Computers & Operations Research 39 (2012) 512–520

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

An improved algorithm for the longest common subsequence problem

Sayyed Rasoul Mousavi *, Farzaneh Tabataba

Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan 84156-83111, Iran

ARTICLE INFO

Available online 2 April 2011

Keywords:
Longest common subsequence
LCS
Beam search
Heuristic function
Algorithms
Bioinformatics

ABSTRACT

The Longest Common Subsequence problem seeks a longest subsequence of every member of a given set of strings. It has applications, among others, in data compression, FPGA circuit minimization, and bioinformatics. The problem is NP-hard for more than two input strings, and the existing exact solutions are impractical for large input sizes. Therefore, several approximation and (meta) heuristic algorithms have been proposed which aim at finding good, but not necessarily optimal, solutions to the problem. In this paper, we propose a new algorithm based on the constructive beam search method. We have devised a novel heuristic, inspired by the probability theory, intended for domains where the input strings are assumed to be independent. Special data structures and dynamic programming methods are developed to reduce the time complexity of the algorithm. The proposed algorithm is compared with the state-of-the-art over several standard benchmarks including random and real biological sequences. Extensive experimental results show that the proposed algorithm outperforms the state-of-the-art by giving higher quality solutions with less computation time for most of the experimental cases.

1. Introduction

The Longest Common Subsequence (LCS) problem asks for a longest string that is a subsequence of every member of a given set of strings. A subsequence of a given string is a string that can be obtained by deleting zero or more characters from the given string. Among various applications of this problem are file comparison [1], text editing [2], data compression [3], query optimization in databases [4], clustering Web users [5], and circuit minimization in field programmable gate arrays (FPGAs) [6]. In addition, LCS is used in molecular biology to compare DNA or RNA sequences and to determine homology in macromolecules [2,7–9].

For two input strings, LCS can be efficiently solved to optimality using dynamic programming in $O(l_1.l_2)$, where l_1 and l_2 are the lengths of the input strings. However, the problem is NP-hard for an arbitrary number of strings [10,11]. Various optimal (exact) algorithms have been proposed for this problem. One approach was to use dynamic programming. In [12,13], dynamic programming algorithms were proposed to solve the problem in $O(l^n)$, where n is the number of the input strings and l is the length of the longest one. These algorithms were improved in [14,15] to reduce the complexity to $O(l^{n-1})$, which is still exponential in the number of strings. Further algorithms based on dynamic programming may be found in the survey by Berghot et al. [16]. Another approach to tackle the LCS problem was based on traversing a search tree. Hsu and Du proposed in [17] an enumeration algorithm based on backtracking. The idea was further enhanced by Easton et al., who adopted a selection heuristic and two new types of branch and bound pruning [18]. The resulting algorithm, called *Specialized Branching* (*SB*), was compared with the previous state-of-the-art algorithms with positive results. In contrary to the above-mentioned dynamic programming algorithms, which are exponential in the number of strings, SB is exponential in the length of the longest common subsequence (LLCS). Among other works on LCS is [19] where an integer programming formulation was proposed whose complexity was still $O(l^n)$.

The above-mentioned optimal algorithms for LCS are impractical for large input sizes, hence the use of non-optimal solutions are inevitable. Until 1994, no heuristic method was introduced for the LCS problem [20]. The first non-optimal algorithm was *Long Run* (LR) which was an approximation algorithm with an approximation ratio of $|\sum|$ [20,21]. This algorithm simply constructs a string, as its output, using only a single character in $\sum$, which is not of interest in practice. Another approximation algorithm called *Expansion* was introduced in [22] which provided the same approximation ratio of $|\sum|$, without the single-character restriction of Long Run. The complexity of Expansion was $O(nl^4 \lg l)$, which was further improved in [23] using minimum-spanning-trees. Huang et al. [24] devised two more approximation

* Corresponding author.
E-mail addresses: srm@cc.iut.ac.ir (S.R. Mousavi), f.tabataba@ec.iut.ac.ir (F. Tabataba).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.02.026

algorithms called *Enhanced Long Run* (ELR) and *Best Next for Maximal Available Symbols* (BNMAS), which were of $O(|\sum|nl)$ and $O(|\sum|^2nl+|\sum|^3l)$ complexities, respectively, still with the approximation ratio of $|\sum|$. They showed that their algorithms were quite successful in practice. In [25], the authors showed that BNMAS was considerably faster than Expansion, especially when $|\sum|$ is small and/or n is large and that it outperformed Expansion in most of the test cases.

In addition to the above-mentioned approximation algorithms, heuristic algorithms, which do not normally guarantee an approximation ratio, were also proposed for the LCS problem. The *Best-Next heuristic* was proposed in [26,27] as a simple heuristic algorithm which is run in $O(|\sum|nl)$, and it was shown to be of superior results compared to some of the above-mentioned approximation algorithms, such as LR, for practical datasets. Guenoche and Vitte [28] proposed a linear-time *dynamic programming heuristic* (DPH), which was further modified by Guenoche [29]. Easton and Singireddy [30] introduced, based on the large-neighborhood search paradigm, a new algorithm called *time horizon specialized branching heuristic* (THSB), which was shown to be superior to DPH. More recently, Shyu and Tsai [25] used ant colony optimization (ACO) to solve LCS. They compared their algorithm with expansion and BNMAS algorithms by implementing and testing them over random and biological datasets obtained from NCBI [31]. According to the experimental results, ACO dominates both of the other algorithms in terms of quality and is faster than Expansion. Finally, Blum et al. proposed a constructive Beam Search algorithm, called *BS*, for the LCS problem [32]. In their algorithm, two different greedy functions were used to evaluate and compare candidate solutions. Their BS algorithm is an extension of a predecessor beam search introduced by Blum and Blesa [33]. In order to compare their BS algorithm with previous leading algorithms in the literature, Blum et al. used two types of parameter settings; one called *low time* aimed at producing quick solutions and the other called *high quality* intended for high quality solutions but at extra computation cost. They compared BS with Expansion, Best-Next, G&V, THSB and ACO algorithms over three benchmarks previously introduced in [33,30,25]. Extensive experimental results showed that Blum et al.'s BS algorithm outperforms, on average, its predecessors in terms of both quality and computation time, concluding that it is the current state-of-the-art.

In this paper, we provide an improved beam search algorithm called IBS-LCS for the LCS problem, which, on average, improves over the state-of-the-art, with respect to both quality and computation time. It has been inspired by the Blum et al.'s beam search algorithm but has the following distinguishing characteristics. First, a novel probability-based heuristic function is used as opposed to the heuristic functions used in BS and the other heuristic algorithms in the literature. We believe that our proposed heuristic function performs better than the existing ones in domains where the given strings are expected to be independent. Second, in contrary to the BS algorithm, it does not use upper bounds for pruning the search tree. Third, BS checks, at each level of the search tree, whether each new candidate solution is dominated by an existing candidate solution. To do so, it compares each new candidate solution with every existing one until it is found to be dominated by some of them or compared by all. However, we use a pre specified number of 'best' solutions, at each level of the search tree, as potential dominators for the other candidate solutions. Instead, the time saved by avoiding the extra comparisons and calculation of upper bounds is invested into larger values of beam size. As the consequence of the above-mentioned modifications, IBS-LCS outperforms the state-of-the-art not only with respect to quality but also with respect to run time, for most standard benchmarks. More specifically, IBS-LCS outperforms the state-of-the-art, on average, over 4 out of the 5 benchmark datasets used in [32]. The only benchmark for which IBS-LCS is not suggested is composed of strings which are highly similar.

