

Probabilistic Beam Search for the Longest Common Subsequence Problem*

Christian Blum and Maria J. Blesa

ALBCOM, Dept. Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya, Barcelona, Spain
{cblum,mjblesa}@lsi.upc.edu

Abstract. Finding the common part of a set of strings has many important applications, for example, in pattern recognition or computational biology. In computer science, this problem is known as the longest common subsequence problem. In this work we present a probabilistic beam search approach to solve this classical problem. To our knowledge, this algorithm is the first stochastic local search algorithm proposed for this problem. The results show the great potential of our algorithm when compared to existing heuristic methods.

1 Introduction

The longest common subsequence (LCS) problem is one of the classical string problems. Given a problem instance $(\mathcal{S}, \Sigma)$, where $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ is a set of n strings over a finite alphabet Σ , the problem consists in finding a longest string t^* that is a subsequence of all the strings in $\mathcal{S}$. Such a string t^* is called a *longest common subsequence* of the strings in $\mathcal{S}$. Note that a string t is called a subsequence of a string s , if t can be produced from s by deleting characters. For example, *dga* is a subsequence of *adagtta*. The LCS problem is in general NP-hard [1]. In case $n = 2$ the problem is polynomially solvable, for example, by dynamic programming [2]. Traditional applications of this problem are in data compression, syntactic pattern recognition, and file comparison [3], whereas more recent applications also include computational biology [4].

To our knowledge, stochastic local search methods have not been applied so far to the LCS problem. Existing approximate methods fall into the class of deterministic heuristics. Examples are simple heuristics based on sequential solution construction (numerous references can be found in [5]), or more sophisticated methods such as the so-called expansion algorithm [6]. The relative performance of these heuristics depends on the characteristics of the problem instances. In this work we propose a so-called probabilistic beam search (PBS) approach to tackle the LCS problem. PBS is a stochastic local search algorithm based on solution construction and lower bounding techniques. It is a probabilistic version of

* This work was supported by grant TIN-2005-08818-C04-01 (OPLINK) of the Spanish government, and by the *Ramón y Cajal* program of the Spanish Ministry of Science and Technology of which Christian Blum is a research fellow.

deterministic beam search, which is an incomplete derivative of branch & bound. In [7] we proposed a similar technique for a related problem, called the shortest common supersequence problem (SCSP). The success of the application to the SCSP was the motivation for applying this approach also to the LCS problem.

This paper is organized as follows. In section 2 we present the constructive heuristic being the basis of our PBS algorithm. In section 3 we present the PBS algorithm, and in section 4 we describe the experimental evaluation of our approach. Finally, in section 5 we offer conclusions and an outlook to future work.

2 A Sequential Construction Heuristic

In this section we outline the so-called BEST-NEXT heuristic [8,9], which is a fast heuristic for the LCS problem. We will use the construction mechanism of this heuristic for our probabilistic beam search approach. Given a problem instance $(\mathcal{S}, \Sigma)$, the BEST-NEXT heuristic produces a common subsequence t sequentially from left to right by appending at each construction step a letter to t such that t maintains the property of being a common subsequence of all strings in $\mathcal{S}$. The algorithm is pseudo-coded in Algorithm 1.. In order to explain all the components of the algorithm we need to introduce the following definitions and notations. They all assume a given common subsequence t of the strings in $\mathcal{S}$.

1. Let $s_i = s_i^A \cdot s_i^B$ be the partition of s_i into substrings s_i^A and s_i^B such that t is a subsequence of s_i^A , and s_i^B has maximal length. Given this partition, which is well-defined, we introduce position pointers $p_i := |s_i^A|$ for $i = 1, \dots, n$ (see Figure 1 for an example).
2. The position of the first appearance of a letter $a \in \Sigma$ in a string $s_i \in \mathcal{S}$ after the position pointer p_i is well-defined and denoted by p_i^a . In case a letter $a \in \Sigma$ does not appear in s_i^B , p_i^a is set to ∞ (see Figure 1).
3. A letter $a \in \Sigma$ is called *dominated*, if exists at least one letter $b \in \Sigma$, $a \neq b$, such that $p_i^b < p_i^a$ for $i = 1, \dots, n$;
4. $\Sigma_t^{\text{nd}} \subseteq \Sigma$ henceforth denotes the set of non-dominated letters of the alphabet Σ . Moreover, for all $a \in \Sigma_t^{\text{nd}}$ it is required that $p_i^a < \infty$. Hence, we require that in each string s_i a letter $a \in \Sigma_t^{\text{nd}}$ appears at least once after position pointer p_i .

