An Effective Branch-and-Bound Algorithm to Solve the k-Longest Common Subsequence Problem

Gaofeng Huang [1] and **Andrew Lim** [1]

Abstract. In this paper, we study the Longest Common Subsequence problem of multiple sequences. Because the problem is NP-hard, we devise an effective Branch-and-Bound algorithm to solve the problem. Results of extensive computational experiments show our method to be effective not only on randomly generated benchmark instances[2], but also on real-world protein sequence instances.

Keywords: Search, Branch-and-Bound, Bioinformatics, real-world protein sequence

1 Introduction

Finding the Longest Common Subsequence(*LCS*) between DNA/Protein sequences is one of the basic problems in modern computational molecular biology[14]. The LCS problem is Related to the "Edit Distance" and "Sequence Alignment"[1]. LCS is more than a classical problem in combinatorial pattern matching[15]; it has many other practical applications such as Web Usage Mining[2], Music Understanding[4], File Comparison[13], etc.

Since 1974, much attention has been focused on the problem of find the *LCS* of 2 sequences with length m and n. Wagner and Fischer[17] first presented a dynamic programming approach, which takes $O(mn)$ time and space. Hirschberg[7] later presented a more efficient implementation which only uses linear space. Many improvements have been proposed. At present the best result is provided by Masek and Paterson [11]. Their algorithm takes $O(mn/\log n)$ time. An extensive survey can be found in [12].

Unfortunately, the *LCS* problem of k sequences is $\mathcal{NP}$-hard (see Maier[10]) even with fixed number of alphabets. A direct extension of the dynamic programming[5] takes $O(n^k)$ time and $O(n^{k-1})$ space to solve *LCS* problem for k sequences of length n. Therefore, even for small values of k, it is not practical since the length of sequence n is usually very large. It is noted in [3] that at least 16Gbyte of memory is required to solve the instances with 5 sequences where each sequence has a length of 400 characters.

Consequently, several heuristic and approximation algorithms were developed. Among these, the Long Run(LR) algorithm developed by Jiang and Li[8] is the first method that guarantees constant performance ratio, while the Expansion Algorithm(EA) proposed by Bonizzoni et al.[3, 16] claimed to outperform LR and is regarded as the current best result. Although these algorithms may deal with instances of 20 sequences each with length 500, these approximation algorithms do not provide the optimal solution.

The purpose of this paper is to present an exact algorithm based on the Branch-and-Bound technique to solve *LCS* problems with multiple sequences. Although the Branch-and-Bound method is an exponential time algorithm, our implementation is extremely efficient through the use of a well-developed upper bound. The effectiveness and efficiency of our method is verified using standard benchmarks.

The rest of this paper is organized as follows. Section 2 briefly describes the problem formulation, while the details of the implementation of our Branch-and-Bound algorithm are presented in Section 3. In Section 4, the computational results of our experiments are given in detail. Finally, we present our conclusions in Section 5.

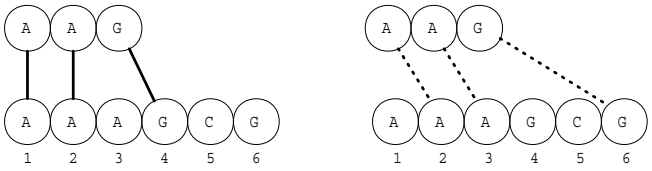
2 Problem Description

A **sequence** $x = x_1x_2...x_n$ over finite **alphabet** Σ may be any combination of n characters from Σ. That is, $x_i \in \Sigma$ and $x \in \Sigma^*$. The **length** of x, can be denoted as $|x|$.

Given a sequence x, we call another sequence $y = y_1y_2...y_m$ a **subsequence** of x, if there exists an **embedding** $I = (i_1, i_2, ..., i_m)$ so that $1 \leq i_1 < i_2 < ... < i_m \leq |x|$ and $x_{i_k} = y_k, \forall k = 1, 2, ..., m$. Let $\mathbf{s(x)} = \{\mathbf{y}|\mathbf{y}$ is a subsequence of $\mathbf{x}\}$.