The rest of the paper is organized as follows. Section 2 provides basic notations and definitions used in the rest of the paper. In Section 3, we present our proposed algorithm. The new heuristic function is developed in Section 4 followed by a brief analysis of the time complexity of the algorithm. Section 5 reports the experimental results, and Section 6 concludes the paper.

2. Basic notations and definitions

Let s be a string of length m. We use s^k, where k is an integer between 1 and m inclusive, to denote the kth character of s. Let s_1 and s_2 be two strings, $A_1=\{i|i\in\mathbf{N}, i\le|s_1|\}$, and $A_2=\{i|i\in\mathbf{N}, i\le|s_2|\}$, where $\mathbf{N}$ is the set of integers greater than zero. We say that s_1 is a subsequence of s_2, and write $s_1\prec s_2$, if there is an injective function g from A_1 to A_2 such that: (1) $\forall k\in A_1$, $s_1^k=s_2^{g(k)}$ and (2) $\forall k,k'\in A_1$, $k<k'\Rightarrow g(k)<g(k')$. We call such a function g a *map of* s_1 to s_2. Note that such a map is not necessarily unique. We define the cost of a map g of s_1 to s_2 as the integer $g(s_1^k)$, where $k=|s_1|$. By an optimal map of s_1 to s_2, we mean a minimum cost map of s_1 to s_2, if any. The null string, i.e. the string of zero length, is considered to be a subsequence of any string.

Let x be a string and S be a nonempty set of strings. We write $x\prec S$ if $\forall s_i\in S$, $x\prec s_i$. The Longest Common Subsequence (LCS[1]) problem is then defined as to obtain a string x of maximum length such that $x\prec S$. By an input string, we mean a string in S. The alphabet over which the input strings are defined is denoted by $\sum$; we assume $|\sum|>1$. We use n to denote the number of input strings; that is, $n=|S|$. Since LCS can be efficiently solved for $n=2$, we assume $n>2$. We further assume that $S=\{s_1,\ldots,s_n\}$; that is, the input strings are denoted by the small letter s indexed from 1 to n. We use m_i to refer to $|s_i|$ and assume $m_i>0$, $i=1,\ldots,n$. In the case all the input strings are of the same length, we use m to denote their length; otherwise, m denotes $\max\{m_i, i=1,\ldots,n\}$. We use (possibly indexed) x to denote a candidate solution. A candidate solution x is called feasible if $x\prec S$; it is otherwise called infeasible. A feasible candidate solution x is optimal if there exists no other feasible solution of a length greater than $|x|$.

Let x be a feasible candidate solution. We use $p_i(x)$ to denote the cost of the optimal map of x to s_i. Then $q_i(x)$ is defined as $m_i-p_i(x)$. By $r_i(x)$, we mean the string obtained by deleting the first $p_i(x)$ characters from s_i (see Fig. 1), and $R(x)$ is defined as the set $\{r_i(x), i=1,\ldots,n\}$. By a random string in this paper, we mean a string each of whose characters obtained by uniformly-randomly selecting one of the characters in $\sum$. Finally, we use $Pr(.)$ to denote the statistical probability function. Although there are two types of beam search, namely constructive and perturbative (Local Beam Search [34]), we use beam search in this paper to refer to the former.

3. The proposed algorithm

The beam search algorithm, in its standard form, is a deterministic, yet heuristic, tree search. It is similar to the breath-first search algorithm except that it does not keep all the leaves but only β of them, where $\beta>0$ is called the beam size. It turns to a pure constructive greedy heuristic in the case $\beta=1$; it also turns to the breath-first search if β is large enough to keep all the

[1] It is also referred to as k-LCS, where $k=|S|$, in the literature.

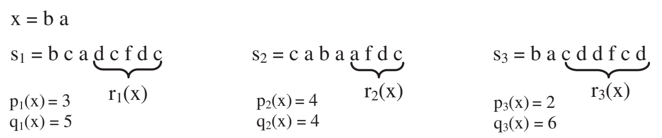


Fig. 1. An instance of the LCS problem with $S=\{s_1,s_2,s_3\}$. A candidate solution is $x=ba$, for which $p_i(x)$, $q_i(x)$, and $r_i(x)$, $i=1,2,3$, are illustrated.

leaves. Therefore, the beam size β may be thought of as a parameter to control a balance between greediness and exhaustiveness and is usually used to avoid excessive running times. Algorithm 1 presents our basic beam search algorithm for the LCS problem.

As a (constructive) beam search, the algorithm starts with an empty candidate solution, here a null string, and incrementally builds longer (feasible) candidate solutions by appending to them characters drawn from the alphabet. However, if the number of feasible candidate solutions exceeds the beam size β, then at most β of them can be kept and the rest must be discarded. In order to determine which candidate solutions to keep, a heuristic function $h(.)$ is used to evaluate the candidate solutions. The set of candidate solutions in this algorithm is denoted by B, which initially contains the null string only. There are three main steps in the while loop. In Step 1, each candidate solution x_i in B is extended by appending at its end a character drawn from $\sum$. The result would then be $|\sum|$ new candidate solutions, one per each letter in $\sum$, for each x_i in B. However, only the feasible ones, determined using the function feasible(.), are kept in the set C. Therefore, the set C contains at most $\beta.|\sum|$ (feasible) candidate solutions. In Step 2, each of the candidate solutions in C is evaluated using the heuristic function. Based on this evaluation, in Step 3, a list κ-best is constructed which consists of the κ best candidate solutions. Each candidate solution in C is then checked for being *dominated* by any member of κ-best and removed if so. A candidate solution x_j is dominated by another candidate solution x_k if $p_i(x_j) \geq p_i(x_k)$, $i=1,2,\ldots,n$. After the removal of the dominated candidate solutions, the best β of the remaining ones in C are selected to make the new set B of candidate solutions for the subsequent iteration of the while loop. Steps 1–3 are repeated within the while loop until C is empty, in which case the algorithm returns a member of B and terminates.

