# Shortest Path Approaches for the Longest Common Subsequence of a Set of Strings

Marina Barsky*, Ulrike Stege*, Alex Thomo*, and Chris Upton†
*Department of Computer Science, University of Victoria, Canada
Email: {mgbarsky,stege,thomo}@cs.uvic.ca
†Department of Biochemistry & Microbiology, University of Victoria, Canada
Email: cupton@uvic.ca

***Abstract*—We investigate the $k$-LCS problem that is finding a longest common subsequence (LCS) for $k$ given input strings. The problem is known to have practical solutions for $k = 2$, but for higher dimensions it is not very well explored. We consider the algorithms by Miller and Myers as well as Wu et al. which solve the 2-LCS problem, and shed a new light on their generalization to higher dimensions. First, we redesign both algorithms such that the generalization to higher dimensions becomes natural. Then we present our algorithms for solving the $k$-LCS problem. We further propose a new approach to reduce the algorithms' space complexity. We demonstrate that our algorithms are practical as they significantly outperform the dynamic programming approaches. Our results stand in contrast to observations made in previous work by Irving and Fraser.**

## I. INTRODUCTION

The longest common subsequence problem is finding a longest sequence which is a subsequence of all strings in a given set of strings. For $k$ strings, the problem is formalized as follows.

**The $k$-LCS Problem**
INPUT: *$k$ strings $s_1, s_2, ..., s_k$ over a finite alphabet $\Sigma$ and of lengths $|s_1|, |s_2|, ..., |s_k|$ respectively.*
OUTPUT: *A longest common subsequence ($LCS$) of the input strings including*

1) *the LCS length $|LCS|$,*
2) *the actual sequence of characters in the $LCS$, and*
3) *the corresponding positions of these characters in the input strings.*

Formulated this way, the $LCS$ problem may serve many practical purposes, for example the $LCS$-based multiple alignment problem [8].

Given two input strings of length $N$ each, a simple dynamic-programming approach yields a time and space complexity of $O(N^2)$ (cf. [18], [23]). A divide-and-conquer variant by Hirschenberg yields a linear space algorithm while preserving the time complexity of $O(N^2)$ [9].

For arbitrary $k$, Maier showed that the problem is NP-complete even for alphabets of size two [15]. Despite its intractability, practical solutions for this fundamental problem are of crucial importance in several fields of computer science where the problem of string comparison arises, e.g. human speech recognition [5], information compression and retrieval [7], codes and error control [23]) and so on. Solving $k$-LCS is even more important in the field of biological data mining and biological sequence analysis. The $k$-LCS itself is an important special case of the Multiple Sequence Alignment (MSA), which is a fundamental problem in bioinformatics (cf. [8], [24]).

$k$-LCS is consistently used for the comparison of strings belonging to the same family [6] or for the computation of consensus patterns in a set of DNA sequences [4]. Remarkably, $k$-LCS is helpful in answering several important questions related to the common structural configuration of a family of macromolecules. We also want to mention here some recent bioinformatics applications using $k$-LCS as an integral part of their method.

One such application of $k$-LCS was shown in the work of Bereg et al. [3]. The authors investigate the problem of folding of noncoding RNA (ncRNA)[1]. Folding of ncRNA into secondary and tertiary structures can be better understood if one takes a set of ncRNAs and looks at their common folding patterns. Notably, Bereg et al. use the solution to $k$-LCS to derive these common folding patterns. Another recent important application of $k$-LCS was shown by Ning et al. in [19]. They derive common patterns from a set of biosequences by using $k$-LCS and $k$-SCS (shortest common supersequence). Both works ([3], [19]) use $k$-LCS as an input for their methods. Due to impracticality and expensiveness of existing algorithms for $k$-LCS, the authors of these papers use heuristic methods for computing $k$-LCS which do not guarantee the output to be a truly longest common subsequence. Therefore, the results based on such non-truly longest common subsequences might be non-reliable and even misleading.

In this paper, we investigate whether better practical solutions for $k$-LCS can be obtained without sacrificing the optimality of the result.

For solving $k$-LCS, a straightforward extension of the dynamic-programming approach to the case of $k \geq 3$ results in an algorithm of $O(N^k)$ time and space complexity (assuming input strings of length $N$). Such an extension was shown to be impractical for inputs having moderate values of $N$ and $k$ (cf. [13]).

Several attempts were undertaken by researchers in order to present a viable solution for the $k$-LCS problem ($k \geq 3$).

[1]The ncRNA molecules are those RNA molecules that are not translated into a protein.

Some of the most important works in this direction are [1], [10], [11], [12].

In such applications as aligning biological sequences of the same family, the input strings are typically similar. Notably, for 2-LCS the proposed algorithms by Miller and Myers [16] and its variant by Wu et al. [25] have a favorable performance in the case of similar strings. We remark that the latter was extended for 3-LCS by Irving and Fraser in [12]. However, they experimentally found that their algorithm outperformed the standard dynamic programming only when $|LCS|$ is very close to $N$ (e.g. 90% of $N$).