Function $\text{ChooseFrom}(\Sigma_t^{\text{nd}})$ is used to choose at each iteration one of the letters from Σ_t^{nd} for extending the common subsequence t . This is done by means of a so-called *greedy function*. In the following we present two different greedy functions that may be used. The first one—henceforth denoted by $\eta_1(\cdot)$ —is known from the literature (see, for example, [8]). The second one—henceforth denoted by $\eta_2(\cdot)$ —is a new development of this work. They are defined as follows:

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

$$\eta_2(a) = \left(\sum_{i=1, \dots, n} \frac{p_i^a - p_i}{|s_i^B|} \right)^{-1} \quad , \forall a \in \Sigma_t^{\text{nd}} \quad (2)$$

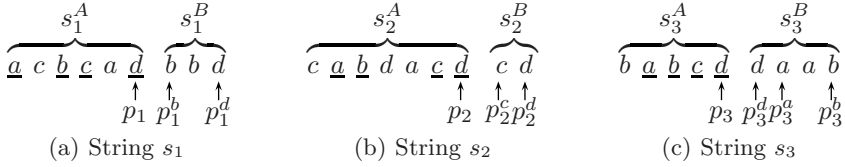


Fig. 1. Given is the problem instance $(\mathcal{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\})$ where $s_1 = acbcadb d$, $s_2 = cabdac d c d$, and $s_3 = babcd d a a b$. Let us assume that $t = abcd$. (a), (b), and (c) show the corresponding division of s_i into s_i^A and s_i^B , as well as the setting of the pointers p_i and the next positions of the 4 letters in s_i^B . Note that in case a letter does not appear in s_i^B , the corresponding pointer is set to ∞ . For example, as letter a does not appear in s_1^B , we set $p_1^a := \infty$.

Algorithm 1. BEST-NEXT heuristic for the LCS problem

- 1: **input:** a problem instance $(\mathcal{S}, \Sigma)$
 - 2: **initialization:** $t := \epsilon$ (where ϵ is the empty string)
 - 3: **while** $|\Sigma_t^{\text{nd}}| > 0$ **do**
 - 4: $a := \text{ChooseFrom}(\Sigma_t^{\text{nd}})$
 - 5: $t := ta$
 - 6: **end while**
 - 7: **output:** common subsequence t
-

In function $\text{ChooseFrom}(\Sigma_t^{\text{nd}})$ we choose $a \in \Sigma_t^{\text{nd}}$ such that $\eta_j(a) \geq \eta_j(b)$ for all $b \in \Sigma_t^{\text{nd}}$.¹ This completes the description of the BEST-NEXT heuristic.

3 Probabilistic Beam Search for the LCS Problem

In the following we present a probabilistic beam search (PBS) approach for the LCS problem. Our algorithm is based on the construction mechanism of the BEST-NEXT heuristic outlined in the previous section. Beam search is a classical tree search method that was introduced in the context of scheduling [10]. The central idea behind beam search is to allow the extension of partial solutions (here: subsequences) in more than one way. The version of beam search that we implemented—see Algorithm 2.—works as follows: The algorithm requires, apart from a problem instance $(\mathcal{S}, \Sigma)$, three input parameters. $k_{\text{bw}} \in \mathbb{Z}^+$ is the so-called *beam width*, $\mu \in \mathbb{R}^+ \geq 1$ is a parameter that is used to determine the number of children that can be chosen at each step, and t_{bsf} is the best solution found so far. At each step of the algorithm is given a set B of subsequences which is called the *beam*. At the start of the algorithm B only contains the empty string ϵ (that is, $B := \{\epsilon\}$). Let C denote the set of all possible extensions

¹ Note that for each application of the BEST-NEXT heuristic, we either use $j = 1$ or $j = 2$.

