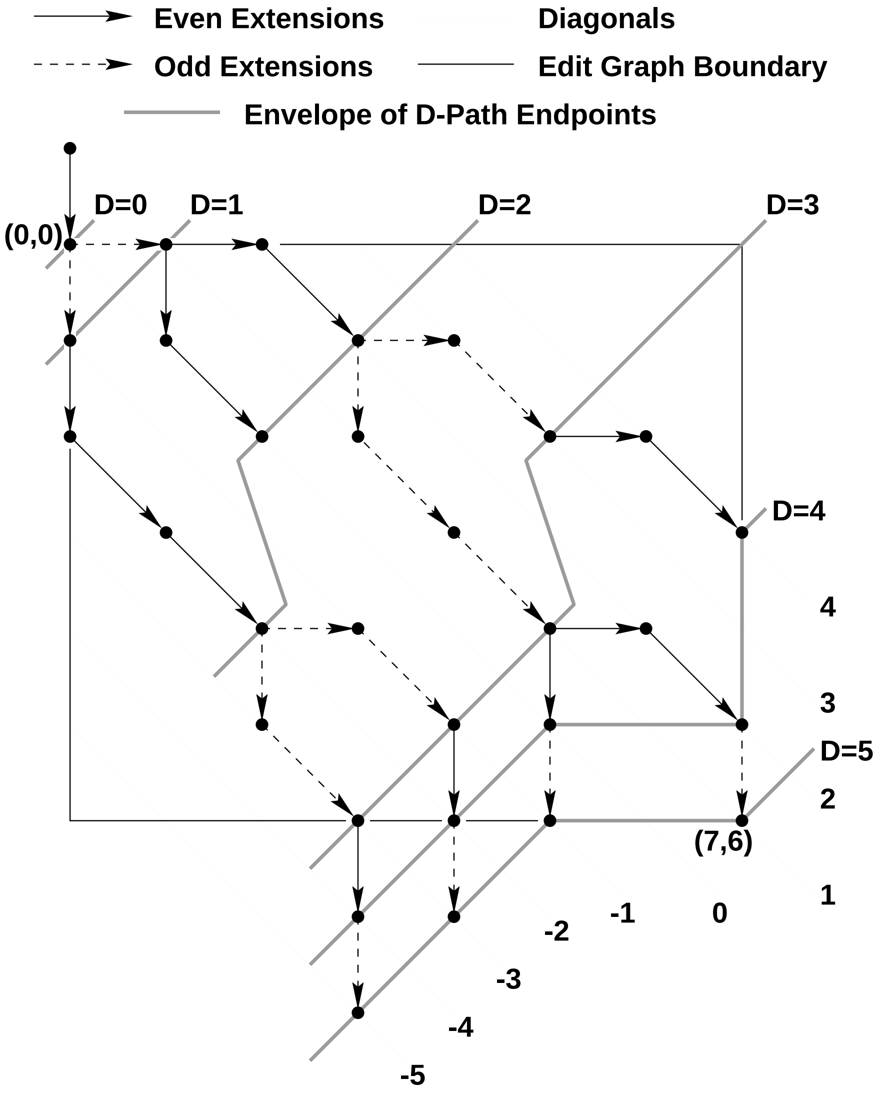

Fig. 3. Furthest reaching paths.

The greedy algorithm takes at most O((M+N)D) time. Lines 1 and 14 consume O(1) time. The inner *For* loop (Line 3) is repeated at most (D+1)(D+2)/2 times because the outer *For* loop (Line 3) is repeated D+1 times and during its k^{th} iteration the inner loop is repeated at most k times. All the lines within this inner loop take constant time except for the *While* loop (Line 9). Thus $O(D^2)$ time is spent executing Lines 2-8 and 10-13. The *While* loop is iterated once for each diagonal traversed in the extension of furthest reaching paths. But at most O((M+N)D) diagonals are traversed since all D-paths lie between diagonals $-D$ and D and there are at most (2D+1)min(N,M) points within this band. Thus the algorithm requires a total of O((M+N)D) time. Note that just Line 9, the traversal of snakes, is the limiting step. The rest of the algorithm is $O(D^2)$. Furthermore the algorithm never takes more than O((M+N)MAX) time in the practical case where the threshold MAX is set to a value much less than M+N.

