Since January 2020 Elsevier has created a COVID-19 resource centre with free information in English and Mandarin on the novel coronavirus COVID-19. The COVID-19 resource centre is hosted on Elsevier Connect, the company's public news and information website.

Engineering Applications of Artificial Intelligence 25 (2012) 457–467

Contents lists available at SciVerse ScienceDirect

Engineering Applications of Artificial Intelligence

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

An enhanced beam search algorithm for the Shortest Common Supersequence Problem

Sayyed Rasoul Mousavi *, Fateme Bahri, Farzaneh Sadat Tabataba

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

ARTICLE INFO

Article history:
Received 22 February 2011
Received in revised form
19 August 2011
Accepted 22 August 2011
Available online 21 September 2011

Keywords:
Heuristic
Optimization
Shortest common supersequence
Dynamic programming
Sequence analysis
Microarray production

ABSTRACT

The Shortest Common Supersequence Problem asks to obtain a shortest string that is a supersequence of every member of a given set of strings. It has applications, among others, in data compression and oligonucleotide microarray production. The problem is NP-hard, and the existing exact solutions are impractical for large instances. In this paper, a new beam search algorithm is proposed for the problem, which employs a probabilistic heuristic and uses the dominance property to further prune the search space. The proposed algorithm is compared with three recent algorithms proposed for the problem on both random and biological sequences, outperforming them all by quickly providing solutions of higher average quality in all the experimental cases. The Java source and binary files of the proposed IBS_SCS algorithm and our implementation of the DR algorithm and all the random and real datasets used in this paper are freely available upon request.

1. Introduction

The Shortest Common Supersequence (SCS) problem asks to obtain a shortest string that is a supersequence of every member of a given set of strings. A supersequence of a given string is a string that can be obtained by inserting zero or more characters anywhere in the given string. Among various applications of this problem are data compression (Storer, 1988; Timkovskii, 1989), AI planning (Foulser et al., 1992), query optimization in databases (Chaudhuri and Bruno, 2008; Sellis, 1988), and bioinformatics, particularly DNA oligonucleotide microarray production (Hubbell et al., 1996; Kasif et al., 2002; Ning et al., 2005; Rahmann, 2003; Sankoff and Kruskal, 1983). Microarrays are precious tools successfully used, among others, in gene clustering and identification, SNP detection, and fusion transcript detection(Ning et al., 2005; Rahmann, 2003; Skotheim et al., 2009). Two well-known types of microarrays are cDNA and oligonucleotide microarrays (Kasif et al., 2002; Ning et al., 2005), the latter known to be of higher sensitivity due to its lower cross-hybridization possibility (Kasif et al., 2002; Ning et al., 2005). Oligonucleotide microarrays are usually manufactured by the photolithographic method. This method involves several synthesis steps, each to append a same nucleotide, which corresponds to a letter in {A,T,C,G}, to several designated probes. Since the process is accomplished by means of light exposure, the other probes, which are not to receive the nucleotide, are protected by a mask. The sequence of the nucleotides used in the synthesis steps is called the *deposition string*, whose length determines the number of the synthesis steps. For several reasons, it is desirable to keep the deposition string as short as possible (Kasif et al., 2002; Ning et al., 2005; Rahmann, 2003). First, the masks and the synthesis steps are expensive. Even a small reduction in the length of the deposition string could lead to a significant reduction in the production cost (Rahmann, 2003). Second, the total manufacturing time is increased as the number of synthesis steps is raised. Third, there exist possibilities for errors in microarray fabrication, because the masking task is not perfect; the probability for a masked probe to be exposed to the light is nonzero. Consequently, the probability for fabrication errors is usually increased as the number of the synthesis steps is raised. Therefore, a shorter deposition sequence is desirable to reduce the manufacturing cost, time, and error. On the other hand, the deposition sequence is a common supersequence of the underlying probes. This motivates the design of high quality algorithms for the SCS problem. Fig. 1 illustrates how the use of a shorter deposition sequence can lead to fewer synthesis steps, hence reducing the production cost, time, and error.

The SCS problem can be optimally solved in polynomial time for a fixed number of input strings, but it is NP-hard in general (Maier, 1978). Consequently, it is highly unlikely to obtain a polynomial-time exact algorithm for the problem, unless $P=NP$ (Garey and Johnson, 1979). Exact algorithms proposed for the

* Corresponding author. Tel.: +98 3113915383; fax: +98 3113912451.
E-mail addresses: srm@cc.iut.ac.ir (S.R. Mousavi), f.bahri@ec.iut.ac.ir (F. Bahri), f.tabataba@ec.iut.ac.ir (F.S. Tabataba).

0952-1976/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.engappai.2011.08.006

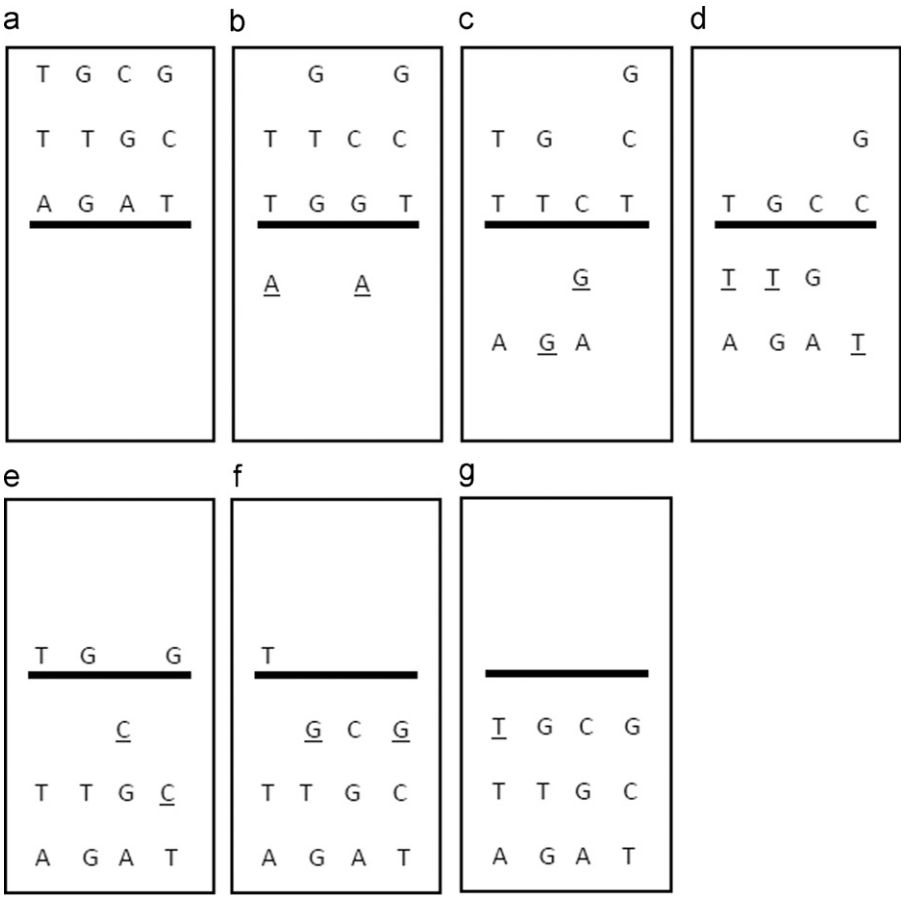

Fig.1. (a) a given set of 3mer oligonucleotides: TTA, GTG, CGA and GCT. (b–g) A step by step illustration of the synthesis process for these oligos. Partially constructed oligos are shown below the black line. The order of adding nucleotides is: AGTCGT (6 steps). If the alphabet method is used, it takes 8 steps (A,C,G,T,A,C,G,T) to build the complete oligos.

problem include a dynamic programming algorithm (Jiang and Li, 1995) and a branch and bound algorithm (Fraser, 1995), which are both exponential, the former in the number of strings and the latter in the size of the corresponding alphabet. Therefore, these algorithms are especially beneficial when, respectively, the number of strings or the alphabet size is restricted. Other research has aimed at devising approximation and (meta) heuristic algorithms, which achieve 'good', but not necessarily optimal, solutions in acceptable time. Approximation algorithms for the SCS include Alphabet (Barone et al., 2001), an approximate A* algorithm (Nicosia and Oriolo, 2003), Reduce_Expand (Barone et al., 2001), and Deposition and Reduction (DR) (Ning and Leong, 2006). The approximation ratio of the algorithms Alphabet, Reduce_Expand, and DR is $|\Sigma|$, which is not appealing. The algorithm DR is in fact a trivial combination of a heuristic mechanism with Alphabet, which, therefore, guarantees the approximation ratio of $|\Sigma|$. The approximate A* algorithm provides a $1+\varepsilon$ approximation ratio, for any fixed $\varepsilon>0$, particularly $\varepsilon=0.2$ in the experiments in Nicosia and Oriolo (2003). However, the algorithm is not efficient (i.e. is not of polynomial time complexity) and the size of the search tree can grow exponentially with the size of the given problem instance.

