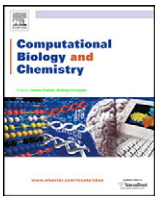
Computational Biology and Chemistry 36 (2012) 42–54

Contents lists available at SciVerse ScienceDirect

Computational Biology and Chemistry

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



A hyper-heuristic for the Longest Common Subsequence problem

Farzaneh Sadat Tabataba*, Sayyed Rasoul Mousavi

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

Article history:
Received 22 November 2011
Accepted 23 December 2011

Keywords:
Hyper-heuristic algorithms
Longest Common Subsequence
Tree search
Heuristic functions
Sequence analysis

The Longest Common Subsequence Problem is the problem of finding a longest string that is a subsequence of every member of a given set of strings. It has applications in FPGA circuit minimization, data compression, and bioinformatics, among others. The problem is NP-hard in its general form, which implies that no exact polynomial-time algorithm currently exists for the problem. Consequently, inexact algorithms have been proposed to obtain good, but not necessarily optimal, solutions in an affordable time. In this paper, a hyper-heuristic algorithm incorporated within a constructive beam search is proposed for the problem. The proposed hyper-heuristic is based on two basic heuristic functions, one of which is new in this paper, and determines dynamically which one to use for a given problem instance. The proposed algorithm is compared with state-of-the-art algorithms on simulated and real biological sequences. Extensive experimental reveals that the proposed hyper-heuristic is superior to the state-of-the-art methods with respect to the solution quality and the running-time.

1. Introduction

The Longest Common Subsequence (LCS) problem is the problem of finding 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 some characters from the given string. LCS is used in molecular biology to compare DNA or RNA sequences and to determine homology in macromolecules (Bafna et al., 1995; Jiang et al., 2002; Sankoff and Kruskal, 1983; Smith and Waterman, 1981). More similar sequences imply more similarity in structures and functions of bimolecular sequences. Also, it has applications, among others, in data compression (Storer, 1988), file comparison (Aho et al., 1983), text editing (Sankoff and Kruskal, 1983), query optimization in databases (Sellis, 1988), clustering Web users (Banerjee and Ghosh, 2001), and circuit minimization in field programmable gate arrays (FPGAs) (Brisk et al., 2004).

The LCS can be optimally solved for two input strings, using dynamic programming in $O(m_1.m_2)$, where m_1 and m_2 are the lengths of the input strings. However, the problem is NP-hard in general (Garey and Johnson, 1990; Maier, 1978), and any exact (optimal) algorithm proposed for the problem has to be of exponential-time worst-case complexity unless P = NP,[1] a case unlikely to occur according to many people in the computer science community. Existing exact algorithms include those based on dynamic programming, tree search, and integer programming. In Hakata and Imai (1992) and Irving and Fraser (1992), dynamic programming algorithms were proposed to solve the problem in $O(m^n)$, where n is the number of the input strings and m is the length of the (longest) strings. Various improvements were performed on the dynamic programming method in Eppstein et al. (1992), Hakata and Imai (1992), Hirschberg (1975) and Irving and Fraser (1992) to reduce the time complexity to $O(m^{n-1})$, which is still exponential in the number of strings. A more recent dynamic programming algorithm was proposed in Wang et al. (2010a), which is based on a divide-and-conquer technique to construct dominant points. An integer programming formulation for the LCS problem was devised in Singireddy (2003) with a time complexity of $O(m^n)$. Hsu and Du (1984) developed a tree search algorithm, which was further enhanced by Easton and Singireddy (2007) who adopted a selection heuristic and two new types of branch and bound pruning. The resulting algorithm, called *Specialized Branching (SB)*, is exponential in the length of the LCS (LLCS). The most recent tree-search method proposed for the problem, to the best of our knowledge, is an A* algorithm presented in Wang et al. (2010b). Also, some parallel algorithms were proposed to run exact LCS solvers on the multiple processors (Chen et al., 2006; Korkin et al., 2008; Wang et al., 2009). Due to their exponential complexities, all these algorithms are impractical for large input sizes, hence the use of non-optimal solutions are inevitable.

Extensive research has also been performed to devise non-optimal algorithms for the LCS problem, which aim at finding 'good',

* Corresponding author. Tel.: +98 03113915383; fax: +98 03113912451.
E-mail addresses: f.tabataba@ec.iut.ac.ir, farzaneh_tabataba@yahoo.com (F.S. Tabataba), srm@cc.iut.ac.ir (S.R. Mousavi).

[1] Note that not every NP-hard problem is solvable in polynomial-time, or solvable at all, even if P = NP.

1476-9271/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compbiolchem.2011.12.004

but not necessarily optimal, solutions within an affordable time. In general, non-optimal algorithms may be classified into two broad categories of approximation and heuristic algorithms. An approximation algorithm guarantees a bound on the approximation ratio, the ratio of the best objective value, here the length of the optimal solution, to the obtained objective value (the ratio is 1 if an optimal algorithm). A heuristic algorithm, on the other hand, does not provide such a guarantee, though it is usually of a superior performance in practice. The first approximation algorithm proposed for the LCS was *Long Run* (LR) with an approximation ratio of $|\Sigma|$ (Chin and Poon, 1994; Jiang and Li, 1995). LR simply constructs a string, as its output, using only a single character in Σ. However, it does not usually give an impressive solution. Another approximation algorithm called *Expansion* was introduced in Bonizzoni et al. (2001), which is of the same approximation ratio of $|\Sigma|$ but without the single-character restriction of LR. The complexity of Expansion was $O(nm^4 \lg m)$ which was further improved in Tsai and Hsu (2002) using minimum-spanning-trees. Huang et al. (2004) proposed two more approximation algorithms called *Best Next for Maximal Available Symbols* (BNMAS) and *Enhanced Long Run* (ELR). The complexity of these algorithms are, respectively, $O(|\Sigma|^2 nm + |\Sigma|^3 m)$ and $O(|\Sigma|nm)$. They proved that their algorithms were of the same approximation ratio of $|\Sigma|$ and demonstrated their successful performance in practice. In Shyu and Tsai (2009), the authors compared BNMAS with Expansion and showed that BNMAS is both more accurate and faster than Expansion especially when the size of the alphabet is small and the number of the strings is large.

近似演算法

The second category of non-optimal solutions consists of heuristic algorithms which do not normally guarantee a bound on the approximation ratio. The term 'heuristic' here is general and includes meta-heuristics (Blum and Roli, 2003) and hyper-heuristics (Burke et al., 2003) as well. The *Best-Next heuristic* was proposed in Fraser (1995) and Johetla et al. (1996) as a simple heuristic method with the time complexity $O(|\Sigma|nm)$. This algorithm was shown to be superior to the approximation algorithm LR on real datasets. Guenoche and Vitte (1995), devised a linear-time *dynamic programming heuristic* (DPH) which was further improved in Guenoche (2004). In Easton and Singireddy (2008), the authors referred to DPH as G&V and showed that it could obtain better results than LR and Expansion. Easton and Singireddy (2008) introduced their new algorithm called *time horizon specialized branching heuristic* (THSB), based on the large-neighborhood search paradigm, and shown that it outperformed DPH. In Shyu and Tsai (2009), ant colony optimization (ACO) was used to address LCS. The resulting algorithm was compared with Expansion and BNMAS on random and biological datasets obtained from NCBI (NCBI), with positive results. Recently, Blum et al. (2009) proposed a constructive Beam Search algorithm, called *BS*, for the LCS problem. In their algorithm, two different greedy functions were used to evaluate and compare candidate solutions. BS algorithm is an extension of a former approach introduced by Blum and Blesa (2007). In order to compare their BS algorithm with previous leading-edge algorithms in the literature, Blum et al. considered two configurations for their algorithm; one called *low time* which is a fast version of the algorithm aimed at producing quick solutions; the other called *high quality* was intended for obtaining high quality solutions but at an extra computation cost. They compared BS with Expansion, Best-Next, G&V, THSB and ACO algorithms on extensive datasets, previously used in Blum and Blesa (2007), Easton and Singireddy (2008) and Shyu and Tsai (2009). The results indicated that Blum et al.'s BS algorithm is superior to its predecessors in terms of both the solution quality and the running-time, proving as the state-of-the-art. Since proposing BS by Blum et al., three more heuristic algorithms have been proposed. The first one, called IBS-LCS, was inspired from BS though using a different heuristic function (Mousavi and Tabataba, 2012). The second algorithm, called MLCS-APP, is based on A* but with a limited number of leaves in the corresponding search tree (Wang et al., 2010b). The last one, called Deposition&Extention (DEA), which is also the most recent heuristic algorithm for the problem to the best of our knowledge, is based on a post-process technique (Ning, 2010). Neither IBS-LCS nor MLCS-APP, as reported in Mousavi and Tabataba (2012) and Wang et al. (2010b), could outperform Blum et al.'s BS over all the benchmarks used in Blum et al. (2009), although they did so for some of the cases. In particular, IBS-LCS could not outperform BS over the so-called BB benchmark, but it did obtained better average results on the other benchmarks used in Blum et al. (2009). On the other hand, MLCS-APP was compared with BS on only the so-called rat benchmark, i.e. on only 1 out of 5 benchmarks, which is rather limited. Finally, DEA was not compared with BS at all; it was only shown to be better than three primitive algorithms, namely LR, Expansion, and THSB). In this paper, a hyper-heuristic algorithm, HH-LCS, is proposed for the problem which is compared with all the three algorithms BS, IBS-LCS, and MLCS-APP on all the benchmarks used in Blum et al. (2009) and Wang et al. (2010b) for fair comparisons. It is also compared with DEA on two benchmarks used in Ning (2010). The proposed HH-LCS algorithm outperforms all the other algorithms by providing the best (average) solution quality in less (average) time, on all the benchmarks. Hence, it proves as the new state-of-the-art heuristic algorithm for LCS.

超啟發演算法

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 the proposed hyper-heuristic algorithm. The basic heuristic functions used in the hyper heuristic algorithm are described in Section 4. Section 5 reports the experimental results, and Section 6 concludes the paper.

