A Fast Heuristic Search Algorithm for Finding the Longest Common Subsequence of Multiple Strings

Qingguo Wang, Mian Pan, Yi Shang and **Dmitry Korkin**
Computer Science
University of Missouri
Columbia, Missouri 65211
{qwp4b, mpry6}@mail.missouri.edu,{shangy, korkin}@missouri.edu

Abstract

Finding the longest common subsequence (LCS) of multiple strings is an NP-hard problem, with many applications in the areas of bioinformatics and computational genomics. Although significant efforts have been made to address the problem and its special cases, the increasing complexity and size of biological data require more efficient methods applicable to an arbitrary number of strings. In this paper, a novel search algorithm, MLCS-A*, is presented for the general case of multiple LCS (or MLCS) problems. MLCS-A* is a variant of the A* algorithm. It maximizes a new heuristic estimate of the LCS in each search step so that the longest common subsequence can be found. As a natural extension of MLCS-A*, a fast algorithm, MLCS-APP, is also proposed to deal with large volume of biological data for which finding a LCS within reasonable time is impossible. The benchmark test shows that MLCS-APP is able to extract common subsequences close to the optimal ones and that MLCS-APP significantly outperforms existing heuristic approaches. When applied to 8 protein domain families, MLCS-APP produced more accurate results than existing multiple sequence alignment methods.

Introduction

Assume sequences are strings of characters defined over a finite alphabet Σ. Let x and y be two sequences of lengths n and k correspondingly. For a sequence $x = a_1 a_2 \dots a_n$, a sequence $y = a_{i_1} a_{i_2} \dots a_{i_k}$ is called a **subsequence** of x if $1 \leq i_j \leq n$, for $1 \leq j \leq k$, and $i_r < i_t$, for $1 \leq r < t \leq k$. Let $S = \{s_1, s_2, \dots, s_d\}$ be a set of sequences over alphabet Σ. A **multiple longest common subsequence** (MLCS) for set S is a sequence y such that (i) y is a subsequence of s_i, $1 \leq i \leq d$, and (ii) y is the longest one satisfying (i).

The MLCS problem is to find the longest subsequence shared between two or more strings. It is a classical computer science problem, with important applications in many fields, such as information retrieval and computational biology (Sheridan and Venkataraghavan 1992; Attwood and Findlay 1994; Sankoff and Blanchette 1999; Bourque and Pevzner 2002). For over 30 years, significant efforts have been made to find efficient algorithms for the MLCS problem. Many of them, however, address the simplest case of MLCS of two strings, also known as the longest common subsequence (LCS) problem (Hirschberg 1977; Masek and Paterson 1980; Smith and Waterman 1981). The general case of the MLCS problem of any given number of strings is NP-hard (Maier 1978). Methods that find the longest common subsequence for arbitrary number of strings are few (Hakata and Imai 1998; Chen, Wan, and Liu 2006; Korkin, Wang, and Shang 2008) and they could benefit greatly from improving their computation times. A method that solves the general MLCS problem efficiently can be applied to many computational biology and computational genomics problems that deal with biological sequences (Sankhoff and Kruskal 1983; Bork and Koonin 1996; Korkin and Goldfarb 2002). With the increasing volume of biological data and prevalent usage of computational sequence analysis tools, it is expected that the general MLCS algorithm will have a significant impact on computational biology methods and their applications.

The contribution of this paper is twofold. First, an algorithm, MLCS-A*, is presented to find a LCS for any given number of sequences. MLCS-A* is a variant of the A* algorithm, a provably optimal best-first search algorithm (Pearl 1984). But unlike A* that finds the least-cost path in a graph, MLCS-A* searches in a multidimensional matrix for a longest path that corresponds to a LCS. Second, a fast approximate algorithm MLCS-APP, which is derived by discarding points with low heuristic function values, is presented. This allows one to apply MLCS-APP to the conservation analysis of the real-world biological data, which includes both protein and gene families, and for which finding an optimal MLCS is often computationally not feasible. The benchmark test shows that MLCS-APP is able to efficiently extract common subsequences that are close to the optimal solutions and that on the benchmark sets of biological sequences MLCS-APP outperforms existing heuristic approaches.