Algorithm 2. Probabilistic beam search (PBS) for the LCS problem

```

1: input: a problem instance  $(\mathcal{S}, \Sigma)$ ,  $k_{\text{bw}}$ ,  $\mu$ ,  $t_{\text{bsf}}$ 
2:  $B_{\text{compl}} := \{\epsilon\}$ ,  $B := \{\epsilon\}$ 
3: while  $B \neq \emptyset$  do
4:    $C := \text{Children\_of}(B)$ 
5:    $B := \emptyset$ 
6:   for  $k = 1, \dots, \min\{\lfloor \mu \cdot k_{\text{bw}} \rfloor, |C|\}$  do
7:      $\langle t, a \rangle := \text{Choose\_From}(C)$ 
8:      $t := ta$ 
9:     if  $\text{UB}(t) = |t|$  then
10:       $B_{\text{compl}} := B_{\text{compl}} \cup \{t\}$ 
11:      if  $|t| > |t_{\text{bsf}}|$  then  $t_{\text{bsf}} := t$  end if
12:    else
13:      if  $\text{UB}(t) \geq |t_{\text{bsf}}|$  then  $B := B \cup \{t\}$  end if
14:    end if
15:     $C := C \setminus \{t\}$ 
16:  end for
17:   $B := \text{Reduce}(B, k_{\text{bw}})$ 
18: end while
19: output:  $\text{argmax}\{|t| \mid t \in B_{\text{compl}}\}$ 

```

(children) of the subsequences in B .² At each step, $\lfloor \mu \cdot k_{\text{bw}} \rfloor$ different extensions from C are selected; each selection step is either performed probabilistically, or deterministically. A chosen extension, which is also a subsequence, is either stored in set B_{compl} in case it is a complete solution, or—in case its upper bound value $\text{UB}(\cdot)$ is greater than the length of the best-so-far solution t_{bsf} —it is stored in the new beam B . At the end of each step the new beam B is reduced in case it contains more than k_{bw} (the *beam width*) subsequences. This is done by evaluating the subsequences in B by means of the upper bound $\text{UB}(\cdot)$ mentioned already above, and by subsequently selecting the k_{bw} subsequences with the greatest upper bound values.

In the following we explain some aspects of Algorithm 2. in more detail. The algorithm uses three different functions. Given the current beam B as input, function $\text{Children_of}(B)$ produces the children (extensions) of all the subsequences in B in the form of a set C of tuples $\langle t, a \rangle$ for each pair $t \in B$ and $a \in \Sigma_t^{\text{nd}}$.

The second function— $\text{Choose_From}(C)$ —is used for choosing one of the (remaining) children in C . This is done by means of one of the greedy functions outlined in the previous section. However, note that the weights given by a greedy function are only meaningful for the comparison of two children $\langle t, a \rangle$ and $\langle t, b \rangle$, while they are meaningless for the comparison of two children $\langle t, a \rangle$ and $\langle z, b \rangle$ (where $t \neq z$). We solved this problem as follows. First, instead of the weights given by a greedy function, we use the corresponding ranks. More

² Remeber that the construction mechanism of the BEST-NEXT heuristic is based on extending a subsequence t by appending one letter from Σ_t^{nd} .

in detail, given all children $\{\langle t, a \rangle \mid a \in \Sigma_t^{\text{nd}}\}$ descending from a subsequence t , the child $\langle t, b \rangle$ with $\eta_j(\langle t, b \rangle) \geq \eta_j(\langle t, a \rangle)$ for all $a \in \Sigma_t^{\text{nd}}$ receives rank 1, denoted by $r_j(\langle t, b \rangle) = 1$.³ The child with the second highest greedy weight receives rank 2, and so on. Note that here we extend the notation $\eta_j(a)$ (as introduced in the previous section) to $\eta_j(\langle t, a \rangle)$, with the same meaning. Next, for evaluating a child $\langle t, a \rangle$ we use the sum of the ranks of the greedy weights that correspond to the construction steps performed to construct subsequence ta , that is