Among (meta) heuristic algorithms for the SCS are Tournament and Greedy (Irving and Fraser, 1993), Majority Merge (Branke et al., 1998), algorithms based on Genetic Algorithms (Branke and Middendorf, 1996; Branke et al., 1998), Ant System and Ant Colony Optimization (Michel and Middendorf, 1998; Michel and Middendorf, 1999), and Min_Height and Sum_Height (Kasif et al., 2002); the latter two specifically proposed for DNA sequences. More recent metaheuristic algorithms include a hybridization of Memetic Algorithms with Beam Search called Hybrid MA_BS (Gallardo et al., 2007), to which we simply refer as MA_BS, and a randomized Beam Search called Probabilistic Beam Search (PBS) (Blum et al., 2007). Another recent algorithm POEMS, together with its variants POEMS_f and POEMS_fw, was also proposed in Kubalik (2010). However, as reported in Kubalik (2010), it was outperformed by MA_BS in all the experimental cases. Based on the results reported in Blum et al. (2007), PBS outperforms MA_BS in most the experimental cases. On the other hand, DR outperforms Alphabet, Tournament, Greedy, and Majority merge in all the experimental cases as reported in Ning and Leong (2006). DR also outperforms Reduce_Expand for strings of length 50–100 (Ning and Leong, 2006). However, no comparison of DR and PBS has yet been made, leaving unclear which one is the state-of-the-art. The time complexity of DR, as specified in Ning and Leong (2006), is $O(|\Sigma|^3 nm^2)$, where $|\Sigma|$, n and m are, respectively, the size of the alphabet, the number of strings and the maximum length of the strings. No complexity of PBS or Hybrid MA_BS was reported in their proposing papers (Gallardo et al., 2007; Blum et al., 2007).

In this paper, we provide an improved beam search algorithm called IBS_SCS for the SCS problem, which, on average, outperforms all the three recent algorithms, namely DR, MA_BS, and PBS, in *all* experimental cases. A similar approach has been successfully used for the Longest Common Subsequence (LCS) problem in Mousavi

and Tabataba (2012). The proposed IBS_SCS algorithm has been inspired by the Blum et al.'s PBS algorithm but has the following distinct characteristics. First, it employs a different probability-based heuristic function than the one used in PBS. Using dynamic programming and at polynomial time and space costs, an array of probabilistic values is populated to facilitate the calculation of the heuristic values. Second, we use a technique called domination to further prune the search tree. The domination pruning technique has been inspired by Easton and Singireddy (2007) and Blum et al., (2009) used for the purpose of the LCS problem. However, our usage of this technique is different from those in Easton and Singireddy (2007) and Blum et al. (2009)). To be precise, in Blum et al. (2009), a candidate solution is checked for being dominated by every existing candidate solution, which is rather time-consuming. On the other hand, in Easton and Singireddy (2007), only the 'best-so-far' solution is used as the potential dominator for a new candidate solution. In our algorithm, we use the κ best-so-far solutions for this purpose, where κ is a control parameter in the algorithm. This approach gives us a control to achieve a good amount of pruning in reasonable time. Finally, given the same beam size, PBS does more work than IBS_SCS, which implies that IBS_SCS can benefit from a larger beam size than that of PBS, given the same amount of execution time. The IBS_SCS algorithm outperforms PBS in all the experimental cases, as reported in this paper. It also outperforms MA_BS by providing solutions of the same or higher (average) quality in all the experimental cases, including the cases where PBS was outperformed by MA_BS as reported in Blum et al. (2007).

Finally, IBS_SCS outperforms DR in all the experimental cases, which were set up based on the experiments conducted in Ning and Leong (2006). The DR algorithm consists of two stages called deposition and reduction. In the deposition stage, a number of candidate supersequences are created, which are then (tried to be) improved in the reduction stage. It uses Alphabet and a variant of Majority Merge in the deposition stage to generate the candidate supersequence. In fact, the use of Alphabet makes DR to be an approximation algorithm, which guarantees an approximation ratio of $|\sum|$. The DR algorithm is not deterministic only because of using a random tie braking; the rest of the logic is deterministic.

The proposed algorithm IBS_SCS is scalable. Its time complexity is polynomial in input size, and its computational cost can be arbitrarily reduced by reducing the beam size. The heuristic function used in the algorithm to evaluate candidate solutions does not suffer from the scalability issue of the heuristic proposed in Mousavi and Tabataba (2012) as an estimation mechanism is used for long strings. While the algorithm is significantly faster than other recent algorithm for the SCS, it yields superior solution quality in most of the cases. Because the proposed algorithm is scalable and sufficiently fast compared to other recent algorithms for the SCS, the main concentration of the reported experimental results is on the solution quality, i.e. on the length of the returned supersequences.

The rest of the paper is organized as follows. Section 2.1 provides the basic notations and definitions used in the paper. In Section 2.2, we describe how candidate solutions are evaluated and compared using the employed heuristic function. The proposed algorithm, together with its complexity analysis, is presented in Section 2.3. Section 3 reports the experimental results, and Section 4 concludes the paper.

2. Methods

2.1. Basic notations and definitions

Let s be a string of length m. We denote the length of s by $|s|$. We use $s[k]$, where k is an integer between 1 and m inclusive, to denote the kth character of s. We also use $s[k_1..k_2]$, where $1 \le k_1 \le k_2 \le |s|$, to indicate the substring of s obtained by removing its first k_1-1 characters and its last $|s|-k_2$ characters. Let s_1 and s_2 be two strings, $A_1=\{i|i\in\mathbb{N}_+, i\le|s_1|\}$ and $A_2=\{i|i\in\mathbb{N}_+, i\le|s_2|\}$, where $\mathbb{N}_+$ is the set of integers greater than zero. We say that s_2 is a *supersequence* of s_1, and write $s_1\prec s_2$, if there exists a monotone increasing total function g from A_1 to A_2 such that $s_1[k]=s_2[g(k)], \forall k\in A_1$. We call such a function g a *map of* s_1 *to* s_2. Note that such a map is not necessarily unique. Every string is considered to be a supersequence of the null string, i.e. the string of zero length. Finally, we say that s_1 is a *subsequence* of s_2 if s_2 is a supersequence of s_1.

Let x be a string and $S=\{s_1, \ldots, s_n\}$ be a nonempty set of n strings over the alphabet Σ. We write $S\prec x$ if $\forall s_i\in S, s_i\prec x$. The Shortest Common Supersequence (SCS) problem is then defined, given an input set S of strings, as to obtain a string x of the minimum length such that $S\prec x$. By an input string, we mean a string in S. Since the SCS can be efficiently solved for $n=2$, we assume $n>2$. We use m_i to mean $|s_i|$ and assume $m_i>0, \forall i\in\{1,\ldots,n\}$. We use m to denote $Max_{i\in\{1,\ldots,n\}}\{m_i\}$. A candidate solution is a string over Σ. We use (possibly indexed) x to denote a candidate solution. A candidate solution x is called *feasible* if $S\prec x$; it is otherwise called *infeasible*. A feasible candidate solution x is *optimal* if no feasible solution of a shorter length exists.

Let x be a candidate solution. We use $p_i(x)$ to denote the maximum possible integer k such that $s_i[1..k]\prec x$. By $r_i(x)$, we mean the string obtained by deleting the first $p_i(x)$ characters from s_i (see Fig. 2), and $R(x)$ is defined as the set $(r_i(x), i=1, \ldots, n)$. By a random string, we mean a string each character of which is obtained by selecting uniformly at random one of the characters in Σ. Finally, we use $pr(.)$ to denote the statistical probability function. Although there are two types of beam search, namely constructive and perturbative (local search), we use beam search in this paper to refer to the former.

2.2. Evaluation of candidate solutions

In this section, we explain how candidate solutions are evaluated and compared. The method used here to evaluate candidate solutions is adapted from Mousavi and Tabataba (2012) where a similar problem, the LCS, was addressed. To evaluate a candidate solution x, we use the probability of $R(x)\prec y$, where y is a random string and the strings in $R(x)$ are assumed to be independent in the sense that $\Pr(r_i(x)\prec y)=\Pr(r_i(x)\prec y|r_j(x)\prec y), \forall i,j\in\{1,\ldots,n\}, i\neq j$. Our intuition for this heuristic function is that a candidate solution x_1 is likely to be superior to another candidate solution x_2 (of the same length) if, given a random string y of length k, $x_1.y$ is more likely than $x_2.y$ to be a common supersequence of the input strings, where $x_i.y$, $i=1$ or 2, indicates the string obtained by appending y at the end of x_i. Of course, the probability of $R(x)\prec y$ depends on the candidate solution x. In the extreme case where x is a supersequence of all the input strings, i.e., when $S\prec x$, this probability is 1 because $R(x)$ would only include null strings.