Related Works

Classical methods for the MLCS problem are based on dynamic programming (Sankoff 1972; Smith and Waterman 1981). In its simplest case, given two sequences s_1 and s_2 of length n_1 and n_2 respectively, a dynamic programming algorithm iteratively builds a $n_1 \times n_2$ score matrix L, in which $L[i, j], 0 \leq i \leq n_1, 0 \leq j \leq n_2$, is the length of a

LCS between two prefixes $s_1[1,\ldots,i]$ and $s_2[1,\ldots,j]$.

$$L[i,j]=\begin{cases}0, & \text{if i or j} = 0\\ L[i-1,j-1]+1, & \text{if } s_1[i]=s_2[j]\\ \max(L[i,j–1],L[i–1,j]), & \text{if } s_1[i]\neq s_2[j]\end{cases} \quad (1)$$

In a straightforward implementation of dynamic programming, all the entries in L are calculated. The resulting algorithm has time and space complexity of $O(n^d)$ for d sequences of length n. Various approaches have been introduced to reduce the complexity of dynamic programming (Hirschberg 1977; Masek and Paterson 1980; Hsu and Du 1984; Apostolico, Browne, and Guerra 1992). Unfortunately, these approaches primarily address the special case of 2 sequences.

Employing the idea of dynamic programming, the dominant point approach limits its search to exploring a smaller set of dominant points rather than the whole set of positions in L. Dominant points are minimal points in a multidimensional search space. Knowing those points allows one to reduce the search space size and the computation time by orders of magnitude. The initial idea of dominant points was introduced by Hirschberg (1977). The dominant-point approach has since been successfully applied to the two-sequences cases (Chin and Poon 1990; Apostolico, Browne, and Guerra 1992). Dominant-point algorithms for more than 2 sequences have also been proposed (Hakata and Imai 1998; Chen, Wan, and Liu 2006; Wang, Korkin, and Shang 2009) and they are shown to be overwhelmingly faster than dynamic programming.

Besides the approaches that find exact LCS, heuristic algorithms for the MLCS problem have also been proposed. Initial heuristic algorithms (Chin and Poon 1994; Jiang and Li 1994) (and some later efforts) returned a common subsequence with a guaranteed approximation factor. Despite the theoretical importance of these algorithms, they were not attractive since the common subsequence extracted by them contains only one letter. Thereafter, various algorithms with improved solution, such as THSB (time horizon specialized branching heuristic) (Easton and Singireddy 2008) and ACO (ant colony optimization) (Shyu and Tsai 2009), have been presented and many successful applications have been exhibited. More recently, Blum et al. studied the Beam search (BS) algorithm (2009) and showed that BS outperforms other approaches from the literature in solution quality as well as in algorithm efficiency.

Definitions and Basic Properties

Let $\Sigma=\{a_1,a_2,\ldots,a_{|\Sigma|}\}$ be a finite alphabet of size $|\Sigma|$. Let $S=\{\mathbf{s}_1,\mathbf{s}_2,\ \ldots,\ \mathbf{s}_d\}$ be a set of d (assume $d\geq 2$) sequences of length $n_1,n_2,\ldots,n_d$, respectively, over Σ. A point p on S is denoted as $p=(p_1,p_2,\ldots,p_d)$, where each p_i , $1\leq i\leq d, 1\leq p_i\leq n_i$, is a coordinate of p for the corresponding sequence $\mathbf{s}_i$.

For any two sequences $\mathbf{s}_i$ and $\mathbf{s}_j$ in S of length n_i and n_j respectively, a $n_i\times n_j$ score matrix M_{ij} is iteratively built, in which $M_{ij}[x,y], 0\leq x\leq n_i, 0\leq y\leq n_j$, is the length of the longest common subsequence (LCS) between two suffixes $\mathbf{s}_i[x+1,\ldots,n_i]$ and $\mathbf{s}_j[y+1,\ldots,n_j]$. Specifically, the score matrix M_{ij} is defined as follows:

$$M_{ij}[x,y]=\begin{cases}0, & x=n_i \text{ or } y=n_j\\ M_{ij}[x+1,y+1]+1, & \mathbf{s}_i[x+1]=s_j[y+1]\\ \max(M_{ij}[x,y+1],M_{ij}[x+1,y]), & otherwise\end{cases} \quad (2)$$