It is noticeable that one subsequence of x may have several embeddings in x. For example, AAG is a subsequence of AAAGCG, which has 6 embeddings: $(1, 2, 4)$, $(1, 2, 6)$, $(1, 3, 4)$, $(1, 3, 6)$, $(2, 3, 4)$, $(2, 3, 6)$.

Figure 1. one subsequence with more embeddings



The *k-LCS* problem $LCS(\mathcal{X})$ can be described as:

$$\begin{array}{ll} \textbf{Instance}: & \text{a set } \mathcal{X} \text{ including } k \text{ sequences } x^{(1)}, x^{(2)}, ...x^{(k)} \\ \textbf{Solution}: & \text{a Longest Common Subsequence } y \\ \textbf{Objective}: & LCS(\mathcal{X}) = \max |y|, \text{subject to } y \in s(x^{(i)}), \\ & \forall i = 1, 2, ..., k \end{array}$$

Obviously, *k-LCS* is more general, while the *2-LCS* problem is its well-known, polynomial-time solvable, special case.

[1] Department of Industrial Engineering & Engineering Management, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong **EMAIL :** {gfhuang, iealim}@ust.hk

[2] The authors would like to express special thanks to Gianluca Della Vedova for providing the benchmark instances.

3 Branch-and-Bound Algorithm

As stated earlier, a direct extension of dynamic programming is not practical since it takes $O(n^k)$ time and $O(n^{k-1})$ space. At the same time, the existing heuristic and approximation algorithms, such as LR and EA, cannot provide any optimality guarantee. In this section, we shall present an Branch-and-Bound algorithm that consists of a well-developed upper bound, the elimination conditions, and the depth-first search strategy.

3.1 Upper Bound

In [3, 16], Bonizzoni et al. used the length of the shortest sequence in $\mathcal{X}$ as a trivial upper bound of the length of *k-LCS*. However, this upper bound is rather loose, for example:

Example 1 *Look at sequence* ATTAAAATTAAAT *and* CGCGCCGCGCGCCG*, the shorter one has 13 characters. The length of LCS is 0, as there is no common character at all.*

Based to this consideration, we derive our first upper bound:

$$UB_c = \sum_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{number of character } \sigma \text{ in sequence } x^{(i)})$$

UB_c reflects the total number of Common Characters among all k sequences, for example:

Example 2 *Look at sequence* AACCACGCG*,* ACCCCGCCACCAA *and* GCCACCAAGC*. There are* 3 *"*A*",* 4 *"*C*" and* 1 *"*G*" in common among all 3 sequences. Hence,* $UB_c = 3 + 4 + 1 = 8$.

This upper bound has a nice mathematical property:

Lemma 1 *The* UB_c *upper bound has* $|\Sigma|$ *guaranteed performance ratio, that is,* $\frac{UB_c}{LCS} \leq |\Sigma|$.

Proof: A trivial lower bound can be defined as:

$$LB_c = \max_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{number of character } \sigma \text{ in sequence } x^{(i)})$$

Which means the common subsequence only contains one kind of symbol σ from the alphabet Σ.

$$\begin{aligned} \text{Therefore, } \frac{UB_c}{LB_c} &= \frac{\sum_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{ of } \sigma \text{ in } x^{(i)})}{\max_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{ of } \sigma \text{ in } x^{(i)})} \\ &\leq \frac{\sum_{\sigma \in \Sigma} \max_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{ of } \sigma \text{ in } x^{(i)})}{\max_{\sigma \in \Sigma} \min_{i=1}^{k} (\#\text{ of } \sigma \text{ in } x^{(i)})} \\ &= \frac{|\Sigma| \times LB_c}{LB_c} = |\Sigma| \end{aligned}$$

A constant ratio is given by: $\frac{LB_c}{LCS} \geq \frac{LB_c}{UB_c} \geq \frac{1}{|\Sigma|}$

Although LB_c is trivial, it has the same guaranteed performance ratio as the approximation algorithm LR[8] and EA[3, 16].