2. Basic notations and definitions

We mainly follow the notations used in Mousavi and Tabataba (2012). Let s be a string of length m. We use $s[k]$, where k is an integer between 1 and m inclusive, to indicate the kth character of s. We also use $s[1...k]$ to indicate the string of the first k characters of s. We denote the length of s by $|s|$. Let s_1 and s_2 be two strings of the lengths m_1 and m_2, respectively. Also let $A_1 = \{i| 1 \le i \le m_1\}$ and $A_2 = \{j| 1 \le j \le m_2\}$. We say that s_1 *is a subsequence* of s_2, 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) $\forall k,k' \in A_1, k < k' \Rightarrow g(k) < g(k')$. Note that the function g is not necessarily unique. The null string, i.e. the string of zero length, is trivially a subsequence of any string.

Let x be a string and S be a nonempty set of strings. We write $x \prec S$ if $\forall s_i \in S, x \prec s_i$. Then, the LCS is defined, given the set S, as the problem of finding a longest string x such that $x \prec S$. Each string in S is called an input string. The set of the characters used in input strings is called the *alphabet* and is denoted by Σ; we assume $|\Sigma| > 1$. An *alphabet character* is an element in Σ. The number of input strings is denoted by n; that is, $n = |S|$. Because LCS can be efficiently solved for $n = 2$, we assume $n > 2$. We further assume that $S = \{s_1, ..., s_n\}$; that is, the input strings are denoted by the small letter s indexed from 1 to n. We use m_i to indicate $|s_i|$ and assume $m_i > 0$, $i = 1, ..., n$. We also use m to denote $max\{m_i, i = 1, ..., n\}$ and (possibly indexed) x to denote a candidate solution. A candidate solution x is called *feasible* if $x \prec S$; it is called *infeasible* otherwise. A feasible candidate solution x is *optimal* if no other feasible solution of a greater length exists.

For a feasible candidate solution x, we use $p_i(x)$ to indicate the smallest integer k such that $x \prec s_i[1...k]$. Then, $q_i(x)$ is defined as $m_i - p_i(x)$. Also, $r_i(x)$ indicates the string obtained by deleting

$x = d\,a$

$s_1 = d\,c\,a\,\underbrace{h\,b\,c\,f\,e\,c}_{r_1(x)}$ $p_1(x) = 3$ $q_1(x) = 6$

$s_2 = c\,a\,d\,a\,\underbrace{a\,f\,h\,c}_{r_2(x)}$ $p_2(x) = 4$ $q_2(x) = 4$

$s_3 = b\,d\,a\,\underbrace{c\,h\,b\,f\,c}_{r_3(x)}$ $p_3(x) = 3$ $q_3(x) = 5$

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

the first $p_i(x)$ characters from s_i (see Fig. 1), and $R(x)$ is defined as the set $\{r_i(x),\ i = 1, \ldots, n\}$. A candidate solution x_k is dominated by another candidate solution x_j if $p(x_j) \le p(x_k)$, $\forall i = 1, \ldots, n$. By a random string, we mean a string each character of which is a random alphabet character (obtained based on the uniform distribution). Finally, $Pr(.)$ indicates the statistical probability function.

3. The proposed algorithm

The beam search algorithm is a deterministic heuristic tree search procedure, in its standard form. As a (constructive) beam search, the algorithm starts with an initially singleton (the null string) set B of candidate solutions and in each generation, builds new longer (feasible) candidate solutions by adding alphabet characters at their end. However, the number of feasible candidate solutions is restricted to the beam size β. Fig. 2 shows the tree structure of constructive beam search algorithm with $\beta = 2$ and $|\Sigma| = 4$ used for constructing common subsequences of two strings.

Beam search is similar to the best-first search algorithm in the sense that it uses a heuristic function to evaluate the leaves but saves only β best of them. It becomes a pure constructive greedy heuristic when β is set to 1; it also turns to the breath-first search if β is large enough to keep all the leaves. Therefore, the beam size β is used to control a balance between the greediness and the exhaustiveness of the search procedure. In order to determine the β best candidate solutions, a heuristic function denoted $h(x)$ is used to evaluate each candidate solution x. Algorithm 1, adopted from Mousavi and Tabataba (2012), presents a high level pseudo code for our basic beam search algorithm BS-LCS for the LCS problem.

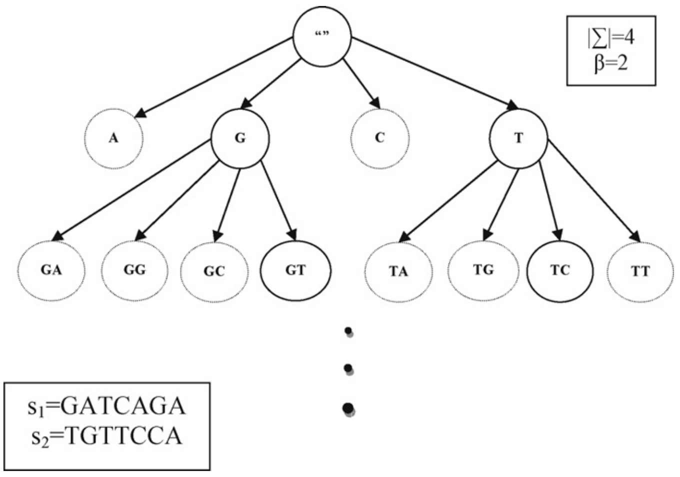


Fig. 2. Tree structure of constructive beam search algorithm with $\beta = 2$ and $|\Sigma| = 4$ used for constructing common sequence between two strings.

Algorithm 1. The basic beam search algorithm BS-LCS 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 dente by $\Sigma$
//output: a string $x$ such that $x \prec S$
//parameter: the beam size $\beta$
//parameter: the beam size $\kappa$

//initialization
$B = \{``"\}$ //the set of candidate solutions initially contains the null string only
finished = false
while Not finished
{
  //step 1: extension
  $C = \{\}$
  for each $x_i \in B$
    for each letter $l \in \Sigma$
      $x$ = 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: dominance pruning
    κ_Best_List = a list of the $\kappa$ best solutions in $C$, with respect to their
heuristic values
    for each $x_i \in C$
      if $x_i$ is dominated by some member of κ_Best_List
        $C = C - \{x_i\}$

  //step 4: selection
    if $C = \{\}$
        finished = true
    else
        $B$ = a set of $\beta$ members of $C$ which have the highest heuristic
values//$B = C$ if $|C| < \beta$
}
return an $x \in B$
```

There are four main steps in the `while` loop. In Step 1, each candidate solution x_i in B is extended by appending an alphabet character to its end. For each x_i in B, $|\Sigma|$ new candidate solutions are generated, one per each letter in Σ. However, feasible solutions are determined by `feasible(.)` function and kept in the set C. Therefore, the set C contains at most $\beta|\Sigma|$ (feasible) candidate solutions. In Step 2, candidate solutions in C are evaluated by using the heuristic function `h(x)`, and the κ best of them are designated and kept in a list called κ_Best_List to be used for possible dominance pruning. That is, in Step 3, each member of C is checked against the designated best solutions in κ_Best_List to decide whether it is dominated by any of them, in which case it is discarded from C. Finally, in Step 4, the (remaining) candidate solutions in C are compared and the best β of them are selected to construct the new set B of candidate solutions. The Steps 1–4 are repeated within the while loop until C becomes empty, in which case the algorithm returns a member of B and terminates. The proposed algorithm runs in polynomial time in the size of its inputs (n, m, and $|\sum|$) and its parameters (β and κ).

Our proposed hyper heuristic algorithm HH-LCS is built at the top of the basic beam search algorithm BS-LCS described above. Informally speaking, it acts as an outer layer to BS-LCS to determine dynamically which heuristic function to use within BS-LCS. More specifically, there are two candidate heuristic functions for `h(.)`, one the heuristic function is a new heuristic function proposed further in this paper and the other developed in Mousavi and Tabataba (2012). We refer to these two heuristic functions in the rest of the paper as *h-power*(.) and *h-prob*(.), respectively. The HH-LCS algorithm is then determines which of *h-power*(.) and *h-prob*(.) to use as for *h*(.) within BS-LCS. However, to this end, it still uses BS-LCS algorithm; it invokes BS-LCS, twice once with *h-power*`(.)` as `h(.)` and once with *h-prob* as `h(.)`. These two runs of BS-LCS are performed using a small beam size β_h, in the favor of low computation overhead. Based on the outcome of these two

為什麼要看最小字尾數？因為這是最大共有的字元數，所以理論上最小字尾數越大越好

runs of BS-LCS, either *h-power*(.) or *h-prob*(.) will be selected as the final heuristic function *h(.)* used in the final run of the BS-LCS algorithm, as illustrated in Algorithm 2.

Algorithm 2. The hyper-heuristic HH-LCS algorithm for LCS.