The heuristic estimate $h(p)$, where $p=(p_1,p_2,\ldots,p_d)$, of the length of the longest common subsequences among d suffixes $\mathbf{s}_i[p_i+1,\ldots,n_i], 1\leq i\leq d$, is computed from M_{ij}, i.e. from 2-dimensional LCS,

$$h(p)=\min_{1\leq i,j\leq d} M_{ij}[p_i,p_j] \quad (3)$$

Lemma 1. *$h(p)$ is equal to or greater than the length of the LCS among sequences $\mathbf{s}_1[p_1+1,\ldots,n_1]$,$\mathbf{s}_2[p_2+1,\ldots,n_2]$, …, and $\mathbf{s}_d[p_d+1,\ldots,n_d]$.*

Proof. Let C be a LCS among sequences $\mathbf{s}_1[p_1+1,\ldots,n_1]$,$\mathbf{s}_2[p_2+1,\ldots,n_2]$, …, and $\mathbf{s}_d[p_d+1,\ldots,n_d]$. The length of C is no greater than $M_{ij}[p_i,p_j], 1\leq i,j\leq d$, since C is also a common subsequence between $\mathbf{s}_i[p_i+1,\ldots,n_i]$ and $\mathbf{s}_j[p_j+1,\ldots,n_j]$. □

A point $p=(p_1,p_2,\ldots,p_d)$ on a set S is called a *match* if $\mathbf{s}_1[p_1]=\mathbf{s}_2[p_2]=\ldots=\mathbf{s}_d[p_d]$. For example, for two sequences, $\mathbf{s}_1=GATTACA$ and $\mathbf{s}_2=GTAATCTAAC$, points $(1,1)$ and $(2,3)$ are two matches, corresponding to symbols G and A, respectively. For any two points $p=(p_1,p_2,\ldots,p_d)$ and $q=(q_1,q_2,\ldots,q_d)$, if $p_i<q_i, 1\leq i\leq d$, we denote $p<q$. A match q of symbol $a\in\Sigma$ is called a *a-successor* of a point p, if $p<q$ and there is no other match r of a symbol such that $p<r<q$. The set of a-successor for p is denoted as $Suc(p,a)$. The set of all successors for p is denoted as $Suc(p,\Sigma)=\cup_{a\in\Sigma}Suc(p,a)$. The set of all successors for a set of points A is denoted as $Suc(A,\Sigma)=\cup_{p\in A}Suc(p,\Sigma)$.

The following Corollary 1 can be inferred directly from the definition of $h(p)$.

Corollary 1. *$h(q)<h(p)$ if $q\in Suc(p,\Sigma)$.*

Let $|MLCS|$ represent the length of the LCS among sequences $\mathbf{s}_1$, $\mathbf{s}_2$, …, $\mathbf{s}_d$ and $L(p)$ be the length of the LCS of prefixes $\mathbf{s}_1[1,\ldots,p_1]$, $\mathbf{s}_2[1,\ldots,p_2]$, …, $\mathbf{s}_d[1,\ldots,p_d]$. Then, function $f(p)$ is defined to estimate $|MLCS|$,

$$f(p)=L(p)+h(p) \quad (4)$$

The following Corollary 2 can be inferred trivially from Lemma 1.

Corollary 2. *$f(p)\geq|MLCS|$ for any point p on a LCS.*

In Section below, function $g(p)$ is used to compute $L(p)$. Correspondingly, the definition of function $f(p)$ in Eq. 4 is modified as $f(p)=g(p)+h(p)$.

To compare two points p and q and determine which one should be visited first, we define $>_H$, a lexicographical order on f and h.

$$p>_H q\Leftrightarrow(f(p)>f(q))\vee(f(p)=f(q)\wedge h(p)>h(q)) \quad (5)$$

Two New MLCS Algorithms

In this section, two new algorithms for the general case of the MLCS problems are presented.

