Computers & Operations Research 36 (2009) 3178 – 3186

Contents lists available at ScienceDirect

Computers & Operations Research

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

Beam search for the longest common subsequence problem

Christian Blum[a,*], Maria J. Blesa[a], Manuel López-Ibáñez[b]

[a]ALBCOM, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain
[b]School of Engineering and The Built Environment, Napier University, Edinburgh, UK

ARTICLE INFO

Available online 20 February 2009

Keywords:
Beam search
Longest common subsequence problem

ABSTRACT

The longest common subsequence problem is a classical string problem that concerns finding the common part of a set of strings. It has several important applications, for example, pattern recognition or computational biology. Most research efforts up to now have focused on solving this problem optimally. In comparison, only few works exist dealing with heuristic approaches. In this work we present a deterministic beam search algorithm. The results show that our algorithm outperforms the current state-of-the-art approaches not only in solution quality but often also in computation time.

1. Introduction

The longest common subsequence (LCS) problem is a classical string problem. Given a string s over an alphabet Σ, each string that can be obtained from s by deleting characters is called a subsequence of s. Given a problem instance $(\mathscr{S}, \Sigma)$, where $\mathscr{S} = \{s_1, s_2, \ldots, s_n\}$ is a set of n strings over a finite alphabet Σ, the LCS problem seeks to find the longest string t^* that is a subsequence of all the strings in $\mathscr{S}$. Such a string t^* is called a *LCS* of the strings in $\mathscr{S}$. Traditional computer science applications of this problem are in data compression [31], syntactic pattern recognition [22], file comparison [1], text edition [26] and query optimization in databases [27]. More recent applications include computational biology [30,19] and the production of circuits in field programmable gate arrays [7].

The LCS problem was shown to be NP-hard [23] for an arbitrary number n of input strings. If n is fixed, the problem is polynomially solvable by dynamic programming [16]. Standard dynamic programming approaches for this problem require $O(l^n)$ of time and space, where l is the length of the longest input string and n is the number of strings. While several improvements may reduce the complexity of standard dynamic programming to $O(l^{n-1})$ (Bergoth et al. [3] provided numerous references), dynamic programming becomes quickly impractical when n grows.

An alternative to dynamic programming was proposed by Hsu and Du [17]. This algorithm was further improved in [11,29] by incorporating branch and bound techniques. The resulting algorithm, called specialized branching (SB), has a complexity of $O(n|\Sigma|^{|t^*|})$, where t^* is the LCS. According to the empirical results of Easton and Singireddy [10], SB outperforms dynamic programming for large n and small l. Additionally, Singireddy [29] proposed an integer programming approach based on branch and cut.

Approximate methods for the LCS problem were first proposed by Chin and Poon [8] and Jiang and Li [20]. The long run algorithm (LR) [20] returns the LCS consisting of a single letter, which is always within a factor of $|\Sigma|$ of the optimal solution. The expansion algorithm (EXPANSION) proposed by Bonizzoni et al. [6] and the BEST-NEXT heuristic [12,18] also guarantee an approximation factor of $|\Sigma|$; however, their results are typically much better than those of LR in terms of solution quality. Guenoche and Vitte [15] described a greedy algorithm that uses both forward and backward strategies, and the resulting solutions are merged subsequently. Earlier approximate algorithms for the LCS problem can be found in Bergroth et al. [2] and Brisk et al. [7].

More recently, Easton and Singireddy [10] proposed a large neighbourhood search heuristic called time horizon SB (THSB) that makes internal use of the SB algorithm. In addition, they implemented a variant of Guenoche and Vitte's algorithm [15], referred to as G&V, that selects the best solution obtained from running the original algorithm with four different greedy functions proposed by Guenoche [14]. Easton and Singireddy [10] compared their algorithm with G&V, LR and EXPANSION, showing that THSB was able to obtain better results than all the competitors in terms of solution quality. Their results also showed that G&V outperforms EXPANSION and LR with respect to solution quality, while requiring less computation time than EXPANSION and a computation time comparable to the one of LR. Finally, Shyu and Tsai [28] studied the application of ant colony optimization (ACO) to the LCS problem and concluded that their

* Corresponding author.
E-mail addresses: cblum@lsi.upc.edu (C. Blum), mjblesa@lsi.upc.edu (M.J. Blesa), m.lopez-ibanez@napier.ac.uk (M. López-Ibáñez).

0305-0548/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2009.02.005

algorithm dominates both Expansion and Best-Next in terms of solution quality, while being much faster than Expansion.

In this work we propose the application of beam search (BS) to the LCS problem. BS is a classical tree search method that was introduced in the context of scheduling [24]. The central idea behind BS is the parallel and non-independent construction of a limited number of solutions with the help of a greedy function and an upper bound to evaluate partial solutions. The algorithm presented in this paper is an extended version of the preliminary approach presented by Blum and Blesa [4]. Extensions with respect to the preliminary approach include the use of an additional method for pruning the search space and an exhaustive experimental evaluation. We applied two configurations of our algorithm to three different sets of benchmark instances and compared the results to the best techniques from the literature. The first configuration (henceforth called *low time*) is chosen such that the algorithm is fast, whereas the second configuration (referred to as *high quality*) aims for higher solution quality. With regard to the benchmark set introduced in [4], BS (*low time*) obtains on average an improvement of 7.25% over the best known solutions, while the average improvement of BS (*high quality*) is 13.625% over the best known solutions. For the benchmark set introduced in [10], the heuristic THSB (introduced in the same paper) is the best known algorithm. THSB was also tested with *low time* and *high quality* configurations. When compared with THSB (*low time*), BS (*low time*) obtains an average improvement of 18.8% in terms of solution quality. Moreover, BS (*low time*) improves the results obtained by THBS (*high quality*) on average by 10.1%. This result is achieved using only 3.7% of the computation time required by THBS (*high quality*). On the other hand, BS (*high quality*) improves the results of THBS (*high quality*) on average by 12.7%, while requiring only half of its computation time. That is, computation time is decreased by 49.4%. With respect to the third benchmark set, we compared the ACO algorithm that is the best algorithm known for these instances. BS (*low time*) obtains an average overall improvement of 2.6% in solution quality while reducing computation time by 90.3%. On the other hand, BS (*high quality*) is able to improve the ACO results on average by 5.9%. However, this comes at the cost of spending about twice as much computation time.

This paper is organized as follows. In Section 2 we present the BS approach to the LCS problem. The experimental evaluation of the algorithms is shown in Section 3. Finally, in Section 4 we offer conclusion.

2. Beam search

BS is an incomplete derivative of branch and bound that was introduced in [24]. In the following we briefly describe the working of a standard version of BS. The central idea behind BS is to allow the extension of partial solutions in several possible ways. At each step, the algorithm chooses at most $\lfloor \mu k_{\text{bw}} \rfloor$ feasible extensions of the partial solutions stored in a set B, called the *beam*. Hereby, k_{bw} is the so-called *beam width* that limits the size of B, and $\mu \geqslant 1$ is a parameter of the algorithm. The choice of feasible extensions is done deterministically by means of a greedy function that assigns a weight to each feasible extension. At the end of each step, the algorithm creates a new beam B by selecting up to k_{bw} partial solutions from the set of chosen feasible extensions. For that purpose, BS algorithms calculate—in the case of maximization—an upper bound value for each chosen extension. Only the maximally k_{bw} best extensions—with respect to the upper bound—are chosen to constitute the new set B. Finally, the best found complete solution is returned.