Algorithm 1.

```
**The basic beam search algorithm for LCS**
//input: $S=\{s_1,s_2,\ldots,s_n\}$, $n>2$, each $s_i$ a string of at least one character. The alphabet of characters used in any of the strings is denoted by $\sum$
//output: a string $x$ such that $x \prec S$
//parameter: 1. the beam size $\beta$ and 2. the number κ of potential dominators
//initialization
B={ ”” } //the set of candidate solutions initially contains the null string
finished=false
while Not finished
{
  //step 1: extension
  $C=\{\}$
  for each $x_i \in B$
    for each letter $l \in \sum$
      $x=$ the string obtained by adding $l$ at the end of $x_i$
      if feasible($x$)
        $C=C\cup\{x\}$
  //step 2: calculation of heuristic values
  for each $x_i \in C$
    calculate heuristic value $h(x_i)$//$h(.)$ is the heuristic function
  //step 3: selection
  if $C=\{\}$
    finished=true
  else
    $\kappa$_best= a set of $\kappa$ best members of $C$
    for each $x_i \in C$
      if $x_i$ is dominated by any member of $\kappa$_best
        $C=C-\{x_i\}$
  B=a set of $\beta$ best members of $C$//$B=C$ if $|C| \leq \beta$
}
return an $x \in B$
```

Our beam search algorithm is different from the beam search algorithm presented in [32]. First, pruning the search tree in our algorithm is simply performed by discarding candidate solutions dominated by some member of κ-best, at each level of the tree (in addition to the standard beam search restriction of $|B|<\beta$). However, Blum et al. performs this type of pruning by comparing each new candidate solution with every other candidate solution, at each level of the search tree. In other words, a new candidate solution may be compared with $\beta.|\Sigma|-1$ other candidate solutions, where it is at most compared with κ ones in our algorithm. Second, Blum et al. adopts a further type of pruning, which is pruning by upper bounds. This type of pruning is not performed in our algorithm. Instead, the computational cost saved by avoiding this type of pruning and by reducing the number of comparisons for dominance pruning is invested into larger values of beam size than those of [32]. Another distinction between the algorithm proposed in this paper and the algorithm in [32] is the use of a novel heuristic function in our algorithm. We believe that the heuristic function proposed in this paper performs better than the existing ones, at least in domains where the given strings are expected to be independent. Finally, we have devised a dynamic programming approach to quickly calculate the heuristic values at some extra, yet affordable, memory cost. As a beam search, our algorithm is of polynomial-time complexity in the input size (n, m, and $|\sum|$).

4. The probabilistic heuristic

Let x be a candidate solution, we use as the heuristic function $h_k(x)$ the probability for $s \prec R(x)$, where s is a random string of length k. This heuristic function is used to evaluate and compare candidate solutions. However, it is dependent not only on x but also on k. Therefore, for a fair comparison, k has to be the same for all candidate solutions which are to be compared. We present at the end of this section a simple formula to determine a value for k. However, how to find the best value for k still remains as an open question.

To determine $h_k(x)$, we make the assumption that the strings in S are 'independent'. To be precise, we assume that, for a given random string s, $\Pr(s \prec s_i)=\Pr(s \prec s_i | s \prec s_j)$, for all distinct strings s_i and s_j in S. Under this assumption, we have:

$$\Pr(s \prec R(x)) = \prod_{i=1}^{n} \Pr(s \prec r_i)$$

We now show how to determine $\Pr(s \prec r_i)$, $i=1,\ldots,n$. The following theorem establishes a recurrence for the function $\Pr(s \prec r)$, which can then be efficiently determined via dynamic programming.

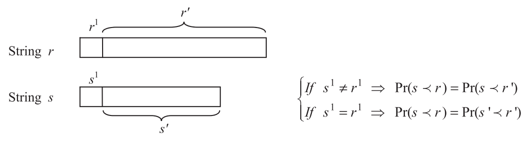

String r
r^1
r'
String s
s^1
s'
$If\ s^1 \neq r^1 \Rightarrow \Pr(s \prec r) = \Pr(s \prec r')$
$If\ s^1 = r^1 \Rightarrow \Pr(s \prec r) = \Pr(s' \prec r')$

Fig. 2. Recurrence can be obtained to determine whether a string is a subsequence of another. Two strings s and r such that $|s|>0$ and $|r|>0$ are shown, and s' and r' are defined as the strings obtained by deleting the first characters from s and r, respectively. In the case the first characters of s and r are different, the probability for s being a subsequence of r is equivalent to the probability for s being a subsequence of r'; otherwise, it is equivalent to the probability for s' being a subsequence of r'.

Theorem. *Let r be a string of length q and s be a random string of length k, $q \geq 0$, $k \geq 0$. Then*:

$$\Pr(s \prec r) = \begin{cases} 1 & \text{if } k=0 \\ 0 & \text{if } k>q \\ \frac{1}{|\Sigma|}\Pr(s' \prec r') + \frac{|\Sigma|-1}{|\Sigma|}\Pr(s \prec r') & \text{otherwise} \end{cases}$$

where s' and r' are the (possibly null) strings obtained by deleting the first letters from s and *r, respectively, when $|s|>0$ and $|r|>0$.*

Proof. It is clear by the definition of subsequence that the null string is a subsequence of every string and that a string cannot be a subsequence of a shorter one. Therefore, the first two cases of $k=0$ and $k>q$ are trivial. We now concentrate on the remaining case, i.e. when $0<k\leq q$. In this case, each of the strings s and r has at least one character; hence, s' and r' are well-defined. One, and only one, of the following two cases holds (see Fig. 2):

Case (i): $s^1 \neq r^1$. In this case, $s \prec r$ if and only if $s \prec r'$. Therefore $\Pr(s \prec r) = \Pr(s \prec r')$.

Case (ii): $s^1 = r^1$. In this case, we show that $s \prec r$ if, and only if, $s' \prec r'$. To that end, first assume $s' \prec r'$. Since $s^1 = r^1$, this implies $s \prec r$. Now assume that $s \prec r$. Then there exists some map g of s to r. This in turn splits into two possible cases of $g(1)=1$ and $g(1)\neq 1$. In either case, $g(i)>1$, $\forall i=2,\ldots,k$. This means that the injective function g' defined as $g'(i)=g(i+1)-1$ is a map of s' to r', which implies $s' \prec r'$. Therefore, $\Pr(s \prec r) = \Pr(s' \prec r')$.

Since the probability for case (ii) is $1/|\sum|$, we conclude:

$$\Pr(s \prec r) = \frac{1}{|\Sigma|}\Pr(s' \prec r') + \frac{|\Sigma|-1}{|\Sigma|}\Pr(s \prec r'). \qquad \square$$

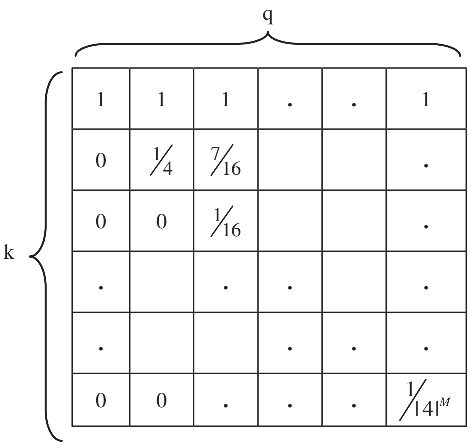
The probability $\Pr(s \prec r_i)$ is only dependent on $|s|$ and $|r_i|$, given an alphabet $\sum$, because s is assumed to be random. Therefore, we simply use $P(k,q)$, where $k=|s|$ and $q=|r_i|$, to refer to $\Pr(s \prec r_i)$. Using Theorem 1, this means:

$$P(k,q) = \begin{cases} 1 & \text{if } k=0 \\ 0 & \text{if } k>q \\ \frac{1}{|\Sigma|}P(k-1,q-1) + \frac{|\Sigma|-1}{|\Sigma|}P(k,q-1) & \text{othewise} \end{cases}$$

Based on dynamic programming, Algorithm 2 calculates $P(k,q)$ for $0 \leq k \leq m$ and $0 \leq q \leq m$ (recall that m is the length of the longest input string). The first few rows and columns of the resulting array P, for $|\sum|=4$, are shown in Fig. 3.