MLCS-A* algorithm

Let T be a set of points whose successors, $Suc(T, \Sigma)$, are known and Q be a subset of successors of T whose successors, $Suc(Q, \Sigma)$, are unknown. The following algorithm in Fig. 1 iteratively explores points in Q.

```
Algorithm MLCS-A*(s_1,s_2,...,s_d)

    Preprocessing;
01  p_0 = (0,0,...,0);
02  parent(p_0) =null;
03  g(p_0) = 0;
04  f(p_0) = h(p_0);
05  Q = {p_0};
06  T = ∅;

07  while (|Q| > 0)
08      p ← PriorityQueue(Q);
09      if (h(p) == 0)
10          return CommonSeq(p);
11      for each (q ∈ Suc(p,Σ))
12          if (q ∉ Q)
13              Q = Q ∪ {q};
14              UpdateSuc(p, q);
15          else if (g(q) < g(p) + 1)
16              UpdateSuc(p, q);
17      T=T∪{p}; //For the purpose of correctness proof only

 //Update point q
 function UpdateSuc(p, q)
18  parent(q) = p;
19  g(q) = g(p) + 1;
20  f(q) = g(q) + h(q);

 //Extract common subsequence
 function CommonSeq(p)
21  if (parent(p) ≠null)
22      CommonSeq(parent(p));
23  print s_1[p];
24  return g(p);
```

Figure 1: The pseudocode of the algorithm MLCS-A*

The main part of the algorithm consists of three functions. The first one, MLCS-A*(), iteratively selects a point p from a priority queue and finds all successors of p. The priority queue in line 8 extracts a point p from Q such that there is no another point q in Q satisfying $q >_H p$. Another function UpdateSuc() updates the value of $f(q)$ for a point q and sets p as the parent of q. Finally, the function CommonSeq() prints a common subsequence and returns the length of the corresponding subsequence.

Proof of the algorithm

Theorem 1 below demonstrates that MLCS-A* algorithm is optimal, i.e. it finds the longest common subsequence. In order to prove Theorem 1, two Lemmas are first presented.

Lemma 2. *$p >_H q$ holds for any points $p \in T, q \in Q$.*

Proof. The Lemma is proved by induction. After the first iteration of the while loop between lines 7 and 17, we have $T = \{p_0\}$, $Q = Suc(p_0, \Sigma)$. $p_0 >_H q$ is true for each $q \in Q$, since $g(q) = 1$ and $h(p_0) > h(q)$(from Corollary 1). Suppose the Lemma is still true after $k, k \geq 1$, iterations of the while loop. Then, at the $k + 1^{th}$ iteration of the loop, let p be a point selected from Q in line 8 for expansion. For any successor q of p, it can be inferred from Corollary 1 that $h(p) > h(q)$. If q is a new point to be added into Q, then $f(q) = g(p)+1+h(q) \leq g(p)+h(p) = f(p)$. If q is already in Q, $f(q) \leq f(p)$ is also true, since $g(q)$ is no greater than $g(p) + 1$. In either case, $p >_H q$ holds. Therefore, after moving p to T in line 17, the Lemma remains true. □

Lemma 3. *In Q, there is at least one point p satisfying: (i) p is on a longest common subsequence and, (ii) $g(p) = L(p)$.*

Proof. The Lemma is proved by induction. The first iteration of the while loop between lines 7 and 17 adds the first letters of all the longest common subsequences into Q and correctly sets 1 to the function $g(p), p \in Q$. So the Lemma is true in the initial step.

Suppose the Lemma remains true after $k, k \geq 1$, iterations of the while loop. Then, let p be a point extracted from Q at the $k + 1^{th}$ iterations of the while loop in line 8. For simplicity, suppose p is the only point in Q satisfying both conditions (i) and (ii) (otherwise, the Lemma trivially holds after processing p). Additionally, suppose $h(p) > 0$ (otherwise the program terminates). Let C be a longest common subsequence to which p belongs. Let q be a point on C immediately following p. Then, $p >_H q$, since $h(p) > h(q)$ and $g(p) + 1 = L(p) + 1 = L(q) \geq g(q)$. The function $g(q)$ of q will be set exactly to $L(q) = L(p) + 1$ by function UpdateSuc() after lines 12-16. Thus, a point q is found in Q satisfying both (i) and (ii). Therefore, the Lemma still holds after the $k + 1^{th}$ iteration. □

Theorem 1. *Let p be a point extracted from Q in line 8. If $h(p) = 0$, then $g(p) = |MLCS|$.*

Proof. Lemma 3 shows that in Q there is at least one point q satisfying that q belongs to a longest common subsequence and that $g(q) = L(q)$. Then, it can be inferred from Corollary 2 that $g(p) = f(p) \geq f(q) = L(q) + h(q) \geq |MLCS|$. □

Implementation details and complexity analysis

For convenience, let's assume below that the lengths of all sequences are equal to n.