The search of the greedy algorithm traces the optimal D-paths among others. But only the current set of furthest reaching endpoints are retained in V. Consequently, only the length of an SES/LCS can be reported in Line 12. To explicitly generate a solution path, $O(D^2)$ space* is used to store a copy of V after each iteration of the outer loop. Let V_d be the copy of V kept after the d^{th} iteration. To list an optimal path from (0,0) to the point $V_d[k]$ first determine whether it is at the end of a maximal snake following a vertical edge from $V_{d-1}[k+1]$ or a horizontal edge from $V_{d-1}[k-1]$. To be concrete, suppose it is $V_{d-1}[k-1]$. Recursively list an optimal path from (0,0) to this point and then list the vertical edge and maximal snake to $V_d[k]$. The recursion stops when d = 0 in which case the snake from (0,0) to $(V_0[0], V_0[0])$ is listed. So with O(M+N) additional time and $O(D^2)$ space an optimal path can be listed by replacing Line 12 with a call to this recursive procedure with $V_D[N-M]$ as the initial point. A refinement requiring only O(M+N) space is shown in the next section.

As noted in Section 2, the LCS/SES problem can be viewed as an instance of the single-source shortest paths problem on a weighted edit graph. This suggests that an efficient algorithm can be obtained by specializing Dijkstra's algorithm [3]. A basic exercise [2: 207-208] shows that the algorithm takes O(ElgV) time where E is the number of edges and V is the number of vertices in the subject graph. For an edit graph E < 3V since each point has outdegree at most three. Moreover, the lgV term comes from the cost of managing a priority queue. In the case at hand the priorities will be integers in [0,M+N] as edge costs are 0 or 1 and the longest possible path to any point is M+N. Under these conditions, the priority queue operations can be implemented in constant time using ''bucketing'' and linked-list techniques. Thus Dijkstra's algorithm can be specialized to perform in time linear in the number of vertices in the edit graph, i.e. O(MN). The final refinement stems from noting that all that is needed is the shortest path from the source (0,0) to the point (M,N). Dijkstra's algorithm determines the minimum distances of vertices from the source in increasing order, one vertex per iteration. By Lemma 1 there are at most O((M+N)D) points less distant from (0,0) than (M,N) and the previous refinements reduce the cost of each iteration to O(1). Thus the algorithm can stop as soon as the minimum distance to (M,N) is ascertained and it only spends O((M+N)D) time in so doing.

It has been shown that a specialization of Dijkstra's algorithm also gives an O(ND) time algorithm for the LCS/SES problem. However, the resulting algorithm involves a relatively complex discrete priority queue and this queue may contain as many as O(ND) entries even in the case where just the length of the LCS/SES is being computed. While one could argue that further refinement leads to the simple algorithm of this paper, the connection becomes so tenuous that the direct and easily motivated derivation used in this section is preferable. The aim of the discussion is to expose the close relationship between the shortest paths and LCS/SES problems and their algorithms.

---

* If only $O(D^2)$ space is to be *allocated*, the algorithm is first run to determine D in O(N) space, then the space is allocated, and finally, the algorithm is run again to determine a solution path.

4. Refinements

The basic algorithm can be embellished in a number of ways. First, the algorithm's expected performance is $O(M+N+D^2)$, which is much superior to the worst case prediction of O((M+N)D). While not shown here, experiments reveal that the variance about the mean is small especially as the alphabet size becomes large. Thus while there are pathological cases that require O((M+N)D) time they are extremely rare (e.g. like $O(N^2)$ problems for quicksort). Second, the algorithm can be refined to use only linear space when reporting an edit script. The only other algorithm that has been shown to admit such a refinement is the basic O(MN) dynamic programming algorithm [7]. A linear space algorithm is of practical import since many large problems can reasonably be solved in $O(D^2)$ time but not in $O(D^2)$ space. Finally, an $O((M+N)\lg(M+N) + D^2)$ worst-case time variation is obtained by speeding up the traversal of snakes with some previously developed techniques [6,14]. The variation is impractical due to the sophistication of these underlying methods but its superior asymptotic worst-case complexity is of theoretical interest.