Algorithm 2.

```
Populates the two-dimensional array P such that
P[k][q]=P(k,q)
Let sigSize be $|\sum|$
for $q=0$ to $m$
    for $k=0$ to $m$
        if $k=0$
            $P[k][q]=1$
        else
            if $k>q$
                $P[k][q]=0$
            else
                $P[k][q]=(1/\text{sigSize})*P[k-1][q-1]+$
                        $((\text{sigSize-1})/\text{sigSize})*P[k][q-1]$
```



	q					
k	1	1	1	.	.	1
	0	1/4	7/16			.
	0	0	1/16			.
	.		.	.		.
	.			.	.	.
	0	0	.	.	.	$1/\lvert 4\rvert^M$

Fig. 3. The first few rows and columns of the two-dimensional array P, populated by dynamic programming (Algorithm 2) to hold the probability values used to calculate the proposed heuristic function.

In order to determine k, we use the formula $k=\min\{q_i(x), i=1,\ldots,n, x\in C\}/|\sum|$, where C is the set of candidate solutions to be compared; we set k to 1 if the above formula gives 0. Our intuition for this formula was that the probability for finding a longer common subsequence of $R(x)$, for a candidate solution x, is decreased as $|\sum|$ is increased and as $\min\{q_i(x), i=1,\ldots,n\}$ is decreased. However, this is a fairly simple support for the formula and how to determine the best value for k requires further investigation. Nevertheless, as will be seen in the subsequent section, the current formula yields satisfactory results, although there is still room for further improvement in this regard.

Having all the required information, we now analyze the time complexity of the algorithm. Before the while loop of Algorithm 1, two data structures are populated. First, a three-dimensional $(n \times m \times |\sum|)$ array to keep, for each location j, $j=1,\ldots,m$, of each string s_i, $i=1,\ldots,n$, where the next occurrence of each alphabet letter in the string is. This array is used in the algorithm in order to run the function feasible(.) in $O(n)$. Using dynamic programming, this array can be populated in $O(nm|\sum|)$. We avoid the pseudo-codes here for brevity. The second data structure populated before the main loop of the algorithm is the two-dimensional array P, used to keep all possible values of $P(k,q)$, which is populated in $O(m^2)$ again using dynamic programming.

Inside the while loop, Step 1 requires $O(\beta\ |\sum|n)$, taking the advantage of the three-dimensional array already populated to perform each feasibility check in $O(n)$. Step 2 determines the heuristic values for all the candidate solutions (at most $\beta.|\sum|$) in C, each requiring $O(n)$ (multiplication of n probability values retrieved from the two-dimensional array P). Therefore, it yields $O(\beta|\sum|n)$ as for the complexity of this step too. In Step 3, at first the dominance pruning is performed, which required the selection of κ best candidate solutions, in $O(\kappa lg(\beta|\sum|))$ using red-black trees[2] (recall that $|C| \leq \beta|\sum|$), together with the dominance

[2] Our current code does not use red-black trees and requires $O(\kappa\beta|\sum|)$ for this step.

checks, in $O(\kappa\beta|\sum|n)$. Then, still in Step 3, the best (at-most) β members of C are selected to construct the new set B, which can be performed in $O(\beta lg(\beta|\sum|))$ using red-black trees. Since the while loop iterates L^* times, where L^* is the length of the LCS returned by the algorithm, the algorithm is run in $O(m^2+nm|\sum|+L^*(\beta\ lg\ \beta+\kappa\beta|\sum|n))$.

5. Experimental results

In this section, we report the results of comparing our proposed algorithm with the BS (Beam Search) algorithm proposed by Blum et al. in [32]. Although there are other recent algorithms such as those proposed in [25] and [30], we do not compare our algorithm with those ones, because such algorithms have already been evaluated and compared with BS [32] and BS is known as to be the current state-of-the-art. We implemented our algorithm in Java using eclipse Platform. In order to provide meaningful comparison of run time, we did not use a recent machine. Instead, we tried to find in our department a machine with specifications as close as possible to the machine used in [32]; that is, we used a Pentium (R) IV desktop machine with 3.40 GHz clock speed, 1 GB of RAM, and 2 MB of L2 cache. The machine we used, however, should be even slower than the machine used in [32], based on the CPU performance test benchmarks in [35]; our machine′s benchmark is ranked 541, whereas the benchmark's rank for the machine used in [32] is 805.

We used four of the five datasets used in [32], namely ES,[3] ACO-random, ACO-rat, and ACO-virus.[4] The ES benchmark was originally used in [30], and the other three datasets were previously used in [25]. We do not propose our algorithm for domains, such as the BB benchmark used in [32], where the input strings are highly related because of the independence assumption made in developing our heuristic function. However, in order to observe the experimental outcomes and its consistency with the theoretical support for the proposed heuristic function, we have also included the results on the BB benchmark first introduced in [33]. We used the parameters $\beta=200$ and $\kappa=7$ in our algorithm.

Table 1 compares the results of our algorithm with those reported in [32] for the ES benchmark. The first two columns in this table show the alphabet size and the number and the length of the input strings (the strings of an instance are all of the same length). The next four columns report the results for BS as specified in [32]. There are two types of runs in their experiments: low-time and high-quality. In the former, low run time is of the main concern, whereas high quality of the solutions is the main goal in the latter. We have included the results of both runs, calling them BS-low-time and BS-high-quality, respectively. The third and the fourth columns show the average length of the returned LCS and the run time for BS-low-time and the fifth and the sixth columns show these quantities for BS-high-quality. The next two columns show the respective values for our algorithm. Finally, the last column calculates the improvement percentage α defined as $(L_2-L_1)/L_1$, where L_2 and L_1 are the average lengths of the solutions returned by IBS-LCS and BS-high-quality, respectively, for the corresponding datasets. Except for the last column which has up to two decimal figures, the other columns are rounded up to one decimal figure.

Table 1
Comparison of IBS-LCS and BS over the ES benchmark for various values of $|\sum|$, n, and m. The last column shows the quality improvement with respect to BS-high-quality. As can be seen, IBS-LCS provides superior quality than BS-high-quality in all the cases, in most of which in even less time than BS-low-time.

n	***m***	**BS-low-time**		**BS-high-quality**		**IBS-LCS**		
		LLCS	**Time**	**LLCS**	**Time**	**LLCS**	**Time**	**α**
$\|\sum\|=$**2**								
10	1000	579.9	0.7	592.6	14.8	**610.2**	0.9	2.97
50	1000	516.3	3.7	521.9	43.5	**535.0**	1.5	2.51
100	1000	502.1	7.4	506	78.6	**517.3**	2.5	2.24
$\|\sum\|=$**10**								
10	1000	185.5	0.5	192.2	9.4	**199.7**	0.9	3.90
50	1000	127.9	1.5	129.6	18.8	**134.6**	1.4	3.87
100	1000	116.5	2.7	117.9	30.6	**122.0**	2.3	3.48
$\|\sum\|=$**25**								
10	2500	214.3	2.7	224.3	51.5	**231.6**	3.1	3.25
50	2500	131.3	5.5	133	76.6	**137.2**	4.6	3.14
100	2500	116.3	9.1	118.1	118.6	**121.1**	7.7	2.51
$\|\sum\|=$**100**								
10	5000	132.5	19.1	139.6	394.6	**142.1**	9.0	1.79
50	5000	67.9	27.8	69.5	490.2	**70.4**	13.6	1.24
100	5000	57.6	42.2	59	602	**59.6**	69.9	1.02
								2.66

As can be seen in Table 1, in all the 12 cases IBS-LCS provides solutions of higher quality than those of the BS-high-quality. In 8 out of the 12 cases, it does so in even less time than those of BS-low-time. In all the remaining four cases, it takes less time than those of the BS-high-quality. On average, our algorithm achieves about 2.66% improvement in solution quality compared to BS-high-quality while it consumes less time than BS-low-time by about 6.5% (not shown in the table).