Finally, $\frac{UB_c}{LCS} \leq \frac{UB_c}{LB_c} \leq |\Sigma|$ □

The UB_c is still loose in practice. For example,

Example 3 *For the sequences* AACCCTTTTGGGGG *and* GGGGGTTTTCCCAA*,* $LB_c = \max(2, 3, 4, 5) = 5$, $UB_c = 2 + 3 + 4 + 5 = 14$*, while the optimal LCS, which is* GGGGG*, has a length of* 5.

Indeed there exist some instances where $\frac{UB_c}{LCS} = |\Sigma|$(see Example 4).

Example 4 (Special Instance)

Sequence $x^{(1)}$: $\underbrace{\sigma_1\sigma_1\sigma_1...\sigma_1}_{p}\underbrace{\sigma_2\sigma_2\sigma_2...\sigma_2}_{p}$ ..... $\underbrace{\sigma_{|\Sigma|}\sigma_{|\Sigma|}\sigma_{|\Sigma|}...\sigma_{|\Sigma|}}_{p}$

Sequence $x^{(2)}$: $\underbrace{\sigma_{|\Sigma|}\sigma_{|\Sigma|}\sigma_{|\Sigma|}...\sigma_{|\Sigma|}}_{p}$ ..... $\underbrace{\sigma_2\sigma_2\sigma_2...\sigma_2}_{p}\underbrace{\sigma_1\sigma_1\sigma_1...\sigma_1}_{p}$

$$UB_c = p \times |\Sigma|$$
$$LCS = p$$
$$\frac{UB_c}{LCS} = |\Sigma|$$

All of these examples motivated us to develop more practical upper bounds. In Example 2, because LCS of the first two sequences AACCACGCG and ACCCCGCCACCAA is ACCCCG, no matter what the third sequence is, the LCS of all 3 sequences is not more than 6. Due to this consideration, we get the following lemmas:

Lemma 2 $\forall \mathcal{X}' \subset \mathcal{X}, LCS(\mathcal{X}') \geq LCS(\mathcal{X})$

Proof: Suppose the LCS of sequence set $\mathcal{X}$ is y^*, according to the definition, $\forall x \in \mathcal{X}, y^* \in s(x)$.

Since $\mathcal{X}' \subset \mathcal{X}, \forall x \in \mathcal{X}' \Rightarrow x \in \mathcal{X} \Rightarrow y^* \in s(x)$.

That means, y^* is also a common subsequence of sequence set $\mathcal{X}'$.

Therefore, $LCS(\mathcal{X}') \geq |y^*| = LCS(\mathcal{X})$. □

Lemma 3 *Let* $UB_i = \min_{\forall \mathcal{X}' \subset \mathcal{X}, |\mathcal{X}'|=i} LCS(\mathcal{X}'), UB_i \leq UB_{i-1}$

Proof: Suppose

$$UB_{i-1} = LCS(\mathcal{X}^*) \tag{1}$$

We get $|\mathcal{X}^*| = i - 1 < k \Rightarrow \exists x \in \mathcal{X}$, but $x \notin \mathcal{X}^*$

Therefore, $\mathcal{X}^* \subset \mathcal{X}^* \cup \{x\}$.

According to Lemma 2,

$$LCS(\mathcal{X}^*) \geq LCS(\mathcal{X}^* \cup \{x\}) \tag{2}$$

Since $|\mathcal{X}^* \cup \{x\}| = i$, due to the definition of UB_i,

$$UB_i \leq LCS(\mathcal{X}^* \cup \{x\}). \tag{3}$$

Finally, deduce from Equation (1)(2) and (3),

$$UB_i \leq LCS(\mathcal{X}^* \cup \{x\}) \leq LCS(\mathcal{X}^*) = UB_{i-1}.$$ □

Using these two lemmas, we develop our new upper bound.

Theorem 1 UB_i *is a upper bound of the k-LCS problem, that is,* $UB_i \geq LCS(\mathcal{X})$

Proof: Due to the definition of UB_i,

$UB_k = \min_{\forall \mathcal{X}' \subset \mathcal{X}, |\mathcal{X}'|=k} LCS(\mathcal{X}')$.

Obviously, $|\mathcal{X}'| = k \Rightarrow \mathcal{X}' = \mathcal{X}$.

So, $UB_k = LCS(\mathcal{X})$.