In this paper, we reinvestigate the approach by Miller and Myers. Based on it, as well as on the Wu et al. variant, we devise algorithms for 3-LCS, and show that they are easily extendable for $k$-LCS ($k > 3$). We further show that our algorithms for 3-LCS significantly outperform dynamic programming even when having an $|LCS|$ of only 50% of the input length. More specifically, the contributions of this paper are as follows. First, we redesign the 2-LCS algorithms by Miller and Myers as well as Wu et al., thereby exposing their potential towards a natural extension for $k$-LCS. Second, we propose efficient $k$-LCS algorithms based on both methods. Third, we present a new technique to reduce the space complexity of our algorithms by an order of magnitude. Finally, with experimental results we show that our 3-LCS algorithms significantly outperform dynamic programming. This sheds a new light on the practicality of extending the algorithms by Miller and Myers as well as Wu et al. for the $k$-LCS problem. Our results stand in contrast to the observations by Irving and Fraser in [12], and thus, we reopen the research in finding feasible optimal solutions to the $k$-LCS problem.

## II. Shortest Path Algorithms for the 2-LCS Problem

In this section, we redesign the 2-LCS algorithm by Miller and Myers [16] (MM) and its variation by Wu et al. [25] (W3M) in a way that allows a natural extension to algorithms for $k$-LCS.

### A. Basic Concepts

The *edit graph* for two input strings $s_1$ and $s_2$ is a grid-like directed graph, where the coordinates of the $X$- and $Y$-axes correspond to the positions of characters in the first and second string. Each point $(x, y) \in [0, 1, \ldots, |s_1|] \times [0, 1, \ldots, |s_2|]$ in the plane represents a node of the edit graph. Every node has at least two outgoing edges (one in the horizontal and one in the vertical direction). Such an edge corresponds to the deletion of one character from either $s_1$ (horizontal edge) or $s_2$ (vertical edge). The horizontal (vertical) edges are directed from left to right (top to bottom). In addition, a node $(x-1, y-1)$ has a third outgoing (diagonal) edge going in the right-bottom direction iff $s_1[x] = s_2[y]$. (cf. Figure 1 [left] for an example).

Several cost assignments for edit graphs are possible. If we assign a cost of 0 to both the vertical and horizontal edges, and a cost of 1 to all diagonal edges, the 2-LCS problem is reduced to finding a longest path in the edit graph (cf. [8]).

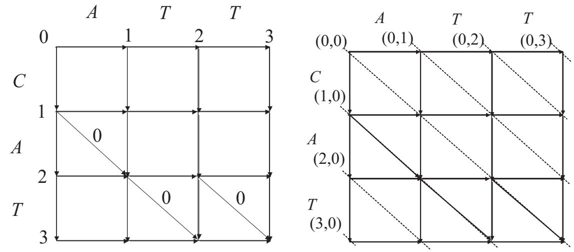



Fig. 1. [Left] The 2D edit graph for input strings *CAT* and *ATT*. The costs of the horizontal and vertical edges are one, and the three 0-edges correspond to matches. [Right] The diagonals and their identifiers in the edit graph.

If we instead assign opposite costs to the edges (that is, 0 to diagonal edges, and 1 to horizontal and vertical edges), the LCS problem is reduced to the problem of finding a shortest path from source node $(0, 0)$ to destination node $(|s_1|, |s_2|)$. The cost $D$ of this shortest path equals the minimum total number of deletions necessary to transform the input strings into the $LCS$. In this case,

$$|LCS| = (|s_1| + |s_2| - D)/2.$$

### B. The algorithm by Miller and Myers (MM)

The latter edge-cost scheme is used by Miller and Myers in their algorithm (MM), described in [16]. We denote edges of cost 1 with 1-*edges* and edges of cost 0 with 0-*edges*.

Algorithm MM works in a diagonal-wise manner. A diagonal in the grid is defined by a sequence of nodes with coordinates $(x, y), (x+1, y+1), \ldots, (x+p, y+p)$. For further convenient use in $k$ dimensions, we identify each diagonal by the coordinates of its starting point. Namely, diagonal $(0, 0)$ is the *main diagonal*, diagonal $(1, 0)$ is the diagonal starting at $(1, 0)$, etc.

We define the *neighbor diagonals* of diagonal $(x, y)$ as the diagonals $(x-1, y)$ and $(x, y-1)$. Whenever we obtain a negative value for either coordinate of a neighbor diagonal, we normalize it to get the true *diagonal identifier*. For example, the neighbor diagonals for diagonal $(3, 0)$ are $(2, 0)$ and $(3, -1)$. The latter is normalized to $(4, 0)$.