The results of our algorithms on the ACO-random, ACO-rat, and ACO-virus datasets are presented and compared with the state-of-the-art in Tables 2–4, respectively. The definitions for the columns of these tables are the same as those of Table 1. As can be seen in Table 2, in none of the 20 cases, IBS-LCS is of an inferior solution quality compared to BS-high-quality. In 14 out of 20 (70% of) cases, it provides solutions of higher quality than those of BS-high-quality. In 12 out of these 14 cases, it does so in even less time than BS-low-time. In one of the remaining 2 cases (the second row of table), it consumes the same amount of time (0.5 s), and in only one case (the first row) it takes more time than that of BS-low-time, where it still gives a higher quality in less time compared to BS-high-quality.

The results of IBS-LCS on the ACO-rat datasets are slightly worse than those of ES and ACO-random, because it is outperformed, with respect to solution quality, by BS-high-quality in 3 out of 20 (15% of) cases. However, in all these cases, it consumes significantly less time even than BS-low-time. In 6 out of the remaining 17 cases, it provides the same quality as BS-high-quality, but again in significantly less time. In 9 out of the remaining 11 cases, it is of higher quality than BS-high-quality while faster than BS-low-time. Only in the first two cases of Table 3, it is not quicker than BS-low-time, though it is of higher quality than BS-high-quality.

On the ACO-virus benchmark, IBS-LCS always performs better than (in 17 cases) or the same as (in 3 cases) BS-high-quality, with respect to solution quality. Except for the first two cases, it is also always faster than BS-low-time. In all the cases where it is of the same quality as BS-high-quality, it is faster than BS-low-time. Similarly, in both the cases where it is not faster than BS-low-time, it is of higher quality than BS-high-quality.

[3] Downloaded from http://www2.imse.ksu.edu/~teaston/publications.php on 23 September 2009. We noticed inconsistencies with some of the instances which we ignored.

[4] Received from Dr. C. Blum; also available on NCBI using the accession numbers specified at http://www.csie.mcu.edu.tw/~sjshyu/resource/accno-aco_lcs.html.

Table 2
Comparison of IBS-LCS and BS over ACO-random benchmark for various values of $|\sum|$, n, and m. The last column shows the quality improvement with respect to BS-high-quality. As can be seen, IBS-LCS provides the same or superior quality compared to BS-high-quality in all the cases, in most of which in even less time than BS-low-time.

n	*m*	**BS-low-time**		**BS-high-quality**		**IBS-LCS**		
		LLCS	**Time**	**LLCS**	**Time**	**LLCS**	**Time**	**α**
$\|\sum\|$=**4**								
10	600	200	0.3	211	9.8	**218**	0.4	3.32
15	600	190	0.5	194	13.2	**203**	0.5	4.64
20	600	178	0.7	184	14.9	**191**	0.5	3.80
25	600	174	0.9	179	15.8	**185**	0.5	3.35
40	600	162	1.4	167	21	**172**	0.7	2.99
60	600	157	2.1	161	27.6	**165**	0.8	2.48
80	600	151	2.7	156	33.5	**161**	1.0	3.21
100	600	150	3.5	154	40.3	**158**	1.2	2.60
150	600	146	5	148	56.4	**151**	1.5	2.03
200	600	144	6.9	146	74.3	**150**	2.1	2.74
$\|\sum\|$=**20**								
10	600	58	0.7	**61**	33.3	**61**	0.5	0.0
15	600	49	0.9	**51**	37.6	**51**	0.4	0.0
20	600	43	1.1	**47**	39.5	**47**	0.5	0.0
25	600	41	1.3	43	39.5	**44**	0.5	2.33
40	600	37	1.7	37	43.2	**38**	0.6	2.70
60	600	34	2.6	34	46.5	**35**	0.8	2.94
80	600	32	3.2	**32**	53.2	**32**	1.0	0.0
100	600	30	3.9	**31**	59.2	**31**	1.2	0.0
150	600	28	5.7	**29**	75.6	**29**	1.5	0.0
200	600	27	7.9	27	98	**28**	1.9	3.70
								2.14

Table 3
Comparison of IBS-LCS and BS over the ACO-rat benchmark for various values of $|\sum|$, n, and m. The last column shows the quality improvement with respect to BS-high-quality. As can be seen, IBS-LCS provides inferior quality in 3 cases, the same quality in 6 cases, and superior quality in 11 cases, compared to BS-high-quality. In most of the cases, it is remarkably faster than BS-low-time.

n	*m*	**BS-low-time**		**BS-high-quality**		**IBS-LCS**		
		LLCS	**Time**	**LLCS**	**Time**	**LLCS**	**Time**	**α**
$\|\sum\|$=**4**								
10	600	189	0.3	191	9.7	**199**	0.4	4.19
15	600	163	0.4	173	12.3	**182**	0.4	5.20
20	600	160	0.6	163	12.6	**168**	0.4	3.07
25	600	160	0.8	162	15.8	**166**	0.4	2.47
40	600	142	1.2	**146**	9.4	**146**	0.5	0.0
60	600	143	1.9	144	26.7	**147**	0.7	2.08
80	600	131	2.3	135	31.8	**141**	0.9	4.44
100	600	129	3	**132**	38.5	**132**	1.0	0.0
150	600	120	4.2	121	51.1	**124**	1.3	2.48
200	600	117	5.6	**121**	69.1	120	1.6	−0.83
$\|\sum\|$=**20**								
10	600	65	0.7	69	27.4	**70**	0.5	1.45
15	600	57	1.1	60	36.7	**61**	0.5	1.67
20	600	50	1.2	51	34.4	**53**	0.5	3.92
25	600	49	1.4	**51**	39	50	0.5	−1.96
40	600	46	2	**49**	47	**49**	0.6	0.0
60	600	44	3.2	**46**	60.3	**46**	0.8	0.0
80	600	42	4	**43**	64.4	**43**	1.0	0.0
100	600	37	4.5	38	64.8	**39**	1.1	2.63
150	600	35	6.7	**36**	77.8	**36**	1.3	0.0
200	600	31	8.3	**33**	101	32	1.7	−3.03
								1.39

An interesting observation of Tables 1–4 is that the improvement in solution quality usually decreases, while the run-time of BS grows rapidly, by increasing n or $|\sum|$. Fig. 4 compares the growth of the run time for IBS-LCS and BS, by increasing the number n of strings, for the ACO-virus instances with $|\sum|=4$. The corresponding graphs for $|\sum|=20$ are also depicted in Fig. 5. As can be seen in these graphs, the run-time of BS grows rapidly, as n increases, whereas the run-time of IBS-LCS grows fairly slowly. This suggests that although BS can achieve solutions of close quality to those of IBS-LCS for larger values of n, its computational cost grows more rapidly compared to that of IBS-LCS. Comparing the run-time of the algorithms for different $|\sum|$, Fig. 6 shows the average run-time of the algorithms per each value of $|\sum|$, for both BS and IBS-LCS. As can be seen in Fig. 6, by increasing the cardinality of the alphabet from 4 to 20, the average run-time of BS rises by about 86% from 34.2 to 63.91. However, as the result of this, the average run time of IBS-LCS is increased from 0.93 to 1.03, i.e. by about 11%. This also suggests that IBS-LCS should better suit larger alphabets.