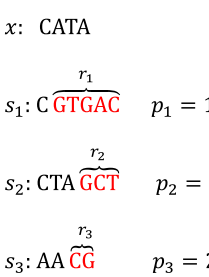

Fig.2. The values $p_i(x)$ and the strings $r_i(x)$, $i=1,2,3$, are illustrated for three input strings s_1, s_2, and s_3, and a candidate (infeasible) solution $x=$CATA.

In another extreme, where x is the null string, it become the probability of a random string being a supersequence of the input strings. As a rule of thumb, a higher value of $Pr(R(x)\prec y)$ is expected for a longer random string y, although this does not necessarily hold. We use $h_k(x)$ to denote the heuristic value of a candidate solution x. we have used the subscript k to emphasis the dependency of heuristic values on the length k of the random string y. Of course, if k is less than the length of the longest string in $R(x)$, the heuristic value, i.e. the probability of $R(x)\prec y$, will be zero. For a fair comparison of candidate solutions, we use the same value of k when evaluating the candidate solutions that are to be compared. A formula used to determine k is presented further in this section. We now show how to calculate heuristic values.

Theorem 1. *Let r be a string of length q and y be a random string of length k. Then:*

$$Pr(r\prec y)=\begin{cases}1 & \text{if } q=0\\ 0 & \text{if } q>k\\ \frac{1}{|\Sigma|}Pr(r[2..q]\prec y[2..k])+\frac{|\Sigma|-1}{|\Sigma|}Pr((r\prec y[2..k]) & \text{otherwise}\end{cases} \quad (1)$$

Proof. First note that in the third case (the *otherwise* case), $0<q\le k$, and the strings $r[2..q]$ and $y[2..k]$ are well-defined. By the definition of supersequence, every string is a supersequence of the null string and a string cannot be a supersequence of a longer one. Therefore, the first two cases of $q=0$ and $q>k$ hold trivially. In the remaining case, because $0<q\le k$, the strings r and y are of at least length 1 and both $r[1]$ and $y[1]$ exist. Depending on whether or not the characters $r[1]$ and $y[1]$ are equal, exactly one of the following two cases holds:

Case (i): $r[1]=y[1]$. In this case, we will prove that $r\prec y\Leftrightarrow r[2..q]\prec y[2..k]$. To than end, we will use the following property, to which we refer as the concatenation property: $\forall s_1,\forall s_2,\forall s_3,\forall s_4,(s_1\prec s_2)\wedge(s_3\prec s_4)\Leftrightarrow s_1.s_2\prec s_3.s_4$.

The "if" direction. We assume $r[2..q]\prec y[2..k]$ and show $r\prec y$:

$r[2..q]\prec y[2..q]\Rightarrow r[1].r[2..q]\prec y[1].y[2..q]$(*by the concatenation property*)

$\Rightarrow r[1..q]\prec y[1..q]\Rightarrow r\prec y$

The "only if" direction. We now assume $r\prec y$ and show $r[2..q]\prec y[2..k]$. Because $r\prec y$, there is, by the definition of supersequence, a total monotone increasing function $g(.)$ from $A_1=\{1,\ldots,q\}$ to $A_2=\{1,\ldots,k\}$ such that $r[i]=y[g(i)]$, $\forall i=1,...,q$. There are two possible cases: either $g(1)=1$ or $g(1)>1$. In either case, $g(i)>1,\forall i=2,...,q$ (note that $g(.)$ is monotone increasing). Now let $g'(.)$ be a total function from $\{1,\ldots,q-1\}$ to $\{1,\ldots,k-1\}$ defined as $g'(i)=g(i+1)-1$, $\forall i=1,...,q-1$. Of course, $g'(.)$ is also monotone increasing. Let r' and y' denote, respectively, $r[2..q]$ and $y[2..k]$. Then, we will have $r'[i]=r[i+1]$, $\forall i=1,...,q-1$, and $y'[i]=y[i+1]$, $\forall i=1,...,k-1$. Therefore, $\forall i=1,...,q-1$,

$y'[g'(i)]=y[g'(i)+1]=y[g(i+1)-1+1]$ (by the definition of$g'(.)$)

$=y[g(i+1)]$

$=r[i+1]$ (because y is a supersequence of r using the mapping $g(.)$)

$=r'[i]$

This means y' is a supersequence of r' using the mapping $g'(.)$. That is, $r[2..q]\prec y[2..k]$.

Case (ii): $r[1]\neq y[1]$. In this case, we show that $r\prec y\Leftrightarrow r\prec y[2..k]$.

The "if" direction. We assume $r\prec y[2\ldots q]$ and show $r\prec y$:

$r\prec y[2..q]\Rightarrow \varepsilon.r\prec y[1].y[2..q]$
(because $\varepsilon\prec y$ [1] and by the concatenation property)
$\Rightarrow r\prec y[1..q]$
$\Rightarrow r\prec y$

The "only-if" direction. We assume $r\prec y$ and show $r\prec y[2..q]$. Because $r\prec y$, there is a total monotone increasing function $g(.)$ from $A_1=\{1,\ldots,q\}$ to $A_2=\{1,\ldots,k\}$ such that $r[i]=y[g(i)]$, $\forall i=1,...,q$. However, $g(1)\neq 1$ because $r[1]\neq y[1]$. Therefore, $g(i)>1,\forall i=1,...,q$ (note that $g(.)$ is monotone increasing). Now let $g'(.)$ be a total function from $\{1,\ldots,q\}$ to $\{1,\ldots,k-1\}$ defined as $g'(i)=g(i)-1$, $\forall i=1,...,q$. Of course, $g'(.)$ is also monotone increasing. Let y' denote $y[2..k]$. Then, we will have $y'[i]=y[i+1]$, $\forall i=1,...,k-1$. Therefore, $\forall i=1,...,q$,

$y'[g'(i)]=y[g'(i)+1]$
$=y[g(i)-1+1]$ (by the definition of $g'(.)$)
$=y[g(i)]$
$=r[i]$ (because y is a supersequence of r using the mapping $g(.)$)

This means y' is a supersequence of r using the mapping $g'(.)$. That is, $r\prec y[2..k]$.

We have so far shown that in Case (i), where $r[1]=y[1]$, $r\prec y\Leftrightarrow r[2..q]\prec y[2..k]$ and that in Case (ii), where $r[1]\neq y[1]$, $r\prec y\Leftrightarrow r\prec y[2..k]$. Therefore, $Pr(r\prec y)=Pr(r[2..q]\prec y[2..k])$ in Case (i) and $Pr(r\prec y)=Pr(r\prec y[2..k])$ in Case (i). On the other hand, y is a random string based on the uniform probability distribution and the probability for case (i) is $1/|\Sigma|$. Consequently, the probability for case (ii) is $1-1/|\Sigma|=(|\Sigma|-1)/|\Sigma|$. Therefore,

$$Pr(r\prec y)=\frac{1}{|\Sigma|}Pr(r[2..q]\prec y[2..k])+\frac{|\Sigma|-1}{|\Sigma|}Pr(r\prec y[2..k]) \quad \square \quad (2)$$

Proposition. *Given a candidate solution x and a positive integer k, the heuristic value for a candidate solution x is calculated as*

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

where y is a random string of length k.

Proof. By the definition of the heuristic function, we have $h_k(x)=Pr(R(x)\prec y)$, where *y is a random string of length k*, and it is assumed $Pr(r_i(x)\prec y)=Pr(r_i(x)\prec y|r_j(x)\prec y),\forall i,j\in\{1,\ldots,n\},i\neq j$. Therefore,

$$h_k(x)=Pr(r_i(x)\prec y,\quad \forall i\in\{1,\ldots,n\})=\prod_{i=1}^{n}Pr(r_i(x)\prec y) \quad \square$$

Let C be a set of candidate solutions that are to be compared. Then, we use the formula $\underset{x\in C}{\text{Max}}_{i\in\{1,\ldots,n\}}\{r_i(x)\}\times lg|\Sigma|$ to determine the value for k. We have used the fact that a greater alphabet size and longer strings in $R(x)$ usually correspond to a longer supersequence of them. However, how to determine the best value for k requires further investigation and remains as an open question.

2.3. The IBS_SCS algorithm

In this section the proposed algorithm IBS_SCS is proposed, which is a constructive beam search metaheuristic algorithm. The standard beam search algorithm is a deterministic heuristic tree

search. It is similar to the breadth-first search in the sense that it incrementally constructs partial solutions and explores the search tree one level at a time. However, contrary to breadth-first search, it does not keep all the candidate solutions. The maximum number of candidate solutions to keep is called *beam size*, which we denote as β. Informally speaking, in the extreme case where the beam size is sufficiently large, the algorithm will act as the breadth-first search. Another extreme case is when the beam size is only 1, in which case it will act as a purely greedy algorithm. The beam search algorithm is also similar to the best-first search in the sense that it also uses a heuristic function to evaluate and compare candidate solutions.