$$\nu(\langle t, a \rangle) := r_j(\langle \epsilon, t_1 \rangle) + \left(\sum_{i=1}^{|t|-1} r_j(\langle t_1 \dots t_i, t_{i+1} \rangle) \right) + r_j(\langle t, a \rangle) \quad , \quad (3)$$

where ϵ is the empty string, and t_i denotes the letter on position i in subsequence t . Moreover, $t_1 \dots t_i$ denotes the substring of t from position 1 to position i . In contrast to the greedy function weights, these newly defined $\nu(\cdot)$ -values can be used to compare children descending from different subsequences, which is done as follows. Given the set of children C , each $\langle t, a \rangle \in C$ is assigned the following probability:

$$\mathbf{p}(\langle t, a \rangle) := \frac{\nu(\langle t, a \rangle)}{\sum_{\langle z, b \rangle \in C} \nu(\langle z, b \rangle)} \quad , \forall \langle t, a \rangle \in C \quad (4)$$

When function **Choose_From**(C) is called, it is first decided whether to perform the choice of a child deterministically, or probabilistically. This is done by drawing a random number $q \in [0, 1]$ and comparing it to a parameter $0 \leq d \leq 1$. If $q < d$, the child $\langle t, a \rangle$ to be returned by function **Choose_From**(C) is selected such that $\langle t, a \rangle := \text{argmax}\{\mathbf{p}(\langle z, b \rangle) \mid \langle z, b \rangle \in C\}$. Otherwise, the child to be returned is selected by roulette-wheel-selection according to the probabilities defined in Equation 4.

The last function used by the PBS algorithm is **Reduce**(B, k_{bw}). In case $|B| > k_{\text{bw}}$, this function removes from B step-by-step those subsequences t that have an upper bound value $\text{UB}(t)$ smaller or equal to the upper bound value of all the other subsequences in B . The removal process stops once $|B| = k_{\text{bw}}$. Remember that k_{bw} is the beam width of the PBS algorithm, that is, the maximum number of subsequences that the algorithm is allowed to consider for extension.

Finally, we outline the upper bound function $\text{UB}(\cdot)$ that we used in our algorithm. Remember that a given subsequence t splits each string $s_i \in \mathcal{S}$ into a first part s_i^A and into a second part s_i^B , that is, $s_i = s_i^A \cdot s_i^B$ (see previous section). Henceforth, $|s_i^B|_a$ denotes the number of occurrences of letter a in s_i^B for all $a \in \Sigma$. Then,

$$\text{UB}(t) := |t| + \sum_{a \in \Sigma} \min\{|s_i^B|_a \mid i = 1, \dots, n\} \quad . \quad (5)$$

In words, for each letter $a \in \Sigma$ we take the minimum of the occurrences of a in s_i^B , $i = 1, \dots, n$. Summing up these minima and adding the result to the length of t results in the upper bound. This completes the description of the PBS algorithm.

³ Remember that $j \in \{1, 2\}$ indicates the used greedy function.

3.1 Deterministic Versus Probabilistic Beam Search

In this work we study two different ways of using the PBS algorithm: (1) Deterministic beam search, and (2) probabilistic beam search used in a multi-start fashion.

Deterministic beam search—henceforth simply denoted by BS—is obtained from algorithm PBS by setting $d = 1$. With this setting each choice of a child from C is done deterministically. Moreover, the best-so-far solution that is required as input of PBS is set to the empty string ϵ , that is, $t_{\text{bsf}} := \epsilon$.

A more interesting use of PBS is achieved by a setting of $d < 1$, and by using PBS in a multi-start fashion (see Algorithm 3.). The resulting algorithm—see Algorithm 3.—is denoted by MS-PBS. As stopping condition we use a maximum number of iterations, that is, calls of Algorithm 2.