It takes $O(n^2)$ time to build a 2-dimensional score matrix $M_{ij}, 1 \leq i, j \leq d$ (defined by Eq. 2). Instead of calculating $d(d-1)/2$ score matrices, one for each pair of sequences, only $d-1$ matrices, $M_{i,i+1}, 1 \leq i \leq d-1$, are computed in practice, which are enough to provide tight upper bound of the length of the LCS. The calculation of $d-1$ matrices takes $O(dn^2)$ time. With these score matrices, the function $h(p)$ can be computed in $O(d)$ time.

To efficiently find all successors of a point, a matrix $M_T = \{[a, j, i]\}, a \in \Sigma, 0 \leq j \leq \max_{1 \leq k \leq d}\{|s_k|\}, 1 \leq i \leq d$, is calculated in the preprocessing step of MLCS-A*, where each element $M_T[a, j, i]$ specifies the position of the first occurrence of character a in the i-th sequence, starting from the $(j + 1)$-st position in that sequence. If a does not occur any more in the i-th sequence, the value of $M_T[a, j, i]$ is equal to $1 + max_{1 \leq k \leq d}\{|s_k|\}$. With the matrix M_T, the a-successor $q = (q_1, q_2, \ldots, q_d)$ of a point $p = (p_1, p_2, \ldots, p_d)$ can be calculated in $O(d)$ time, using the formula $q_i = M_T(a, p_i, i), 1 \leq i \leq d$. The calculation of matrix M_T takes $O(\,|\Sigma|\,d\,n)$ time.

Let N be the total number of points visited (i.e. $N = |T| + |Q|$). Then, it takes $O(\log N)$ time to extract a point p from the priority queue in line 8. p has at most $|\Sigma|$ successors, each requiring $O(d)$ time from previous analysis. The operations on set Q take constant time by storing it in a hash table. Therefore, totally it takes $O(\log N + |\Sigma|d)$ time to process one point.

In summary, it takes $O(dn^2 + |\Sigma|dn + |\Sigma|dN + N \log N)$ time altogether to extract a LCS. As the MLCS problem is NP-hard, it is expected that N grows non-polynomially with the size of input (otherwise $P = NP$). By ignoring insignificant terms, the time complexity of MLCS-A* is obtained as follows,

$$O(\,N \log N\,) \tag{6}$$

While an accurate estimate of N remains an open question, it is not hard to show that N is far smaller than n^d, the number of points that classical dynamic programming algorithm visits, since the search of MLCS-A* is restricted to a subset of matches in matrix L (defined by Eq. 1). Suppose the occurrence of each letter in Σ is evenly distributed in sequences. Then, the total number of matches in L can be estimated to $(n^d/|\Sigma|^{d-1})$. Hence, $N < n^d/|\Sigma|^{d-1} << n^d$.

MLCS-APP algorithm

It is computationally expensive to find a LCS for multiple strings since N grows quickly with the size of input. With increasing volume of data, MLCS-A* algorithm would quickly become impractical. Therefore, it is essential to adapt MLCS-A* to large volume applications.

Fig. 2 presents the MLCS-APP algorithm, an adaption of MLCS-A*. MLCS-APP differs from MLCS-A* in lines 8-13, which preserves for later expansion points with high value of $f(p)$ (i.e. points p satisfying $y - c \leq f(p)$), and discards those with low $f(p)$. Here, parameter c is a constant. By adjusting the value of c, the length of the returned common subsequences and the running time can be balanced. Besides, to restrict memory consumption, an upper bound k is set to Q' (lines 10-11) so that $|Q'| \leq k$. If there are more than k points in Q', $|Q'| - k$ points with the lowest $f(p)$ will be discarded. The parameters k and c jointly determine the solution quality. In the Experimental Section below, k and c are fixed at 2000 and 20 respectively.