Finally, there exists a scalability issue with the described heuristic function for large problem instances. To be precise, the probability $Pr(r \prec y)$ decreases rapidly as the length of r becomes close to the length of y, especially when r and y are long strings. To overcome this issue, we estimate $P(q,k)$ with $P(q-cut, k-cut)$, where *cut* is dynamically determined in such a way that $q-cut$ is positive and is either less than or as close as possible to 100. This approximation overcomes the scalability issue of the heuristic function and makes the algorithm sufficiently robust for long biological sequences.

Algorithm 1 presents a high-level pseudo-code of our proposed `IBS_SCS` algorithm for the SCS. As a beam search, the algorithm starts with an initially-singleton (the null string) set B of candidate solutions and incrementally builds longer ones by appending to them alphabet characters from the alphabet Σ, until a feasible candidate solution is obtained; the feasible solution is then returned and the algorithm terminates. However, (at most) β best candidate solutions are kept, using a heuristic function, and the others are eliminated.

The algorithm consists of an initialization section followed by a **while** loop, which consists of four steps. In the initialization section, the algorithm constructs an efficient data structure to speed up the calculation of heuristic values. The core idea behind using this data structure is the property of the heuristic function $h_k(.)$, described in the previous section, that, given an alphabet Σ, a string r of length q and a random string y of length k, both over Σ, the probability of $r \prec y$ is only dependent on q and k (see Theorem 1). Therefore, we construct a two-dimensional array P such that $P[q][k]$ holds the probability $Pr(r \prec y)$. Using dynamic programming, and based on Theorem 1, the array P is populated by the following recurrence:

$$P(q,k) = \begin{cases} 1 & \text{if } q = 0 \\ 0 & \text{if } q > k \\ \frac{1}{|\Sigma|}P(q-1,k-1) + \frac{|\Sigma|-1}{|\Sigma|}P(q,k-1) & \text{otherwise} \end{cases} \quad (3)$$

In the initialization section, the set B of candidate solutions is also initialized to a singleton containing the null string only. Having completed the initialization section, the **while** loop is run, which consists of four steps. In Step 1, each candidate solution x in B is extended by appending at its right end a character drawn from Σ, so obtaining $|\Sigma|$ new candidate solutions. The algorithm ends as soon as a feasible solution, determined using the function `feasible(.)`, is obtained. Every (infeasible) candidate solution is added to a set C, provided that it is 'usable'. More specifically, a candidate solution x^{new} obtained by appending a character l to a string x is called usable if the first character of at least one of the strings $r_i(x)$, $i=1, \ldots, n$, is l. This is checked by the function `usable(.)` in the algorithm. Therefore, the set C contains at most $\beta \times |\Sigma|$ (infeasible yet usable) candidate solutions. In Step 2, the heuristic values of the candidate solutions in C are computed, based on which the κ best candidate solutions comprise a list κ_Best_List of potential dominators to be used for dominance pruning. That is, in Step 3, each member of C is checked against the designated best solutions to decide whether it is dominated by any of them, in which case it is discarded from C. A candidate solution x_k is dominated by another candidate solution x_j if $p_i(x_k) \le p_i(x_j), \forall i = 1,\ldots,n$. 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 proposed algorithm runs in polynomial time in its input size (n, m, and $|\Sigma|$) and the values of the parameters β and κ.

Algorithm 1: The basic `IBS_SCS` algorithm for the SCS problem
Input: $S=\{s_1,s_2,\ldots,s_n\}$, each s_i is a string of at least one character
the alphabet of characters used in any of the strings is denoted by Σ
Output: a string x such that $S \prec x$
Parameter: the beam size β
Parameter: the number κ of best solutions used for dominance pruning
//initialization
For $q=0$ to m
  For $k=0$ to M //M is the maximum value for k, *i.e.* $M=m \times lg|\Sigma|$
    If $q=0$ **Then** $P[q][k]=1$;
    Else If $q > k$ **Then** $P[q][k]=0$;
    Else $P[q][k] = \frac{1}{|\Sigma|}P[q-1,k-1] + \frac{|\Sigma|-1}{|\Sigma|}P[q,k-1]$
$B=\{\varepsilon\}$ //the set of candidate solutions initially contains the null string only
While (true)
{

//Step 1: extension
$C=\emptyset$
For each $x \in B$
  For each letter $l \in \Sigma$
    x^{new}=the string obtained by adding l at the end of x
    If feasible(x^{new}) **Then**
      return x^{new}//the algorithm terminates here
    If usable(x^{new}) **Then**
      $C=C \cup \{x^{new}\}$//add it to C only if it contributes to covering a letter in some input strings

//Step 2: Evaluation of candidate solutions
$k = \max_{i \in \{1,\ldots,n\}, x \in C} \{r_i(x)\} \times lg|\Sigma|$//determine k
For each $x \in C$
  tmpH=1
  For $i=1$ to n
    tmpH=tmpH $\times P[|r_i(x)|][k]$
  $h_k(x)$=tmpH

//Step 3: dominance pruning
κ_Best_List=a list of κ best dominators
For each $x \in C$
  If x is dominated by some member of κ_Best_List **Then**
    $C=C-\{x\}$

//Step 4: selection
B=a set of β members of C with the highest heuristic values
//in the case $|C| < \beta$, let $B=C$
}

Proposition. *Algorithm* `IBS_SCS` *(Algorithm 1) is of complexity* $O(m^2 lg|\Sigma| + L^*(n\kappa\beta|\Sigma| + \beta|\Sigma| lg(\beta|\Sigma|)))$ *in computing time, where* L^* *is the length of the returned solution.*

Proof. In the initialization section before the **While** loop, the two-dimensional array P is populated using dynamic programming. The value of each entry is determined in $\Theta(1)$, in terms of two entries in its previous rows and columns. As can be seen, there are two nested loops and it takes $\Theta(m \times M)$ to populate the array. Because M is the maximum value used for k, and that we have used the formula $k = \underset{x \in C}{\text{Max}}_{i \in \{1,\ldots,n\}} \{r_i(x)\} \times lg|\Sigma|$ to determine the values of k (see Step 2 inside the **While** loop), $M = m \times lg|\Sigma|$. Therefore, the array P is populated in $\Theta(m \times m \times lg|\Sigma|) = \Theta(m^2 lg|\Sigma|)$.

There are four Steps inside the **While** loop, which we analyze in turn. Step 1 consists of two nested **For** loops, one iterating $O(\beta)$ times and the other iterating $|\Sigma|$ times. Inside these loops, a feasibility check (using the function feasible(.)) and a usability check (using the function usable(.)) are performed. The feasibility check involves checking whether a given candidate solution, here x^{new}, is a supersequence of the input strings. If we keep n indices of $p_i(x)$ associated with each candidate solution x in B, then we only need to update these indices for x^{new} and check whether $p_i(x^{new}) = m_i, \forall i = 1,\ldots,n$ (recall that $m_i = |s_i|$), which are performed in $\Theta(n)$. The usability check returns true if, and only if, $p_i(x^{new}) = p_i(x)+1, \exists i = 1,\ldots,n$, which is also performed in $\Theta(n)$. Therefore, Step 1 is run in $O(\beta|\Sigma|n)$.

Step 2 determines the heuristic values for all candidate solutions in C, the number of which is $O(\beta|\Sigma|)$. For each candidate solution x in C, the **For** loop iterates n times. Because of using the array P, each probability value (i.e., $(r_i(x) \prec y)$) is determined in $\Theta(1)$ inside the **For** loop, which are all multiplied together, making the heuristic value $h_k(x)$ for the candidate solution x. Therefore, Step 2 requires $O(\beta|\Sigma|n)$ (which is the same as that of Step 1).

Step 3 first determines the κ best solutions in C (stored in the κ_Best_List) and then examines each member of C against the designated best solutions for dominance pruning. Recall that there are at most $\beta \times |\Sigma|$ candidate solutions in C. Therefore, to determine the κ_Best_List can be performed in $O(\kappa\beta|\Sigma|)$. To check whether a candidate solution in C is dominated by a member of κ_Best_List is performed in $O(n)$; hence all the dominance checks are performed in $O(\beta|\Sigma|\kappa n)$. Therefore, the total complexity of Step 3 is $O(\beta|\Sigma|\kappa n)$.

Finally, Step 4 selects the best (at-most) β members of C to construct the new B, which can also be performed in $O(\beta|\Sigma|lg(\beta|\Sigma|))$ using red-black trees.[1]

Therefore, Steps 1–4 inside the **while** loop are performed in $O(\beta|\Sigma|n+\beta|\Sigma|n+\beta|\Sigma|\kappa n+\beta|\Sigma|lg(\beta|\Sigma|))$. Because the initialization section is run in $\Theta(m^2 lg|\Sigma|)$ and the **While** loop iterates L^* times, the whole algorithm is run in $O(m^2 lg|\Sigma| + L^*(\beta|\Sigma|\kappa n + \beta|\Sigma|lg(\beta|\Sigma|))$, which completes the proof. □

Note that the time complexity of the algorithm is polynomial in the input size (n, m, and $|\Sigma|$). Also note that $L^* = O(nm)$ (recall that m is the length of the longest input string), because, informally speaking- each character of a candidate solution must contribute to covering a character of at least one input string and there are at most a total of $n \times m$ such characters. Therefore, the time complexity of the algorithm may also be presented as $O(m^2 lg|\Sigma| + n^2 m\beta|\Sigma|\kappa + nm\beta|\Sigma|lg(\beta|\Sigma|)$, although this does not provide a tight bound.