We perform an initialization and $D$ iterations.

1) In the initialization phase, we build the unique path of cost 0, starting at source node $(0, 0)$ and following (as long as possible) 0-edges along the main diagonal.
2) During each iteration $I$ of the algorithm, we extend the paths built in the previous iteration obtaining all the paths starting at the source node and having cost $d = I$. These paths must end on one of the $2d + 1$ diagonals surrounding (and including) the main diagonal: if a path ends outside this area then it contains more than $d$ 1-edges.
   More specifically, for a given diagonal (call it the *current diagonal*) we do the following.
   (a). We consider the two paths which ended (in the previous iteration) on the two neighbor diagonals. We

extend them by a 1-edge each in order to reach the current diagonal.

(b). We select the furthest reaching path among them.

(c). We expand this furthest reaching path with all the possible subsequent 0-edges along the current diagonal.

3) In iteration $I = D$ the destination node is finally reached, and the path reaching it can be traced back to obtain the sequence of deletions and therefore an $LCS$ of $s_1$ and $s_2$.

In fact, only $d+1$ diagonals (out of the $2d+1$) are extended in each iteration. Such a diagonal, say $(x, y)$ with $x = 0$ or $y = 0$, satisfies $(x + y)$ mod 2 = $I$ mod 2 (see [16]).

*C. The Algorithm by Wu et al. (W3M)*

We describe next the modification of the MM algorithm by Wu et al. (W3M algorithm) [25], which is based on the shortest path heuristic proposed by Sedgewick and Vitter [21]. The use of this heuristic puts the W3M algorithm among the fastest practical algorithms solving the 2-LCS problem [2].

It is hard to see how the W3M algorithm as described in [25] can be extended to $k$ dimensions. This is due to the asymmetry introduced by the so called *compressed distance (p-value)*, which is equivalent to the estimated distance (defined below), but restricted only to one dimension.

To overcome this, we define the *estimated distance* $e = ds + dd$ of a node in the edit graph as the sum of the true distance of this node from the source ($ds$) and an optimistic estimation of the distance from this node to the destination ($dd$), which we call *heuristic destination distance*. Observe that, in each iteration, we know distance $ds$ from the source to the end node of each path built so far. The second term, $dd$, is defined as the minimum number of vertical and horizontal edges needed to return to the diagonal of the destination node. Note that when $|s_1| = |s_2|$, the destination node belongs to diagonal $(0, 0)$. The intuition behind term $dd$ is that in order to reach the destination node, at least $dd$ 1-edges must be traversed. This is because in this edit graph there is no other way of moving from diagonal to diagonal.

The main steps of the W3M algorithm for input strings of equal lengths can be described as follows.

1) The initialization phase is identical to the one in algorithm MM.
2) During each iteration $I$ of the algorithm, we build all the paths ending in nodes with estimated distance $e = 2I$. These paths must end on one of the $2I + 1$ diagonals surrounding (and including) the main diagonal. If a path ends outside this area, then the path contains more than $I$ 1-edges and its end node has $dd > I$. Since also $ds > I$ the estimated distance of the path's end node must be larger than $2I$.

In each iteration, when considering an additional diagonal increasing so $ds$ by one, we also increase $dd$ by one. This is true since increasing $ds$ by one means moving further from the destination diagonal, and this automatically means that we need one more 1-edge to reach the destination diagonal. Hence, the estimated distance of end nodes of the paths is increased by two in each iteration.

Given a diagonal, a path $\pi$ ending on this diagonal is obtained from paths $\pi_1$ and $\pi_2$ ending on the two neighbor diagonals. As in algorithm MM, $\pi$ is an extension of $\pi_1$ or $\pi_2$ whichever yields further when extending to the current diagonal. In difference to algorithm MM, one of these two paths, say $\pi_1$, is built in the same iteration, while the other one, $\pi_2$, is built in the previous iteration. The reason is that we can "trade" $dd$ for $ds$ keeping their sum $e$ constant. In other words, path $\pi$ is an extension of $\pi_1$ only if the destination distance of $\pi$'s end node is smaller than the one of $\pi_1$'s end node.

3) Once the destination node is reached in the last iteration $I_{last}$, it has estimated distance $e_{last} = 2I_{last}$. Since for the destination node $dd = 0$, we have that $e_{last} = D$ (total number of deletions), and

$$|LCS| = (|s_1|+|s_2|-e_{last})/2 = (|s_1|+|s_2|-2I_{last})/2.$$

In the case that $|s_1| \neq |s_2|$, say $|s_1| < |s_2|$, the initialization of the algorithm is different. In iteration 0, we build all the paths which end in nodes with estimated distance $\Delta = |s_2| - |s_1|$. The first path in iteration 0 is comprised of all the consecutive 0-edges from node $(0, 0)$. The end node of this path has $ds = 0$ and $dd = \Delta$. The next path we build ends on diagonal $(0, 1)$, and its end node has $ds = 1$ and $dd = \Delta - 1$. The end node of the last path, which ends on diagonal $(0, \Delta)$, has $ds = \Delta$ and $dd = 0$. This means that, already in the initialization phase, we build all $\Delta + 1$ paths with estimated distance $e = \Delta$. Next, we perform $I_{last}$ total iterations as described above, until node $(|s_1|, |s_2|)$ is reached. This time, the total cost of the best path from source to destination is:

$$E = 2I_{last} + \Delta,$$

and the length of the $LCS$ is

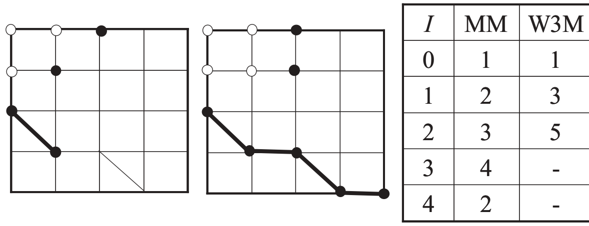
$$|LCS| = (|s_1| + |s_2| - 2I_{last} - \Delta)/2.$$

Now we explain the practical performance gain of W3M in comparison to MM. In each iteration of algorithm MM, we extend the paths built in the previous iteration. On the other hand, in algorithm W3M, in each iteration the same path can be extended several times. In practice, the W3M algorithm performs faster than the MM algorithm (cf. [2]). For an example see Figure 2.

## III. Algorithms for 3-LCS

Now, we can naturally extend the above 2D algorithms into three dimensions. This also builds the basis for solving $k$-LCS.

We define the edit graph for three input strings in a way similar to the edit graph for two input strings. This time we have three axes, $X$, $Y$, and $Z$, which are labeled with the positions of the characters in the three input strings. These axes define a 3D space. Each point in this space represents a node of the edit graph. Each node has at least three outgoing edges: an $X$-edge, a $Y$-edge and a $Z$-edge. Also, there is an



| $I$ | MM | W3M |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 3 |
| 2 | 3 | 5 |
| 3 | 4 | - |
| 4 | 2 | - |

Fig. 2. Comparative run of algorithms WM [left] and W3M [middle] for 2-LCS. We show a snapshot after the second iteration. The filled nodes illustrate the nodes visited during the second iteration, while the hollow ones were visited before. The paths built so far are shown in bold. The table [right] illustrates the number of expansion steps executed per iteration in each of the two algorithms.

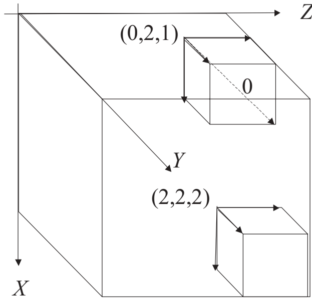



Fig. 3. The 3D edit graph for strings *ACC* ($X$-axis), *GGA* ($Y$-axis), and *TAT* ($Z$-axis). For better visibility, only the edges starting at nodes (0,2,1) and (2,2,2) are shown.

additional edge of cost zero from a node $(x-1, y-1, z-1)$ to node $x, y, z$ iff there is a match $s_1[x] = s_2[y] = s_3[z]$. This edge forms a $45°$ angle with all the three axes (See for an example Figure 3).

As before, a diagonal in 3D is defined as a sequence of points with coordinates $(x, y, z), (x+1, y+1, z+1), \ldots, (x+p, y+p, z+p)$. Each diagonal is uniquely identified by the coordinates of its starting point. The main diagonal is identified by $(0, 0, 0)$. As another example, the diagonal identified by $(1, 2, 0)$ starts at plane $XY$ and is parallel to the main diagonal.

### A. MM+: Extending the MM Algorithm to 3D

The main steps are analogous to the 2-LCS algorithm:

1) In the initialization phase, we build the (unique) 0-cost path, starting at source node $(0, 0, 0)$ and following only 0-edges on the main diagonal $(0, 0, 0)$.
2) Then, during an iteration $I$ of the algorithm, we build all paths starting at the source node and having cost $d = I$. It is easy to see that

*Theorem 1:* Any path starting at the source node and having a cost of at most $d$ can only end on one of at most $1 + 3d + 3\frac{d(d-1)}{2}$ diagonals.

*Corollary 1:* The running time of algorithm MM+ is $O(ND^2)$.

**Proof.** The claimed running time follows from the fact that the algorithm never considers nodes outside of the area of size $D^2N$ around the main diagonal. ■

Similarly to the 2D case, due to the special nature of the graph, in each iteration $I$ only the diagonals $(x, y, z)$ are considered where $(x + y + z)$ mod 3 = $I$ mod 3.