Table 4
Comparison of IBS-LCS and BS over the ACO-virus benchmark for various values of $|\sum|$, n, and m. The last column shows the quality improvement with respect to BS-high-quality. As can be seen, IBS-LCS provides the same or better quality compared to BS-high-quality in all the cases, in most of which in even less time than BS-low-time.

n	*m*	**BS-low-time**		**BS-high-quality**		**IBS-LCS**		
		LLCS	**Time**	**LLCS**	**Time**	**LLCS**	**Time**	**α**
$\|\sum\|$=**4**								
10	600	203	0.4	212	11.6	**225**	0.5	6.13
15	600	192	0.5	193	15.4	**203**	0.5	5.18
20	600	179	0.7	181	17.2	**189**	0.5	4.42
25	600	178	0.9	185	17.9	**193**	0.5	4.32
40	600	158	1.3	162	21.9	**168**	0.6	3.70
60	600	153	2	158	29.1	**165**	0.8	4.43
80	600	148	2.6	153	36	**158**	0.9	3.27
100	600	149	3.4	150	43.9	**158**	1.2	5.33
150	600	143	5	148	64.5	**156**	1.7	5.41
200	600	143	6.8	145	84.5	**154**	2.1	6.21
$\|\sum\|$=**20**								
10	600	67	0.7	**75**	27.2	**75**	0.5	0.0
15	600	58	1	**63**	38.6	**63**	0.5	0.0
20	600	55	1.2	57	40.3	**60**	0.5	5.26
25	600	50	1.4	53	38.9	**54**	0.5	1.89
40	600	47	2.1	**49**	48.4	**49**	0.7	0.0
60	600	44	3.1	45	56.1	**47**	1.0	4.44
80	600	43	4	44	67.4	**45**	1.1	2.27
100	600	41	5	43	74.2	**44**	1.5	2.33
150	600	43	7.8	44	108	**45**	1.8	2.27
200	600	43	11	43	140	**44**	2.2	2.33
								3.46

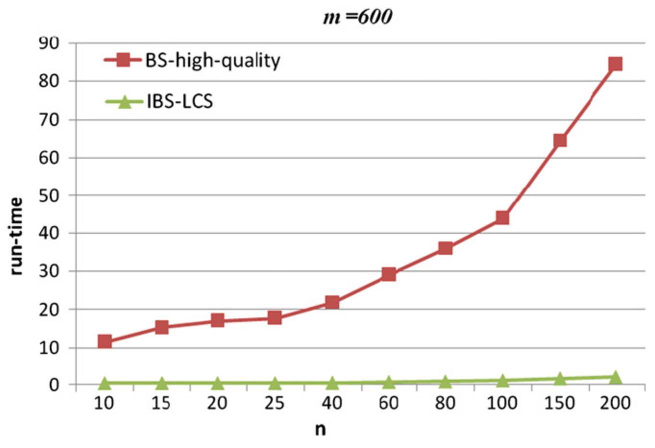


Fig. 4. Comparison of IBS-LCS and BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\sum|=4$.

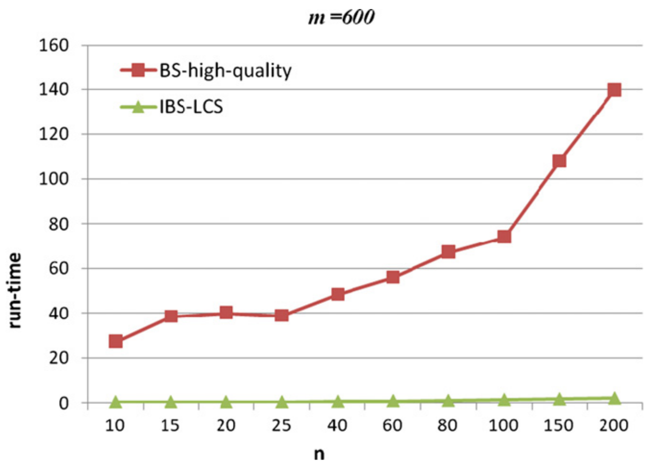


Fig. 5. Comparison of IBS-LCS and BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\sum|=20$.

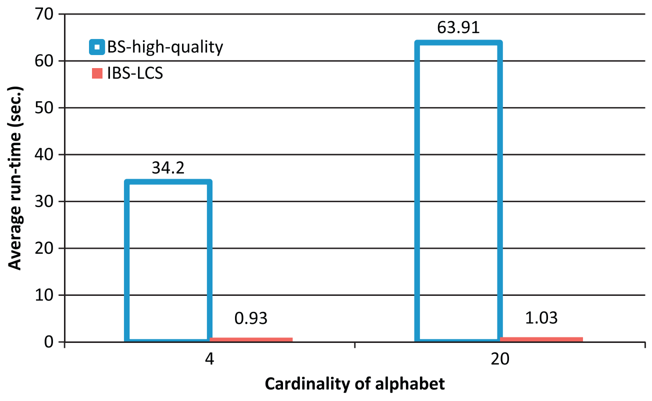


Fig. 6. Comparison of average run-time of IBS-LCS and BS-high-quality, for alphabet sizes of 4 and 20, on the ACO-virus benchmark.

Table 5
Comparison of IBS-LCS and BS over the BB benchmark for various values of $|\sum|$, n, and m. The last column shows the quality improvement with respect to BS-high-quality. As can be seen, IBS-LCS provides inferior quality compared to BS-high-quality in half of the cases. It still provides higher quality solutions compared to BS-low-time in majority of cases.

n	m	BS-low-time		BS-high-quality		IBS-LCS		
		LLCS	Time	LLCS	Time	LLCS	Time	α
$\|\sum\|=2$								
10	1000	613.2	0.6	648	13.6	**672.1**	1.1	3.72
100	1000	531.6	6.3	541	72.5	**554.2**	2.4	2.44
$\|\sum\|=4$								
10	1000	477.3	0.9	534.7	18.1	**543.7**	1.5	1.68
100	1000	350.7	9.3	**369.3**	121.6	361.7	2.8	−2.06
$\|\sum\|=8$								
10	1000	420	0.7	**462.3**	21.2	461.9	2.2	−0.09
100	1000	241.5	10.6	**258.7**	154.7	240.9	3.5	−6.88
$\|\sum\|=24$								
10	1000	382.6	1.3	**385.6**	37.4	**385.6**	4.4	0.0
100	1000	140.3	13.5	**147.7**	268.3	130.5	4.9	−11.6
								−1.60

Table 6
Average improvement achieved by IBS-LCS with respect to solution quality and run-time over BS on the various benchmarks. As can be seen, IBS-LCS is of higher quality and less run-time on majority of the benchmarks.