Most BS applications from the literature are to scheduling problems (see, e.g., [25,13,32]). Only few applications to other types of problems exist (see, e.g., [21]). Crucial components of BS are the underlying constructive heuristic that defines the feasible extensions of partial solutions and the upper bound function for evaluating partial solutions. In the following we present our implementation of BS for the LCS problem, first focusing on these two crucial components.

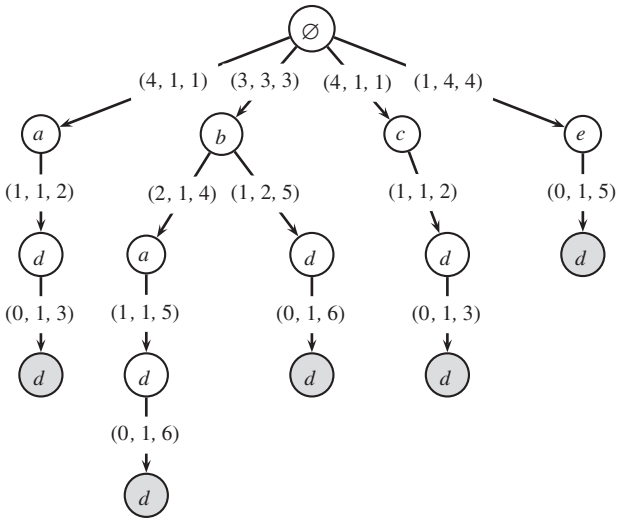

Fig. 1. Search tree defined by the construction mechanism of Best-Next, for example instance $I^{\text{ex}} = (\mathcal{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\})$, where $s_1 = bcadcdc$, $s_2 = caabadd$, and $s_3 = bacddcd$. The arc labels are explained in the text of Sections 2.1 and 2.3.

2.1. Constructive heuristic

The so-called Best-Next heuristic [12,18] is a fast heuristic for the LCS problem. Given a problem instance $(\mathcal{S}, \Sigma)$, the Best-Next heuristic produces a common subsequence t from left to right, adding at each construction step exactly one letter of the alphabet to the current subsequence. The algorithm stops when no more letters can be added, that is, when each further letter would produce an invalid solution. The pseudo-code of this heuristic is shown in Algorithm 1.

Algorithm 1.

Best-Next heuristic for the LCS problem
1: **input:** a problem instance $(\mathcal{S}, \Sigma)$
2: **initialization:** $t := \emptyset$ (where $\emptyset$ denotes the empty string)
3: **while** $|\Sigma_t^{\text{nd}}| > 0$ **do**
4: $a :=$ Choose_From(Σ_t^{nd})
5: $t := ta$
6: **end while**
7: **output:** common subsequence t

For explaining and illustrating in detail the working of Best-Next we consider the problem instance $I^{\text{ex}} = (\mathcal{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\})$, where $s_1 = bcadcdc$, $s_2 = caabadd$, and $s_3 = bacddcd$. Fig. 1 shows all possible solution constructions with respect to the construction mechanism of Best-Next in the form of a search tree. The root node of this tree contains the empty string denoted by $\emptyset$. The arc labels that are shown in the form of triples of integers will be explained in the context of the greedy functions later on. The definitions and notations that we introduce in the following assume a given partial solution t to a problem instance $(\mathcal{S}, \Sigma)$:

1. Let $s_i = x_i \cdot y_i$ be the partition of s_i into substrings x_i and y_i such that t is a subsequence of x_i, and y_i has maximal length. Given this partition, which is well defined, we keep track of *position pointers* $p_i := |x_i|$ for $i = 1, \ldots, n$. The partition into substrings as well as the

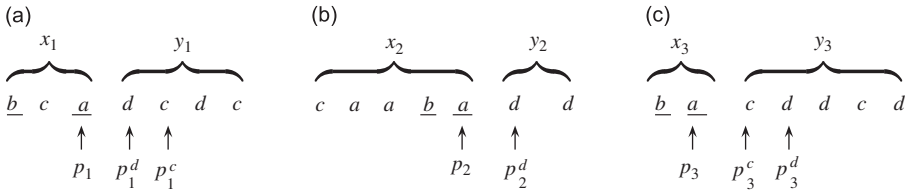

Fig. 2. In this graphic we consider instance $I^{ex} = (\mathscr{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\})$, where $s_1 = bcadcdc$, $s_2 = caabadd$, and $s_3 = bacddcd$. Moreover, the current partial solution is $t = ba$. (a), (b), and (c) show the corresponding division of s_i into x_i and y_i, as well as the setting of the position pointers p_i and the next positions of the four letters in y_i. In case a letter does not appear in y_i, the corresponding pointer is set to ∞. This is the case for letters a and b in y_1: $p_1^a := \infty$ and $p_1^b := \infty$.

corresponding position pointers is shown with respect to problem instance I^{ex} and partial solution $t = ba$ as shown in Fig. 2.

2. The *position of the first appearance* of a letter $a \in \Sigma$ in a string $s_i \in \mathscr{S}$ after the position pointer p_i is well defined and denoted by p_i^a. In case letter $a \in \Sigma$ does not appear in y_i, p_i^a is set to ∞. Again, see Fig. 2 for an example.
3. Letter $a \in \Sigma$ is called *dominated*, if there exists at least one letter $b \in \Sigma$, $a \neq b$, such that $p_i^b < p_i^a$ for $i = 1, \ldots, n$. For an example consider partial solution $t = b$ for problem instance I^{ex}. As letter a always appears before letter d in y_i ($\forall i \in \{1, 2, 3\}$), we say that a dominates d.
4. $\Sigma_t^{nd} \subseteq \Sigma$ denotes the set of non-dominated letters of the alphabet Σ with respect to t. Obviously it is required that a letter $a \in \Sigma_t^{nd}$ appears in each string s_i at least once after the position pointer p_i.

Function Choose_From(Σ_t^{nd})—see line 4 of Algorithm 1—is used to choose at each iteration exactly one letter from Σ_t^{nd}. The chosen letter is subsequently appended to t. A letter is chosen by means of a *greedy function*. In the following we present two different greedy functions that may be used. The first one—henceforth denoted by $\eta_1()$—is known from the literature (see, e.g., Fraser [12]). The second one—henceforth denoted by $\eta_2()$—is new. They are defined as follows:

$$\eta_1(a) := \min\{|s_i| - p_i^a | i = 1, \ldots, n\}, \quad \forall a \in \Sigma_t^{nd} \tag{1}$$

$$\eta_2(a) := \left(\sum_{i=1,\ldots,n} \frac{p_i^a - p_i}{|y_i|} \right)^{-1}, \quad \forall a \in \Sigma_t^{nd} \tag{2}$$

For an example, see Fig. 1. Note that the values assigned by greedy function $\eta_1()$ can be found at the first position of the triples of integers that serve as arc labels.