4a. A Probabilistic Analysis

Consider the following stochastic model for the sequences A and B in a shortest edit script problem. A and B are sequences over an alphabet Σ where each symbol occurs with probability p_σ for $\sigma \in \Sigma$. The N symbols of A are randomly and independently chosen according to the probability densities, p_σ. The $M = N-\delta+\iota$ symbol sequence B is obtained by randomly deleting δ symbols from A and randomly inserting ι randomly chosen symbols. The deletion and insertion positions are chosen with uniform probability. An equivalent model is to generate a random sequence of length $L = N-\delta$ and then randomly insert δ and ι randomly generated symbols to this sequence to produce A and B, respectively. Note that the LCS of A and B must consist of at least L symbols but may be longer.

An alternate model is to consider A and B as randomly generated sequences of length N and M which are constrained to have an LCS of length L. This model is not equivalent to the one above except in the limit when the size of Σ becomes arbitrarily large and every probability p_σ goes to zero. Nonetheless, the ensuing treatment can also be applied to this model with the same asymptotic results. The first model is chosen as it reflects the edit scripts for mapping A into B that are assumed by the SES problem. While other edit script commands such as ''transfers'', ''moves'', and ''exchanges'' are more reflective of actual editing sessions, their inclusion results in distinct optimization problems from the SES problem discussed here. Hence stochastic models based on such edit process are not considered.

In the edit graph of A and B there are L diagonal edges corresponding to the randomly generated LCS of A and B. Any other diagonal edge, ending at say (x,y), occurs with the same probability that $a_x = b_y$ as these symbols were obtained by independent random trials. Thus the probability of an *off-LCS diagonal* is $\rho = \sum_{\sigma \in \Sigma} p_\sigma^2$. The SES algorithm searches by extending furthest reaching paths until the point (N,M) is reached. Each extension consists of a horizontal or vertical edge followed by the longest possible snake. The maximal snakes consist of a number of LCS and off-LCS diagonals. The probability that there are exactly t off-LCS diagonals in a given extension's snake is $\rho^t(1-\rho)$. Thus the expected number of off-LCS diagonals in an extension is $\sum_{t=0}^{\infty} t\rho^t(1-\rho) = \rho/(1-\rho)$. At most d+1 extensions are made in the d^{th} iteration of the outer *For* loop of the SES algorithm. Therefore at most $(D+1)(D+2)\rho/2(1-\rho)$ off-LCS diagonals are traversed in the expected case. Moreover, at most L LCS diagonals are ever traversed. Consequently, the critical *While* loop of the algorithm is executed an average of $O(L+D^2)$

times when ρ is bounded away from 1. The remainder of the algorithm has already been observed to take at worst $O(D^2)$ time. When ρ = 1, there is only one letter of nonzero probability in the alphabet Σ, so A and B consists of repetitions of that letter, with probability one. In this case, the algorithm runs in O(M+N) time. Thus the SES Algorithm takes $O(M+N+D^2)$ time in the expected case.

4b. A Linear Space Refinement

The LCS/SES problem is symmetric with respect to the orientation of edit graph edges. Consider reversing the direction of every edge in the edit graph for sequences A and B. Subsequences and edit scripts for A and B are still modeled as paths in this reverse edit graph but now the paths start at (N,M) and end at (0,0). Also, the interpretation of paths alters just slightly to reflect the reversal of direction. Each diagonal edge *beginning* at (x,y) gives a symbol, a_x (= b_y), in the common subsequence; each horizontal edge *from* point (x,y) corresponds to the delete command ‘‘x D’’; etc. So the LCS/SES problem can be solved by starting at (N,M) and progressively extending furthest reaching paths in the reverse edit graph until one reaches (0,0). Hereafter, forward paths will refer to those in the edit graph and reverse paths will refer to those in the reverse edit graph. Since paths in opposing directions are in exact correspondence, the direction of a path is distinguished only when it is of operational importance.

As in the linear space algorithm of Hirschberg [7], a divide-and-conquer strategy is employed. A D-path has D+1 snakes some of which may be empty. The divide step requires finding the $\lceil D/2 \rceil + 1$ or middle snake of an optimal D-path. The idea for doing so is to simultaneously run the basic algorithm in both the forward and reverse directions until furthest reaching forward and reverse paths starting at opposing corners ‘‘overlap’’. Lemma 3 provides the formal observation underlying this approach.