Benchmark	Quality improvement		Run-time improvement	
	vs. BS-low-time	vs. BS-high-quality	vs. BS-low-time	vs. BS-high-quality
ES	5.04	2.66	6.5	93.9
BB	4.3	−1.6	−42.38	93.99
ACO-random	5.12	2.14	51.89	97.6
ACO-rat	4.74	1.39	55.29	97.69
ACO-virus	7.1	3.46	53.44	97.76
Average	**5.43**	**1.99**	**36.89**	**96.75**

We now present the results of running our proposed algorithm on the BB benchmark. Recall that we do not propose our heuristic function for such domains where the input strings are highly related. The reason for this restriction is that we presumed that the input strings are independent in order to facilitate the development of a statistical formula for our proposed heuristic. How to extend the formula to the case where no such independence can be assumed is an open question which we leave as a potential future work. However, in order to see the outcome of running the proposed algorithm on such datasets as BB, we present the results.

Table 5 shows the results of running IBS-LCS on the BB benchmark. The definition of the columns of Table 5 is similar to those of Tables 1–4. As can been in Table 5, in the first three cases, IBS-LCS improves over BS_high-quality with respect to the quality of solutions. In only one case it gives the same quality as BS-high-quality, and in the rest (i.e. 4 out of 8 cases) it achieves less quality than BS-high-quality. However, in all the case, its run-time is remarkably less than that of BS-high-quality. Comparing with BS-low-time, in only 2 out of 8 cases (the sixth and the last cases), it yields inferior quality. In all the other cases (i.e. 6 out of 8 cases), it provides solutions of higher quality. In half of the cases, it takes less time and in the rest it takes more time than BS-low-time.

Table 6 summarizes the comparison of IBS-LCS with BS-low-time and BS-high-quality with respect to both quality and run-time over all the five benchmarks. With respect to quality, the highest improvement is due to the ACO-virus benchmark followed by ES and ACO-random. The worst performance, as expected, is on the BB benchmark. IBS-LCS yields the quality improvement of -1.6% over BS-high-quality (i.e. outperformed by BS-high-quality). However, its run-time is reduced by about 94% compared to BS-high-quality. Compared with BS-low-time, on average, IBS-LCS achieves the improvement ratio of 4.30% with respect to solution quality, while its run-time is worse by about 43%. Recall that the BB benchmark is of different characteristics than the other four benchmarks in [32]. Even in [32], a setting (i.e. beam size and heuristic function) different from those of the other benchmarks was chosen when running the proposed algorithm on the BB benchmark.

It is important to note that the above results were obtained by our algorithm with only a predetermined beam size of 200. In contrary to [32], we used a fixed setting for all the datasets. However, for most instances, the time consumed by IBS-LCS is significantly less than those of the state-of-the-art, and increasing the beam size could further improve the solution quality of IBS-LCS.

Finally, to show how important the setting of k, used in the proposed heuristic function is, we performed another experiment, where we changed k by $\pm 5\%$ and $\pm 10\%$ and observed the outcome on two of the benchmarks, namely ACO-virus and BB, which are shown in Tables 7 and 8, respectively. As can be seen in the first half of Table 7, the quality can change by up to 3 units (but mostly one unit) as k varies. In the second half of Table 7, however, the quality remains intact as k varies, except for one out of the ten

Table 7
Changes in solution quality by IBS-LCS as a result of changing the value of k, on the ACO-virus benchmark. There are five columns, from left to right, for $0.9k_{org}$, 0.95 k_{org}, k_{org}, 1.05 k_{org}, and $1.1k_{org}$, where k_{org} is the original value for k used in the previous experiments. As can be seen, most of the changes are in the first half of the table, which are up to 3 units.

n	***m***	**0.9 K_{org}**	**0.95 K_{org}**	**K_{org}**	**1.05 K_{org}**	**1.1 K_{org}**
$\vert\sum\vert$=**4**						
10	600	**226**	225	225	225	225
15	600	202	**203**	**203**	201	203
20	600	**189**	188	**189**	**189**	**189**
25	600	192	191	**193**	**193**	192
40	600	166	**169**	168	**169**	**169**
60	600	165	164	165	165	**166**
80	600	**159**	158	158	157	**159**
100	600	155	155	**158**	**158**	155
150	600	155	155	**156**	155	**156**
200	600	153	**154**	**154**	**154**	**154**
$\vert\sum\vert$=**20**						
10	600	**75**	**75**	**75**	**75**	**75**
15	600	62	62	**63**	**63**	62
20	600	**60**	**60**	**60**	**60**	**60**
25	600	**54**	**54**	**54**	**54**	**54**
40	600	**49**	**49**	**49**	**49**	**49**
60	600	**47**	**47**	**47**	**47**	**47**
80	600	**45**	**45**	**45**	**45**	**45**
100	600	**44**	**44**	**44**	**44**	**44**
150	600	**45**	**45**	**45**	**45**	**45**
200	600	**44**	**44**	**44**	**44**	**44**
Quality improvement with respect to BS-high-quality		3.15	3.16	**3.46**	3.37	3.35
Quality improvement with respect to BS-low-time		6.78	6.79	**7.10**	7.01	6.98

Table 8
Changes in solution quality by IBS-LCS as a result of changing the value of k, on the BB benchmark. There are five columns, from left to right, for $0.9k_{org}$, 0.95 k_{org}, k_{org}, 1.05 k_{org}, and 1.1 k_{org}, where k_{org} is the original value for k used in the previous experiments. As can be seen, the smallest value of $0.9k_{org}$ results in improved solutions.

n	***m***	**0.9 K_{org}**	**0.95 K_{org}**	**K_{org}**	**1.05 K_{org}**	**1.1 K_{org}**
$\vert\sum\vert$=**2**						
10	1000	666.5	670.9	**672.1**	671.7	670.4
100	1000	**561.4**	558.2	554.2	548.8	543.8
$\vert\sum\vert$=**4**						
10	1000	**544.4**	**544.4**	543.7	532.4	532.3
100	1000	**370.8**	365.4	361.7	358.6	354.3
$\vert\sum\vert$=**8**						
10	1000	**462.6**	461.9	461.9	461.7	461.7
100	1000	**245.1**	243.9	240.9	237.6	235.9
$\vert\sum\vert$=**24**						
10	1000	**385.6**	**385.6**	**385.6**	**385.6**	**385.6**
100	1000	**136.0**	133.9	130.5	131.4	129.0
Quality improvement with respect to BS-high-quality		−0.53	−0.96	−1.60	−2.19	−2.77
Quality improvement with respect to BS-low-time		5.43	4.98	4.30	3.67	3.06

cases. On average (shown at the bottom of the table), the best performance is due to the original value for k. The results are slightly different for the BB benchmark: the quality could be further improved by using a smaller k. As can be seen, in most of the cases, the quality changes as k varies. The best average quality is due to the smallest k, where it achieves 5.43% improvement over BS-low-time and −0.53% over BS-high-quality (i.e. still outperformed by BS-high-quality with respect to solution quality). This suggests that there is still room for further improvement of the algorithm by appropriately choosing the value of k.