For each of these diagonals, in iteration $I$, the algorithm finds the longest reaching path ending on this (current) diagonal. The neighbor diagonals of a current diagonal, say $(x, y, z)$, are the diagonals $(x-1, y, z)$, $(x, y-1, z)$ and $(x, y, z-1)$. Whenever we obtain a negative value for either coordinate of a neighbor diagonal, we normalize to the true *diagonal identifier*. For example, the neighbor diagonals for diagonal $(3, 6, 0)$ are calculated as $(2, 6, 0)$,$(3, 5, 0)$ and $(3, 6, -1)$. The identifier $(3, 6, -1)$ is normalized to $(4, 7, 0)$.

In each iteration, a path ending on the current diagonal is built in three steps:

1) We consider the three paths which ended on neighbor diagonals in the previous iteration. These paths are extended by one 1-edge each in order to reach the current diagonal.
2) The furthest reaching path among them is selected.
3) The end of this furthest reaching path is expanded with all the possible subsequent 0-edges along the current diagonal.

In iteration $I$=$D$ the destination node is finally reached, and the path can be traced back to obtain the sequence of deletions and therefore an $LCS$ of the three strings, and

$$|LCS| = (|s_1| + |s_2| + |s_3| - D)/3.$$

### B. W3M+: Extending Algorithm W3M to 3D

The estimated distance $e$ of a node in the 3D edit graph is defined similarly to the 2D case: $e = ds + dd$.

Note that in 3D, when one $X$-edge ($Y$-edge/$Z$-edge) is added to a path increasing the distance from the source by one, then two additional edges, a $Y$-edge and a $Z$-edge (an $X$-edge and a $Z$-edge / an $X$-edge and a $Y$-edge), are needed in order to return to the main diagonal.

Here again, the initialization differs from the one of MM+ for the case of input strings with unequal lengths. Let us assume that $|s_1| \leq |s_2| \leq |s_3|$, and denote $|s_2| - |s_1|$ and $|s_3| - |s_2|$ by by $\Delta_{21}$ and $\Delta_{32}$ respectively. Then in the initialization phase, we expand the paths ending on the following $(\Delta_{21} + 1) \times (\Delta_{32} + 1)$ diagonals:

$$(0, 1, 0), \ldots, (0, \Delta_{32}, 0)$$

$$(0, 1, 1), \ldots, (0, \Delta_{32}, 1)$$

$$\ldots$$

$$(0, 1, \Delta_{21}), \ldots, (0, \Delta_{32}, \Delta_{21}).$$

In order to obtain the performance gain of algorithm W3M+, in each iteration, for any given pair of neighbor diagonals, we make sure that we first process the diagonal where the end node of the corresponding path has greater heuristic destination distance (cf. Figure 4).

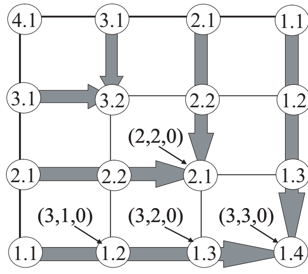



Fig. 4. Illustration of the processing order for diagonals [starting at plane $XY$] in algorithm W3M+. The first number inside the circle shows the order of outer loop iteration, the second number is the step inside this iteration. For example, diagonal $(3, 2, 0)$ is processed before diagonal $(3, 1, 0)$, and diagonal $(2, 2, 0)$ is processed before diagonal $(3, 3, 0)$. This order ensures that neighbor diagonals are processed in decreasing heuristic destination distance.

The pseudocode for the W3M+ algorithm for 3-LCS is presented in Figure 5. We use a structure, "*Frontier*," for storing the end nodes of the paths ending on each diagonal. This structure contains three 2-dimensional arrays, called $XY$, $XZ$, and $YZ$, which store the end points for the diagonals starting on the corresponding planes. In an iteration $I$, the algorithm calls routine *buildExtensions*, which builds all the paths with estimated distance $2I$. Routine *buildExtensions* calls routine *bestExtension*, which in turn computes the furthest reaching path on a given diagonal. Structure *Frontier* is used and updated by *bestExtension*. Routine *getZeroCostPath* starting from a node in a given diagonal $(x, y, z)$ traverses all the subsequent 0-edges (on the diagonal) until it reaches a node without an outgoing 0-edge. The routine returns this last node.

The worst-case running time of our W3M+ algorithm is also $O(ND^2)$. However, in practice we found that the W3M+ algorithm perfoms much better than MM+.

### C. The $k$-LCS Algorithm

Based on the detailed descriptions of the algorithms for the 2D and 3D cases, one can easily generalize for $k$-LCS.

For the complexity analysis, we generalize our result from the 3D case and obtain

*Theorem 2:* All the paths starting from the source node and having a cost of at most $d$ can only end on one in at most

$$1 + kd + k\frac{d(d-1)}{2} + k\frac{d(d-1)(d-2)}{3} + \ldots + k\frac{d(d-1)(d-2)\cdots(d-(k-2))}{k-1}$$

diagonals.