According to Lemma 3, we get a series of structured upper bound:

$$UB_1 \geq UB_2 \geq UB_3 ... \geq UB_k = LCS(\mathcal{X})$$ □

In fact, UB_1 means exactly "the length of the shortest sequence in $\mathcal{X}$ " which is the loosest one used in [3, 16]. For upper bound UB_i, we need to compute the sub-problem $LCS(\mathcal{X}')$, which can be solved in $O(n^i)$ time by applying Dynamic Programming technique. At the same time there are totally $\binom{k}{i}$ such subsets for $|\mathcal{X}'| = i$. Therefore, the computation of UB_i will take $O(n^i) \times \binom{k}{i}$ time. Due to the time and space constraint, we use UB_2 in our Branch-and-Bound algorithm.

3.2 Elimination Conditions

The general idea of the Branch-and-Bound algorithm is to construct a search tree and then apply a carefully selected criterion to determine which node to expand during the search. Therefore, elimination conditions are useful in curtailing the enumeration tree of a branch-and-bound scheme.

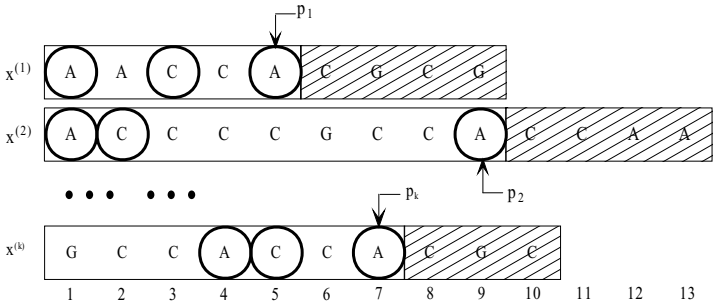
As stated earlier, one subsequence may have many embeddings in a specific sequence. Here, given sequence $y = y_1 y_2 ... y_m$, we define the **dominant embedding** $I^* = (i_1^*, i_2^*, ..., i_m^*)$ in sequence x as:

$$i_a^* = \min_{x_i = y_a, i > i_{a-1}^*} i \tag{4}$$

Thus, scanning the sequence x and y from left to right, to verify whether y is a subsequence of x, can be done in $O(n)$ time.

In the branch-and-bound scheme, we can maintain that each branch π of the search tree corresponds to a **partial common subsequence** of all k sequences, that is, $\pi = \pi_1 \pi_2 ... \pi_m$ can be embedded into each of the k sequences. As shown in Figure 2, let p_i denote the last dominant embedding position in sequence $x^{(i)}$, a **partial sub-problem** can be represented as $(p_1, p_2, ..., p_k)$, which means *LCS* of the k shadowed parts: $x^{(1)}_{p_1+1..|x^{(1)}|}, x^{(2)}_{p_2+1..|x^{(2)}|}, ..., x^{(k)}_{p_k+1..|x^{(k)}|}$. Thus, the partial state (valid branch) during search can be represented as $\pi // (p_1, p_2, ..., p_k)$.

Figure 2. the partial state during search



Theorem 2 *A branch $\pi // (p_1, p_2, ..., p_k)$ can be eliminated if there exists a common subsequence y so that $|\pi| + UB_i(p_1, p_2, ..., p_k) \leq |y|$.*

Proof: We have proved in Theorem 1 that UB_i is a upper bound, that is, $UB_i \geq LCS$. Therefore, $|\pi| + LCS(p_1, p_2, ..., p_k) \leq |\pi| + UB_i(p_1, p_2, ..., p_k) \leq |y|$, which means that there is no better solution in this branch. Consequently this branch can be eliminated. □

3.3 Implementation of Branch-and-Bound

The implementation of our Branch-and-Bound algorithm includes two parts: precomputing and depth-first search strategy, where the precomputing part is used to accelerate the embedding (valid branch) checking, and upper bound computation.