Function Choose_From(Σ_t^{nd}) chooses $a \in \Sigma_t^{nd}$ such that $\eta_1(a) \geqslant \eta_1(b)$ (respectively, $\eta_2(a) \geqslant \eta_2(b)$) for all $b \in \Sigma_t^{nd}$. If more that one letter fulfills this condition, the one that is lexicographically smaller is taken.

2.2. Upper bound

A second crucial element of BS is the upper bound function for evaluating partial solutions. Remember that a given common subsequence t splits each string $s_i \in \mathscr{S}$ into a first part x_i and into a second part y_i, that is, $s_i = x_i \cdot y_i$. Henceforth, $|y_i|_a$ denotes the number of occurrences of letter $a \in \Sigma$ in y_i. The upper bound value of t is defined as follows:

$$\mathrm{UB}(t) := |t| + \sum_{a \in \Sigma} \min\{|y_i|_a | i = 1, \ldots, n\} \tag{3}$$

In words, for each letter $a \in \Sigma$ the minimum number (over $i = 1, \ldots, n$) of its occurrences in y_i is taken. Summing up these minima and adding the resulting sum to the length of t result in the upper bound value. For an example, consider the partial solution $t = ba$ of the example instance shown in Fig. 2. As letters a, b, and c do not appear in the remaining part of string (y_2), they do not contribute to the upper bound value. On the other side, letter d appears at least twice in each y_i ($\forall i \in \{1, 2, 3\}$). Therefore, the upper bound value of ba is $|ba| + 2 = 2 + 2 = 4$.

Note that this upper bound function can be efficiently computed by keeping appropriate data structures. Even though the resulting upper bound values are not very tight, we will show in the section on experimental results that the bound is able to effectively guide the search process of BS.

2.3. The BS algorithm

In the following we give a technical description of the BS that we implemented. A practical example will be given afterwards. Our BS algorithm—see Algorithm 2—works roughly as follows: apart from a problem instance $(\mathscr{S}, \Sigma)$, the algorithm requires three input parameters: $k_{bw} \in \mathbb{Z}^+$ is the so-called *beam width*, $\mu \in \mathbb{Z}^+ \geqslant 1$ is a parameter that is used to determine the number of extensions that can be chosen at each step, and $\eta()$ is the particular greedy function used, either $\eta_1()$ or $\eta_2()$. Throughout the execution of the algorithm, there is also a set B of subsequences called the *beam*. At the start of the algorithm B only contains the empty string denoted by $\emptyset$, that is, $B := \{\emptyset\}$. Let C denote the set of all possible extensions of the subsequences in B.[1] At each step, the best $\lfloor \mu k_{bw} \rfloor$ extensions from C are selected with respect to the greedy function. In case a chosen extension represents a complete solution, it is stored in set B_{compl}. Otherwise, it is added to set B, in case its upper bound value UB() is greater than the length of the best-so-far solution t_{bsf}. At the end of each step, the new beam B is reduced if it contains more than k_{bw} partial solutions. This is done by evaluating the subsequences in B by means of the upper bound function UB(), and by selecting the k_{bw} subsequences with the greatest upper bound values.

Algorithm 2.

Beam search (BS) for the LCS problem
1: **input:** a problem instance $(\mathscr{S}, \Sigma)$, k_{bw}, μ, $\eta()$
2: $B_{compl} := \emptyset$, $B := \{\emptyset\}$, $t_{bsf} := \emptyset$
3: **while** $B \neq \emptyset$ **do**
4: $C :=$ Produce_Extensions(B)
5: $C :=$ Filter_Extensions(C) {this function is optional}
6: $B := \emptyset$
7: **for** $k = 1, \ldots, \min\{\lfloor \mu k_{bw} \rfloor, |C|\}$ **do**
8: $za :=$ Choose_Best_Extension($C, \eta()$)
9: $t := za$
10: **if** $\mathrm{UB}(t) = |t|$ **then**

[1] Remember that the construction mechanism of the Best-Next heuristic is based on extending a subsequence t by appending one letter from Σ_t^{nd}.

```
11:          B_compl := B_compl ∪ {t}
12:          if |t| > |t_bsf| then t_bsf := t end if
13:        else
14:          if UB(t) ⩾ |t_bsf| then B := B ∪ {t} end if
15:        end if
16:        C := C\{t}
17:      end for
18:      B := Reduce(B, k_bw)
19:    end while
20:    output: argmax{|t||t ∈ B_compl}
```

We next explain the functions of Algorithm 2 in detail. The algorithm uses four different functions. Given the current beam B as input, function Produce_Extensions(B) produces the set C of non-dominated extensions of all the subsequences in B. More specifically, C is a set of subsequences ta, where $t \in B$ and $a \in \Sigma_t^{\mathrm{nd}}$. The second function, Filter_Extensions(C), extends the non-domination relation—as defined in Section 2.1—from the extensions of one specific subsequence to the extensions of different subsequences of the same length. Formally, given two extensions $ta, zb \in C$, where $t \neq z$ but not necessarily $a \neq b$, ta dominates zb if and only if the position pointers concerning a appear before the position pointers concerning b in the corresponding remaining parts of the n strings. For an example consider subsequences $t = ba$ and $z = ad$ with respect to the example instance shown in Fig. 2. Moreover, consider in both cases the extension with letter d, that is, the considered extensions are td and zd. The position pointers in the case of td are $p_1^d = 4$, $p_2^d = 6$, and $p_3^d = 4$, whereas in the case of zd the position pointers are $p_1^d = 6$, $p_2^d = 7$, and $p_3^d = 5$. Therefore, td dominates zd.

The third function, Choose_Best_Extension($C, \eta()$), is used for choosing extensions from C. This function requires one of the two greedy functions outlined before as a parameter. Note that for the comparison of two extensions ta and zb from C the greedy function is only useful in case $t = z$, while it might be misleading in case $t \neq z$. We solved this problem as follows. First, instead of the weights assigned by a greedy function, we use the corresponding ranks. More specifically, given all extensions $\{ta | a \in \Sigma_t^{\mathrm{nd}}\}$ of a subsequence t, the extension tb with $\eta(tb) \geqslant \eta(ta)$ for all $a \in \Sigma_t^{\mathrm{nd}}$ receives rank 1, denoted by $r(tb) = 1$. The extension with the second highest greedy weight receives rank 2, etc. Note that the notation $\eta(a)$, as introduced in Section 2.1, is extended here to the notation $\eta(ta)$. For an example, see the search tree shown in Fig. 1. The second integer in the triples that serve as arc labels shows the rank that corresponds to the weight assigned by greedy function $\eta_1()$.