Algorithm 3. Multi-start probabilistic beam search (MS-PBS)

```

1: input: a problem instance  $(\mathcal{S}, \Sigma)$ ,  $k_{\text{bw}}$ ,  $\mu$ 
2:  $t_{\text{bsf}} := \epsilon$ 
3: while stopping conditions not satisfied do
4:    $t := \text{PBS}((\mathcal{S}, \Sigma), k_{\text{bw}}, \mu, t_{\text{bsf}})$  {see Algorithm 2.}
5:   if  $|t| > |t_{\text{bsf}}|$  then  $t_{\text{bsf}} := t$ 
6: end while
7: output:  $t_{\text{bsf}}$ 
    
```

4 Experimental Evaluation

Apart from MS-PBS and BS, we also included the following algorithms in the experimental evaluation. The BEST-NEXT heuristic was applied with greedy function $\eta_1(\cdot)$ (see Equation 1), as well as with greedy function $\eta_2(\cdot)$ (see Equation 2). The former version is henceforth denoted by BN(1) and the latter one by BN(2). Moreover, we applied to all instances the expansion algorithm [6], henceforth denoted by EXP. We implemented all these algorithms in ANSI C++ using GCC 3.2.2 for compiling the software. Our experimental results were obtained on a PC with an AMD64X2 4400 processor and 4 Gb of memory.

4.1 Parameter Setting of Probabilistic Beam Search

Remember that Algorithm 2. has 4 paramters: k_{bw} is the beam width, that is, the maximum number of subsequences that the algorithm is allowed to keep at any time; μ is used to determine the number of children that can be chosen from set C at each step of the algorithm; d is the probability that determines the balance between probabilistic and deterministic choices of children from C ; ⁴ finally, we have to decide between greedy functions $\eta_1(\cdot)$ and $\eta_2(\cdot)$.

⁴ When $d = 0$ the algorithm is completely random, while a setting of $d = 1$ results in deterministic beam search.

For the experimental evaluation presented in this paper we decided to fix the parameters after an initial tuning by hand. A thorough tuning process is left for future work. We decided for a setting of $k_{bw} = 10$ and $\mu = 3$.⁵ Concerning the choice of the greedy function, we decided for $\eta_1(\cdot)$ for the application to the instances of **Set1**, whereas we chose $\eta_2(\cdot)$ for the instances of **Set2** (see Section 4.2 for a description of the problem instances). As we will explain in the following section, $\eta_1(\cdot)$ seems to work much better than $\eta_2(\cdot)$ for instances from **Set1**, whereas the opposite is the case for instances of **Set2**. Finally, after tuning by hand we chose a setting of $d = 0.8$, which means that on average 20% of the choices of children from set C are performed probabilistically.

4.2 Benchmark Instances

Two different sets of benchmark instances have been used in the experimentation. The instances of the first set—henceforth denoted by **Set1**—were produced such that the length of an optimal solution is known. All instances of **Set1** have the following characteristics: Each of them consists of 10 strings of length $l \in \{100, 200, \dots, 1000\}$, derived from an alphabet Σ , where $|\Sigma| \in \{2, 4, 8, 24\}$. For each combination of l and $|\Sigma|$ we produced randomly 10 instances as follows. The first 2 strings of each instance are randomly generated. Then, dynamic programming is applied in order to produce the longest common subsequence t^* of these 2 strings. The 8 remaining strings of an instance are produced on the basis of t^* , which is extended by adding randomly chosen letters from Σ at random positions until length l is reached. As there are 40 combinations of l and $|\Sigma|$, we produced a total of 400 instances. Note that due to the above outlined construction mechanism, the length of an optimal solution of an instance I is $|t^*|$.

The second set of instances—henceforth denoted by **Set2**—was generated quite differently. Given $h \in \{100, 200, \dots, 1000\}$ and Σ (where $|\Sigma| \in \{2, 4, 8, 24\}$), an instance is produced as follows. First, a string s of length h is produced randomly from the alphabet Σ . The string s is in the following called *base string*. Each instance contains again 10 strings. Each of these strings is produced from the base string s by traversing s and by deciding for each letter with a probability of 0.1 whether to remove it, or not. Note that the 10 strings of such an instance are not necessarily of the same length. As we produced 10 instances for each combination of h and $|\Sigma|$, 400 instances were generated in total. Note that the values of optimal solutions of these instances are unknown. However, a lower bound is obtained as follows. While producing the 10 strings of an instance, we record for each position of the base string s , whether the letter at that position was removed for the generation of at least one of the 10 strings. The number of positions in s that were never removed constitutes the lower bound value henceforth denoted by LB_I with respect to an instance I .