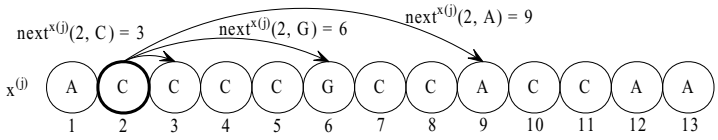
During search, by adding a symbol to a branch $\pi = \pi_1 \pi_2 ... \pi_m$, we get its child $\pi' = \pi_1 \pi_2 ... \pi_m \pi_{m+1}$. If the dominant embedding of π in sequence $x^{(j)}$ is $I^{(j)*} = (i_1^{(j)*}, i_2^{(j)*}, ..., i_m^{(j)*})$, the dominant embedding of π' should be $I'^{(j)*} = (i_1^{(j)*}, i_2^{(j)*}, ..., i_m^{(j)*}, i_{m+1}^{(j)})$. Therefore, in order to check whether π' is a valid branch(common subsequence), we only need to compute $i_{m+1}^{(j)}$. However, a direct implementation of Equation(4) will take $O(nk)$ time for a total of k sequences.

In our algorithm, we use precomputing to reduce the time complexity of each branch valid checking from $O(nk)$ to $O(k)$. The basic idea is that the next embedding position $i_{m+1}^{(j)}$ is only concerned with three parameters: the sequence $x^{(j)}$, position $i_m^{(j)*}$ and symbol π_{m+1}(see Figure 3). Thus, by defining:

$$next^{(j)}(i, \sigma) = \min_{x^{(j)}_{i'} = \sigma, i' > i} i', \text{ where } j = 1...k, \quad \begin{matrix} i = 1...|x^{(j)}|, \\ \sigma \in \Sigma. \end{matrix} \tag{5}$$

we can compute $i_{m+1}^{(j)} = next^{(j)}(i_m^{(j)*}, \pi_{m+1}), j = 1, 2, ...k$ in $O(k)$ time.

Figure 3. precomputing of $next$



The precomputation of $next$ itself requires $O(nk|\Sigma|)$ space and $O(n^2 k|\Sigma|)$ time. Moreover, it can be further reduced to $O(nk|\Sigma|)$ time using the following equation:

$$next^{(j)}(i, \sigma) = \begin{cases} \text{invalid} & \text{if } i \geq |x^{(j)}| \\ i + 1 & \text{if } x_{i+1}^{(j)} = \sigma \\ next^{(j)}(i+1, \sigma) & \text{if } x_{i+1}^{(j)} \neq \sigma \end{cases} \tag{6}$$

Another precomputing concerns the upper bound. As mention in Section 3.1, we adopt UB_2 as the upper bound in our real implementation, which takes $O(n^2) \times \binom{k}{2}$ time and space. For the partial sub-problem $(p_1, p_2, ..., p_k)$(see Figure 2), UB_2 can be rewritten as:

$$UB_2(p_1, p_2, ..., p_k) = \min_{i<j=1,2,...,k} \textit{2-LCS}(x^{(i)}_{p_i+1..|x^{(i)}|}, x^{(j)}_{p_j+1..|x^{(j)}|}) \tag{7}$$

And a well-known Dynamic Programming approach for the *2-LCS* is:

$$d^{(i)(j)}(p_i, p_j) = \begin{cases} 0 & \begin{matrix} \text{if } p_i \geq |x^{(i)}| \\ \text{or } p_j \geq |x^{(j)}| \end{matrix} \\ d^{(i)(j)}(p_i + 1, p_j + 1) + 1 & \text{if } x_{p_i+1}^{(i)} = x_{p_j+1}^{(j)} \\ \max \begin{cases} d^{(i)(j)}(p_i+1, p_j) \\ d^{(i)(j)}(p_i, p_j+1) \end{cases} & else \end{cases} \tag{8}$$

where $d^{(i)(j)}(p_i, p_j)$ means the length of Longest Common Subsequence between sequence $x^{(i)}_{p_i+1..|x^{(i)}|}$ and $x^{(j)}_{p_j+1..|x^{(j)}|}$.

Eventually, the depth-first search strategy is implemented recursively as Algorithm 1.