[1] The current code of the proposed algorithm does not use red-black trees and requires $O(\beta^2|\Sigma|)$ for this step.

How to determine the appropriate values for the parameters κ and β depends on the underlying problem. In particular, a larger beam size β usually, but not necessarily, corresponds to a more accurate solution at a greater computational cost. However, if the solution quality using a specific beam size is near the optimal value, there will be not much point further increasing the beam size. On the other hand, using a "too small" beam size may adversely affect the solution quality. In fact, it depends on the underlying problem instance and such factors as what level of accuracy is required and how much run-time is affordable. Similarly, there is no strict rule for determining the best value for κ. A larger κ corresponds to a more computational time spent for dominance pruning. However, the pruning can lead to the reduction of computational cost that would otherwise be required for processing the pruned sub trees.

3. Results

In this section, we report the results of comparing our proposed algorithm with DR (Ning and Leong, 2006), MA_BS (Gallardo et al., 2007), and PBS (Blum et al., 2007), as three recent algorithms proposed for the SCS problem. Although there are other algorithms proposed in the literature as mentioned earlier in this paper, we do not compare our algorithm with them, because the most significant of them have already been shown to be outperformed by these three recent algorithms as reported in Ning and Leong (2006), Gallardo et al. (2007), and Blum et al. (2007).

The implementations of DR, MA_BS, and PBS were not available. The whole datasets used in Gallardo et al. (2007) and Blum et al. (2007) were available, and we used the reported results in Gallardo et al. (2007), and (Blum et al., 2007) to compare our algorithms with MA_BS and PBS. However, only real instances used in (Ning and Leong, 2006) were available (http://www.biomedcentral.com/content/supplementary/1471-2105-7-S4-S12-S1.zip; http://www.biomedcentral.com/content/supplementary/1471-2105-7-S4-S12-S2.zip). For random instances, we used their random instance generator (http://www-personal.umich.edu/~kning/random.html), and to compare IBS_SCS with DR on random instances, we implemented DR, precisely based on its specifications in (Ning and Leong, 2006). We implemented DR and IBS_SCS in Java using the Eclipse Platform on a Pentium IV machine with 2.4 GHz clock speed, 2 GB of RAM, and 2 MB of L2 cache. We allowed Java to use (at most) 1 GB of RAM.

In order to compare IBS_SCS with DR, the random instances were generated with exactly the same values for the parameters n, m, and $|\Sigma|$ as used in (Ning and Leong, 2006). There are altogether sixteen problem instances; the first eight, which are of relatively smaller numbers and lengths of strings, correspond to those in Table 4 and the other eight correspond to those in Table 1 of Ning and Leong (2006). The real instances are the DNA/protein instances used, respectively, in Tables 3 and 5 of (Ning and Leong, 2006). There are altogether 11 datasets, the first six for DNA and the other five for protein sequences, and each datasets includes ten instances. It is important to note that in some of the sequences in the real data, there were characters such as n- outside the underlying alphabet. Such characters were randomly replaced with one of their candidate characters. We ran IBS_SCS with the parameters $\beta = 100$ and $\kappa = 7$. We ran our implementation of DR on the random instances but used the reported results in Ning and Leong (2006) for real instances.

Table 1 provides the comparison of IBS_SCS with DR on random DNA sequences. The first and the second columns show, respectively, the number and the length of the sequences in each problem instance. Each row of the table corresponds to ten problem instances of the specified n and m. The third and the

Table 1
Comparison of IBS_SCS with DR on random DNA datasets (hence $|\Sigma|=4$). Each row corresponds to a dataset of ten random problem instances, and the average length of the solutions is reported for each algorithm. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The run-time (in seconds) of IBS_SCS and the improvement percentage (ρ%) are also reported (the last two columns). The improvement percentage, obtained by IBS_SCS, is defined as the reduction percentage in the average length of the solution. There are a total of sixteen datasets. The first eight, which contain relatively small instances, are of the same specifications (i.e., the same number n and length m of strings) as those in Table 4 (Ning and Leong, 2006). The other eight datasets are of the same specifications as those in Table 1 of Ning and Leong (2006). As can be seen, IBS_SCS outperforms DR by providing solutions of higher quality (i.e. with shorter lengths) in *all* the sixteen cases.

Number of strings	Length of strings	Average length of the returned solutions		Run-time of IBS_SCS	Improvement percentage
n	m	DR	IBS_SCS	$T_{\text{IBS\_SCS}}$	ρ%
5	10	21.2	20.1	0	5.18
10	10	25.3	24.2	0	4.34
50	10	30.9	29.6	0	4.2
100	10	32.2	31.1	0	3.41
5	100	196.7	185.5	0	5.69
10	100	228.2	211.8	0	7.18
50	100	262.3	251.9	0	3.96
100	100	269.9	261.6	1	3.07
100	100	268.3	260.6	1	2.86
500	100	278.2	274	4	1.5
1000	100	279.3	276.1	7	1.14
5000	100	282.3	281.7	43	0.21
100	1000	2529.4	2467.6	15	2.44
500	1000	2573.1	2542	44	1.2
1000	1000	2575.5	2557	81	0.71
5000	1000	2580.9	2571.7	469	0.35

fourth columns report the average length of the string returned by DR and IBS_SCS, respectively, over the ten instances of each row. The fifth column reports the average run-time of IBS_SCS, including the time needed to read in the data files. Finally, the last column calculates the (average) reduction percentage ρ% in the length of the string by IBS_SCS, defined as $\rho\%=(L_{\text{DR}}-L_{\text{IBS\_SCS}})/L_{\text{DR}}\times 100$, where L_{DR} and $L_{\text{IBS\_SCS}}$ denote the average lengths of the strings returned by DR and IBS_SCS, respectively.

As can be seen in Table 1, IBS_SCS outperforms DR by achieving shorter strings in *all* the sixteen cases. The reduction percentage ρ% varies from 0.21 (for the case $n=5000$ and $m=100$) to 7.18 (for the case $n=10$ and $m=100$), with an average of 2.95 (not shown in the table). No complete run-time report was provided in Ning and Leong (2006); it was only mentioned that DR took less than 10 s for the random instance ($n=100$, $m=100$) and an average of 5–10 min for the random instance ($n=1000$, $m=1000$). The run-time of our (efficient) implementation of DR observed for these two cases are 20 s and 18091 s (about 30 min), respectively. However, the run-times of IBS_SCS for the corresponding instances are 1 s and 81 s (less than two minutes). This suggests that IBS_SCS should be significantly faster than DR. It is important to note the DR was observed to take more than 27 h for the last instance ($n=5000$, $m=1000$), for which no run-time was reported in Ning and Leong (2006). The time taken by IBS_SCS for that instance was 469 s (less than 8 min). Fig. 3 depicts the growth of run-time for both our implemented DR and IBS_SCS with the number of strings n for a fixed sequence length of $m=100$ (our implemented DR and IBS_SCS algorithms are available on request).

Table 2 provides the comparison of IBS_SCS with DR on real biological sequences. The first column in this table represents the dataset name. The definitions for the second to the seventh

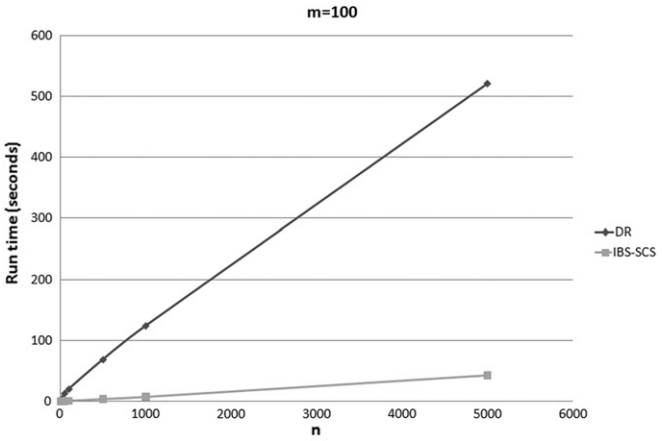

Fig. 3. The growth of run-time for both DR and IBS-SCS with n, for the eight (middle) cases of Table 1, where $m=100$. For these datasets, $|\Sigma|=4$, and the number of strings n are 5, 10, 50, 100, 500, 1000, and 5000. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$.