⁵ The only exception occurs when instances with $|\Sigma| = 2$ are concerned. In this case we chose $\mu = 1.5$, because a setting of $\mu = 2$ would mean that at each step all children of C are selected.

4.3 Results for Set1

We applied each of the 5 algorithms exactly once to each problem instance. The MS-PBS algorithm was applied with 100 applications of algorithm PBS. We present the results averaged over the 10 instances for each combination of l (the length of the 10 strings per instance), and the alphabet size $|\Sigma|$. Two measures are presented:

1. The (average) length of the solutions, expressed in percent with respect to the length of the optimal solutions. That is, 100% performance is achieved when the length of a solution is equal to the length of an optimal solution.
2. The computation time of the algorithms. In case of MS-PBS, this refers to the time the best solution was found within the 100 applications of algorithm PBS (averaged over the 10 instances of the same type).

The results are shown graphically in Figure 2. The graphics on the left hand side show the algorithm performance (first measure), and the graphics on the right hand side show the computation times. The results allow us to draw the following conclusions. First, the performance of BN(1) and EXP strongly decreases with increasing alphabet size. Surprisingly, the performance of the other three algorithms increases with increasing alphabet size. Second, algorithm MS-PBS is clearly the winner of the comparison. When $|\Sigma| > 4$, the algorithm nearly always finds the optimal solutions, regardless of l . Moreover, when the alphabet size is rather small and l is large the algorithm is clearly better than the 4 competitors. In particular, MS-PBS is never worse than BS, its deterministic version. With respect to computation times, algorithm EXP clearly uses much more computation time than the other approaches, while the heuristics are much faster than the other approaches. Algorithms BS and MS-PBS produce their results generally within a few seconds.

4.4 Results for Set2

We applied each of the 5 algorithms exactly once to each problem instance. The MS-PBS algorithm was again applied with 100 applications of algorithm PBS. We present the results averaged over the 10 instances for each combination of h (the length of the base string that was used to produce an instance), and the alphabet size $|\Sigma|$. Apart from the average computation time (see previous section) we present the following measure: The (average) length of the solutions expressed in percent deviation from the respective lower bounds, which is computed as follows:

$$\left(\frac{f}{\text{LB}_I} - 1 \right) \cdot 100 , \quad (6)$$

where f is the length of the solution achieved by the respective algorithm. The results are shown graphically in Figure 3. The graphics on the left hand side show again the algorithm performance (in percentage deviation from the lower bound), and the graphics on the right hand side show the computation times. The following observations are of interest. First, in contrast to the results for