4 Experimental Results

In this section, we conduct elaborate experiments to demonstrate the effectiveness of our Branch-and-Bound algorithm. All the codes are

Algorithm 1 DepthFirstSearch ($\pi_1\pi_2...\pi_t//(p_1, p_2, ..., p_k)$)

1: **for** $\pi_{t+1} = \sigma \in \Sigma$ **do**
2: $\forall j, nextp_j = next^{(j)}(p_j, \sigma)$
3: **if** $\forall j, nextp_j \neq$ invalid {**check of valid branch**} **then**
4: $UB_2 \leftarrow \min_{i<j=1,2,...,k} d^{(i)(j)}(nextp_i, nextp_j)$
5: **if** $(t+1+UB_2) > currentBest$ {**elimination condition**} **then**
6: Depth-First-Search($\pi_1\pi_2...\pi_t\pi_{t+1}//(nextp_1, nextp_2, ..., nextp_k)$)
7: **end if**
8: **if** $t+1 > currentBest$ {**update currentBest**} **then**
9: $currentBest \leftarrow t+1$
10: $currentLCS \leftarrow \pi_1\pi_2...\pi_t\pi_{t+1}$
11: **end if**
12: **end if**
13: **end for**

implemented in C/C++ and run on a PentiumIII 800Mhz PC with 128M memory.

4.1 Random instances

All the random benchmark instances used in [3, 16](87700 instances in total) are tested in our experiments. These instances are generated through the following two random types:

Type A Random instances: There are 82000 instances of this type, while each instance consists of exactly $k = 4$ sequences with a length n that varies from 50 to 100. All the sequences are randomly generated according to the uniform distribution. And the alphabet size will be either $4(likeDNA)$ or $20(likeProtein)$.

Experiment results of our algorithm are described in Table 1. On average, our Branch-and-Bound will give the optimal solution within 10 seconds.

Table 1. Results of random type A (82000 instances)

alphabet size = 4 (DNA)						{41000 instances in total}
#number of instances	k	n_{max}	n_{min}	Avg. UB_2	Avg. k-LCS	Avg. Time(sec)
8000	4	70	50	33.41	26.31	0.07
8000			60	37.61	28.99	0.26
8000		75	50	34.41	27.40	0.12
8000			60	38.85	30.23	0.43
3000		100	80	42.83	31.41	1.53
3000			90	46.17	33.53	3.76
3000			95	47.85	34.67	6.25
alphabet size = 20 (Protein)						{41000 instances in total}
#number of instances	k	n_{max}	n_{min}	Avg. UB_2	Avg. k-LCS	Avg. Time(sec)
8000	4	70	50	13.36	6.38	0.004
8000			60	15.16	7.17	0.004
8000		75	50	13.95	6.74	0.004
8000			60	15.80	7.49	0.005
3000		100	80	14.20	6.12	0.004
3000			90	15.46	6.66	0.005
3000			95	16.12	6.88	0.005

Type B Random instances: Another 5700 instances have $k = 5, 10$ or 20 sequences each with up to $n = 500$ lengths. In every instance, all the sequences are generated by simulating an evolution process on a same random sequence $base(\mathcal{S})$ according to the Jukes-Cantor model[9].

Table 2. Results of random type B (5700 instances)

alphabet size = 4 (DNA)						{3000 instances in total}
#number of instances	k	n_{max}	n_{min}	Avg. UB_2	Avg. k-LCS	Avg. Time(sec)
300	5	500	400	99.72	98.17	0.02
300			450	270.56	269.89	0.05
400			480	405.16	405.01	0.11
300	10		400	97.92	95.22	0.20
300			450	262.60	260.85	0.21
400			480	402.53	402.12	0.42
300	20		400	96.83	93.87	1.30
300			450	257.75	255.10	1.22
400			480	400.28	399.48	1.51
alphabet size = 20 (Protein)						{2700 instances in total}
#number of instances	k	n_{max}	n_{min}	Avg. UB_2	Avg. k-LCS	Avg. Time(sec)
300	5	500	400	100.53	99.01	0.02
300			450	270.50	269.79	0.07
300			480	406.95	406.90	0.17
300	10		400	97.98	95.21	0.23
300			450	262.36	260.58	0.24
300			480	403.43	403.27	0.54
300	20		400	96.73	93.83	1.01
300			450	257.07	254.40	1.03
300			480	401.35	400.93	1.76

Although with larger k and n, the experiment results of our algorithm (see Table 2) indicate that these instances are even easier than random type A, since our upperbound UB_2 is quite close to the optimal solution $k - LCS$.