Lemma 3: There is a D-path from (0,0) to (N,M) if and only if there is a $\lceil D/2 \rceil$-path from (0,0) to some point (x,y) and a $\lfloor D/2 \rfloor$-path from some point (u,v) to (N,M) such that:

(feasibility) **u+v ≥ $\lceil D/2 \rceil$ and x+y ≤ N+M–$\lfloor D/2 \rfloor$, and**
(overlap) **x–y = u–v and x ≥ u.**

Moreover, both D/2-paths are contained within D-paths from (0,0) to (N,M).

Proof:

Suppose there is a D-path from (0,0) to (N,M). It can be partitioned at the start, (x,y), of its middle snake into a $\lceil D/2 \rceil$-path from (0,0) to (x,y) and a $\lfloor D/2 \rfloor$-path from (u,v) to (N,M) where (u,v) = (x,y). A path from (0,0) to (u,v) can have at most u+v non-diagonal edges and there is a $\lceil D/2 \rceil$-path to (u,v) implying that u+v ≥ $\lceil D/2 \rceil$. A path from (x,y) to (N,M) can have at most (N+M)–(x+y) non-diagonal edges and there is a $\lfloor D/2 \rfloor$-path to (x,y) implying that x+y ≤ N+M–$\lfloor D/2 \rfloor$. Finally, u–v = x–y and u≤x as (x,y) = (u,v).

Conversely, suppose the $\lceil D/2 \rceil$- and $\lfloor D/2 \rfloor$-paths exist. But u≤x implies there is a k-path from (0,0) to (u,v) where k≤$\lceil D/2 \rceil$. By Lemma 1, Δ = $\lceil D/2 \rceil$–k is a multiple of 2 as both the k-path and $\lceil D/2 \rceil$-path end in the same diagonal. Moreover, the k-path has (u+v–k)/2 ≥ Δ/2 diagonals as u+v ≤ $\lceil D/2 \rceil$. By replacing each of Δ/2 of the diagonals in the k-path with a pair of horizontal and vertical edges, a $\lceil D/2 \rceil$-path from (0,0) to (u,v) is obtained. But then there is a D-path from (0,0) to (N,M) consisting of this $\lceil D/2 \rceil$-path to (u,v) and the the given $\lfloor D/2 \rfloor$-path from (u,v) to (N,M). Note that the $\lfloor D/2 \rfloor$-path is part of this D-path. By a symmetric argument the $\lceil D/2 \rceil$-path is also part of a D-path from (0,0) to (N,M). □

The outline below gives the procedure for finding the middle snake of an optimal path. For successive values of D, compute the endpoints of the furthest reaching forward D-paths from (0,0) and then compute the furthest reaching reverse D-paths from (N,M). Do so in V vectors, one for each direction, as in the basic algorithm. As each endpoint is computed, check to see if it overlaps with the path in the same diagonal but opposite direction. A check is needed to ensure that there is an opposing path in the given diagonal because forward paths are in diagonals centered about 0 and reverse paths are in diagonals centered around Δ = N−M. Moreover, by Lemma 1, the optimal edit script length is odd or even as Δ is odd or even. Thus when Δ is odd, check for overlap only while extending forward paths and when Δ is even, check for overlap only while extending reverse paths. As soon as a pair of opposing and furthest reaching paths overlap, stop and report the overlapping snake as the middle snake of an optimal path. Note that the endpoints of this snake can be readily delivered as the snake was just computed in the previous step.

> $\Delta \leftarrow$ **N−M**
> **For D $\leftarrow$ 0 to** $\lceil (M+N)/2 \rceil$ **Do**
>   **For k $\leftarrow$ −D to D in steps of 2 Do**
>   **Find the end of the furthest reaching forward D-path in diagonal k.**
>   **If Δ is odd and** $k \in [\Delta-(D-1), \Delta+(D-1)]$ **Then**
>   **If the path overlaps the furthest reaching reverse** $(D-1)$**-path in diagonal k Then**
>   **Length of an SES is 2D−1.**
>   **The last snake of the forward path is the middle snake.**
>   **For k $\leftarrow$ −D to D in steps of 2 Do**
>   **Find the end of the furthest reaching reverse D-path in diagonal k+Δ.**
>   **If Δ is even and** $k+\Delta \in [-D, D]$ **Then**
>   **If the path overlaps the furthest reaching forward D-path in diagonal k+Δ Then**
>   **Length of an SES is 2D.**
>   **The last snake of the reverse path is the middle snake.**