For evaluating an extension ta we use the sum of the ranks of the greedy weights that correspond to the construction steps performed to construct subsequence ta, that is

$$v(ta) := r(t_1) + \left(\sum_{i=1}^{|t|-1} r(t_1 \cdots t_i t_{i+1}) \right) + r(ta) \tag{4}$$

where $t_1 \cdots t_i$ denotes the substring of t from position 1 to position i, and t_{i+1} denotes the letter at position $i+1$ of subsequence t. In contrast to the greedy function weights, these newly defined $v()$ values can be used to compare the extensions of different subsequences. In fact, a call of function Choose_Best_Extension($C, \eta()$) returns the extension from C with maximal $v()^{-1}$ value. For illustrating this concept we again refer to Fig. 1. The third integer in the triples that serve as arc labels represents the value $v()$ defined above. For example, the value of $v(td)$, where $t = ba$, is 5 because $r(b)$ is 3, the rank of the extension ba is 1, and the rank of the extension td is again 1.

Finally, the last function used by the BS algorithm is Reduce(B, k_{bw}). In case $|B| > k_{\mathrm{bw}}$, this function removes from B step-by-step those subsequences t that have an upper bound value UB(t) smaller or equal to the upper bound value of all the other subsequences in B. The removal process stops once $|B| = k_{\mathrm{bw}}$.

2.4. Example for the working of BS

In the following we outline the steps of BS when applied to example instance I^{ex} (see Fig. 2). We use greedy function $\eta_1()$, a beam width of 2 (that is, $k_{\mathrm{bw}} = 2$), and $\mu = 1.5$. This means that, at each step, we are allowed to choose maximally $2 \times 1.5 = 3$ extensions of the subsequences that are in the beam B. The search tree that is shown in Fig. 1 shows that solution *badd* is the unique optimal solution of instance I^{ex}. Note that the result of applying Best-Next to I^{ex} is *add*, which is a suboptimal solution.

BS starts with a beam that only contains the empty string, that is, $B = \{\emptyset\}$. In the first step we choose the three extensions of the empty string with minimal rank-sum. Remember that the rank-sums are shown in the third position of the triples that serve as arc labels in the search tree. Accordingly, the chosen extensions are a, b and c. As the beam width is two, we have to discard one of the chosen extensions. For that purpose the upper bound value is computed for all three extensions. It can be easily verified that UB(a) = 3, UB(b) = 4, and UB(c) = 3. Given these values, extension c is discarded because its upper bound value is smaller than the upper bound value of b, and because c is lexicographically greater than a. Therefore, after the first step, it holds that $B = \{a, b\}$. The possible extensions of b are ba and bd. However, as explained before, ba dominates bd. Therefore, bd is not considered. We neither consider extension ad, since it is also dominated by ba. Therefore, the only extension chosen in the second step is ba, that is, $B = \{ba\}$. In the third and fourth steps of BS, the only possible extensions are chosen and the algorithm provides as output the optimal solution *badd*.

3. Experiments

We implemented our algorithm in ANSI C++ and compiled the software with GCC 4.1.2 in GNU/Linux 2.6.20. Experiments were run on an Intel Core2 1.66 GHz with 2 MB of cache size.

First, we conducted extensive tuning experiments for finding appropriate parameter settings. Remember that BS has three parameters. The beam width parameter k_{bw} denotes the number of subsequences that BS keeps in the beam B at each step. In general, larger values of k_{bw} produce better results at the cost of higher computation times. The second parameter, μ, determines the number of extensions that are chosen at each step of the algorithm. Low values of μ make the search rely completely on the greedy function $\eta()$, while high values of μ allow solutions with lower heuristic values and, at the same time, with greater upper bound values to be chosen for the beam of the next step. Finally, $\eta()$ may correspond to greedy function $\eta_1()$ as defined in Eq. (1), or $\eta_2()$ as defined in Eq. (2). For space reasons we decided not to present the tuning results. Instead, we refer the interested reader to [5]. The chosen parameter settings will be indicated for each benchmark set.

3.1. Benchmark instances

We considered three different sets of instances for our experiments. The first set, henceforth denoted by BB, was introduced by Blum and Blesa in [4] and generated by the following procedure. First, for each combination of an alphabet Σ, a number of strings n and a length l, a *base string* of length l was created by randomly generating letters from Σ. Then, each of the n strings was produced by traversing the base string and deleting each letter with a probability of 0.1. Ten instances were generated for each of the 80 combinations of $|\Sigma| \in \{2, 4, 8, 24\}$, $n \in \{10, 100\}$ and $l \in \{100, 200, 300, \ldots, 1000\}$.

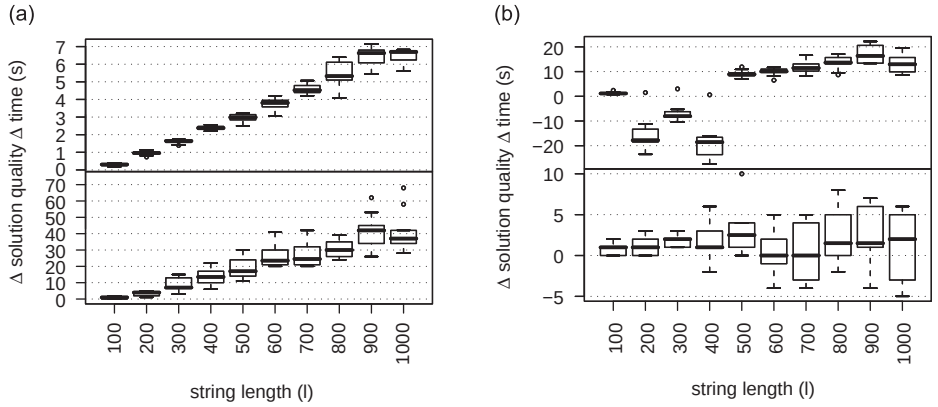

Fig. 3. Benefits of using function Filter_Extensions(C) in BS. (a) Instances with $n = 10$ and (b) instances with $n = 100$.

In addition, we applied BS to two sets of benchmark instances from the literature. Set ES was introduced by Easton and Singireddy [10]. It is composed of 50 instances per combination of $|\Sigma|$, n, and l. These instances were created by sequentially generating each letter with a probability of $1/|\Sigma|$. A third set of instances, denoted by ST, stems from Shyu and Tsai [28]. Their instances are biologically inspired, and thus they consider an alphabet of size $|\Sigma| = 4$, corresponding to DNA sequences, or of size $|\Sigma| = 20$, corresponding to protein sequences. Shyu and Tsai studied three different types of instances. One was randomly generated, presumably in the same way as Easton and Singireddy's [10] instances. The other two sets consist of real DNA and protein sequences of rats and viruses.

3.2. Benefits of filtering

As a first step, we wanted to study the effect of using the function Filter_Children() in Algorithm 2. For this purpose we applied the *high quality* configuration of BS[2] with and without the use of this function to all the problem instances of set BB. As an illustration, we show the results obtained for all instances with $|\Sigma| = 2$ in Fig. 3. Fig. 3(a) presents the results for all instances with 10 strings ($n = 10$), while Fig. 3(b) shows the corresponding results for all instances with 100 strings. The results are shown in terms of the difference (Δ) between the algorithm using filtering and the algorithm that is not using filtering. This means that positive values in terms of solution quality indicate an advantage for the algorithm using filtering. However, positive values in terms of computation time indicate that the algorithm using filtering is slower.