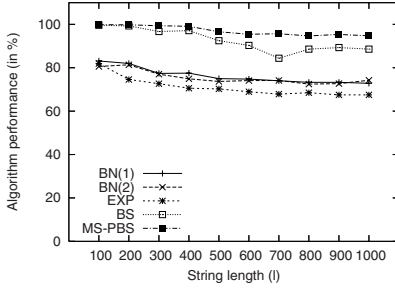
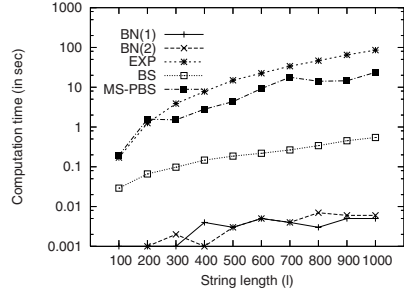
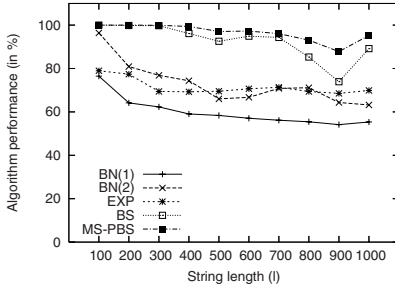
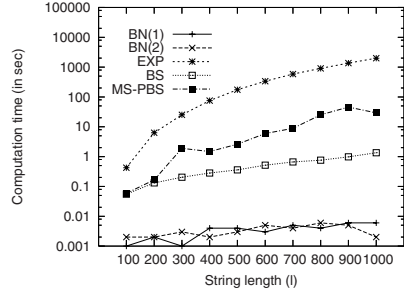
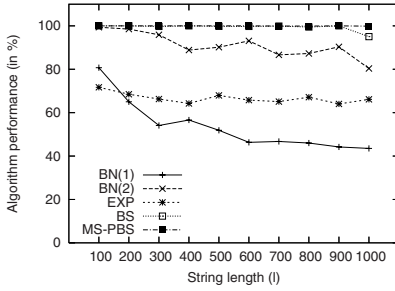
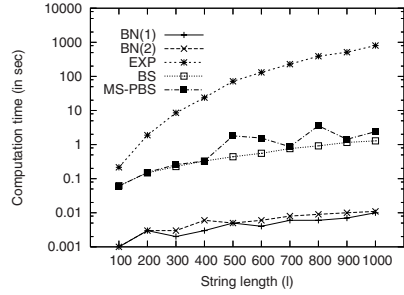
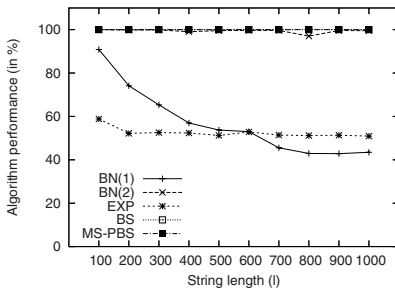
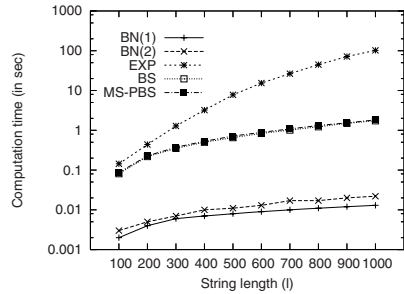
(a) $|\Sigma| = 2$, results(b) $|\Sigma| = 2$, computation times(c) $|\Sigma| = 4$, results(d) $|\Sigma| = 4$, computation times(e) $|\Sigma| = 8$, results(f) $|\Sigma| = 8$, computation times(g) $|\Sigma| = 24$, results(h) $|\Sigma| = 24$, computation times