The correctness of this procedure relies heavily on Lemma 3. Without loss of generality suppose Δ is even. The algorithm stops as soon as the smallest D is encountered for which furthest reaching D-paths in opposite directions overlap. First, the overlapping paths must be shown to satisfy the feasibility condition of Lemma 3. Suppose the reverse furthest reaching path ends at (u,v) where u+v = k. There is always a k-path of non-diagonal edges from (0,0) to (u,v) which when combined with the reverse D-path forms a k+D-path from (0,0) to (N,M). This path and Lemma 3 imply there are overlapping h-paths where h = (k+D)/2 (k+D is divisible by 2 as Δ is even). So certainly there are overlapping furthest reaching h-or-less paths. If k < D, then h < D contradicting the fact that furthest reaching D-paths are the first to overlap. So u+v ≥ D as desired. A similar argument shows that the furthest reaching forward D-path also satisfies the feasibility constraint of Lemma 3. Now the feasible, overlapping D-paths and Lemma 3 imply that there is a solution path of length 2D. This must be optimal for if there is a 2k-path, k<D, then the lemma implies there are k-paths that overlap. But this implies that there are overlapping furthest reaching k-or-less paths, contradicting the fact that there are no overlapping paths for smaller D. Thus by Lemma 3, both of the overlapping paths are parts of optimal 2D-paths from (0,0) to (N,M). The same conclusion can be drawn when Δ is odd. It remains to show that the chosen snake is the middle snake of the optimal path of which it is a part. If Δ is

odd(even) then this snake is the $D+1^{st}$ of a forward(reverse) path which is part of an optimal 2D−1(2D)-path. In either case the snake is the $\lceil k/2 \rceil + 1^{st}$ of an optimal k-path containing it.

The procedure for finding the middle snake of an optimal D-path requires a total of O(D) working storage for the two V vectors. The procedure only requires O((M+N)D) time because the forward and reverse path extension portions both consume O((M+N)D) time by the same argument used for the basic algorithm. In fact, the number of snakes traversed in extending forward and reverse paths can be *half* that of the basic algorithm because only D/2 diagonals are searched. This feature is of practical import — experiments reveal that this procedure for determining the length of an SES is as efficient as the basic algorithm when D = 0 and rapidly becomes twice as fast as D increases.

Given the middle snake procedure, a linear space algorithm for finding an optimal path through the edit graph of A and B of sizes N and M can be devised. For simplicity, the divide-and-conquer algorithm below just lists a longest common subsequence of A and B. (Producing a shortest edit script is left as an exercise.) First, divide the problem by finding a middle snake from, say, (x,y) to (u,v) of an optimal D-path. Begin conquering the problem, by recursively finding a $\lceil D/2 \rceil$-path from (0,0) to (x,y) and listing its LCS. Then list the middle snake (''Output A[x..u]'' lists nothing if u < x). Finally, recursively find the $\lfloor D/2 \rfloor$-path from (u,v) to (N,M) and list its LCS. The recursion ends in two ways. If N=0 or M=0 then L=0 and there is nothing to list. In the other case, N>0 and M>0 and D≤1. If D≤1 then B is obtained from A by either deleting or inserting at most one symbol. But then it follows that the shorter of A and B is the LCS and should be listed.

```
LCS(A,N,B,M)
    If N>0 and M>0 Then
        Find the middle snake and length of an optimal path for A and B.
        Suppose it is from (x,y) to (u,v).
        If D > 1 Then
            LCS(A[1..x],x,B[1..y],y)
            Output A[x+1..u].
            LCS(A[u+1..N],N−u,B[v+1..M],M−v)
        Else If M > N Then
            Output A[1..N].
        Else
            Output B[1..M].
```

Let T(P,D) be the time taken by the algorithm where P is N+M. It follows that T satisfies the recurrence inequality:

$$T(P,D) \le \begin{cases} \alpha PD + T(P_1, \lceil D/2 \rceil) + T(P_2, \lfloor D/2 \rfloor) & \text{if } D > 1 \\ \beta P & \text{if } D \le 1 \end{cases}$$

where $P_1 + P_2 \le P$ and α and β are suitably large constants. Noting that $\lceil D/2 \rceil \le 2D/3$ for D≥2, a straightforward induction argument shows that T(P,D) ≤ 3αPD+βP. Thus the divide-and-conquer algorithm still takes just O((M+N)D) time despite the $\lceil \lg D \rceil$ levels of recursion through which it descends. Furthermore, the algorithm only requires O(D) working storage. The middle snake procedure requires two O(D) space V vectors. But this step is completed before engaging in the recursion. Thus only one pair of global V vectors are shared by all invocations of the procedure. Moreover, only O(lgD) levels of recursion are traversed implying that only O(lgD) storage is needed on the recursion stack. Unfortunately, the input sequences A and B must be kept in memory, implying that a total of O(M+N) space is needed.