The use of filtering typically produces better quality solutions for instances with a small number of strings ($n = 10$), as shown graphically in the lower part of Fig. 3(a). Interestingly, the longer the length of the strings, the greater is the advantage of the algorithm using filtering. In the same way, the computation time overhead of using filtering increases with the length of the strings, as shown in the upper part of Fig. 3(a). On the other hand, for large number of strings ($n = 100$), filtering may actually reduce sometimes the computation time, as shown in Fig. 3(b). For instances with $n = 100$, however, filtering is not always beneficial in terms of solution quality. Nonetheless, the median difference (as denoted by the line within each box) is positive most of the times. Thus, on average filtering does improve solution quality. We conclude from these results that the use of filtering generally pays off in terms of solution quality, while incurring in a typically small computation time overhead. Therefore, filtering will be used in BS for all further experiments.

3.3. Experimental results for set BB

We selected two configurations of BS with respect to the results of a previous experimental analysis of parameters [5]. The first one (referred to as *low time*) aims at short computation times, while the second configuration (*high quality*) aims at high solution quality. The *low time* configuration corresponds to parameters $\eta_1()$, $k_{bw} = 10$, and $\mu = 1.5$ if $|\Sigma| = 2$, and $\mu = 3$ otherwise. The *high quality* configuration is the same but using a larger beam width of $k_{bw} = 100$. The results of the two configurations are compared in Table 1 with the results obtained by the classical Expansion and Best-Next algorithms. The table columns with heading $|t^*|$ present the solution quality, whereas the columns with heading "Time" show the computation time in seconds. Each value in these columns corresponds to the mean and, in parentheses, the standard deviation of the results for 10 instances with the same characteristics. Finally, the table column with heading Δ presents the improvement of BS with respect to the best result obtained between Expansion and Best-Next.[3] This measure is given in % and is computed as $(100 \cdot X/Y) - 100$, where X is the result obtained by BS, and Y is the best result between Expansion and Best-Next.

The results are clearly in favour of BS. First, the best results are always obtained by the *high quality* configuration of BS, which is even noticeably faster than Expansion. Therefore, this configuration of BS outperforms Expansion in all aspects. The average improvement of BS (*high quality*) over the best results obtained between Expansion and Best-Next is 13.625%. Second, although best-next is evidently the fastest approach, the quality of its solutions is very poor compared to the results of BS (*low time*). Note that even BS (*low time*) achieves an average improvement of 7.25% over the best results obtained between Expansion and Best-Next.

3.4. Experimental results for set ES

For the application to the benchmark instances proposed in [10], we also selected a *low time* and a *high quality* configuration for BS. The configuration using $k_{bw} = 10$, $\eta_2()$ and $\mu = 1.5$ gives good solutions for all instances in a relatively short time. On the other hand, configurations $\eta_2()$, k_{bw}=100 and μ=1.5 for $|\Sigma|$=2, and $\eta_2()$, k_{bw}=50,

[2] For the specification of this *high quality* configuration, see Section 3.3.

[3] Note that a *negative improvement* corresponds to a decrease in performance in comparison to the competitors.

Table 1
Comparison of BS with EXPANSION and BEST-NEXT for instances of set BB [4].

Instance			Expansion		BEST-NEXT		BS (low time)			BS (high quality)		
$\|\Sigma\|$	n	l	$\|t^*\|$	Time	$\|t^*\|$	Time	$\|t^*\|$	Δ (%)	Time	$\|t^*\|$	Δ (%)	Time
2	10	1000	515.4 (16.2)	43.9 (3.3)	556.9 (14.4)	0.0 (0.0)	613.2 (14.6)	10.1	0.6 (0.0)	648.0 (15.0)	16.4	13.6 (0.6)
	100	1000	476.7 (5.5)	932.9 (64.7)	503.3 (6.8)	0.1 (0.0)	531.6 (6.1)	5.6	6.3 (0.2)	541.0 (8.0)	7.5	72.5 (3.9)
4	10	1000	484.3 (18.3)	2007.5 (276.5)	382.7 (16.1)	0.0 (0.0)	477.3 (15.7)	−1.4	0.9 (0.0)	534.7 (12.6)	10.4	18.1 (0.5)
	100	1000	266.0 (5.4)	3082.6 (135.8)	319.3 (4.9)	0.1 (0.0)	350.7 (10.1)	9.8	9.3 (0.2)	369.3 (4.6)	15.7	121.6 (2.2)
8	10	1000	436.2 (18.0)	1525.9 (195.0)	293.6 (15.6)	0.0 (0.0)	420.0 (27.0)	−3.7	0.7 (0.0)	462.3 (12.9)	6.0	21.2 (0.6)
	100	1000	159.4 (9.3)	2633.1 (48.6)	208.1 (6.4)	0.1 (0.0)	241.5(4.5)	16.0	10.6 (0.3)	258.7 (4.7)	24.3	154.7 (2.0)
24	10	1000	374.9 (8.2)	794.6 (75.4)	229.2 (25.3)	0.0 (0.0)	382.6 (10.0)	2.1	1.3 (0.0)	385.6 (7.0)	2.9	37.4 (1.4)
	100	1000	64.2 (8.4)	2717.3 (80.4)	117.4 (4.4)	0.2 (0.0)	140.3 (5.7)	19.5	13.5 (0.2)	147.7 (4.6)	25.8	268.3 (8.0)

BS (low time): $\eta_1()$, $k_{bw} = 10$, and $\mu = 1.5$ for $|\Sigma| = 2$ or $\mu = 3$ for $|\Sigma| > 2$.
BS (high quality): $\eta_1()$, $k_{bw} = 100$, and $\mu = 1.5$ for $|\Sigma| = 2$ or $\mu = 3$ for $|\Sigma| > 2$.

Table 2
Comparison of BS with G&V and THSB for instances of set ES [10].

Instance			G&V		THSB (low time)		THSB (high quality)		BS (low time)			BS (high quality)		
$\|\Sigma\|$	n	l	$\|t^*\|$	Time	$\|t^*\|$	Time	$\|t^*\|$	Time	$\|t^*\|$	Δ (%)	Time	$\|t^*\|$	Δ (%)	Time
2	10	1000	562.8	0.0	562.0	0.4	577.2	24.5	579.9 (4.8)	0.5	0.7 (0.0)	592.6 (4.2)	2.7	14.8 (0.3)
	50	1000	503.7	0.1	506.1	1.0	511.3	85.1	516.3 (1.9)	1.0	3.7 (0.1)	521.9 (1.8)	2.1	43.5 (0.4)
	100	1000	489.6	0.1	493.2	2.2	497.9	196.2	502.1 (1.8)	0.8	7.4 (0.1)	506.0 (1.8)	1.6	78.6 (5.0)
10	10	1000	153.4	0.0	156.7	2.0	162.5	90.6	185.5 (2.6)	14.2	0.5 (0.0)	192.2 (2.0)	18.3	9.4 (0.2)
	50	1000	105.4	0.1	107.6	0.8	109.8	69.6	127.9 (1.2)	16.5	1.5 (0.0)	129.6 (1.1)	18.0	18.8 (0.3)
	100	1000	96.6	0.2	98.7	1.1	100.7	58.6	116.5 (0.8)	15.5	2.7 (0.1)	117.9 (0.9)	17.1	30.6 (0.4)
25	10	2500	183.6	0.1	173.8	8.8	188.9	102.2	214.3(2.2)	13.4	2.7 (0.1)	224.3 (1.9)	18.7	51.5 (0.8)
	50	2500	112.7	0.2	106.8	2.2	115.3	52.4	131.3 (0.9)	13.9	5.5 (0.1)	133.0 (0.8)	15.4	76.6 (1.2)
	100	2500	101.5	0.4	97.7	2.5	104.1	81.1	116.3 (0.9)	11.7	9.1 (0.1)	118.1 (0.8)	13.4	118.6 (3.7)
100	10	5000	113.6	0.2	92.2	36.0	117.8	6,099.3	132.5 (1.7)	12.5	19.1 (0.3)	139.6 (1.4)	18.5	394.6 (7.0)
	50	5000	58.4	0.7	52.2	185.7	60.9	4,273.0	67.9 (0.5)	11.5	27.8 (0.5)	69.5 (0.6)	14.1	490.2 (10.7)
	100	5000	50.4	1.3	46.5	353.7	52.7	11,128.3	57.6 (0.6)	9.3	42.2 (0.8)	59.0 (0.3)	12.0	602.0 (8.8)