```
//input: $S = \{s_1, s_2, ..., s_n\}$, $n > 2$, each $s_i$ a string of at least one character
the alphabet of characters used in any of the strings is dente by $\Sigma$
//output: a string $x$ such that $x \prec S$
//parameters: 1 – the beam size $\beta_h$//the beam size for the trial phase, 2 –
  the beam size $\beta$/3- the beam size $\kappa$
len1 = BS-LCS run with $\beta_h$ and $h(.) = h\text{-}power(.)$
len2 = BS-LCS run with $\beta_h$ and $h(.) = h\text{-}prob(.)$
if(len1 > len2)
    return BS-LCS run with $\beta$ and $h(.) = h\text{-}power(.)$
else
    return BS-LCS run with $\beta$ and $h(.) = h\text{-}prob(.)$
```

The basic heuristic functions *h-power*(.) and *h-prob*(.) will be described in the subsequent section.

4. The basic heuristic functions

In this section, the basic heuristic functions *h-power*(.) and *h-prob*(.) adopted in the hyper-heuristic algorithm HH-LCS are described. The former is a new heuristic, whereas the latter was previously developed in Mousavi and Tabataba (2012). The motivation for using a hyper-heuristic mechanism to dynamically select them was the observation that, while both of these functions could yield superior results, e.g. to those of Blum et al. (2009), neither could outperform the other in all of the experimental cases. By dynamically choosing which one to use, the average quality was observed as to be better than those of their individual uses, as extensively reported further in this paper. The next two sections describe these two basic functions.

4.1. The basic heuristic function h-power(.)

In this section, the new heuristic function *h-power*(.) is described. First we define $q_{min}(x) = min\{q_i(x), i = 1, ..., n\}$, for a candidate solution x. Then, the heuristic function *h-power*(.) is defined as

$$h\text{-}power(x) = \left(\prod_{i=1}^{n} q_i(x)\right)^{\rho} \times (q_{\min}(x)) \quad (1)$$

不能只考慮min q值，個別q值也要考慮

where $0 < \rho \leq 1$. This heuristic function is a more generalized form of the heuristic η_1 used in Blum et al. (2009). To be precise, $\eta_1 = q_{\min}(x)$; that is, η_1 is a special case of *h-power*(*x*) where $\rho = 0$. The motivation behind this generalization is that all the values $q_i(x)$, $i = 1, ..., n$, should be considered and not only $q_{\min}(x)$. For example, consider the strings s_1, s_2, s_3, and the candidate solution x_1 and x_2 shown in Fig. 3. As can be seen in Fig. 3, $q_{\min}(x_1) = 5$ whereas $q_{\min}(x_2) = 4$. Using the heuristic $\eta_1 = q_{\min}(x)$ (which ignores the other values $q_i(x_1)$ and $q_i(x_2)$, $i = 1, 3$), x_1 evaluated as to be superior to x_2. However, as can be seen in Fig. 3, x_2 could lead to a much better solution (abde) that the one obtained from x_1 (de).

Our proposed heuristic function *h-power*(.) is based on all the values q_i, $i = 1, ..., n$. However, to emphasize the relative importance of $q_{\min}(x)$, the control parameter ρ, $0 < \rho \leq 1$, is used. A smaller value of ρ corresponds to higher importance of $q_{\min}$. In fact, with the constant ρ, as the number of input strings is increased, the value of $\left(\prod_{i=1}^{n} q_i(x)\right)^{\rho}$ becomes very larger than $q_{\min}(x)$ and the value of $q_{\min}(x)$ would have small or no effect on the value of *h-power*(*x*) consequently. Therefore, we suggest the following strictly decreasing, convex curve for ρ as a function of the number of input strings in Fig. 4. This curve tends to zero for large number of input strings.

Moreover, as the similarity of the input strings is increased (it means that the strings of the dataset are statistically dependent), more common characters will be found in the suffix of the strings.

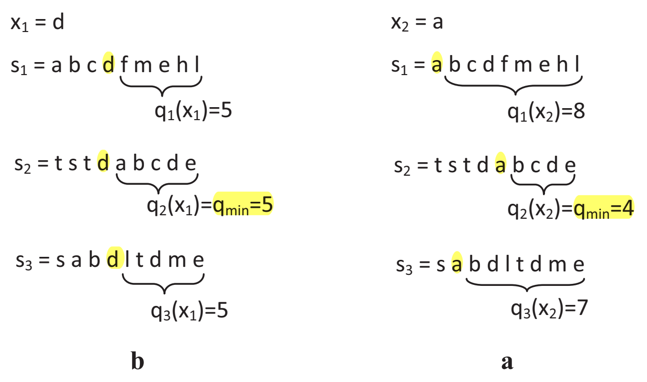

x_1 = d
s_1 = a b c d f m e h l
$q_1(x_1)$=5
s_2 = t s t d a b c d e
$q_2(x_1)$=q_{min}=5
s_3 = s a b d l t d m e
$q_3(x_1)$=5
b
x_2 = a
s_1 = a b c d f m e h l
$q_1(x_2)$=8
s_2 = t s t d a b c d e
$q_2(x_2)$=q_{min}=4
s_3 = s a b d l t d m e
$q_3(x_2)$=7
a

Fig. 3. An instance of the LCS problem with $S = \{s_1, s_2, s_3\}$ and the effect of considering all the values $q_i(x)$, $i = 1, 2, 3$, in evaluating candidate solutions. Candidate solution x_1 *selected as better solution rather than* x_2 *by* $\eta_1 = q_{\min}(x)$, whereas x_2 is could yield in better results.

不能只考慮min q值，個別q值也要考慮

In other words, the probability of finding the characters of the minimum suffix ($r_{\min}$) in the other suffixes ($r_i(x)$) increases. Therefore, $q_{\min}(x)$ should be more effective in determining the value of the heuristic. For this reason, a smaller value of ρ should be used to achieve better result. Therefore, the slope of decreasing curve for ρ, can also be determined by the similarity of the given strings as further described in Section 5.

4.2. The basic heuristic function h-prob(.)

In this section, the heuristic function *h-prob*(.), which was previously used (Mousavi and Tabataba, 2012), is described. In order to evaluate and compare candidate solutions, the heuristic function *h-prob*(.) is defined as follow:

$$h\text{-}prob(x) = Pr(s \prec R(x)) \quad (2)$$

where x is a candidate solution, and $Pr(s \prec R(x))$ indicates the probability of $s \prec R(x)$, where s is a random string of length k. We call this heuristic *h-prob*(.) (for a probabilistic heuristic). Note that *h-prob*(*x*) is dependent not only on x but also on k. Eq. (5) at the end of this section, is presented to determine a value for k. It is assumed that the strings in S are 'independent'. It means 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, $Pr(s \prec R(x))$ is calculated as follows:

$$Pr(s \prec R(x)) = \prod_{i=1}^{n} Pr(s \prec r_i) \quad (3)$$

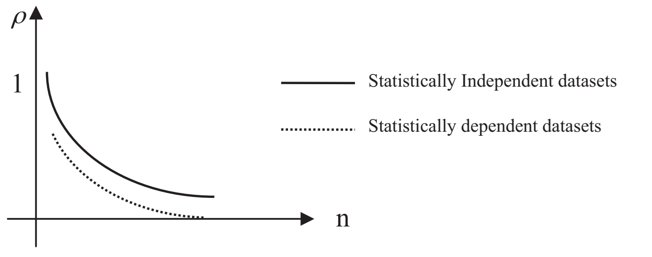


Fig. 4. A strictly decreasing curve for ρ as a function of the number of input strings. The control parameter ρ is used to emphasize the relative importance of $q_{\min}(x)$ in heuristic function H-power.

If r is a string of length q and s is a random string of length k, $q \geq 0$, $k \geq 0$, then the fallowing equation is used to determine $Pr(s \prec r)$ (Mousavi and Tabataba, 2012):

$$Pr(s \prec r) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k > q \\ \frac{1}{|\Sigma|}Pr(s' \prec r') + \frac{|\Sigma|-1}{|\Sigma|}Pr(s' \prec r') & \text{otherwise} \end{cases} \quad (4)$$

where s' and r' are the strings obtained by deleting the first characters from s and r, respectively, when $|s| > 0$ and $|r| > 0$.

The probability $Pr(s \prec r_i)$ is only dependent on $|s|$ and $|r_i|$, given an alphabet Σ, because s is assumed to be random. Therefore, we can simply substitute $Pr(s \prec r_i)$ by $P(k,q)$, where $k = |s|$ and $q = |r_i|$. That is,

$$Pr(s \prec r) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k > q \\ \frac{1}{|\Sigma|}P(k-1, q-1) + \frac{|\Sigma|-1}{|\Sigma|}P(k, q-1) & \text{otherwise} \end{cases} \quad (5)$$

The values $P(k,q)$, $0 \leq k \leq m$ and $0 \leq q \leq m$, are determined using dynamic programming for more efficiency. In order to determine k for the length of random string s, the following formula was used in Mousavi and Tabataba (2012):

$$k = \frac{\min\{q_i(x), i = 1, \ldots, n, x \in C\}}{|\Sigma|} \quad (6)$$

where C is the set of candidate solutions to be compared; k is set to 1 if the above formula gives 0. Note that using this formula, k is conversely proportional to $|\Sigma|$. This is reasonable, because the probability for finding a longer common sequence of $R(x)$ decreases as $|\Sigma|$ is increased.

In the next section we show the result of running HH-LCS on different benchmarks and compared it with most recent algorithms.

5. Experimental results

In this section, we compare the proposed algorithm HH-LCS with (to our best knowledge) three heuristic algorithms proposed for the LCS problem, namely BS (Blum et al., 2009), MLCS-APP (Wang et al., 2010b), and DEA (Ning, 2010). In addition, we include the results obtained by our basic BS-LCS algorithm once with *h-power*(.) and once with *h-prob*(.) as the adopted heuristic function. We implemented our algorithms in Java using the eclipse Platform. To compare HH-LCS with BS and MLCS-APP, we use *all* the benchmarks used in the corresponding papers (Blum et al., 2009; Wang et al., 2010b), respectively. We also compared HH-LCS with DEA over two benchmarks used in Ning (2010) (real dataset and simulated dataset obtained from the author's web site[2]). Moreover, we compare the results of HH-LCS with the results of the other algorithms as reported in their respective papers.