Next the time complexity of MLCS-APP is estimated. Each iteration of the while loop in line 7 expends points $p \in Q'$ and therefore increases $g(q)$ for $q \in Suc(p, \Sigma)$.

Algorithm MLCS-APP($\mathbf{s}_1$,$\mathbf{s}_2$,. . .,$\mathbf{s}_d$)

```
    Preprocessing;
01  p0 = (0,0,...,0);
02  parent(p0) =null;
03  g(p0) = 0;
04  f(p0) = h(p0);
05  Q = {p0};
06  //T = ∅;

07  while (|Q| > 0)
08      y = max_{p∈Q} f(p);
09      Q' = {p|p ∈ Q ∧ y − c ≤ f(p)};
10      if (|Q'| > k)
11          remove points of low f value so that |Q'| = k;
12      Q = ∅;
13      for each (p ∈ Q')
14          if (h(p) == 0)
15              return CommonSeq(p);
16          for each (q ∈ Suc(p,Σ))
17              if (q ∉ Q)
18                  Q = Q ∪ {q};
19                  UpdateSuc(p, q);
20              else if (g(q) < g(p) + 1)
21                  UpdateSuc(p, q);
```

Figure 2: The pseudocode of the algorithm MLCS-APP

Since $g(q) \leq n$, the number of iteration of the while loop is no more than n. Moreover, given $|Q'| \leq k$, the number of points processed within the while loop is at most $k\,|\Sigma|$. From these two observations, the complexity of MLCS-APP is derived as follows,

$$O(\,d\,n^2 \,+\, k\,|\Sigma|\,d\,n\,) \tag{7}$$

Experimental Results

First, MLCS-A* was compared with MLCS-APP. Random DNA sequences independently generated from the alphabet $\Sigma = \{A, C, G, T\}$ were used as test data. The length of sequences was fixed at 100 and the number of sequences was different in each test case. Table 1 shows the lengths of the common subsequences, columns $g(*)$, returned by the two algorithms and the corresponding computation times (in seconds) of them. It indicates that in most cases MLCS-APP successfully found the longest common subsequences. For the last case where MLCS-APP failed to return the longest one, the length of the common subsequence extracted by MLCS-APP is very close to the length of the optimal ones. In addition, the results show that MLCS-APP is significantly faster than MLCS-A* when applied to a larger number of sequences.

Next, the number of sequences was fixed at 5 and the lengths of sequences were changed. Then, both algorithms were run on the new test set. The comparative results in Table 2 indicate again that MLCS-APP achieves a significant improvement in efficiency over MLCS-A* at a very low price of solution precision.

Number of sequences	MLCS-A*		MLCS-APP		Speedup[2]
	$g(*)$[1]	time	$g(*)$[1]	time	
4	46	0.05	46	0.31	0.16
5	43	0.53	43	0.28	1.89
6	40	3.30	40	0.27	12.22
7	37	25.39	37	0.23	110.39
8	36	93.42	36	0.23	406.17
9	35	195.10	34	0.23	848.26

1. g(*) denotes the lengths of the common subsequences
2. Speedup is the ratio of the running time of MLCS-A* to the running time of MLCS-APP.

Table 1: Comparison of MLCS-APP to MLCS-A* on random DNA sequences of length 100

Length of sequences	MLCS-A*		MLCS-APP		Speedup
	$g(*)$	time	$g(*)$	time	
100	43	0.53	43	0.28	1.89
120	51	1.03	51	0.36	2.86
140	60	5.73	59	0.42	13.64
160	70	8.80	69	0.50	17.60
180	77	25.47	76	0.56	45.48
200	84	70.47	83	0.63	111.86

Table 2: Comparison of MLCS-APP to MLCS-A* on 5 random DNA sequences of different lengths

MLCS-APP was also compared with the state-of-the-art heuristic algorithm, Beam search algorithm (Blum, Blesa, and López-Ibáñez 2009), on a large volume of biological sequences. Specifically, a set of gene and protein sequences from the rat ($Rattus\ Norvegicus$) genome, which was proposed by (Shyu and Tsai 2009) and applied later to benchmark Beam search, was used as test data. The length of each biological sequence was set to 600 while the number of sequences in each test was varied from 10 to 200.