Results for G&V and THSB are taken from [10].
BS (low time): $\eta_2()$, $k_{bw} = 10$, $\mu = 1.5$.
BS (high quality): $\eta_2()$, $k_{bw} = 100$, $\mu = 1.5$ for $|\Sigma| = 2$; $\eta_2()$, $k_{bw} = 50$, $\mu = 3$ for $|\Sigma| > 2$.

$\mu = 3$ for $|\Sigma| > 2$, give the best solution quality but require considerably longer computation times. Interestingly, the parameter settings determined by tuning BS for these instances (see [5]) are somewhat different from the ones determined for benchmark set BB. Most noticeably, the greedy function $\eta_2()$ performs clearly better than $\eta_1()$, specially for large n (number of strings). These differences indicate that the methodology used to generate the instances can have an influence on the performance of the greedy functions.

Table 2 presents—both for *high quality* and *low time* configurations—a comparison of BS with the results of THSB and G&V. Note that THSB is currently the best available algorithm for benchmark set ES. The results of THSB and G&V, obtained on a 1.5 GHz Pentium IV that should be slightly slower than our machine, are taken from [10]. The table columns with heading $|t^*|$ provide the solution quality, whereas the columns with heading "Time" show the computation time in seconds. Each value in these columns corresponds to the mean of the results for 50 instances with the same characteristics. In the case of BS we additionally show the standard deviation in parentheses. Finally, the table column with heading Δ presents the improvement of BS with respect to the results obtained by THSB (*high quality*). This measure is given in % and is computed as $(100 \cdot X/Y) - 100$, where X is the result obtained by BS, and Y is the result of THSB (*high quality*).

The results presented in Table 2 show that the *low time* configuration of BS dominates the *high quality* configuration of THSB in all aspects. That is, the *low time* configuration of BS is always able to obtain better solutions in a much shorter time. As for the *low time* configuration of THSB, it is typically faster than BS; however, the resulting sequences are significantly worse than those obtained by BS. More in detail, when compared with THSB (*low time*), BS (*low time*) obtains an average improvement of 18.8% in terms of solution quality. Moreover, BS (*low time*) improves the results obtained by THBS (*high quality*) on average by 10.1%, while reducing the computation time requirements by 96.3%. That is, BS (*low time*) needs, on average, only 3.7% of the computation time required by THBS (*high quality*). On the other hand, BS (*high quality*) improves the results of THBS (*high quality*) on average by 12.7% while requiring only half of its computation time, that is, computation time is reduced by 49.4%.

Finally, note that G&V is definitely the fastest of the compared algorithms, specially for the largest instances, but at the cost of a significant decrease in solution quality with respect to BS.

3.5. Experimental results for set ST

Shyu and Tsai [28] presented an ACO algorithm for the LCS problem. ACO [9] is a metaheuristic inspired by the foraging behaviour of ant colonies. At the core of this behaviour is the indirect communication between the ants by means of chemical pheromone trails, which enables them to find short paths between their nest and food sources. This characteristic of real ant colonies is exploited in ACO

Table 3
Comparison of BS with ACO for Random instances [28].

Instance (Random)		ACO		BS (low time)			BS (high quality)		
$\|\Sigma\|$	n	$\|t^*\|$	Time	$\|t^*\|$	Δ (%)	Time	$\|t^*\|$	Δ (%)	Time
4	10	197.2 (2.0)	10.7 (2.0)	200	1.4	0.3	211	7.0	9.8
	15	185.2 (1.3)	15.7 (5.4)	190	2.6	0.5	194	4.8	13.2
	20	176.2 (1.3)	11.4 (0.8)	178	1.0	0.7	184	4.4	14.9
	25	172.2 (0.7)	15.4 (1.7)	174	1.0	0.9	179	3.9	15.8
	40	161.4 (1.3)	23.8 (10.3)	162	0.4	1.4	167	3.5	21.0
	60	155.4 (1.3)	24.7 (3.2)	157	1.0	2.1	161	3.6	27.6
	80	151.6 (0.8)	32.5 (5.9)	151	−0.4	2.7	156	2.9	33.5
	100	148.8 (1.3)	43.6 (10.4)	150	0.8	3.5	154	3.5	40.3
	150	143.4 (0.8)	57.2 (17.1)	146	1.8	5.0	148	3.2	56.4
	200	141.0 (0.6)	59.1 (9.6)	144	2.1	6.9	146	3.5	74.3
20	10	54.0 (1.1)	7.4 (2.1)	58	7.4	0.7	61	13.0	33.3
	15	46.2 (1.6)	9.3 (2.5)	49	6.1	0.9	51	10.4	37.6
	20	42.4 (1.3)	11.4 (4.9)	43	1.4	1.1	47	10.8	39.5
	25	40.0 (1.1)	10.5 (2.3)	41	2.5	1.3	43	7.5	39.5
	40	34.2 (0.7)	14.1 (4.8)	37	8.2	1.7	37	8.2	43.2
	60	30.6 (0.8)	17.3 (1.3)	34	11.1	2.6	34	11.1	46.5
	80	29.0 (1.1)	22.9 (3.0)	32	10.3	3.2	32	10.3	53.2
	100	28.4 (0.8)	25.6(0.1)	30	5.6	3.9	31	9.2	59.2
	150	26.0 (0.4)	40.8 (7.4)	28	7.7	5.7	29	11.5	75.6
	200	25.0 (0.2)	55.4 (4.7)	27	8.0	7.9	27	8.0	98.0

BS (low time): $\eta_2()$, $k_{bw} = 10$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 10$, $\mu = 5$ for $|\Sigma| = 20$.
BS (high quality): $\eta_2()$, $k_{bw} = 100$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 100$, $\mu = 5$ for $|\Sigma| = 20$.

Table 4
Comparison of BS with ACO for Rat instances [28].