It's meaningless to directly compare the result of an exact algorithm with heuristic algorithms. However, for all of these random instances, it is evident that our algorithm dominates heuristic algorithm EA and LR, since the solutions can be obtained and guaranteed to be optimal in only a few seconds.

4.2 Real-world instances

In real-world, DNA/Protein sequences are neither uniformly distributed nor strictly Jukes-Cantor model fitted. Therefore, it will be challenging to test our algorithm on real-world data.

We try our algorithm on protein families from "Blocks Database" (`http://blocks.fhcrc.org/`), where a "block" contains of multiply aligned ungapped segments which correspond to the most highly conserved regions of proteins[6]. For example, "block" BL00355 contains "HMG14 and HMG17 proteins". To retrieve the protein sequence data of block BL00355, you can access the URL: `http://blocks.fhcrc.org/blocks-bin/getblock.sh?BL00355`. Under the link "Block Map" , 12 typical real protein sequences are included, such as, HG14_HUMAN, HG14_MOUSE, HG17_HUMAN, HG17_PIG, HG17_RAT etc.

50 instances are selected from the database, where the number of sequences k varies from 8 to 75. Experiment results for these 50 real protein families are shown in Table 3. It is evident that the upperbound UB_2 that we proposed is much tighter than UB_1, which is used in [3, 16].

Compared with BB_1(Branch-and-Bound by using UB_1), as you can see, our algorithm BB_2 works well for these real-world data. And even for the hard instances (such as BL00264 and BL00053), our algorithm can give the optimal solution in few minutes.

Table 3. Results for 50 instances from real-world protein families, (">10mins" means the algorithm does not terminate in 10 minutes)

BLOCK ID	k	n_{max}	UB_1	UB_2	LCS	Time(sec)	
						BB_1	BB_2
BL01181	8	70	55	26	15	0.03	0.01
BL00234	8	248	70	28	21	70.12	0.1
BL00634	9	270	85	42	26	>10mins	23.25
BL00051	10	52	49	23	15	0.01	0
BL01108	10	194	111	45	24	>10mins	12.41
BL00361	10	241	101	43	22	>10mins	19.32
BL00256	11	110	61	26	14	0.56	0.04
BL00257	11	330	82	34	20	16.92	0.14
BL01167	11	238	116	50	26	>10mins	33.86
BL01143	12	97	66	30	16	1.84	0.11
BL00355	12	104	69	45	37	>10mins	2.32
BL00282	13	472	81	32	17	23.52	0.12
BL01169	14	256	100	40	19	22.24	1.59
BL00582	15	67	49	21	11	0.04	0.01
BL00045	15	100	90	40	25	>10mins	0.43
BL01048	15	215	95	36	15	10.8	0.98
BL00025	15	1840	78	38	27	>10mins	1.1
BL00936	16	159	59	26	15	0.32	0.04
BL00831	16	371	82	48	27	>10mins	1.79
BL00258	16	141	89	39	22	468.96	3.28
BL00056	16	237	109	46	20	>10mins	10.19
BL01015	17	131	113	51	25	>10mins	29.62
BL00286	18	63	28	14	11	0.01	0.01
BL00285	18	119	68	27	13	0.28	0.07
BL00732	20	162	75	32	14	0.88	0.15
BL00362	20	286	88	40	20	305.6	5.76
BL00057	22	170	64	27	13	0.56	0.1
BL00264	22	168	125	68	39	>10mins	576.57
BL00269	24	100	93	39	21	7.64	0.37
BL00784	25	51	42	16	10	0.01	0.03
BL00579	25	200	63	23	10	0.01	0.09
BL00259	25	234	58	28	18	3.92	0.26
BL00937	25	129	111	43	17	30.24	5.34
BL00783	25	250	137	49	18	>10mins	77.72
BL00828	26	45	37	17	10	0.01	0.02
BL00475	28	322	82	32	18	119.44	6.73
BL00268	30	64	31	11	7	0.01	0.05
BL00360	32	278	103	43	18	33.4	1.34
BL00646	35	184	114	45	18	94.04	7.58
BL00265	38	131	36	12	6	0.01	0.1
BL00053	40	152	129	45	16	>10mins	199.91
BL00527	41	115	50	21	8	0.01	0.25
BL00352	41	798	65	26	14	0.88	0.33
BL00050	43	263	84	26	9	0.2	0.61
BL00049	43	141	119	43	18	>10mins	66.62
BL00280	44	1416	55	22	11	0.23	0.13
BL00323	45	212	78	31	13	0.56	0.56
BL00048	47	68	46	26	22	0.24	0.14
BL00054	48	173	116	40	15	20.6	15.27
BL00260	67	206	27	10	3	0.01	0.05
BL00055	75	188	67	23	11	2.36	0.16