To provide meaningful comparisons of run time, we used a relatively old Pentium (R) IV desktop machine with 3.40 GHz clock speed, 1 GB of RAM, and 2 MB of L2 cache in order to compare HH-LCS with BS and MLCS-APP. The machine we used should even be slower that of Blum et al. (2009), based on the CPU performance tests benchmarks in cpu-benchmark; our machine's benchmark is ranked 541, whereas the benchmark's rank for the machine used in Blum et al. (2009) is 805 (it is said in Wang et al. (2010b) that a machine with the same specification as the one used (Blum et al., 2009) was used to run MLCS-APP for fair comparison). Because no precise run-time of DEA was reported in Ning (2010), we did not perform a run-time comparison with DEA and used a more recent (laptop) machine with Intel(R) Core(TM)2 Due p8600CPU and 4 GB of RAM.

We first describe how we determined the parameter value ρ for power heuristic. As shown in Fig. 4, we used a curve for ρ. More specifically, we used $\rho(n) = a \times \exp(-b \times n) + c$, where a, b, and c are constants to be determine experimentally. In order to determine these coefficients, at least three points of the curve must be given. We obtained experimentally three points and calculated the coefficients. However, we used three datasets used in Blum et al. (2009) for these constants, two for the cases where the independence condition holds and one for the cases where is does not. To determine the required three points, we used the ACO-rat and ACO-virus datasets for the former case and the BB datasets for the latter; these datasets were previously used in Blum et al. (2009) and Shyu and Tsai (2009), and they are also used in the subsequent section to compare the proposed algorithm with BS. The resulting values are, respectively ($a = 1.82$, $b = 0.066$, $c = 0.07$) and ($a = 3.0$, $b = 0.24$, $c = 0$). Except for the BB benchmark, where the sequences are highly related, we used the first set of values in all the experiments.

5.1. Comparison with BS

We used all the five benchmarks used in Blum et al. (2009), namely BB, ES,[3] ACO-random, ACO-rat, and ACO-virus.[4] The BB and ES benchmarks were originally used in Blum and Blesa (2007) and Easton and Singireddy (2008), respectively, and the other three datasets were previously used in Shyu and Tsai (2009). We set $\beta = 200$, $\beta_h = 10$ and $\kappa = 7$. The results of comparing HH-LCS with BS on these benchmarks are shown, respectively, in Tables 1–5. The first three columns in these tables show, respectively, the size of the alphabet, the number of input strings, and the length of the strings. The next four columns report the results for BS, the first two of which correspond to the low-time and the other two correspond to the high-quality runs, as reported in Blum et al. (2009). In the low-time run, as the name stands, low run time is of the main concern, whereas high quality of the solutions is the main goal in the high-quality run. For simplicity, we call these two types of runs as BS-low-time and BS-high-quality, respectively. For each run, both the (average) length of the returned LCS and the (average) running-time (in seconds) are shown. The next three pairs of columns report the respective results for HH-LCS with, respectively, the power heuristic (H-power), the probabilistic heuristic (H-prob), and the hyper heuristic (HH-LCS). The best results are shown in boldface.

Table 1 reports the results obtained by running the algorithms on the BB benchmark. As shown in this table, in all the 8 cases, the best results are obtained by one or more of our heuristics. In one case only, the seventh case, BS-high-quality also obtains the same best solution. The numbers of cases where H-power, H-prob, and HH-LCS give the best quality are, respectively, 7, 3, and 7. Hence, in most of the time, the best results are due to HH-LCS. For each setting ($\sum$, n, and m), 10 sets of sequences were generated and the result for each row is the average of the common sequences' length over all 10 instances. Therefore, the hyper heuristic may use both probabilistic and power heuristics for each row. As a result, the hyper heuristic algorithm could obtain better solutions than both of the individual heuristics.

There are some statistics underneath the table which report the improvements obtained by these three heuristics over

[2] Downloaded from http://wwwpersonal.umich.edu/kning/Data/ on October 2010.

[3] Downloaded from http://www2.imse.ksu.edu/~teaston/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 1
Comparison of power, probabilistic and hyper heuristics with Blum et al.'s beam search over BB dataset.

	n	m	BS-low-time		BS-high-quality		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
|Σ| = 2	10	1000	613.2	0.6	648	13.6	669.2	0.9	**672.1**	1.1	672	1.0
	100	1000	531.6	6.3	541	72.5	555	2.2	554.2	2.4	**555.9**	2.5
|Σ| = 4	10	1000	477.3	0.9	534.7	18.1	542	1.3	**543.7**	1.5	543.2	1.4
	100	1000	350.7	9.3	369.3	121.6	**381.2**	2.8	361.7	2.8	**381.2**	3.0
|Σ| = 8	10	1000	420	0.7	462.3	21.2	**462.4**	2.0	461.9	2.2	**462.4**	2.1
	100	1000	241.5	10.6	258.7	154.7	**267.4**	3.6	240.9	3.5	**267.4**	3.8
|Σ| = 24	10	1000	382.6	1.3	385.6	37.4	**385.6**	3.9	**385.6**	4.4	**385.6**	4.4
	100	1000	140.3	13.5	147.7	268.3	**148.6**	5.0	130.5	4.9	**148.6**	5.3
Quality improvement with respect to BS-high-quality							**1.81**		**−1.60**		**1.91**	
Speed up with respect to BS-high-quality							94.62		93.99		94.09	
Quality improvement with respect to BS-low-time							**7.91**		**4.30**		**8.02**	
Speed up with respect to BS-low-time							−26.37		−42.38		−39.03	

Table 2
Comparison the Length of Common Sequence over ES dataset produced by power, probabilistic and hyper heuristics with Blum et al.'s beam search.

	n	m	BS-low-time		BS-high-quality		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
|Σ| = 2	10	1000	579.9	0.7	592.6	14.8	**611.6**	0.8	610.2	0.9	611.2	1.0
	50	1000	516.3	3.7	521.9	43.5	533.4	1.4	**535.0**	1.5	534.9	1.7
	100	1000	502.1	7.4	506	78.6	515.1	2.2	517.3	2.5	517.3	2.6
|Σ| = 10	10	1000	185.5	0.5	192.2	9.4	**200.8**	0.9	199.7	0.9	200.6	1.0
	50	1000	127.9	1.5	129.6	18.8	**134.6**	1.3	**134.6**	1.4	**134.6**	1.5
	100	1000	116.5	2.7	117.9	30.6	121.5	2.1	**122.0**	2.3	121.9	2.5
|Σ| = 25	10	2500	214.3	2.7	224.3	51.5	**233.0**	2.8	231.6	3.1	232.4	3.5
	50	2500	131.3	5.5	133	76.6	**137.4**	4.3	137.2	4.6	137.3	4.9
	100	2500	116.3	9.1	118.1	118.6	120.9	7.0	**121.1**	7.7	121.0	7.9
|Σ| = 100	10	5000	132.5	19.1	139.6	394.6	**143.6**	8.6	142.1	9.0	143.5	8.6
	50	5000	67.9	27.8	69.5	490.2	**70.9**	13.0	70.4	13.6	70.8	13.6
	100	5000	57.6	42.2	59	602	**59.6**	21.2	**59.6**	69.9	**59.6**	22.5
Quality improvement with respect to BS-high-quality							**2.82**		**2.66**		**2.88**	
Speed up with respect to BS-high-quality							94.99		93.90		94.24	
Quality improvement with respect to BS-low-time							**5.22**		**5.04**		**5.27**	
Speed up with respect to BS-low-time							22.83		6.50		11.17	

Table 3
Comparison the Length of Common Sequence over ACO-Random dataset produced by power, probabilistic and hyper heuristics with Blum et al.'s beam search.

	n	m	BS_low-time		BS_high_quality		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
|Σ| = 4	10	600	200	0.3	211	9.8	**221**	0.4	218	0.4	**221**	0.5
	15	600	190	0.5	194	13.2	202	0.4	**203**	0.5	202	0.4
	20	600	178	0.7	184	14.9	**191**	0.5	**191**	0.5	**191**	0.5
	25	600	174	0.9	179	15.8	**186**	0.5	185	0.5	185	0.5
	40	600	162	1.4	167	21	**174**	0.6	172	0.7	172	0.7
	60	600	157	2.1	161	27.6	**165**	0.8	**165**	0.8	**165**	0.9
	80	600	151	2.7	156	33.5	**162**	0.9	161	1.0	161	1.0
	100	600	150	3.5	154	40.3	157	1.0	**158**	1.2	**158**	1.2
	150	600	146	5	148	56.4	**151**	1.5	**151**	1.5	**151**	1.7
	200	600	144	6.9	146	74.3	**150**	1.9	**150**	2.1	**150**	2.0
|Σ| = 20	10	600	58	0.7	61	33.3	**62**	0.5	61	0.5	61	0.5
	15	600	49	0.9	51	37.6	**52**	0.4	51	0.4	**52**	0.5
	20	600	43	1.1	47	39.5	**47**	0.5	**47**	0.5	**47**	0.5
	25	600	41	1.3	43	39.5	**44**	0.5	**44**	0.5	**44**	0.5
	40	600	37	1.7	37	43.2	**38**	0.6	**38**	0.6	**38**	0.6
	60	600	34	2.6	34	46.5	**35**	0.7	**35**	0.8	**35**	0.8
	80	600	32	3.2	32	53.2	**33**	0.9	32	1.0	**33**	0.9