Instance (Rat)		ACO		BS (low time)			BS (high quality)		
$\|\Sigma\|$	n	$\|t^*\|$	Time	$\|t^*\|$	Δ (%)	Time	$\|t^*\|$	Δ (%)	Time
4	10	182.0 (2.4)	7.4 (1.9)	189	3.8	0.3	191	4.9	9.7
	15	166.6 (1.3)	10.5 (2.4)	163	−2.2	0.4	173	3.8	12.3
	20	160.0 (1.3)	12.5 (3.8)	160	0.0	0.6	163	1.9	12.6
	25	155.8 (1.3)	15.9 (4.0)	160	2.7	0.8	162	4.0	15.8
	40	143.4 (0.8)	21.0 (4.6)	142	−1.0	1.2	146	1.8	19.4
	60	142.4 (1.7)	26.2 (8.9)	143	0.4	1.9	144	1.1	26.7
	80	128.8 (0.7)	29.9 (4.9)	131	1.7	2.3	135	4.8	31.8
	100	124.6 (2.0)	48.8 (17.9)	129	3.5	3.0	132	5.9	38.5
	150	115.6 (1.3)	35.0 (6.8)	120	3.8	4.2	121	4.7	51.1
	200	114.6 (2.3)	65.5 (14.0)	117	2.1	5.6	121	5.6	69.1
20	10	63.4 (1.3)	9.2 (2.5)	65	2.5	0.7	69	8.8	27.4
	15	56.6 (0.8)	8.9 (2.4)	57	0.7	1.1	60	6.0	36.7
	20	47.8 (0.7)	14.6 (5.8)	50	4.6	1.2	51	6.7	34.4
	25	46.2 (1.3)	11.5 (1.2)	49	6.1	1.4	51	10.4	39.0
	40	44.2 (1.3)	14.6 (3.2)	46	4.1	2.0	49	10.9	47.4
	60	43.0 (0.4)	31.5 (6.9)	44	2.3	3.2	46	7.0	60.3
	80	39.6 (0.8)	32.4 (10.1)	42	6.1	4.0	43	8.6	64.4
	100	37.0 (1.1)	42.4 (8.8)	37	0.0	4.5	38	2.7	64.8
	150	34.0 (1.1)	49.1 (8.1)	35	2.9	6.7	36	5.9	77.8
	200	32.4 (1.3)	75.7 (17.0)	31	−4.3	8.3	33	1.9	101.0

BS (low time): $\eta_2()$, $k_{bw} = 10$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 10$, $\mu = 5$ for $|\Sigma| = 20$.
BS (high quality): $\eta_2()$, $k_{bw} = 100$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 100$, $\mu = 5$ for $|\Sigma| = 20$.

algorithms for solving, for example, discrete optimization problems. In general, ACO attempts to solve an optimization problem by iterating the following two steps: (1) candidate solutions are constructed using a pheromone model, that is, a parameterized probability distribution over the solution space; and (2) the candidate solutions are used to modify the pheromone values in a way that is deemed to bias future sampling towards high quality solutions.

First, we conducted tuning experiments with BS for the application to benchmark set ST instroduced in [28]. According to these results [5], we choose $\eta_2()$, $k_{bw} = 100$, with $\mu = 3$ for $|\Sigma| = 4$ and $\mu = 5$ for $|\Sigma| = 20$ for the *high quality* configuration of BS, whereas the *low time* configuration uses $\eta_2()$, $k_{bw} = 10$, with $\mu = 3$ for $|\Sigma| = 4$ and $\mu = 5$ for $|\Sigma| = 20$. We then compared the two configurations of BS with the results of ACO obtained from [28]. ACO has been implemented in C++ and the experiments were run on a AMD Athlon 2100+ CPU, which should be slightly faster than our machine. Tables 3–5 show this comparison for Random, Rat and Virus instances, respectively. In each table, the columns with heading $|t^*|$ contain the solution quality and the columns with heading "Time" present the computation time in seconds. Results for ACO show the mean and, in parentheses, the standard deviation of 10 independent runs for a single instance. Since BS is deterministic, and each result shown is the one obtained for a single instance. Finally, the table columns with heading Δ present the improvement (respectively, the

Table 5
Comparison of BS with ACO for Virus instances [28].

Instance (Virus)		ACO		BS (low time)			BS (high quality)		
$\|\Sigma\|$	n	$\|t^*\|$	Time	$\|t^*\|$	Δ (%)	Time	$\|t^*\|$	Δ (%)	Time
4	10	197.6 (1.3)	3.7 (0.7)	203	2.7	0.4	212	7.3	11.6
	15	183.6 (1.3)	7.9 (2.0)	192	4.6	0.5	193	5.1	15.4
	20	173.8 (2.5)	20.4 (6.5)	179	3.0	0.7	181	4.1	17.2
	25	179.0 (1.8)	18.3 (5.3)	178	−0.6	0.9	185	3.4	17.9
	40	155.0 (2.1)	20.5 (3.4)	158	1.9	1.3	162	4.5	21.9
	60	150.6 (1.3)	30.8 (9.3)	153	1.6	2.0	158	4.9	29.1
	80	145.8 (1.3)	45.5 (6.9)	148	1.5	2.6	153	4.9	36.0
	100	143.4 (2.7)	23.8 (10.3)	149	3.9	3.4	150	4.6	43.9
	150	141.6 (0.8)	50.0 (21.3)	143	1.0	5.0	148	4.5	64.5
	200	140.6 (1.3)	65.6 (15.6)	143	1.7	6.8	145	3.1	84.5
20	10	65.6 (0.8)	3.5 (1.2)	67	2.1	0.7	75	14.3	27.2
	15	55.8 (1.3)	10.4 (1.6)	58	3.9	1.0	63	12.9	38.6
	20	53.6 (1.3)	10.8 (0.5)	55	2.6	1.2	57	6.3	40.3
	25	49.6 (0.8)	13.2 (4.5)	50	0.8	1.4	53	6.9	38.9
	40	46.4 (0.8)	17.1 (2.6)	47	1.3	2.1	49	5.6	48.4
	60	43.4 (0.8)	27.7 (4.2)	44	1.4	3.1	45	3.7	56.1
	80	43.0 (0.4)	38.1 (11.4)	43	0.0	4.0	44	2.3	67.4
	100	42.0 (1.1)	23.4 (5.1)	41	−2.4	5.0	43	2.4	74.2
	150	42.6 (0.8)	71.4 (19.8)	43	0.9	7.8	44	3.3	108.0
	200	41.0 (0.2)	78.9 (21.7)	43	4.9	11.0	43	4.9	140.0

BS (low time): $\eta_2()$, $k_{bw} = 10$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 10$, $\mu = 5$ for $|\Sigma| = 20$.
BS (high quality): $\eta_2()$, $k_{bw} = 100$, $\mu = 3$ for $|\Sigma| = 4$; $\eta_2()$, $k_{bw} = 100$, $\mu = 5$ for $|\Sigma| = 20$.

decrease in performance) of BS with respect to the results obtained by ACO. This measure is given in % and is computed as $(100 \cdot X/Y) - 100$, where X is the result obtained by BS, and Y is the result of ACO.