Table 3 shows the running times and the lengths of common subsequences returned by MLCS-APP and Beam search for each test case. As Beam search is not publicly available, its results were taken directly from the published paper (Blum, Blesa, and López-Ibáñez 2009). Beam search was benchmarked on an Intel Core2 1.66GHz. MLCS-APP was run on the same computer for comparative purpose. As indicated in Table 3, MLCS-APP is on average 10 times faster than Beam search, while in 90% of cases it extracts common subsequences generally longer than or equal to Beam search.

Finally, MLCS-APP was compared with current multiple sequence alignment programs used in practice, ClustalW (version 2) (Larkin et al. 2007) and MUSCLE (version 4) (Edgar 2004). Two basic command line options, "–input" and "–log", were used for the execution of MUSCLE. Eight protein domain families were chosen as the test data set from the Pfam database (Finn et al. 2008), a collection of protein families that includes their annotations and multiple sequence alignments. Eight sequences of roughly the same length, i.e., around 200 amino acids, were selected from each family. The names of protein families are provided in

$\|\Sigma\|$	Number of sequences	Beam search		MLCS-APP	
		$g(*)$	time	$g(*)$	time
4	10	191	9.7	194	2.0
	15	173	12.3	180	1.9
	20	163	12.6	165	1.7
	25	162	15.8	164	2.0
	40	146	19.4	151	2.0
	60	144	26.7	147	2.8
	80	135	31.8	137	3.1
	100	132	38.5	134	3.7
	150	121	51.1	125	4.9
	200	121	69.1	122	6.6
20	10	69	27.4	70	3.3
	15	60	36.7	61	3.0
	20	51	34.4	53	2.4
	25	51	39.0	51	2.7
	40	49	47.4	48	2.9
	60	46	60.3	46	3.7
	80	43	64.4	43	3.9
	100	38	64.8	38	4.3
	150	36	77.8	35	5.2
	200	33	101.0	33	7.0

Table 3: Comparison of MLCS-APP to Beam search for protein and gene sequences in $Rattus\ Norvegicus$

Table 4, sorted in increasing order of the average pairwise sequence identities, which were computed using MUSCLE.

Table 4 shows the lengths of the common subsequences extracted by the three methods from the given sequences. The common subsequences were retrieved from alignment by counting the number of residues that are in common among all the sequences in the alignment. Table 4 indicates that the common subsequences calculated by MLCS-APP are consistently longer than those extracted by ClustalW and MUSCLE. It also demonstrates that MLCS-APP found the LCS in 7 out of 8 cases, while ClustalW and MUSCLE found none. More importantly, in the cases such as the protein family AP_endonuc_2, when the pairwise identity among sequences is poor, MLCS-APP still worked well, while both ClustalW and MUSCLE failed to find residues that are in common among all the sequences.

Acknowledgements

This research was supported by NIH Grant 5R21GM078601-02. We are also grateful to the authors of Beam search algorithm for sharing with us its test data.

References

Apostolico, A.; Browne, S.; and Guerra, C. 1992. Fast linear-space computations of longest common subsequences. *Theor. Comput. Sci.* 92(1):3–17.

Attwood, T., and Findlay, J. 1994. Fingerprinting g-protein-coupled receptors. *Protein Eng.* 7(2):195–203.

Blum, C.; Blesa, M. J.; and López-Ibáñez, M. 2009. Beam

Family ID (accession number)	Average length of sequences	Pairwise sequence identity	Length of optimal subsequences	Length of common subsequences		
				ClustalW	MUSCLE	MLCS-APP
AP_endonuc_2(PF01261)	205	19.6%	30	0	0	29
DUF2077(PF09850)	205	25.2%	35	7	7	35
NikM(PF10670)	207	35.7%	40	10	10	40
Nop25(PF09805)	211	45.9%	67	30	30	67
Exon_PolB(PF10108)	213	56.5%	76	63	63	76
Frag1(PF10277)	205	65.1%	93	87	87	93
G6PD_bact(PF10786)	203	74.9%	105	100	100	105
Adeno_hexon_C(PF03678)	215	85.2%	136	133	133	136

Table 4: The lengths of the common subsequences extracted by ClustalW, MUSCLE and MLCS-APP from the protein domain families selected from Pfam database. Eight sequences of roughly the same length, i.e., around 200 amino acids, were selected from each family.

search for the longest common subsequence problem. *Comput. Oper. Res.* 36(12):3178–3186.