4c. An $O((M+N)\lg(M+N) + D^2)$ Worst-Case Variation

The final topic involves two previous results, each of which are just sketched here. First, suffix trees [12,14] are used to efficiently record the common sublists of the sequences being compared. The term sublist is used as opposed to subsequence to emphasize that the symbols must be contiguous. Second, a recent RAM-based algorithm for anwering Q on-line queries for the lowest common ancestors of vertices in a fixed V-vertex tree takes O(V+Q) time [6]. The efficient variation centers on quickly finding the length or endpoint of a maximal snake starting at point (x,y). This is shown to reduce to finding the lowest common ancestor of two leaves in a suffix tree. This can be done in O((M+N)lg(M+N)) pre-processing time and O(1) time per query using the two techniques above. The ensuing paragraphs embellish these ideas.

A suffix or Patricia tree [12,14] for a sequence S of length L has edges labelled with sublists of S, has L leaves labelled with the positions of S, and satisfies the following three properties.

1. Concatenating the edge labels traversed on the path from the root to the leaf for position j, gives the suffix, S[j..L], of S starting at j. Thus every path within the tree denotes a sublist of S.
2. Every interior vertex has out-degree greater than one.
3. The labels of the out-edges of every vertex begin with distinct symbols.

These properties can only be satisfied if the last symbol of S is distinct from every other symbol in S. This condition is usually met by appending a special symbol to the target sequence and once satisfied, the suffix tree is unique. Property 2 guarantees that there are less than L interior vertices. Moreover, the substrings labelling edges can be represented by just storing indices to their first and last characters in S. Thus suffix trees can be stored in O(L) space. The efficient construction of suffix trees is beyond the scope of this paper. The reader is referred to a paper by McCreight [14] giving an algorithm that constructs a suffix tree in O(L) steps. Most of the steps are easily done in O(1) time but some require selecting an out-edge based on its first symbol. When the alpabet is finite, the out-degree of vertices is finite and the selection takes O(1) time. When the alphabet is unrestricted, height-balanced trees or some other worst-case efficient search structure permits selection in O(lgL) time. Thus suffix tree construction takes O(L) time for finite alphabets and O(LlgL) time otherwise.

Consider the two paths from the root of S's suffix tree to leaves i and j. Each path from the root to a common ancestor of i and j, denotes a common prefix of the suffixes S[i..L] and S[j..L]. From Property 3 it follows that the path to the lowest common ancestor of i and j, denotes the *longest* prefix of their respective suffixes. This observation motivates the following suffix tree characterization of the maximal snake starting at point (x,y) in the edit graph of A and B of lengths N and M respectively. Form the position tree for the sequence S = A.$\$_1$.B.$\$_2$ where the symbols $\$_1$ and $\$_2$ are not equal to each other or any symbol in A or B. The maximal snake starting at (x,y) is denoted by the path from the root of S's suffix tree to the lowest common ancestor of positions x and y+N+1. This follows because neither $\$_1$ or $\$_2$ can be a part of this longest common prefix for the suffixes A[x..N].$\$_1$.B.$\$_2$ and

B[y..M].$\$_2$. So to find the endpoint of a snake starting at (x,y), find the lowest common ancestor of leaves x and y+N+1 in the suffix tree and return (x+m,y+m) where m is the length of the sublist denoted by the path to this ancestor. In a linear preprocessing pass the sublist lengths to every vertex are computed and the auxiliary structures needed for the O(V+Q) lowest common ancestor algorithm of Harel and Tarjan [6] are constructed. This RAM-based algorithm requires O(V) preprocessing time but can then answer each on-line query in O(1) time. Thus with O((M+N)lg(M+N)) preprocessing time (building the suffix tree is the dominant cost), a collection of on-line queries for the endpoints of maximal snakes can be answered in O(1) time per query.