6. Conclusion

In this paper, a deterministic algorithm for the longest common subsequence problem was developed. The algorithm is a constructive beam search which uses a new heuristic function to evaluate and compare candidate solutions. This heuristic function has been inspired by the probability theory. In devising the heuristic, it was assumed that the input strings are independent in the sense that whether or not a given candidate solution is a subsequence of one string does not affect the likelihood of it being a subsequence of another string. Positive results on several random and real datasets showed that the assumption is reasonable, although the extension of the heuristic function to the cases where input strings are highly related is expected to lead to further improved results. Relying on efficient data structures and optimized codes, the algorithm is relatively fast, run within a few seconds in most of the experimental cases.

The proposed algorithm was compared with the state-of-the-art, the BS algorithm proposed in [32], over several benchmarks.

Compared to the high-quality version of BS reported in [32] and on average over all the benchmarks, the proposed algorithm achieved about 2% improvement in solution quality while it saved more than 96% of the time used by BS. Compared to the low-time version of BS reported in [32] and on average over all the four benchmarks, it achieved about 5.4% quality improvement while saving more than 37% of the time consumed by BS.

A possible direction for future work is to relax the independence assumption and generalize the heuristic formula to the case where the input strings are highly related. Another possibility is to hybridize the proposed heuristic function with other heuristic functions proposed in the literature to make a so-called hyper heuristic for the problem.

Acknowledgement

The authors would like to thank Dr. C. Blum for his precious help and providing us with some of the datasets. Thanks also to Dr. T. Easton for kindly making the ES datasets available. Precious comments by Mrs. M. Montazerian are appreciated. Finally, thanks to the anonymous reviewer for their constructive comments.

References

[1] Aho A, Hopcroft J, Ullman J. Data structures and algorithms. Reading, MA: Addison-Wesley; 1983.

[2] Sankoff D, Kruskal JB. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. Reading, UK: Addison-Wesley; 1983.

[3] Storer J. Data compression: methods and theory. Maryland: Computer Science Press; 1988.

[4] Sellis T. Multiple query optimization. ACM Transactions on Database Systems 1988;13(1):23–52.

[5] Banerjee A, Ghosh J. Clickstream clustering using weighted longest common subsequences. In: Proceedings of the web mining workshop at the first SIAM conference on data mining, Chicago, 2001. p. 33–40.

[6] Brisk P, Kaplan A, Sarrafzadeh M. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In: Proceedings of the 41st design automation conference. IEEE Press; 2004. p. 395–400.

[7] Smith T, Waterman M. Identification of common molecular subsequences. Journal of Molecular Biology 1981;147(1):195–7.

[8] Jiang T, Lin G, Ma B, Zhang K. A general edit distance between RNA structures. Journal of Computational Biology 2002;9(2):371–88.

[9] Bafna V, Muthukrishnan S, Ravi R. Computing similarity between RNA strings. In: Proceedings of the sixth annual symposium on combinatorial pattern matching. Espoo, Finland, 1995. p. 1–16.

[10] Maier D. The complexity of some problems on subsequences and supersequences. Journal of the ACM 1978;25:322–36.

[11] Garey MR, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness. New York: W.H. Freeman and Company; 1979.

[12] Hakata K, Imai H. The longest common subsequence problem for small alphabet size between many strings. Lecture notes in computer science, vol. 650. Berlin: Springer; 1992. p. 469–78.

[13] Irving RW, Fraser CB. Two algorithms for the longest common subsequence of three (or more) strings. Lecture notes in computer science, vol. 644. Berlin: Springer; 1992. p. 214–29.

[14] Hirschberg DS. A linear space algorithm for computing maximal common subsequences. Communication of the Association for Computing Machinery 1975;18(6):341–3.

[15] Eppstein D, Galil Z, Italiano Giancarlo R. Sparse dynamic programming. II. convex and concave cost functions. Journal of the Association for Computing Machinery. 1992;39(3):546–67.

[16] Bergroth L, Hakonen H, Raitta T. A survey of longest common subsequence algorithms. In: Proceedings seventh international symposium on string processing and information retrieval, SPIRE'2000. 2000. p. 39–48.

[17] Hsu WJ, Du MW. Computing a longest common subsequence for a set of strings. BIT Numerical Mathematics 1984;24(1):45–59.

[18] Easton T, Singireddy A. A specialized branching and fathoming technique for the longest common subsequence problem. International Journal of Operations Research 2007;4(2):98–104.

[19] Singireddy A. Solving the longest common subsequence problem in bioinformatics. Master's thesis, Industrial and Manufacturing Systems Engineering, Kansas State University, Manhattan, KS, 2003.

[20] Chin F, Poon CK. Performance analysis of some simple heuristics for computing longest common subsequences. Algorithmica 1994;12(4–5): 293–311.

[21] Jiang T, Li M. On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 1995;24(5): 1122–39.

[22] Bonizzoni P, Della Vedova G, Mauri G. Experimenting an approximation algorithm for the LCS. Discrete Applied Mathematics 2001;110(1):13–24.

[23] Tsai YT, Hsu JT. An approximation algorithm for multiple longest common subsequence problems. In: Proceedings of the sixth world multi conference on systemics, cybernetics and informatics, SCI2002, 2002. p. 456–60.

[24] Huang KS, Yang CB, Tseng KT. Fast algorithms for finding the common subsequence of multiple sequences. In: Proceedings of international computer symposium, 2004. p. 90–95.

[25] Shyu SJ, Tsai C-Y. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers & Operations Research 2009;36(1):73–91.

[26] Fraser CB. Subsequences and supersequences of strings. PhD thesis, Computing Science Department Research Report, TR-1995-16, University of Glasgow, 1995.

[27] Johetla T, Smed J, Hakonen H, Raita T. An efficient heuristic for the LCS problem. In: Proceedings of the third South American workshop on string processing, WSP'96, 1996. p. 126–40.

[28] Guenoche A, Vitte P. Longest common subsequence with many strings: exact and approximate methods. Technique et Science Informatiques 1995;14(7): 897–915. [in French].

[29] Guenoche A. Supersequence of masks for oligo-chips. Journal of Bioinformatics and Computational Biology 2004;2(3):459–69.

[30] Easton T, Singireddy A. A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics 2008;14(3):271–83.

[31] National Center for Biotechnology Information (NCBI), ⟨http://www.ncbi.nlm.nih.gov/⟩.

[32] Blum C, Blesa MJ, Ibáñez ML. Beam search for the longest common subsequence problem. Computers and Operations Research 2009;36(12): 3178–86.

[33] Blum C, Blesa M. Probabilistic beam search for the longest common subsequence problem. In: Proceedings of SLS 2007—engineering stochastic local search algorithms. Lecture notes in computer science, vol. 4638. Berlin, Germany: Springer; 2007. p. 150–61.

[34] Russel S, Norvig P. Artificial intelligence: a modern approach. 2nd ed. Prentice-Hall; 2003. [Chapter 4].

[35] http://cpubenchmark.net/.