5 Conclusion

Unlike the approximation/heuristic algorithm proposed in previous research, an exact Branch-and-Bound algorithm is developed in this paper, where the key idea is to construct a better upperbound by using Dynamic Programming.

For those random instances, experiment results show that our Branch-and-Bound algorithm dominates heuristic algorithms(EA, LR) since the optimal solutions can be obtained only in several seconds. Moreover, our algorithm works well for real-world protein families.

REFERENCES

[1] A. Apostolico, 'String editing and longest common subsequence', in *Handbook of Formal Languages, 2 Linear Modeling: Background and Application*, pp. 361–398. Springer-Verlag, Berlin, (1997).

[2] A. Banerjee and J. Ghosh, 'Clickstream clustering using weighted longest common subsequences', in *Proceedings of the Web Mining Workshop at the 1st SIAM Conference on Data Mining*, pp. 361–398. Chicago, (April 1997).

[3] Paola Bonizzoni, Gianluca Della Vedova, and Giancarlo Mauri, 'Experimenting an approximation algorithm for the lcs', *Discrete Applied Mathematics*, **110**(1), 13–24, (2001).

[4] Dannenberg, 'Recent work in real-time music understanding by computer', in *Proceedings of the Intl Symposium on Music, Language, Speech and Brain*, (1991).

[5] K. Hakata and H. Imai, 'The longest common subsequence problem for small alphabet size between many strings', in *Proc. 3rd International Symposium on Algorithms and Computaion(ISAAC)*, volume 650, pp. 469–478. Springer Verlag, (1992).

[6] J. G. Henikoff, E. A. Greene, S. Pietrokovski, and S. Henikoff, 'Increased coverage of protein families with the blocks database servers', *Nucl. Acids Res.*, **28**, 228–230, (2000).

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

[8] T. Jiang and M. Li, 'On the approximation of shortest common supersequences and longest common subsequences', *SIAM Journal on Computing,*, **24**(5), 1122–1139, (1995).

[9] W. H. Li, 'Molecular evolution', *Sinauer Assoc.*, (1997).

[10] D. Maier, 'The complexity of some problems on subsequences and supersequences', *Journal of the ACM*, **25**, 322–336, (1978).

[11] W. J. Mask and M. S. Paterson, 'A faster algorithm computing string edit distances', *Journal of Computer and System Sciences*, **20**(1), 18–31, (1980).

[12] Michael S. Paterson and Vlado Dancik, 'Longest common subsequences', *Mathematical Foundations of Computer Science*, 127–142, (1994).

[13] I. Simon, 'Sequence comparison: some theory and some practice', *Electronic Dictionaries and Automata in Computational Linguistics*, **377**, 79–92, (1987).

[14] T. Smith and M. Waterman, 'Identification of common molecular subsequences', *Journal of Molecular Biology*, **147**, 195–197, (1981).

[15] Zdenek Tronicek, 'Problems related to subsequences and supersequences', *SPIRE/CRIWG*, 199–205, (1999).

[16] Gianluca Della Vedova, *Multiple Sequence Alignment and Phylogenetic Reconstruction: Theory and Methods in Biological Data Analysis*, Ph.D thesis, 2000.

[17] R. A. Wagner and M. J. Fischer, 'The string-to-string correction problem', *Journal of the ACM*, **21**(1), 168–173, (1974).