	100	600	30	3.9	31	59.2	**31**	1.0	**31**	1.2	**31**	1.2
	150	600	28	5.7	29	75.6	**29**	1.3	**29**	1.5	**29**	1.4
	200	600	27	7.9	27	98	27	1.7	**28**	1.9	27	1.8
Quality improvement with respect to BS-high-quality							**2.43**		**2.14**		**2.26**	
Speed up with respect to BS-high-quality							97.80		97.60		97.60	
Quality improvement with respect to BS-low-time							**5.42**		**5.12**		**5.24**	
Speed up with respect to BS-low-time							55.51		51.89		51.82	

Table 4
Comparison the Length of Common Sequence over ACO-rat dataset produced by power and hyper and probabilistic heuristics with Blum et al.'s beam search.

	n	*m*	BS-low-time		BS-high-quality		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
$\|\Sigma\|=4$	10	600	189	0.3	191	9.7	**200**	0.3	199	0.4	**200**	0.4
	15	600	163	0.4	173	12.3	**183**	0.4	182	0.4	**183**	0.4
	20	600	160	0.6	163	12.6	**171**	0.4	168	0.4	168	0.4
	25	600	160	0.8	162	15.8	**168**	0.5	166	0.4	**168**	0.5
	40	600	142	1.2	146	9.4	**153**	0.5	146	0.5	**153**	0.5
	60	600	143	1.9	144	26.7	**149**	0.7	147	0.7	**149**	0.7
	80	600	131	2.3	135	31.8	**141**	0.8	**141**	0.9	**141**	0.9
	100	600	129	3	132	38.5	**134**	1.0	132	1.0	**134**	1.0
	150	600	120	4.2	121	51.1	**125**	1.3	124	1.3	124	1.4
	200	600	117	5.6	121	69.1	**122**	1.6	120	1.6	120	1.7
$\|\Sigma\|=20$	10	600	65	0.7	69	27.4	**70**	0.5	**70**	0.5	**70**	0.7
	15	600	57	1.1	60	36.7	**62**	0.5	61	0.5	**62**	0.5
	20	600	50	1.2	51	34.4	**54**	0.4	53	0.5	53	0.5
	25	600	49	1.4	51	39	**51**	0.4	50	0.5	**51**	0.5
	40	600	46	2	49	47	**49**	0.6	**49**	0.6	**49**	0.6
	60	600	44	3.2	46	60.3	**47**	0.8	46	0.8	**47**	0.8
	80	600	42	4	43	64.4	**43**	0.9	**43**	1.0	**43**	1.1
	100	600	37	4.5	38	64.8	**39**	1.0	**39**	1.1	**39**	1.0
	150	600	35	6.7	36	77.8	**37**	1.2	36	1.3	**37**	1.3
	200	600	31	8.3	33	101	**34**	1.6	32	1.7	**34**	1.7
Quality improvement with respect to BS-high-quality							**2.94**		**1.39**		**2.62**	
Speed up with respect to BS-high-quality							97.77		97.69		97.58	
Quality improvement with respect to BS-low-time							**6.35**		**4.74**		**6.03**	
Speed up with respect to BS-low-time							57.49		55.29		52.79	

BS-high-quality and BS-low-time. For example, it can be seen that the average quality improvements obtained by the power, probability and hyper heuristics over BS-high-quality are, respectively, 1.81, −1.60, and 1.91. As can be seen, the hyper heuristic has the best average quality.

As can be observed in Tables 2–5, our algorithms perform even better on the other benchmarks, compared to BS. Table 2 shows that in all the cases, the best results are obtained by HH-LCS. In 8 out of the 12 cases, our algorithms perform in even less time than those of BS-low-time. In all the remaining four cases, they consume less time than those of the BS-high-quality. On average, quality improvements of HH-LCS with power, probabilistic and hyper heuristics over BS-high-quality are 2.82%, 2.66% and 2.88%, respectively. These improvements over BS-low-time are 5.22%, 5.04% and 5.27%. The statistics shows that our algorithms consume less time rather than BS-low-time on average. Again hyper heuristic obtains the best average quality over this dataset.

We compared our algorithms with BS over ACO-random, ACO-rat, and ACO-virus datasets and reported the results in Tables 3–5, respectively. As can be seen in Table 3, all proposed heuristics could

Table 5
Comparison the Length of Common Sequence over ACO-virus dataset produced by power and hyper and probabilistic heuristics with Blum et al.'s beam search.

	n	*m*	BS-low-time		BS-high-quality		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
$\|\Sigma\|=2$	10	600	203	0.4	212	11.6	224	0.4	**225**	0.5	**225**	0.5
	15	600	192	0.5	193	15.4	202	0.4	**203**	0.5	202	0.4
	20	600	179	0.7	181	17.2	**191**	0.5	189	0.5	189	0.5
	25	600	178	0.9	185	17.9	191	0.5	**193**	0.5	**193**	0.6
	40	600	158	1.3	162	21.9	165	0.6	**168**	0.6	165	0.6
	60	600	153	2	158	29.1	164	0.8	**165**	0.8	164	0.8
	80	600	148	2.6	153	36	156	0.8	**158**	0.9	**158**	1.0
	100	600	149	3.4	150	43.9	152	1.1	**158**	1.2	**158**	1.2
	150	600	143	5	148	64.5	150	1.5	**156**	1.7	**156**	1.7
	200	600	143	6.8	145	84.5	144	1.8	**154**	2.1	**154**	2.1
$\|\Sigma\|=20$	10	600	67	0.7	75	27.2	**75**	0.5	**75**	0.5	**75**	0.6
	15	600	58	1	63	38.6	**63**	0.5	**63**	0.5	**63**	0.5
	20	600	55	1.2	57	40.3	**60**	0.5	**60**	0.5	**60**	0.5
	25	600	50	1.4	53	38.9	**55**	0.6	54	0.5	**55**	0.5
	40	600	47	2.1	49	48.4	**49**	0.7	49	0.7	**49**	0.7
	60	600	44	3.1	45	56.1	**47**	0.8	47	1.0	**47**	1.0
	80	600	43	4	44	67.4	**46**	1.0	45	1.1	**46**	1.2
	100	600	41	5	43	74.2	**44**	1.1	**44**	1.5	**44**	1.3
	150	600	43	7.8	44	108	**45**	1.8	**45**	1.8	**45**	1.9
	200	600	43	11	43	140	43	2.0	**44**	2.2	43	2.1
Quality improvement(relative to Blum-h-q)							2.57		3.46		3.40	
Speed up(relative to Blum-h-q)							97.94		97.76		97.75	
Quality improvement(relative to Blum-l-t)							6.19		7.10		7.05	
Speed up(relative to Blum-l-t)							56.67		53.44		52.95	

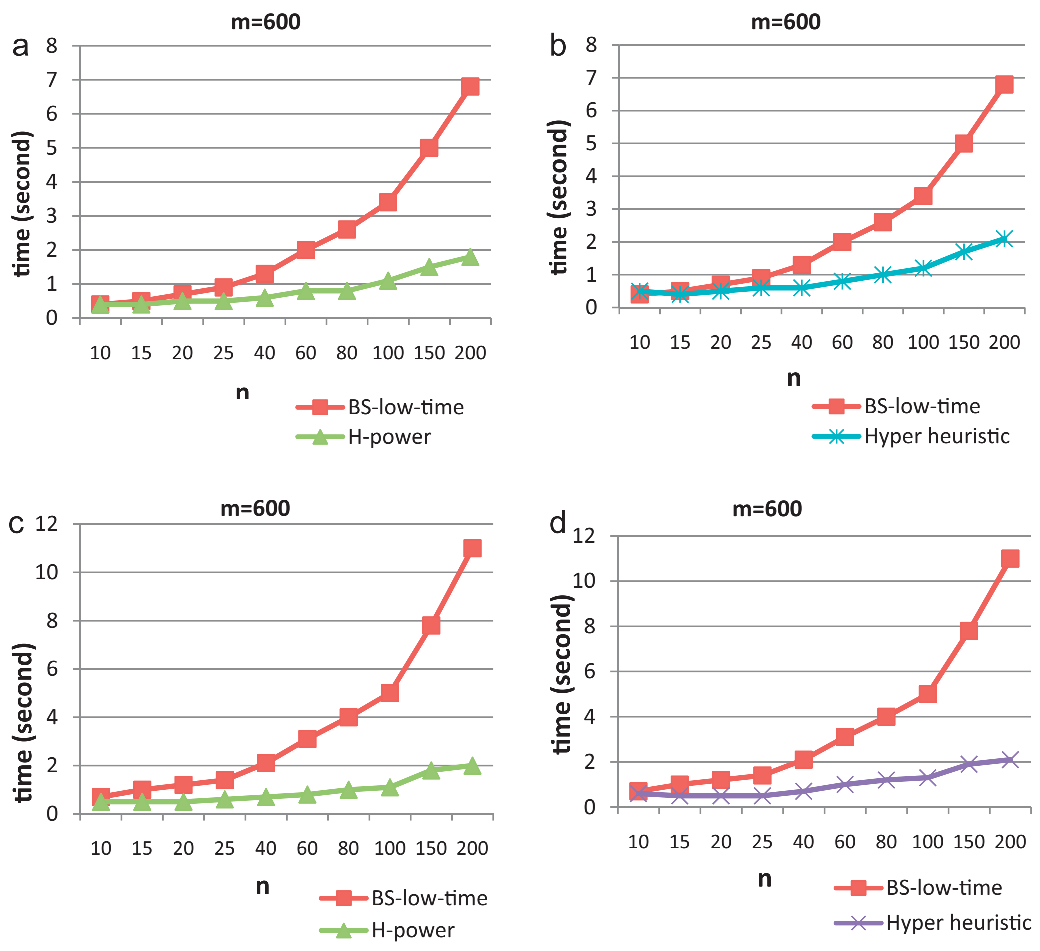


Fig. 5. (a) Comparison of power heuristic (H-power) to BS-low-time with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 4$. (b) Comparison of hyper heuristic (HH-LCS) to BS-low-time with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 4$. (c) Comparison of power heuristic and BS-low-time with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 20$. (d) Comparison of (HH-LCS) and BS-low-time with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 20$.

obtain higher quality solutions than BS-low-time, whereas they consume less time in all cases except in one (the first row). The results of our algorithms in none of 20 cases are inferior to those of BS-high-quality in term of solution quality. In 16 out of 20(80% of) cases, power heuristic provides higher quality solutions than BS-high-quality, while probabilistic and hyper heuristics outperform BS-high-quality in 70% and 75% of cases, respectively. In summary, quality improvements for HH-LCS with power, probabilistic and hyper heuristics rather than BS-high-quality are 2.43%, 2.14% and 2.26%, respectively. The quality improvements with respect to BS-low-time for our algorithms are 5.42%, 5.12% and 5.24%. On average, power heuristic obtains the highest quality improvement on ACO-random dataset.

Table 4 compares the results of HH-LCS with BS algorithm on ACO-rat benchmark. As reported in Table 4, our algorithms provide better or as good performance as BS-high-quality (in term of quality) in all cases, while they are also much faster. They are even faster than BS-low-time, except for the first row. On average, the speed up of HH-LCS with power, probabilistic and hyper heuristics with respect to BS-low-time are 57.49%, 55.29% and 52.79% and the average improvements are 6.35%, 4.74% and 6.03%, respectively. In summary, power heuristic has the best performance on this dataset again.

As can be seen in Table 5, HH-LCS with power heuristic can produce higher or the same solution quality compared to BS-high-quality except in the tenth row (95% of the cases) for ACO-virus benchmark, whereas it performs faster than BS-low-time (average speed up with respect to BS-low-time is 56.67%). In contrast to other datasets, the probabilistic heuristic shows better results than power heuristic on this dataset; therefore, hyper heuristic could improve the performance of power heuristic. Quality improvements of power, probabilistic and hyper heuristics with respect to BS-low-time over ACO-virus benchmark are 6.19%, 7.10% and 7.05%, respectively. Average improvements of our algorithms over BS-high-quality are 2.57%, 3.46% and 3.40%.

With respect to the running-time, it is evident that HH-LCS is more similar to BS-low-time than BS-high-quality. The running-time for HH-LCS has even slower growth than that of BS-low-time. Fig. 5 compares the growth of running-time of HH-LCS with BS-low-time, by increasing the number of strings. As can be seen, the running-time of BS-low-time grows rapidly, compared to our algorithms. The difference between the running time of our algorithms and BS-high-quality is even more significant, which is depicted in Fig. 6. These figures show that our algorithms are much faster than Blum algorithms, especially for large $|\Sigma|$ and n.