Table 2
Comparison of IBS_SCS with DR on real DNA and protein sequences ($|\Sigma|=4$ for DNA instances, and $|\Sigma|=20$ for protein instances). The datasets are those used in Tables 3 and 5 in Ning and Leong (2006), whose names are specified in the first column. Each row corresponds to ten instances, and the average length of the solutions returned by each algorithm is reported. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The run-time (in seconds) of IBS_SCS and the improvement percentage (ρ%) are also reported. The improvement percentage is defined as the reduction percentage in the average length of the solutions, obtained by IBS_SCS. There are a total of 11 datasets, the first six for DNAs and the other for protein sequences. The results for DR are directly taken from Tables 3 and 5 in Ning and Leong (2006). As can be seen, IBS_SCS outperforms DR by obtaining higher quality solutions in *all* the eleven cases.

Benchmark name	Number of strings	Length of strings	Average length of the returned solutions		Run-time of IBS_SCS	Improvement percentage
	n	m	DR	IBS_SCS	$T_{\text{IBS\_SCS}}$	ρ%
DNA-1	100	500	1364.4	1284.6	6	5.84
DNA-2	500	500	1420.7	1351.6	22	4.86
DNA-3	100	1000	2675.7	2540.1	16	5.06
DNA-4	500	1000	2769.1	2662.9	48	3.83
DNA-5	100	100	284.4	272.3	1	4.25
DNA-6	500	100	296.8	288.1	4	2.93
PROT-1	100	500	4846.3	4374.9	159	9.72
PROT-2	500	500	5548.6	5162.5	464	6.95
PROT-3	1000	500	5734	5394.5	815	5.92
PROT-4	100	100	1008.7	910.6	16	9.72
PROT-5	500	100	1199.8	1118.1	74	6.8

columns are as those for the first to the sixth columns of Table 1. As indicated by Table 2, IBS_SCS outperforms DR by obtaining shorter strings in *all* the eleven cases. The reduction percentage ρ% ranges from 2.93 (for DNA-6) to 9.72 (for PROT-1 and PROT-4), with an average of 5.98 (not shown in the table). Again IBS_SCS was observed to be significantly faster than DR. An interesting observation is that the reduction percentage gained by IBS_SCS is significantly higher for real than random biological sequences. The minimum ρ% for real instances is, as already-mentioned, 2.93, which is about the average ρ% (2.95) for random instances. This indicates that IBS_SCS is promising for practical use and further research for this purpose.

To compare our algorithm with PBS and MA_BS, we used the same random and real instances as used in Blum et al. (2007).The datasets consist of one random and five biological benchmarks. The random datasets are categorized into 5 classes, each of which is specified with a different alphabet size, namely 2, 4, 8, 16, and

24. Each class contains five instances, and each instance consists of eight strings, four of length 40 and the other four of length 80. On the other hand, each biological instance is characterized by a biological sequence s and a probability p. More specifically, the strings within each instance are obtained from the same biological sequence s by removing each of its symbols with a fixed probability p. The number of the strings in each instance is 10. Five biological sequences each with three probabilities of 0.1, 0.15, and 0.20 have been used to construct a total of 15 instances. The five biological sequences are two SARS Coronavirus DNA sequences obtained from a genomic database (http://gel.ym.edu.tw/sars/genomes.html) and three protein sequences obtained from Swiss-Prot (http://www.expasy.org/sprot). The DNA sequences are of the lengths 158 and 1269, and the protein sequences are Oxytocin, p53 and Estrogen, which are of the lengths 125, 393, and 595, respectively. The lengths of the optimal SCSs for these instances are, respectively, 158, 1269, 125, 393, and 595 (Blum et al., 2007).

For MA_BS and PBS, we used the reported results in (Blum et al., 2007). Contrary to IBS_SCS, MA_BS and PBS are not deterministic; they were run more than once in Blum et al. (2007), and the best, the mean, and the standard deviation of their solution quality and run-time were reported. However, we ran only IBS_SCS once on each instance. We used $\kappa=7$ and $\beta=700$ for random and $\beta=100$ for real instances.

Table 3 compares IBS_SCS with MA_BS and PBS on the random datasets. The first column shows the alphabet size. The second and the third columns show the length of the solutions returned by MA_BS and PBS, respectively. The fourth column reports the length of the solutions returned by IBS_SCS. The fifth column reports the average run-time of IBS_SCS. Finally, the last column calculates the reduction percentage ρ% achieved by IBS_SCS with respect to PBS (which is almost superior to MA_BS), defined as $\rho\%=(L_{\mathrm{PBS}}-L_{\mathrm{IBS\_SCS}})/L_{\mathrm{PBS}}\times 100$, where L_{PBS} and $L_{\mathrm{IBS\_SCS}}$ denote the (average) length of the solutions returned by PBS and IBS_SCS, respectively.

As can be seen in Table 3, in *all* the five cases, IBS_SCS outperforms MA_BS and PBS, even with respect to their best runs. We do not intend to provide a precise run-time comparison, because of using different machines. However, it can be inferred that IBS_SCS should not be any slower than the other two algorithms; the run-time limit for PBS and MA_BS were reported in Blum et al. (2007) and Gallardo et al. (2007) as to be 350 and 600 s, respectively, whereas the longest run-time of IBS_SCS is only 8 s (the last row of Table 3). Note that with a smaller beam size of 200, the longest run-time of our algorithm was even less than 1.5 s (not shown in the table), while it still outperformed PBS in all the five cases. Finally, as can be seen in the last column of Table 3, IBS_SCS achieves the reduction percentage of 1.35–3.52 over PBS, with an average of more than 2.5%. Note that MA_BS outperforms PBS with respect to the average length of the solutions for the case $|\Sigma|=2$, but it is still outperformed by IBS_SCS in this case.

The results of experiments over the biological sequences are reported in Tables 4–8. The first columns in these tables show the value for the probability p. The next column shows the maximum length m of input strings. The next three columns show the results of the algorithms MA_BS, PBS, and IBS_SCS, respectively. The last column shows the run-time of IBS_SCS. Tables 4 and 5

Table 4
Comparison of IBS_SCS with MA_BS and PBS on the 158-Nucleotide SARS dataset (hence $|\Sigma|=4$). The dataset and the results for MA_BS and PBS are those in Blum et al. (2007). The first column shows p, the probability of each letter in a sequence being deleted. The number of strings is $n=10$. Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The last column shows the run-time of IBS_SCS (in seconds). As can be seen, IBS_SCS obtains the optimal solution, within 1 s, in *all* of the cases.

Probability	Length of strings	average length of the returned solutions			Run-time of IBS_SCS
p	m	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\mathrm{IBS\_SCS}}$
0.10	146	158 158 ± 0	158 158 ± 0	158	0
0.15	137	158 158 ± 0	158 158 ± 0	158	0
0.20	130	158 158 ± 0	158 158 ± 0	158	0

Table 5
Comparison of IBS_SCS with MA_BS and PBS on the 1269-Nucleotide SARS dataset (hence $|\Sigma|=4$). The dataset and the results for MA_BS and PBS are those in (Blum et al., 2007). The first column shows p, the probability of each letter in a sequence being deleted. The number of strings is $n=10$. Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The last column shows the run-time of IBS_SCS (in seconds). It can be seen that IBS_SCS achieves the optimal solution, within 2 s, in *all* the cases.

Probability	Length of strings	Average length of the returned solutions			Run-time of IBS_SCS
p	m	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\mathrm{IBS\_SCS}}$
0.10	1156	1269 1269 ± 0	1269 1269 ± 0	1269	2
0.15	1097	1269 1269 ± 0	1269 1303.8 ± 36.6	1269	2
0.20	1039	1269 1269 ± 0	1571 1753.2 ± 61.0	1269	2

Table 3
Comparison of IBS_SCS with MA_BS and PBS on random instances. There are five different alphabet sizes (first column), for each of which there are five instances. Each instance consists of eight strings (hence $n=8$), four of length 40 and the other four of length 80 (hence $m=80$ – recall that m is the length of the longest input string). These instances are exactly the random instances used in Blum et al. (2007). The results of MA_BS and PBS are taken from Blum et al. (2007). Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance, which are then averaged on all the 5 instances for each row. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=700$ and $\kappa=7$. The run-time of IBS_SCS (in seconds) and the reduction percentage $\rho\%$ are also reported in the last two columns. As can be seen, in all of the five cases, IBS_SCS outperforms MA_BS and PBS, even with respect to their best runs.