Bork, P., and Koonin, E. 1996. Protein sequence motifs. *Curr.Opin.Struct.Biol.* 6:366–376.

Bourque, G., and Pevzner, P. 2002. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Research* 12(1):26–36.

Chen, Y.; Wan, A.; and Liu, W. 2006. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. *BMC Bioinformatics* 7(Suppl 4):S4.

Chin, F. Y. L., and Poon, C. K. 1990. A fast algorithm for computing longest common subsequences of small alphabet size. *J. Inf. Process.* 13(4):463–469.

Chin, F., and Poon, C. K. 1994. Performance analysis of some simple heuristics for computing longest common subsequences. *Algorithmica* 12:293–311.

Easton, T., and Singireddy, A. 2008. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics* 14(3):271–283.

Edgar, R. C. 2004. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research* 32(5):1792–1797.

Finn, R. D.; Tate, J.; Mistry, J.; Coggill, P. C.; Sammut, S. J. J.; Hotz, H.-R. R.; Ceric, G.; Forslund, K.; Eddy, S. R.; Sonnhammer, E. L.; and Bateman, A. 2008. The pfam protein families database. *Nucleic acids research* 36(Database issue):D281–288.

Hakata, K., and Imai, H. 1998. Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima. *Optimization Methods and Software* 10:233–260.

Hirschberg, D. S. 1977. Algorithms for the longest common subsequence problem. *J. ACM* 24(4):664–675.

Hsu, W. J., and Du, M. W. 1984. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics* 24(1):45 – 59.

Jiang, T., and Li, M. 1994. *Automata, Languages and Programming*. chapter On the approximation of shortest common supersequences and longest common subsequences, 191 – 202.

Korkin, D., and Goldfarb, L. 2002. Multiple genome rearrangement: a general approach via the evolutionary genome graph. *Bioinformatics* 18(suppl_1):S303–311.

Korkin, D.; Wang, Q.; and Shang, Y. 2008. An efficient parallel algorithm for the multiple longest common subsequence (mlcs) problem. In *ICPP '08: Proc. 37th Intl. Conf. on Parallel Processing*, 354–363. Washington, DC, USA: IEEE Computer Society.

Larkin, M.; Blackshields, G.; Brown, N.; Chenna, R.; McGettigan, P.; McWilliam, H.; Valentin, F.; Wallace, I.; Wilm, A.; Lopez, R.; Thompson, J.; Gibson, T.; and Higgins, D. 2007. Clustal w and clustal x version 2.0. *Bioinformatics* 23(21):2947–2948.

Maier, D. 1978. The complexity of some problems on subsequences and supersequences. *J. ACM* 25(2):322–336.

Masek, W. J., and Paterson, M. S. 1980. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.* 18 – 31.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Sankhoff, D., and Kruskal, J. B. 1983. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison*. AddisonWealey.

Sankoff, D., and Blanchette, M. 1999. Phylogenetic invariants for genome rearrangements. *Journal of Computational Biology* 6:431–445.

Sankoff, D. 1972. Matching sequences under deletion/insertion constraints. *Proc. Natl. Acad. Sci. USA* 69(1):4 – 6.

Sheridan, R. P., and Venkataraghavan, R. 1992. A systematic search for protein signature sequences. *Proteins: Structure, Function, and Genetics* 14(1):16–28.

Shyu, S. J., and Tsai, C.-Y. 2009. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Comput. Oper. Res.* 36(1):73–91.

Smith, T. F., and Waterman, M. S. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147(1):195 – 197.

Wang, Q.; Korkin, D.; and Shang, Y. 2009. Efficient dominant point algorithms for the multiple longest common subsequence (mlcs) problem. In *IJCAI*, 1494–1500.