Table 6 summarizes the results of comparing HH-LCS with BS-low-time and BS-high-quality over all the five benchmarks. On average, the proposed hyper heuristic algorithm has the best performance. With respect to solution quality, it achieves the improvement ratio of 2.69% and 6.17% over BS-high-quality and BS-low-time, respectively. Table 6 shows that power heuristic obtains better solutions than probabilistic heuristic on average. Quality improvements of power heuristic with respect to BS-high-quality and BS-low-time are 2.59% and 6.06%, respectively.

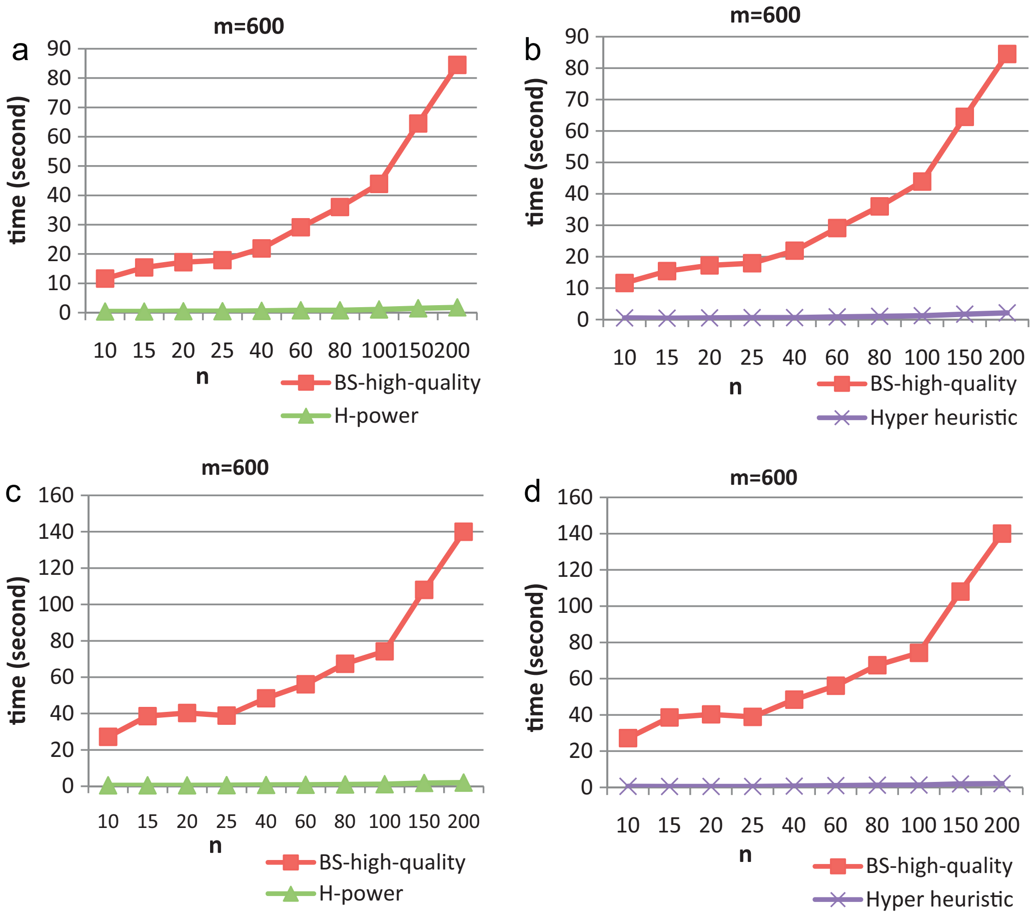


Fig. 6. (a) Comparison of power heuristic (H-power) to BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 4$. (b) Comparison of hyper heuristic (HH-LCS) to BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 4$. (c) Comparison of power heuristic and BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 20$. (d) Comparison of (HH-LCS) and BS-high-quality with respect to run-time growth, for the ACO-virus benchmark with $|\Sigma| = 20$.

These values are, respectively, 1.99% and 5.43% for the probabilistic heuristic.

In summary, HH-LCS provides superior solution quality than BS-high-quality while using less time than BS-low-time, in almost all the experimental cases. Although, in most of the cases the heuristics H-power and H-prob result in good solutions, their performance is relatively poor in a few cases. However, the hyper heuristic overcomes their shortcoming in such cases and, on average, yields superior results. It illustrates the effectiveness of hybridizing heuristics, especially when they are complement.

5.2. Comparison with MLCS-APP

In order to compare HH-LCS with MLCS-APP algorithm, we used all four benchmarks provided in Wang et al. (2010b). The first two datasets used in Wang et al. (2010b) are Random DNA sequences generated independently from the alphabet $\sum = \{A,G,T,C\}$, which we call *DNA-Random1* and *DNA-Random2*, respectively. Table 7 shows the results of running IBS-LCS on the *DNA-Random1* benchmark. The first three columns in this table show the size of the alphabet, the number of input strings and the length of the strings, respectively. The next two columns report the results for MLCS-A* algorithm which is an exact algorithm proposed in Wang et al. (2010b) and is only applicable for small datasets (datasets with small number of strings and/or short strings). The sixth and seventh columns show the results of the MLCS-APP algorithm. The next six columns show the respective results for our algorithms with the power heuristic (H-power), the probabilistic heuristic (H-prob), and the hyper heuristic (HH-LCS). We used the beam size $\beta = 300$ and $\beta_h = 10$. As can be seen in Table 7, HH-LCS with power and hyper heuristics provides optimum solution in all cases whereas MLCS-APP algorithm cannot achieve the optimum solution in the

Table 6
Average improvement in solution quality and time over BS for each benchmark.

Benchmark	Quality improvement vs. BS-low-time			Quality improvement vs. BS-high-quality			Time improvement vs. BS-low-time			Time improvement vs. BS-high-quality		
	H-power	H-prob	HH-LCS	H-power	H-prob	HH-LCS	H-power	H-prob	HH-LCS	H-power	H-prob	HH-LCS
ES	5.22	5.04	5.27	2.82	2.66	2.88	22.83	6.5	11.17	94.99	93.9	94.24
BB	7.91	4.3	8.02	1.81	−1.6	1.91	−26.37	−42.38	−39.03	94.62	93.99	94.09
ACO-random	5.42	5.12	5.24	2.43	2.14	2.26	55.51	51.89	51.82	97.8	97.6	97.6
ACO-rat	6.35	4.74	6.03	2.94	1.39	2.62	57.49	55.29	52.79	97.77	97.69	97.58
ACO-virus	6.19	7.1	7.05	2.57	3.46	3.4	56.67	53.44	52.95	97.94	97.76	97.75
Total	**6.06**	**5.43**	**6.17**	**2.59**	**1.99**	**2.69**	**43.21**	**36.89**	**37.16**	**97.09**	**96.75**	**96.78**

Table 7
Comparison the Length of Common Sequence and computation time for power, probabilistic and hyper heuristics with MLCS-APP and MLCS-A* over DNA-Random1 dataset.

	n	m	MLCS-A*		MLCS-APP		H-power		H-Prob		HH-LCS	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
$\lvert\Sigma\rvert = 4$	4	100	46	0.05	46	0.31	46	0.17	46	0.17	46	0.08
	5	100	43	0.53	43	0.28	43	0.13	43	0.14	43	0.08
	6	100	40	3.30	40	0.27	40	0.08	40	0.08	40	0.08
	7	100	37	25.39	37	0.23	37	0.08	37	0.08	37	0.08
	8	100	36	93.42	36	0.23	36	0.08	36	0.08	36	0.08
	9	100	35	195.1	34	0.23	**35**	0.08	34	0.08	**35**	0.08

Table 8
Comparison the Length of Common Sequence and computation time for power, probabilistic and hyper heuristics with MLCS-APP and MLCS-A* over DNA-Random2 dataset.

	n	m	MLCS-A*		MLCS-APP		H_power		H_Prob		HH-LCS	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
$\lvert\Sigma\rvert = 4$	5	100	43	0.53	43	0.28	**43**	0.08	**43**	0.09	**43**	0.08
	5	120	51	1.03	51	0.36	**51**	0.11	**51**	0.11	**51**	0.11
	5	140	60	5.73	59	0.42	**60**	0.14	**60**	0.14	**60**	0.14
	5	160	70	8.80	69	0.50	**70**	0.16	**70**	0.17	**70**	0.17
	5	180	77	25.47	76	0.56	76	0.20	76	0.19	76	0.20
	5	200	84	70.47	83	0.63	**85**	0.22	**84**	0.22	**84**	0.22

Table 9
Comparison the Length of Common Sequence over ACO-rat dataset produced by power and hyper and probabilistic heuristics with MLCS-APP.

	n	m	BS-high-quality		MLCS-APP		H-power		H-Prob		Hyper heuristic (HH-LCS)	
			LCS	Time	LCS	Time	LCS	Time	LCS	Time	LCS	Time
$\lvert\Sigma\rvert = 4$	10	600	191	9.7	194	2.0	**203**	1.9	199	1.5	**203**	1.4
	15	600	173	12.3	180	1.9	182	1.8	**184**	1.4	182	1.4
	20	600	163	12.6	165	1.7	**171**	1.2	168	1.0	168	1.2
	25	600	162	15.8	164	2.0	**169**	1.2	166	1.4	**169**	1.5
	40	600	146	9.4	151	2.0	**149**	1.5	145	1.4	**149**	1.2
	60	600	144	26.7	147	2.8	**150**	1.9	149	1.8	**150**	1.9
	80	600	135	31.8	137	3.1	**142**	2.1	**142**	2.1	**142**	1.8
	100	600	132	38.5	134	3.7	**134**	2.2	**134**	2.2	**134**	2.2
	150	600	121	51.1	125	4.9	**128**	2.7	125	2.7	125	2.6
	200	600	121	69.1	122	6.6	**122**	2.8	120	3.2	120	3.0
$\lvert\Sigma\rvert = 20$	10	600	69	27.4	70	3.3	**71**	1.8	70	1.6	**71**	1.8
	15	600	60	36.7	61	3.0	**62**	2.8	**62**	1.7	**62**	1.5
	20	600	51	34.4	53	2.4	**54**	1.3	**54**	1.5	**54**	1.4
	25	600	51	39	51	2.7	**51**	1.4	**51**	1.3	**51**	1.3
	40	600	49	47	48	2.9	**49**	1.5	**49**	1.6	**49**	1.6
	60	600	46	60.3	46	3.7	**47**	1.9	46	2.2	**47**	2.0
	80	600	43	64.4	43	3.9	43	3.5	**44**	2.3	43	2.0
	100	600	38	64.8	38	4.3	**39**	2.4	**39**	2.3	**39**	2.1
	150	600	36	77.8	35	5.2	**37**	2.3	36	2.5	**37**	2.4
	200	600	33	101	33	7.0	**34**	2.8	33	3.2	**34**	3.0
Quality improvement with respect to MLCS_APP							**1.99**		**1.03**		**1.70**	
Speed up with respect to MLCS_APP							36.05		41.09		42.89	

last row. Also, HH-LCS consumes less than half the time consumed by MLCS-APP algorithm.

In Table 8, we compare HH-LCS with MLCS-A* and MLCS-APP over DNA-Random2 benchmark. We set $\beta = 300$ and $\beta_h = 10$. It can be seen that our algorithms can provide optimum solution for all heuristics in all cases except for the fifth row. In addition, HH-LCS with H-power heuristic provides higher quality solutions than optimal solution produced by MLCS-A* in the sixth row. (We carefully verified the obtained solution,[5] which indicates an issue in Wang et al.'s report or in their implementation of MLCS-A*.) Table 8 indicates the superiority of HH-LCS compared to MLCS-APP with respect to both the solution quality and the running-time.

Wang et al. (2010b) compared their algorithm with BS only using ACO-rat dataset previously introduced in Section 5.1. They compared MLCS-APP with BS-high-quality and provided almost better solutions in terms of both quality and time. Table 9 compares the result of our algorithms with MLCS-APP on ACO-rat. We set $\beta = 450$ and $\beta_h = 10$. As can be seen in this table, HH-LCS with the power heuristic provides better or as good performance as MLCS-APP (in terms of quality) in all but one case. Another observation is that HH-LCS with the power heuristic achieves, on average, better solutions than the probabilistic and hyper heuristics. On average, compared to the MLCS-APP, HH-LCS with the power, probabilistic and hyper heuristics provides 1.99%, 1.03% and 1.70% improvements in solution quality, respectively, while performing faster than MLCS-APP (note that we have used a machine even slower than the machine

[5] The common sequence produced by power heuristic over the last instance of DNA-random2 benchmark is = "TGAAAAAAGGCTTCGGGTCGGATACCGAAGCAGCC-AGGGGCGTCGCCCAGTGTGGTGGTTCCAAATGGGGATATAAGAGTTACTG" which its length is 85.

Table 10