Alphabet size	Average length of the returned solutions			Run-time of IBS_SCS	Improvement percentage
$\lvert\Sigma\rvert$	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\mathrm{IBS\_SCS}}$	$\rho\%$
2	110.6 110.7 ± 0.0	110.8 110.9 ± 1.7	109.4	1	1.35
4	145.6 146.4 ± 0.5	144.8 145.4 ± 1.5	142.4	2	2.06
8	191.6 192.6 ± 1.4	186.4 187.2 ± 1.7	180.6	3	3.52
16	242.8 244.0 ± 1.0	240.4 241.9 ± 3.4	235.6	6	2.6
24	280.2 281.2 ± 0.8	276.4 277.9 ± 4.0	268.8	8	3.27

Table 6
Comparison of IBS_SCS with MA_BS and PBS on the 125-Aminoacid Oxytocin dataset (hence $|\Sigma|=20$). The dataset and the results for MA_BS and PBS are those in Blum et al. (2007). The first column shows p, the probability of each letter in a sequence being deleted. The number of strings is $n=10$. Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The last column shows the run-time of IBS_SCS (in seconds). IBS_SCS obtains the optimal solution, within 1 s, in *all* of the cases.

probability	Length of strings	Average length of the returned solutions			Run-time of IBS_SCS
p	m	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\text{IBS\_SCS}}$
0.10	115	125 125 $\pm$ 0	125 125 $\pm$ 0	125	0
0.15	108	125 125 $\pm$ 0	125 125 $\pm$ 0	125	0
0.20	102	125 125 $\pm$ 0	125 125 $\pm$ 0	125	0

Table 7
Comparison of IBS_SCS with MA_BS and PBS on the 393-Aminoacid p53 dataset (hence $|\Sigma|=20$). The dataset and the results for MA_BS and PBS are those in Blum et al. (2007). The first column shows p, the probability of each letter in a sequence being deleted. The number of strings is $n=10$. Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The last column shows the run-time of IBS_SCS (in seconds). It is observed that IBS_SCS obtains the optimal solution, within 1 s, in *all* the cases.

Probability	Length of strings	Average length of the returned solutions			Run-time of IBS_SCS
p	m	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\text{IBS\_SCS}}$
0.10	366	393 393 $\pm$ 0	393 393 $\pm$ 0	393	1
0.15	348	393 393 $\pm$ 0	393 393 $\pm$ 0	393	1
0.20	335	393 393 $\pm$ 0	393 393 $\pm$ 0	393	1

Table 8
Comparison of IBS_SCS with MA_BS and PBS on the 595-Aminoacid Estrogen dataset (hence $|\Sigma|=20$). The dataset and the results for MA_BS and PBS are those in (Blum et al., 2007). The first column shows p, the probability of each letter in a sequence being deleted. The number of strings is $n=10$. Because MA_BS and PBS are not deterministic, their reported results are statistical values of the best, the mean, and the standard deviation of their several runs on the same instance. However, IBS_SCS is deterministic and is run only once on each instance. IBS_SCS was run with the parameters $\beta=100$ and $\kappa=7$. The last column shows the run-time of IBS_SCS (in seconds). IBS_SCS obtains the optimal solution, within 1 s, in *all* the cases.

Probability	Length of strings	Average length of the returned solutions			Run-time of IBS_SCS
p	m	MA_BS best mean $\pm\sigma$	PBS best mean $\pm\sigma$	IBS_SCS	$T_{\text{IBS\_SCS}}$
0.10	546	595 595 $\pm$ 0	595 595 $\pm$ 0	595	1
0.15	522	595 595 $\pm$ 0	595 595 $\pm$ 0	595	1
0.20	501	595 595 $\pm$ 0	596 596 $\pm$ 0	595	1

report the results of the experiments on the Nucleotide SARS datasets (hence of $|\Sigma|=4$), and Tables 6–8 provide the results for the Aminoacid protein datasets (hence of $|\Sigma|=20$).

As can be seen in Tables 4–8, both MA_BS and PBS obtain optimal solutions in all but the last two datasets of Table 5 and the last dataset of Table 8, where PBS is outperformed by MA_BS. However, IBS_SCS obtains optimal solutions in *all* the cases in these tables. As shown in the last columns of Tables 4–8, IBS_SCS needs, at worst, about a couple of seconds to find the optimal solutions; the run-time limit reported in Gallardo et al. (2007) and Blum et al. (2007) are 600 and 350 s, respectively.

Finally, to observe how the performance of IBS_SCS varies with the parameters κ and β, we conducted two more types of experiments, on the datasets of Table 3. In the first series of experiments, we used the same value of 7 for κ but used the values 100, 400, 700, 1000, and 1300 for β. We used the results for the case $\beta=700$ as the reference and calculated the percentages of the changes due to using the other values of β. To be precise, for each value $v=100$, 400, 1000, 1300, we calculated $(L_2-L_1)/L_1\times 100$, where L_2 and L_1 are the lengths of the solutions obtained using, respectively, $\beta=v$ and $\beta=700$. Because there are five instances for each alphabet size $|\Sigma|$ in Table 3, we then averaged the percentages of changes over the five instances of each alphabet size.

The results are shown in Table 9. The first column in this table shows the alphabet size. The second and the third columns show, respectively, the average and the variance of the percentages of changes due to using $\beta=100$, as opposed to 700, over the five instances with $|\Sigma|=2$. These values are denoted by D and V, respectively. The next three pairs of columns report the respective values for the other beam sizes of 400, 1000, and 1300. The last row shows the average of the values D over all the instances.

It can be observed from Table 9 that the results are not much sensitive to the specified beam size values in that the average percentage of the changes (D) is under 1% in majority of cases. It is also observed that there is no obvious pattern for changing the results with the beam size. However, on average (shown in the last row of the table), the worst results correspond to the smallest beam size 100 (with the average D of 1.28%).

Fig. 4 depicts how the average percentage of changes D varies with the beam size, for different values of alphabet size. As can be seen in Fig. 4, the curves tend to fall with increasing the beam size, but a number of exceptions are also observed, e.g. for $|\Sigma|=24$.

In the second series of experiments, we used the same beam size of $\beta=700$ but different values of 3, 5, 7, 9, and 11 for the parameter κ. Similarly, we used the results for the case ($\beta=700$ and) $\kappa=7$ as the reference and calculated the percentages of the changes due to using the other values of κ. More specifically, for each value $v=3$, 5, 9, 11, we calculated $(L_2-L_1)/L_1\times 100$, where L_2 and L_1 are the lengths of the solutions obtained using, respectively $\kappa=v$ and $\kappa=7$. Then, we averaged the percentages of changes D over the five instances of each alphabet size.

The results are presented in Table 10. The layout of Table 10 is similar to that of Table 9, except that the results in Table 10 are presented for different values of κ, as opposed to β. As shown in Table 10, except for two cases of $\kappa=3$ and $\kappa=5$ in the last row

Table 9
The average and the variance of the percentage of changes in the length of the returned solution by IBS_SCS, for different beam sizes of 100, 400, 1000, and 1300, with a fixed $\kappa=7$, in comparison with that for the reference beam size of 700. The dataset is the simulated biological sequences used in Blum et al. (2007), with $n=8$, $m=80$, and $|\Sigma|=2$, 4, 8, 16, and 24, which is also used in Table 3.

$\lvert\Sigma\rvert$	$\beta=100$		$\beta=400$		$\beta=1000$		$\beta=1300$	
	D	V	D	V	D	V	D	V
2	0.4	0.3	0.2	0.2	0	0	0	0
4	−0.2	0.4	−0.3	0.4	−0.8	0.3	−0.6	0.2
8	1.8	1.7	0.6	0.5	0.8	2.4	0.5	0.6
16	2.8	0.3	1.5	2.4	0.3	0.8	0.1	0.6
24	1.6	4.8	0.8	2.1	1	4.1	1.1	1.9
Total average of *D*	**1.28**		**0.56**		**0.26**		**0.22**	

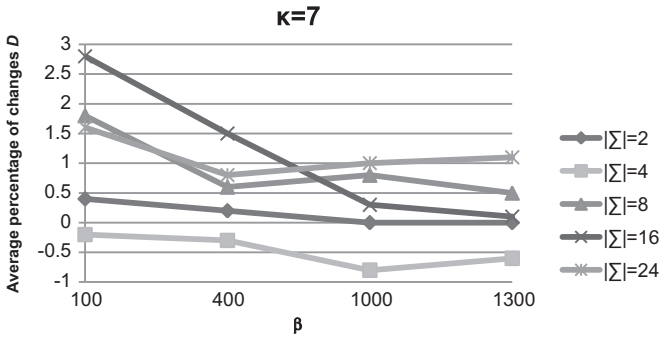

Fig. 4. The average percentage of changes D versus the beam size β, by running IBS_SCS on the datasets of Table 3, with $n=8$ and $m=80$. The beam sizes are 100, 400, 1000, and 1300, with the beam size 700 as the reference. The parameter κ is fixed to 7. For each alphabet size $|\Sigma|$, a separate curve is shown.

Table 10
The average D and the variance V of the changes in the length of the returned solution by IBS_SCS, for different values of 3, 5, 9, and 11 for the parameter κ, with a fixed beam size of 700, in comparison with that for the reference value 7 for κ. The dataset is the simulated biological sequences used in Blum et al. (2007), with $n=8$, $m=80$, and $|\Sigma|=2$, 4, 8, 16, and 24, which is also used in Table 3.