*Corollary 2:* The running time for both algorithms in $k$ dimensions is $O(ND^{k-1})$.

So far, in order to recover the sequence of deletions and therefore an $LCS$, we need $O(ND^{k-1})$ space. In the next section, we disuss how to reduce the space complexity of the two algorithms to $O(kN + D^{k-1})$.

## IV. SPACE REDUCTION

We remind the reader on the memory reduction trick by Hirschenberg that makes it possible to perform standard dynamic programming for 2-LCS in linear space (cf. [9]).

We first explain the extension of this idea for standard dynamic programming in 3D. Here, the search space is divided into two sub-spaces by the plane $x = |s_1|/2$. Then for each sub-space the cells are computed (starting from opposite corners of the table). In every iteration, all cell-values but the ones of the previous iteration are discarded. Once both values are computed (in opposite directions) for each cell of the dividing plane, we select a cell having the minimum sum of these values. This cell is on a shortest path between source and destination node. We record its coordinates, say $(x, y, z)$. We then recursively solve the two subproblems, namely for substrings $s_1[0 \ldots x], s_2[0 \ldots y], s_3[0 \ldots z]$ and for substrings $s_1[(x+1) \ldots |s_1|], s_2[(y+1) \ldots |s_2|], s_3[(z+1) \ldots |s_3|]$. The recursive calls are performed until the sizes of the sub-spaces are small enough to be solved using the memory of the available machine.

For simplicity, let us assume that $|s_1| = |s_2| = |s_3| = N$. Then given a minimim-sum cell $(x, y, z)$ in the dividing plane $x = N/2$ obtained as above, the divide step cuts the search space into two sub-spaces of size $\frac{N}{2}(N-y)(N-z)$ and $\frac{N}{2}yz$ respectively. Since $[(N-y)(N-z) + yz] \leq N^2$, we obtain a total running time of

$$N^3 + \frac{N}{2}N^2 + \frac{N}{4}N^2 + \ldots \leq 2N^3 = O(N^3).$$

Unfortunately, this approach alone cannot directly be applied to algorithms MM+ and W3M+. If we divide the space of the edit graph into two equal-sized sub-spaces and then perform one of the above algorithms starting from opposite corners of the edit graph, the cells containing the end nodes of the expanded paths may not overlap. Therefore, we are not always able to choose a cell, which is on the shortest path betweeen source and destination.

In order to overcome this problem, we combined the idea above with the bidirectional search technique presented in [17]. We call this method "divide and conquer by half-cost points."

We consider the 3D edit graph as two different edit graphs: one (called the *direct edit graph*), which is identical to the original edit graph, and another one (called the *opposite edit graph*), which coincides with the original edit graph with the exception that the origin and destination of every edge are swapped. In the opposite edit graph the source node is $(|s_1|, |s_2|, |s_3|)$ and the destination node is $(0, 0, 0)$.

The main changes are as follows.

1) Each iteration is performed in a synchronized manner in both the direct and the opposite edit graph.
2) For each 0-edge traversal, routine *getZeroCostPath* checks whether for both graphs (the direct and the opposite edit graph) the computed node corresponds to the end node of the other graph. The algorithm terminates whenever the coordinates of those computed nodes in the two edit graphs coincide. We remark, that it is sufficient to detect only one such *meeting point*, say $(x, y, z)$, which is recorded. This meeting point belongs to a shortest path from source to destination.

```
Algorithm W3M

destinationReached:=false,
I=0

while destinationReached=false do
    buildExtensions (I)
    I = I+1

buildExtensions (I)
    for i=I down to 1
        for p=0 to i do
         if p=i or p=0 then
              bestExtension (Frontier, (i,p,0))
              bestExtension (Frontier, (i,0,p))
              bestExtension (Frontier, (0,i,p))
         else
              bestExtension (Frontier, (i,p,0))
              bestExtension (Frontier, (i,0,p))
              bestExtension (Frontier, (0,i,p))
              bestExtension (Frontier, (p,i,0))
              bestExtension (Frontier, (0,p,i))
              bestExtension (Frontier, (p,0,i))
    /* main diagonal at last */
   bestExtension (Frontier, (0,0,0))
```

```
bestExtension (Frontier, x, y,z)
    /* In the following we denote by reachFromX, reachFromY, and reachFromZ
       the X coordinates of the point on the current diagonal reached by adding
       a 1-edge from the three neighbor diagonals. */
  if x=0 then
      reachFromX= Frontier.YZ[x-1, y, z]+1
      reachFromY= Frontier.YZ[x, y-1, z]
      reachFromZ= Frontier.YZ[x, y, z-1]

      Frontier.YZ[x,y,z] = getZeroCostPath (x,y,z,
                             max(reachFromX,reachFromY,reachFromZ) )
  if y=0 then
      reachFromX= Frontier.XZ[x-1, y, z]+1
      reachFromY= Frontier.XZ[x, y-1, z]
      reachFromZ= Frontier.XZ[x, y, z-1]

      Frontier.XZ[x,y,z] = getZeroCostPath (x,y,z,
                             max(reachFromX,reachFromY,reachFromZ) )

  if z=0 then
      reachFromX= Frontier.XY[x-1, y, z]+1
      reachFromY= Frontier.XY[x, y-1, z]
      reachFromZ= Frontier.XY[x, y, z-1]

      Frontier.XY[x,y,z] = getZeroCostPath (x,y,z,
                             max(reachFromX,reachFromY,reachFromZ)
```