Comparison the result of power, probabilistic and hyper heuristics with MLCS-APP over protein sequences selected from Pfam dataset, $n=8$, $\Sigma=20$, average strings' length almost 200.

Family ID (accession number)	Optimal common sequence	MLCS-APP	H-power		H-Prob		Hyper heuristic (HH-LCS)	
	Length	LCS	LCS	Time	LCS	Time	LCS	Time
AP_endonuc_2 (PF01261)	30	29	**30**	0.25	**30**	0.25	**30**	0.27
DUF2077 (PF09850)	35	35	**35**	0.33	**35**	0.33	**35**	0.31
NikM (PF10670)	40	40	**40**	0.30	**40**	0.30	**40**	0.38
Nop25 (PF09805)	67	67	**67**	0.52	**67**	0.52	**67**	0.52
Exon_PolB (PF10108)	76	76	**76**	0.45	**76**	0.48	**76**	0.42
Frag1 (PF10277)	93	93	**93**	0.63	**93**	0.55	**93**	0.56
G6PD_bact (PF10786)	105	105	**105**	0.52	**105**	0.58	**105**	0.55
Adeno_hexon_C (PF03678)	136	136	**136**	0.28	**136**	0.30	**136**	0.33

used in Wang et al. (2010b)). The last row (Speedup) of Table 9 reports the time improvement of our algorithms in comparison with MLCS-APP.

The fourth benchmark used by Wang et al. consists of eight protein domain families selected from Pfam database (Finn et al., 2008). Eight sequences of approximately the same length (around 200 amino acids) were chosen from each family. The first column includes the names of the protein families. The second column is the length of optimal common sequence and the third column is the result of applying MLCS-APP algorithm on this dataset. Table 10 shows that HH-LCS with all the heuristics can achieve optimum solution in all cases whereas MLCS-APP provides 7 optimal solutions in 8 cases. We set $\beta=200$ and $\beta_h=10$. Wang et al. did not report run time for this benchmark.

Tables 7–10 demonstrate that HH-LCS outperforms MLCS-APP in terms of both the quality and the run time.

5.3. Comparison with Deposition&Extention algorithm (DEA)

The DEA algorithm was not compared BS (which was the state-of-the-art); it was only compared with three primitive algorithms, namely LR, Expansion, and THSB. However, we compare the result of our algorithms with DEA over two real and simulated dataset used in Ning (2010) and show that our algorithms significantly outperforms DEA.

The real dataset are DNA and protein sequences randomly selected from NCBI (NCBI-viruses) and SwissProt (Swiss-Prot) websites, respectively. We obtained this dataset from Ning's website. The first three columns of Table 11 show the size of the alphabet, the number of input strings and the length of the strings, respectively. For each setting (Σ, n, and m), there are 10 sets of sequences randomly selected from corresponding databases and the results are the averages of the common sequences' length over all 10 instances. DEA used two different heuristics called MF (Most Front) and MC (Min Change). The next two columns report the result of DEA with MF and MC heuristics, respectively. The next six columns show the respective values for our algorithms with the power (H-power), the probabilistic (H-prob) and the hyper heuristics (HH-LCS). As shown in Table 11, on average, our algorithm with the power heuristic, probabilistic heuristic and hyper heuristic provides 23%, 24.83% and 25.01% improvements in the solution quality, respectively, compared to DEA with MC heuristic. Quality improvements for HH-LCS compared to DEA with MF heuristic are 19.61%, 21.38%, and 21.56%, respectively. Note that for each row of the Table 11, 10 sets of sequences are averaged to get the result. Therefore, as can be observed in the second and the third rows of this table, the hyper heuristic can provide higher quality solutions than the individual power and probabilistic heuristics. Also, on average, HH-LCS with hyper heuristics outperforms both power and probabilistic heuristics over this dataset.

We have also compared our algorithms with DEA over the simulated dataset used in (Ning, 2010). Ning generated 4-character sequences from $\Sigma=\{A,T,G,C\}$. Each character is distributed among sequences with specified alphabet content (γ). It means that γ% of each sequence consists of two characters G and C. Table 12 shows that on average our algorithm with the power, probabilistic and hyper heuristics provide 3.47%, 4.32% and 4.28% quality improvements, respectively, compared to DEA with MC heuristic. Also, quality improvements of HH-LCS with these heuristics compared to DEA using MF heuristic are 4.15%, 5.01%, and 4.97%, respectively.

Ning tested DEA algorithm on another simulated benchmark but did not express its alphabet content for it explicitly. Therefore, we could not determine which file he used for testing his algorithm. (We asked for the benchmarks and codes but did not receive any response.)

Table 11
Comparison the Length of Common Sequence for power and hyper and probabilistic heuristics with DEA algorithm (MF and MC heuristics) over real dataset (Ning, 2010).

	n	m	DEA		H_power		H_Prob		HH-LCS	
			(MF)	(MC)	LCS	Time	LCS	Time	LCS	Time
$\|\Sigma\|=4$	100	100	14.9	15.8	**18.5**	0.27	18.3	0.33	18.4	0.27
	100	500	98.6	94.3	118.4	2.10	117.8	2.14	**118.7**	2.31
	100	1000	198.2	191.4	240.9	5.44	**243.5**	5.47	243.6	5.67
	500	100	9.8	10.7	**12.3**	0.88	**12.3**	1.09	**12.3**	0.92
	500	500	65.8	71.0	89.2	8.37	**92.6**	8.87	**92.6**	9.26
	500	1000	142.9	152.1	185.0	21.30	200.6	22.24	**200.8**	23.11
$\|\Sigma\|=20$	100	100	2.5	2.8	**3.1**	0.15	**3.1**	0.18	**3.1**	0.17
	100	500	24.8	22.0	**30.6**	1.81	30.5	1.87	**30.6**	2.01
	500	100	0.9	0.9	**0.9**	0.83	0.9	0.77	**0.9**	0.81
	500	500	18.5	13.7	20.8	7.81	**21.3**	7.54	**21.3**	8.04
	1000	100	0.1	0.1	**0.1**	1.44	**0.1**	1.56	**0.1**	1.56
	1000	500	14.1	11.7	16.8	13.21	**17.3**	13.00	**17.3**	13.45
Quality improvement with respect to DEA(MC)					23.00		24.83		25.01	
Quality improvement with respect to DEA(MF)					19.61		21.38		21.56	

Table 12
Comparison the Length of Common Sequence for power and hyper and probabilistic heuristics with DEA algorithm (MF and MC heuristics) over simulated dataset (Ning, 2010).

	n	m	$\beta1$ (%)	DEA		H_power		H_Prob		HH-LCS	
				(MF)	(MC)	LCS	Time	LCS	Time	LCS	Time
$\|\Sigma\|=4$	100	1000	10	443.2	446.0	461.2	8.76	462.7	9.36	462.5	9.17
	100	1000	20	389.8	391.2	407.7	7.86	410.4	8.45	410.4	8.63
	100	1000	30	337.5	338.7	355.4	7.16	357.9	7.59	357.9	7.69
	100	1000	40	285.1	290.0	302.6	6.33	304.2	6.76	304.2	6.81
	100	1000	50	243.6	250.0	268.4	5.81	269.5	6.29	269.5	6.20
$\|\Sigma\|=4$	1000	1000	10	419.8	418.8	425.4	67.07	428.9	67.01	428.7	69.19
	1000	1000	20	369.0	369.7	375.2	63.18	379.7	65.72	379.1	67.57
	1000	1000	30	320.7	320.3	325.8	60.23	330.3	60.36	330.3	62.57
	1000	1000	40	272.1	272.5	277.5	55.97	280.8	54.29	280.5	57.63
	1000	1000	50	224.8	226.7	235.3	51.61	237.8	48.53	237.8	52.52
Quality improvement with respect to DEA(MC)						3.47		4.32		4.28	
Quality improvement with respect to DEA(MF)						4.15		5.01		4.97	

As mentioned before, no precise run-time of DEA was reported in Ning (2010). Ning reported that for the datasets with $n \geq 1000$ and $m \geq 1000$, DEA took less than 10 min whereas the maximum run time for HH-LCS (for $n = 5000$ and $m = 1000$) is about 5 min.

In summary, Tables 1–12 indicate that HH-LCS provides higher quality solutions than BS, MLCS-APP and DEA in most of the experiments while it even takes less time. This suggests the HH-LCS is the new state-of-the-art heuristic algorithm for the LCS.

6. Conclusion

Various optimal and non-optimal algorithms have already been proposed for the LCS problem. The optimal algorithms are of exponential complexity due to the NP-hardness of the problem, hence not affordable in practice for large or even moderate size problem instances. The non-optimal algorithms proposed so-far include approximation, simple heuristic and metaheuristic algorithms. Our proposed hyper-heuristic algorithm in this paper is deterministic and based on beam search, initially proposed by Blum et al. for the LCS problem, although it has its own distinct features as specified in this paper. The hyper heuristic mechanism is responsible for dynamically deciding which basic heuristic function to use in the main beam search algorithm. There are two candidate heuristic functions for this purpose, one *h-power*(.) proposed for the first time in this paper for the LCS problem and the other *h-prob*(.), recently proposed in Mousavi and Tabataba (2012). The motivation behind incorporating them within a hyper heuristic mechanism was that neither of them was observed to be able to dominate the other in the experimental cases (while each providing superior solutions than those of existing algorithms in the literature). Taking the advantage of beam search, it is dynamically decided which heuristic function better suits a given problem instance. To that end, a low-cost beam search, i.e. one with a small beam size, is run twice, once per each of the candidate heuristic functions, and based on the results, the best heuristic is selected for the final run of the algorithm. Although this mechanism does not guarantee that the better choice (with respect to a given problem instance) is always made, it results in higher (average) solution quality compared to the case an individual heuristic function is always used. This improvement costs the extra light runs of beam search, i.e., with the small beam size, which was less than 5% in our experimental cases. The proposed algorithm was extensively compared with three most recent published algorithms of the non-optimal family over several real biological benchmarks with positive results.