Modify the basic algorithm of Section 3 by (a) prefacing it with the preprocessing needed for the maximal snake queries and (b) replacing Line 9 with the O(1) query primitives. Recall that every line in the innermost loop other than Line 9 is O(1) and that the loop is repeated $O(D^2)$ times. Now that Line 9 takes O(1) time it follows that this modification results in an algorithm that runs in $O(\ (M+N)\lg(M+N)\ +\ D^2\ )$ time. Note that this variation is primarily of theoretical interest. The coefficients of proportionality are much larger for the algorithm fragments employed implying that problems will have to be very large before the variation becomes faster. But suffix trees are particularly space inefficient and two auxiliary trees of equal size are needed for the fast lowest common ancestor algorithm. Thus for problems large enough to make the time savings worthwhile it is likely that there will not enough memory to accomodate these additional structures.

Acknowledgements

Webb Miller originally proposed the problem of finding an O(ND) algorithm. The author would like to thank him for nurturing this work and his many helpful suggestions. The referees comments and corrections improved the paper greatly.

References

1. Aho, A.V., Hirschberg, D.S., and Ullman, J.D. ‘‘Bounds on the Complexity of the Longest Common Subsequence Problem.’’ *Journal of ACM 23*, 1 (1976), 1-12.
2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass. (1983), 203-208.
3. Dijkstra, E.W. ‘‘A Note on Two Problems in Connexion with Graphs.’’ *Numerische Mathematik 1* (1959), 269-271.
4. Gosling, J. ‘‘A Redisplay Algorithm.’’ *Proceedings ACM SIGPLAN/SIGOA Symposium on Text Manipulation* (1981), 123-129.
5. Hall, P.A.V. and Dowling, G.R. ‘‘Approximate String Matching.’’ *Computing Surveys 12*, 4 (1980), 381-402.
6. Harel, D. and Tarjan, R.E. ‘‘Fast Algorithms for Finding Nearest Common Ancestors.’’ *SIAM Journal on Computing 13*, 2 (1984), 338-355.
7. Hirschberg, D.S. ‘‘A Linear Space Algorithm for Computing Maximal Common Subsequences.’’ *Communications of ACM 18*, 6 (1975), 341-343.
8. Hirschberg, D.S. ‘‘Algorithms for the Longest Common Subsequence Problem.’’ *Journal of ACM 24*, 4 (1977), 664-675.
9. Hirschberg, D.S. ‘‘An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem.’’ *Information Processing Letters 7*, 1 (1978), 40-41.
10. Hunt, J.W. and McIlroy, M.D. ‘‘An Algorithm for Differential File Comparison.’’ Computing Science Technical Report 41, Bell Laboratories (1975).

11. Hunt, J.W. and Szymanski, T.G. ‘‘A Fast Algorithm for Computing Longest Common Subsequences.’’ *Communications of ACM 20*, 5 (1977), 350-353.

12. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass. (1983), 490-493.

13. Masek, W.J. and Paterson, M.S. ‘‘A Faster Algorithm for Computing String Edit Distances.’’ *J. of Computer and Systems Sciences 20*, 1 (1980), 18-31.

14. McCreight, E.M. ‘‘A Space-Economical Suffix Tree Construction Algorithm.’’ *Journal of ACM 23*, 2 (1976), 262-272.

15. Miller, W., and Myers, E.W. ‘‘A File Comparison Program.’’ *Software — Practice & Experience 15*, 11 (1985), 1025-1040.

16. Nakatsu, N., Kambayashi, Y., and Yajima, S. ‘‘A Longest Common Subsequence Algorithm Suitable for Similar Text Strings.’’ *Acta Informatica 18* (1982), 171-179.

17. Rochkind, M.J. ‘‘The Source Code Control System.’’ *IEEE Transactions on Software Engineering 1*, 4 (1975), 364-370.

18. Sankoff, D. and Kruskal, J.B. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass. (1983).

19. Tichy, W. ‘‘The String-to-String Correction Problem with Block Moves.’’ *ACM Transactions on Computer Systems 2*, 4 (1984), 309-321.

20. Wagner, R.A. and Fischer, M.J. ‘‘The String-to-String Correction Problem.’’ *Journal of ACM 21*, 1 (1974), 168-173.