Fig. 5. [Left] The pseudocode of the W3M+ algorithm in 3D. [Right] The sub-routine *bestExtension*.

3) Meeting point $(x, y, z)$ divides the search space into two sub-spaces. We recursively continue for each sub-space until the entire path from source to destination (of the direct edit graph) is built.
   The base case of the recursion is reached when the cost of the path between source and destination is zero.

We next show that in each consecutive recursion step the time complexity is reduced four times: if the running time before a recursive call is $O(ND^2)$, then the running time of the recursive call of the "divide and conquer by half-cost points" method is

$$(N-x)(D/2)^2 + x(D/2)^2 = N(D/2)^2 = (ND^2)/4.$$

Thus, the total running time of the 3D algorithm is

$$ND^2 + N\frac{D^2}{4} + N\frac{D^2}{16} + \ldots \leq 2ND^2 = O(ND^2).$$

Thus, the time complexity remains unchanged, but now in the first recursion step it is enough to store end vertices of the paths starting at $3D^2$ diagonals, and this number is reduced 4 times in each following recursion step. The space complexity therefore becomes $O(D^2)$, and for $k$ strings $O(D^{k-1})$. The total space required by the algorithms is $O(kN + D^{k-1})$.

## V. Experimental Evaluation

In this section we present a comparative experimental evaluation of all three algorithms, DP, MM+ and W3M+, extended to the case of three input strings. For all three algorithms we used the memory reduction technique described above.

As mentioned before, [12] proposed another approach for extending W3M to 3D. The authors reported that standard DP outperformed W3M for problem instances where $|LCS|$ was $50\%$ of the input string length $N$. For input strings with an $|LCS| = 90\%N$ of their length they obtained some improvement over DP. However, this improvement was by an order of magnitude less than our improvement over standard DP. Results for other degrees of similarity were not reported in [12]. Our experiments stand in contrast to the results reported in [12].

We implemented our 3D algorithms in Java 1.5. The algorithms then were tested on a Pentium 4 3GHz processor with 1GB RAM. We generated triplets of random input strings over an alphabet of size 20. By skewing the random distribution of characters, we obtained different sets of three input strings with $|LCS| = 50\%N, 60\%N, 70\%N, 80\%N$.

The running time (in seconds) of the algorithms is presented in Figure 6. As can be seen from these results, both 3D implementations outperform the DP 3D algorithm for all the problem instances considered.

Since the worst-case complexity is the same for both of our algorithms, MM+ and W3M+, we show experimental results in 3D for both of them. We observe from Figure 6, that for example algorithm W3M+ performs about five times better than DP, for strings of length 1000 with an $|LCS| = 50\%N$. The W3M+ algorithm performs about 100 times better than DP, for strings of length 1000 with an $|LCS| = 80\%N$. We can also see that the W3M+ algorithm outperforms the MM+ algorithm significantly.

## References

[1] Baeza–Yates R.A. Searching subsequences. *Theoretical Computer Science* 78(2): 363–376, 1992.
[2] Bergroth L., Hakonen H., and Raita T. A survey of longest common subsequence algorithms. *SPIRE'00*: 39–48, 2000.