$\|\Sigma\|$	$\kappa=3$		$\kappa=5$		$\kappa=9$		$\kappa=11$	
	D	V	D	V	D	V	D	V
2	0	0	0	0	0	0	0	0
4	0	0	−0.2	0.2	−0.5	0.4	−0.6	0.4
8	0.9	1.4	0.9	0.4	0.1	0.2	0.3	0.7
16	0.8	3.6	0.4	2.9	−0.3	0.8	−0.9	0.8
24	1.5	1.5	1.2	1.1	0.4	3.2	0.2	2.2
Total average of D	**0.64**		**0.46**		**−0.06**		**−0.2**	

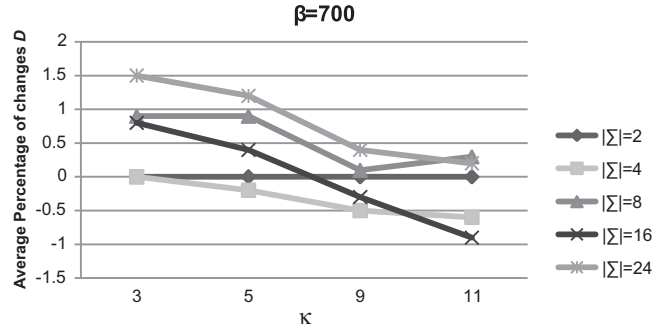

Fig. 5. The average percentage of changes D versus κ, by running IBS_SCS on the datasets of Table 3, with $n=8$ and $m=80$. The values of κ are 3, 5, 9, and 11, with $\kappa=7$ as the reference. A fix beam size of 700 is used, and a curve is depicted for each alphabet size $|\Sigma|$.

$|\Sigma|=24$, the percentage of the changes is under 1%. This suggests that the algorithm is sufficiently tolerant to the choice of κ. However, the solution quality usually increases with increasing κ. This is better observed in Fig. 5, which shows how the average percentage of changes D varies with κ, for different values of alphabet size. As can be seen in Fig. 5, the curves tend to fall with increasing κ, although there are still exceptions, e.g. for the case $|\Sigma|=8$.

4. Conclusion

In this paper, a deterministic heuristic algorithm for the shortest common supersequence problem was proposed. The algorithm is a constructive beam search and uses a heuristic function different from those already proposed in the literature for the SCS problem. The algorithm also uses the dominance property to effectively prune the search tree. However, it does not check for dominance with respect to every existing candidate solution as it would lead to a significant time-consumption. Neither is it restricted to using only the best solution found so far as it would then be not using the true power of dominance pruning. Instead, it selects the κ best solutions found so far as potential dominators of candidate solutions at each iteration, where κ is a control parameter in the algorithm. The proposed algorithm was compared with three recent algorithms proposed for the problem on both simulated and real biological sequences. It outperformed all the three algorithms in all of the experimental cases. This justifies that the proposed algorithm is promising for further research and improvements.

Possible avenues for future work include (i) to devise a method to determine appropriate values for k (see Eq. (3)), because its proper setting can significantly improve the solution quality and there is enough room for improvement in this regard, (ii) to generalize the employed heuristic to the case where the input strings are correlated to further improve the performance of the algorithm in such domains, and (iii) to dynamically determine the appropriate values for the control parameters κ and β.

Acknowledgment

The authors would like to thank Dr. C. Blum, Dr. J.E. Gallardo and Dr. C. Cotta for kindly providing us with their random and real datasets. We would also like to thank Dr. K. Ning and Dr. H.W. Leong for their real benchmarks and random-instance generator. Thanks are also due to the anonymous reviewers for their constructive comments.

Appendix A. Supplementary materials

Supplementary data associated with this article can be found in the online version at doi:10.1016/j.engappai.2011.08.006.

References

Barone, P., Bonizzoni, P., Della Vedova, G., Mauri, G., 2001. An approximation algorithm for the shortest common supersequence problem: an experimental analysis. In: Proceedings of the 2001 ACM Symposium on Applied Computing. ACM, Las Vegas, Nevada, United States, pp. 56–60.

Blum, C., Blesa, M.J., López-Ibáñez, M., 2009. Beam search for the longest common subsequence problem. Computers & Operations Research 36, 3178–3186.

Blum, C., Cotta, C., Fernández, A., Gallardo, J., 2007. A Probabilistic Beam Search Approach to the Shortest Common Supersequence Problem, Evolutionary Computation in Combinatorial Optimization. Springer, Berlin/Heidelberg, pp. 36–47.

Branke, J., Middendorf, M., 1996. Searching for shortest common supersequences by means of a heuristic based genetic algorithm. In: Alander, J.T. (Ed.), Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications. University of Vaasa, Vaasa, Finland, pp. 105–114.

Branke, J., Middendorf, M., Schneider, F., 1998. Improved heuristics and a genetic algorithm for finding short supersequences. OR Spectrum 20, 39–45.

Chaudhuri, S., Bruno, N., 2008. Method and Apparatus for Generating Statistics on Query Expressions for Optimization. Microsoft Corporation (Redmond, WA, US), United States.

Easton, T., Singireddy, A., 2007. A specialized branching and fathoming technique for the longest common subsequence problem. Int. J. Opera. Res. 4, 98–104.

Foulser, D.E., Li, M., Yang, Q., 1992. Theory and algorithms for plan merging. Artif. Intell. 57, 143–181.

Fraser, C.B., 1995. Subsequences and Supersequences of Strings, Computing Science. University of Glasgow.

Gallardo, J.E., Cotta, C., Fernandez, A.J., 2007. On the hybridization of memetic algorithms with branch-and-bound techniques. IEEE Trans. Syst. Man Cybern. B: Cybern. 37, 77–83.

Garey, M., Johnson, D., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co Ltd, New York, NY.
〈http://gel.ym.edu.tw/sars/genomes.html〉.
〈http://www-personal.umich.edu/~kning/random.html〉.
〈http://www.biomedcentral.com/content/supplementary/1471-2105-7-S4-S12-S1.zip〉.
〈http://www.biomedcentral.com/content/supplementary/1471-2105-7-S4-S12-S2.zip〉.
〈http://www.expasy.org/sprot〉.
Hubbell, E.A., Morris, M.S., Winkler, J.L., 1996. Computer-Aided Engineering System for Design of Sequence Arrays and Lithographic Masks. Affymax Technologies N.V. (Curaco, AN), United States.
Irving, R.W., Fraser, C., 1993. On the worst-case behaviour of some approximation algorithms for the shortest common supersequence of k strings. In: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching. Springer Berlin/Heidelberg, pp. 63–73.
Jiang, T., Li, M., 1995. On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 24, 1122–1139.
Kasif, S., Weng, Z., Derti, A., Beigel, R., DeLisi, C., 2002. A computational framework for optimal masking in the synthesis of oligonucleotide microarrays. Nucleic Acids Res. 30, e106.
Kubalik, J., 2010. Efficient stochastic local search algorithm for solving the shortest common supersequence problem. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, GECCO '10. ACM, Portland, Oregon, USA, pp. 249–256.
Maier, D., 1978. The complexity of some problems on subsequences and supersequences. J. ACM 25, 322–336.
Michel, R., Middendorf, M., 1998. An island Model Based Ant System with Lookahead for the Shortest Supersequence Problem, Parallel Problem Solving from Nature—PPSN V. Springer, Berlin/Heidelberg, pp. 692–701.
Michel, R., Middendorf, M., 1999. An ACO Algorithm for the Shortest Common Supersequence Problem, New Ideas in Optimization. McGraw-Hill Ltd, UK, pp. 51–62.
Mousavi, S.R., Tabataba, F., 2012. An improved algorithm for the longest common subsequence problem. Comput. Oper. Res. 39, 512–520.
Nicosia, G., Oriolo, G., 2003. An approximate A* algorithm and its application to the SCS problem. Theor. Comput. Sci. 290, 2021–2029.
Ning, K., Choi, K.P., Leong, H.W., Zhang, L., 2005. A post-processing method for optimizing synthesis strategy for oligonucleotide microarrays. Nucleic Acids Res. 33, e144.
Ning, K., Leong, H., 2006. Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm. BMC Bioinformatics 7, S12.
Rahmann, S., 2003. The shortest common supersequence problem in a microarray production setting. Bioinformatics 19, ii156–ii161.
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 Trans. Database Syst. (TODS) 13, 23–52.
Skotheim, R., Thomassen, G., Eken, M., Lind, G., Micci, F., Ribeiro, F., Cerveira, N., Teixeira, M., Heim, S., Rognes, T., Lothe, R., 2009. A universal assay for detection of oncogenic fusion transcripts by oligo microarray analysis. Mol. Cancer 8, 5.
Storer, J.A., 1988. Data Compression: Methods and Theory. Computer Science Press, Inc.
Timkovskii, V.G., 1989. Complexity of common subsequence and supersequence problems and related problems. Cybern. Syst. Anal. 25, 565–580.