Possible avenues for future work include the incorporation of further heuristic functions within the hyper-heuristic framework and to increase the precision of the hyper-heuristic mechanism in choosing the right heuristic function for a given problem instance.

The proposed algorithm was extensively compared with BS, MLCS-APP, and DEA algorithms, with evident positive results. This suggests that the proposed hyper-heuristic algorithm, HH-LCS is the current state-of-the-art heuristic algorithm for LCS.

Acknowledgement

The authors would like to thank Dr. C. Blum, Dr. T. Easton, and Mr. Q. Wang for making available their datasets.

References

Aho, A.V., Ullman, J.D., Hopcroft, J.E., 1983. Data Structures and Algorithms. Addison-Wesley, Reading, MA.
Bafna, V., Muthukrishnan, S., Ravi, R., 1995. Computing similarity between RNA strings. In: 6th Annual Symposium on Combinatorial Pattern Matching, Espoo, Finland, pp. 1–16.
Banerjee, A., Ghosh, J., 2001. Clickstream clustering using weighted longest common subsequences. In: Proceedings of the Web Mining Workshop at the 1st SIAM Conference on Data Mining, pp. 33–40.
Blum, C., Blesa, M.J.,2007. Probabilistic beam search for the longest common subsequence problem. In: Proceedings of the 2007 International Conference on Engineering Stochastic Local Search Algorithms: Designing, Implementing and Analyzing Effective Heuristics. Springer-Verlag, Brussels, Belgium.
Blum, C., Blesa, M.J., López-Ibáñez, M., 2009. Beam search for the longest common subsequence problem. Computers and Operations Research 36, 3178–3186.
Blum, C., Roli, A., 2003. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, vol. 35. ACM, pp. 268–308.
Bonizzoni, P., Vedova, G.D., Mauri, G., 2001. Experimenting an approximation algorithm for the LCS. Discrete Applied Mathematics 110, 13–24.
Brisk, P., Kaplan, A., Sarrafzadeh, M.,2004. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In: Proceedings of the 41st Annual Design Automation Conference. ACM, San Diego, CA, USA, pp. 395–400.
Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., Schulenburg, S., 2003. Hyper-heuristics: an emerging direction in modern search technology. In: Hillier, F.S. (Ed.), Handbook of Metaheuristics, vol. 57. Springer, New York, pp. 457–474.
Chen, Y., Wan, A., Liu, W., 2006. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. BMC Bioinformatics 7, S4.
Chin, F., Poon, C., 1994. Performance analysis of some simple heuristics for computing longest common subsequences. Algorithmica 12, 293–311.
cpu-benchmark. http://cpubenchmark.net/.
Easton, T., Singireddy, A., 2007. A specialized branching and fathoming technique for the longest common subsequence problem. International Journal of Operations Research 4, 98–104.
Easton, T., Singireddy, A., 2008. A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics 14, 271–283.
Eppstein, D., Galil, Z., Giancarlo, R., Italiano, F.G., 1992. Sparse dynamic programming II: convex and concave cost functions. Journal of the Association for Computing Machinery 39, 546–567.
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., Bateman, A., 2008. The pfam protein families database. Nucleic Acids Research 36, D281–D288 (Database issue).
Fraser, C.B., 1995. Subsequences and Supersequences of Strings. University of Glasgow.
Garey, M., Johnson, D., 1990. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman & Co. Ltd., New York, NY.
Guenoche, A., 2004. Supersequence of masks for oligo-chips. Journal of Bioinformatics and Computational Biology 2 (3), 459–469.

Guenoche, A., Vitte, P., 1995. Longest common subsequence with many strings: exact and approximate methods. Technique et Science Informatiques 14 (7), 897-915.

Hakata, K., Imai, H.,1992. The longest common subsequence problem for small alphabet size between many strings. In: Proceedings of the Third International Symposium on Algorithms and Computation, vol. 650. Springer-Verlag, pp. 469–478.

Hirschberg, D.S., 1975. A linear space algorithm for computing maximal common subsequences. Communication of the Association for Computing Machinery 18, 341–343.

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

Huang, K.S., Yang, C.B., Tseng, K.T., 2004. Fast algorithms for finding the common subsequence of multiple sequences. In: Proceedings of International Computer Symposium, pp. 90–95.

Irving, R.W., Fraser, C.,1992. Two algorithms for the longest common subsequence of three (or more) strings. In: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, vol. 644. Springer-Verlag, Berlin, pp. 214–229.

Jiang, T., Li, M., 1995. On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 24, 1122–1139.

Jiang, T., Lin, G., Ma, B., Zhang, K., 2002. A general edit distance between RNA structures. Journal of Computational Biology 9, 371–388.

Johetla, T., Smed, J., Hakonen, H., Raita, T., 1996. An efficient heuristic for the LCS problem. In: Third South American Workshop on String Processing, WSP'96, pp. 126–140.

Korkin, D., Wang, Q., Shang, Y., 2008. An efficient parallel algorithm for the multiple longest common subsequence (MLCS) problem. In: International Conference on Parallel Processing (ICPP), pp. 354–363.

Maier, D., 1978. The Complexity of Some Problems on Subsequences and Supersequences, vol. 25. ACM, pp. 322–336.

Mousavi, S.R., Tabataba, F., 2012. An improved algorithm for the longest common subsequence problem. Computers and Operations Research 39, 512–520.

NCBI-viruses. http://www.ncbi.nlm.nih.gov/genomes/VIRUSES/viruses.html.

NCBI National Center for Biotechnology Information (NCBI). http://www.ncbi.nlm.nih.gov/.

Ning, K., 2010. Deposition and extension approach to find longest common subsequence for thousands of long sequences. Computational Biology and Chemistry 34, 149–157.

Sankoff, D., Kruskal, J., 1983. Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparisons. Addison-Wesley.

Sellis, T.K., 1988. Multiple-query optimization. ACM Transactions on Database Systems (TODS) 13, 23–52.

Shyu, S.J., Tsai, C.-Y., 2009. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers and Operations Research 36, 73–91.

Singireddy, A., 2003. Solving the longest common subsequence problem in bioinformatics. Master, Kansas State University, KS, Manhattan.

Smith, T.F., Waterman, M.S., 1981. Identification of common molecular subsequences. Journal of Molecular Biology 147, 195–197.

Storer, J.A., 1988. Data Compression: Methods and Theory. Computer Science Press Inc.

Swiss-Prot Release, 45.5, 2005. http://us.expasy.org/sprot/.

Tsai, Y.T., Hsu, J.T., 2002. An approximation algorithm for multiple longest common subsequence problems. In: Proceeding of the 6th World Multiconference on Systemics, Cybernetics and Informatics, SCI2002, pp. 456–460.

Wang, Q., Korikin, D., Shang, Y., 2010a. A fast multiple longest common subsequence (MLCS) algorithm. IEEE Transactions on Knowledge and Data Engineering.

Wang, Q., Korkin, D., Shang, Y., 2009. Efficient dominant point algorithms for the multiple longest common subsequence (MLCS) problem. In: International Joint Conference on Artificial Intelligence (IJCAI), Pasadena, CA, USA, pp. 1494–1500.

Wang, Q., Pan, M., Shang, Y., Korkin, D., 2010b. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In: Conference on Artificial Intelligence (AAAI), Atlanta, GA, USA, pp. 1287–1292.