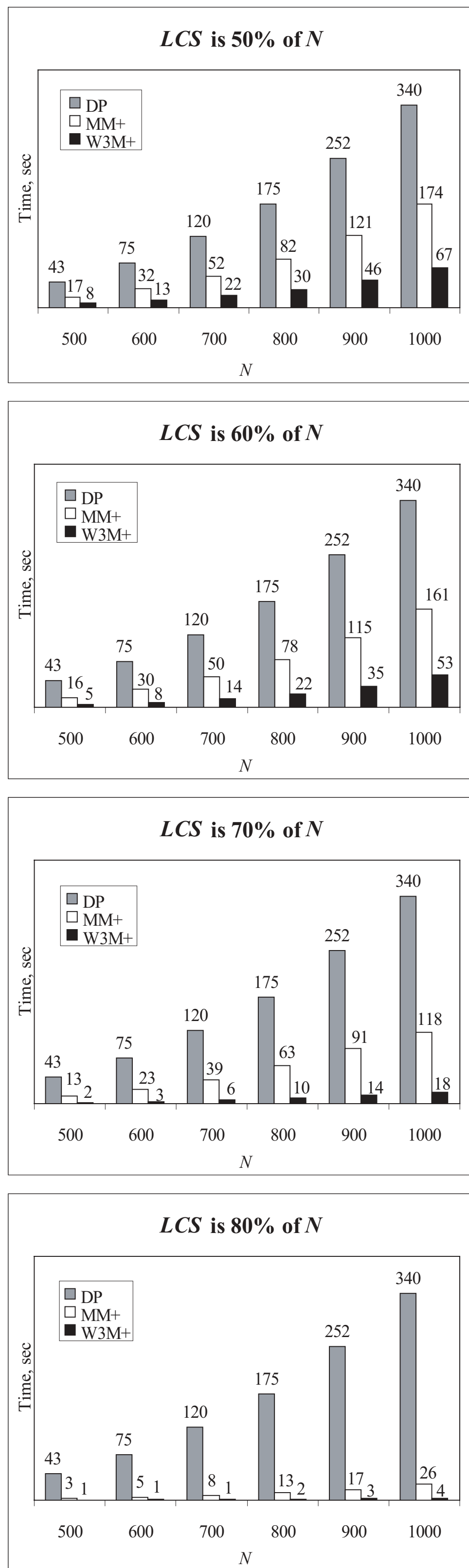



Fig. 6. Running time (in seconds) for DP and algorithms MM+, and W3M+ in 3D for different length $N$ of input strings. DP is not influenced by the similarity of input strings, and so, the running time of DP in all four graphs is identical.


[3] BEREG S, KUBICA M., WALEN T., ZHU B. RNA multiple structural alignment with longest common subsequences. *Journal of Combinatorial Optimization* 13(2): 179-188, 2007.

[4] DAY. H. E. W, MCMORRIS F. R. The computation of consensus patterns in DNA sequences. *Mathematical and Computer Modelling* 17(10): 49-52, 1993.

[5] DIXON N. R., MARTIN T. B. *Automatic speech and speaker recognition*. IEEE Press, New York., 1979.

[6] ELLOUMI M. Comparison of strings belonging to the same family. *Inf. Sci.* 111 (1–4): 49–63, 1998.

[7] FRENCH J., POWELL A., AND SCHULMAN E. Applications of approximate word matching in information retrieval. *CIKM'97*: 9–15, 1997.

[8] GUSFIELD D. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[9] HIRCHENBERG D.S. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18(6): 341–343, 1975.

[10] HAKATA K., IMAI H. The longest common subsequence problem for small alphabet size between many strings. *Algorithms and Computation: LNCS* 650: 469–478, 1992.

[11] HSU W., DU M. Computing a longest common subsequence for a set of strings. *BIT* 24:45–59, 1984.

[12] IRVING R.W., FRASER C.B. Two algorithms for the longest common subsequence of three (or more) strings. *CPM, LNCS* 644: 214–229, 1992.

[13] ITOGA S.Y. The string merging problem. *BIT* 21: 20–30, 1981.

[14] KUO S., AND CROSS, G.R. An improved algorithm to find the length of the longest common subsequence of two strings. *ACM SIGIR* 23(3): 89–99, 1989.

[15] MAIER D. The complexity of some problems on subsequences and supersequences. *Journal of the ACM* 25(2): 322–336, 1978.

[16] MILLER W., MYERS, E.W. A file comparison problem. *Softw. Pract. Exp.* 15(11): 1025–1040, 1985.

[17] MYERS, E.W. An $O(ND)$ difference algorithm and its variations. *Algorithmica* 1: 251–266, 1986.

[18] NEEDLEMAN S.B., WUNSH C.D. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology* 48: 443–453, 1970.

[19] NING K., KEE NG H., WAI LEONG H. Finding Patterns in Biological Sequences by Longest Common Subsequences and Shortest Common Supersequences. *BIBE'06*: 53–60, 2006.

[20] SANKOFF D., KRUSKAL J.B. (EDS.) *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley, MA, 1983.

[21] SEDGEWICK R., VITTER S. Shortest paths in Euclidean graphs. *Algorithmica* 1: 31–48, 1986.

[22] STEPHEN G.A. *String Searching Algorithms*. World Scientific, Singapore, 1994.

[23] WAGNER R.A., FISHER M.J. The string-to-string correction problem. *Journal of the ACM* 21(1) : 168–173, 1974.

[24] WATERMAN, M. S. *Introduction to computational biology: maps, sequences, and genomes*. Chapman & Hall/CRC, 1995.

[25] WU S., MANBER U., MYERS, G., MILLER W. An $O(NP)$ sequence comparison algorithm. *Inf. Proc. Lett.* 35: 317–323, 1990.