For the instances with alphabet size $|\Sigma| = 4$, the *low time* configuration of BS finds slightly better solutions than ACO in a much shorter time, specially for large values of n. On the other hand, the *high quality* configuration of BS clearly outperforms ACO with respect to solution quality while it requires approximately the same amount of computation time. As for alphabet size $|\Sigma| = 20$, BS (*low time*) matches, and often improves over the solutions generated by ACO, while requiring a small fraction of the computation time used by ACO. However, in this case, BS (*high quality*) is clearly slower than ACO. Nonetheless, BS (*high quality*) outperforms both ACO and BS (*low time*). The difference is particularly large for instances with few strings, that is, a small value of n. In summary, BS (*low time*) obtains an average overall improvement of 2.6% in solution quality while reducing computation time by 90.3%. On the other hand, BS (*high quality*) is able to improve the ACO results on average by 5.9% with the drawback of using about twice as much computation time.

4. Conclusions

In this paper we proposed a beam search (BS) algorithm for the LCS problem. The proposed BS was empirically tested on three different sets of LCS benchmark instances. As expected, the use of a larger beam width improves the quality of the solutions obtained by BS, although it also increases the computation time required to reach a certain solution quality. Therefore, we selected for each set of instances two configurations of parameters for BS. The first one is characterized by low computation time requirements, while the second one rather aims at producing high quality solution. These *low time* and *high quality* configurations were compared against the best approaches from the literature for the respective benchmark sets. The results showed that BS outperforms the expansion algorithm as well as the THSB heuristic in solution quality as well as in computation time requirements. As for the comparison to an ant colony optimization (ACO) algorithm, the *low time* configuration of BS produces equally good solutions in a much shorter time, specially for high number of strings, whereas the *high quality* configuration of BS consistently finds better solutions than ACO at the cost of higher running times.

In summary, our experimental analysis showed that the proposed BS algorithm is currently a state-of-the-art method for solving the LCS problem.

Acknowledgements

This work was supported by Grants TIN2007-66523 (FORMALISM), TIN2005-09198 (ASCE), and TIN2005-25859 (AEOLUS) of the Spanish government. In addition, Christian Blum acknowledges the support from the *Ramón y Cajal* program of the Spanish Ministry of Science and Technology of which he is a research fellow.

Moreover, we would like to thank T. Easton, A. Singireddy, S.J. Shyu, and C.-Y. Tsai for providing their benchmark instances, and the anonymous referees for their careful revision of this manuscript.

References

[1] Aho A, Hopcroft J, Ullman J. Data structures and algorithms. Reading, MA: Addison-Wesley; 1983.
[2] Bergroth L, Hakonen H, Raita T. New approximation algorithms for longest common subsequences. In: Proceedings of string processing and information retrieval: a South American symposium. 1998. p. 32–40.
[3] Bergroth L, Hakonen H, Raita T. A survey of longest common subsequence algorithms. In: Proceedings of SPIRE 2000—7th international symposium on string processing and information retrieval. IEEE Press; 2000. p. 39–48.
[4] Blum C, Blesa M. Probabilistic beam search for the longest common subsequence problem. In: Stützle T, Birattari M, Hoos HH, editors. Proceedings of SLS 2007—engineering stochastic local search algorithms. Lecture notes in computer science, vol. 4638. Berlin, Germany: Springer; 2007. p. 150–61.
[5] Blum C, Blesa MJ, López-Ibáñez M. Beam search for the longest common subsequence problem. Technical Report LSI-08-29, Department LSI, Univeristat Politècnica de Catalunya, 2008.
[6] Bonizzoni P, Della Vedova G, Mauri G. Experimenting an approximation algorithm for the LCS. Discrete Applied Mathematics 2001;110(1):13–24.
[7] Brisk P, Kaplan A, Sarrafzadeh M. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In: Proceedings of the 41st design automation conference. IEEE Press; 2004. p. 395–400.
[8] Chin F, Poon CK. Performance analysis of some simple heuristics for computing longest common subsequences. Algorithmica 1994;12(4–5):293–311.
[9] Dorigo M, Stützle T. Ant colony optimization. Cambridge, MA: MIT Press; 2004.

[10] Easton T, Singireddy A. A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics 2008;14(3):271–83.
[11] Easton T, Singireddy A. A specialized branching and fathoming technique for the longest common subsequence problem. International Journal of Operations Research 2007;4(2):98–104.
[12] Fraser CB. Subsequences and supersequences of strings. PhD thesis, University of Glasgow; 1995.
[13] Ghirardi M, Potts CN. Makespan minimization for scheduling unrelated parallel machines: a recovering beam search approach. European Journal of Operational Research 2005;165(2):457–67.
[14] Guenoche A. Supersequence of masks for oligo-chips. Journal of Bioinformatics and Computational Biology 2004;2(3):459–69.
[15] Guenoche A, Vitte P. Longest common subsequence with many strings: exact and approximate methods. Technique et Science Informatiques 1995;14(7):897–915 [in French].
[16] Gusfield D. Algorithms on strings, trees, and sequences. Computer science and computational biology. Cambridge: Cambridge University Press; 1997.
[17] Hsu WJ, Du MW. Computing a longest common subsequence for a set of strings. BIT Numerical Mathematics 1984;24(1):45–59.
[18] Huang K, Yang C, Tseng K. Fast algorithms for finding the common subsequences of multiple sequences. In: Proceedings of the international computer symposium. IEEE Press; 2004. p. 1006–11.
[19] Jiang T, Lin G, Ma B, Zhang K. A general edit distance between RNA structures. Journal of Computational Biology 2002;9(2):371–88.
[20] Tao J, Li M. On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 1995;24(5):1122–39.
[21] Lee G-C, Woodruff DL. Beam search for peak alignment of NMR signals. Analytica Chimica Acta 2004;513(2):413–6.
[22] Lu SY, Fu KS. A sentence-to-sentence clustering procedure for pattern analysis. IEEE Transactions on Systems, Man and Cybernetics 1978;8(5):381–9.
[23] Maier D. The complexity of some problems on subsequences and supersequences. Journal of the ACM 1978;25:322–36.
[24] Ow PS, Morton TE. Filtered beam search in scheduling. International Journal of Production Research 1988;26:297–307.
[25] Sabuncuoglu I, Bayiz M. Job shop scheduling with beam search. European Journal of Operational Research 1999;118(2):390–412.
[26] Sankoff D, Kruskal JB. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. Reading, UK: Addison-Wesley; 1983.
[27] Sellis T. Multiple query optimization. ACM Transactions on Database Systems 1988;13(1):23–52.
[28] Shyu SJ, Tsai C-Y. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers & Operations Research 2009;36(1):73–91.
[29] Singireddy A. Solving the longest common subsequence problem in bioinformatics. Master's thesis, Industrial and Manufacturing Systems Engineering, Kansas State University, Manhattan, KS; 2007.
[30] Smith T, Waterman M. Identification of common molecular subsequences. Journal of Molecular Biology 1981;147(1):195–7.
[31] Storer J. Data compression: methods and theory. Maryland: Computer Science Press; 1988.
[32] Valente JMS, Alves RAFS. Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. Computers & Industrial Engineering 2005;48(2):363–75.