Fig. 2. Results for the instances of Set1

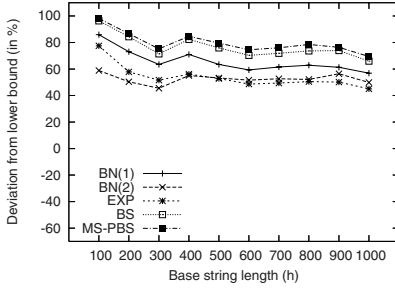
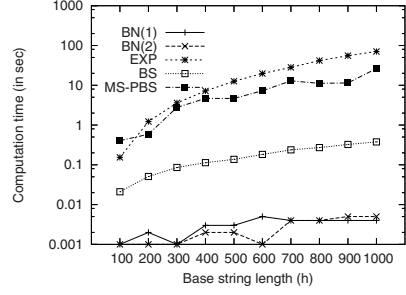
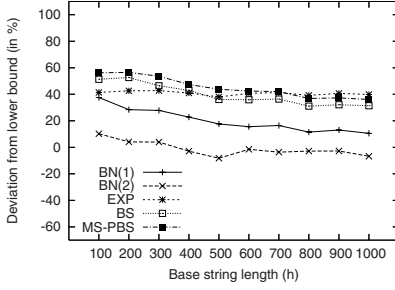
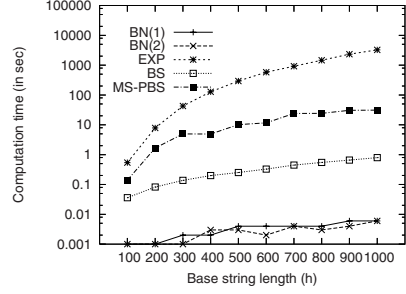
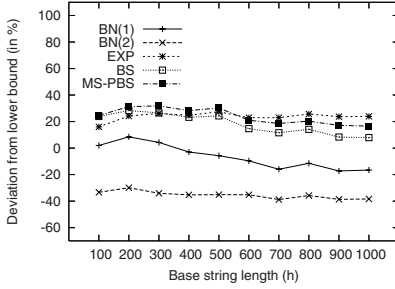
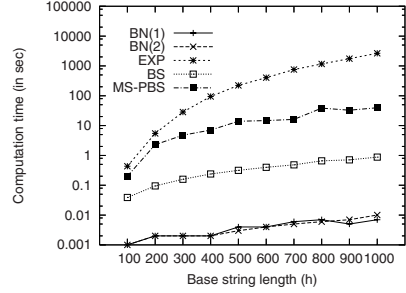
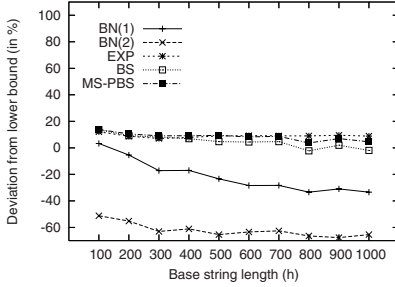
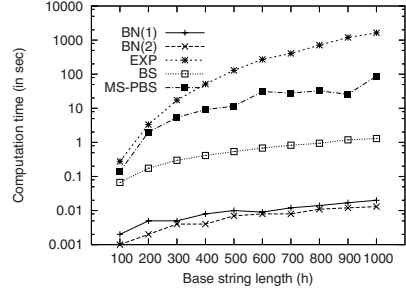

 (a) $|\Sigma| = 2$, results

 (b) $|\Sigma| = 2$, computation times

 (c) $|\Sigma| = 4$, results

 (d) $|\Sigma| = 4$, computation times

 (e) $|\Sigma| = 8$, results

 (f) $|\Sigma| = 8$, computation times

 (g) $|\Sigma| = 24$, results

 (h) $|\Sigma| = 24$, computation times

Fig. 3. Results for the instances of Set2

instance set **Set1**, algorithms BN(1) and EXP perform (in comparison) much better on the instances of **Set2**. Algorithm EXP is even slightly better than MS-PBS when $|\Sigma| \geq 4$ and h is large. However, EXP uses much more computation time than MS-PBS. Second, heuristic BN(1) is clearly better than heuristic BN(2) on this instance set. Concerning the beam search approaches, MS-PBS is again consistently better than BS, which shows the potential of a probabilistic beam search approach. However, especially when the alphabet size grows and h is large, there is room for improvement, which is shown by the fact that EXP is slightly better than MS-PBS in these cases. This might also be an indication that MS-PBS is not properly tuned for these combinations of $|\Sigma|$ and h .

5 Conclusions and Future Work

In this work we presented a probabilistic beam search approach for the longest common subsequence problem, which is one of the classical string problems with applications in pattern recognition and computational biology. We generated two sets of each 400 problem instances. In summary, the results indicated that the probabilistic beam search algorithm is clearly the best algorithm for the first set of instances, whereas it is—with the current parameter settings—slightly inferior by the expansion algorithm on certain instances from the second set. Concerning computation times, the probabilistic beam search approach needs much less computation time for reaching—in most cases improving—the quality level of the expansion algorithm.

Future work will focus on two lines. First, we intend to improve probabilistic beam search in particular for the cases in which it performed slightly worse than the expansion algorithm. Possible directions include the implementation of a lookahead mechanism and the addition of a learning component, as for example in Beam-ACO [11]. The second line of future work concerns the problem instances. We plan to test our algorithms also for instances with a varying string number. In this study we only regarded instances with 10 strings. Moreover, we would like to learn the reasons for the different behaviour of the greedy functions and the expansion algorithm when comparing between the two sets of differently generated problem instances.

References

1. Maier, D.: The complexity of some problems on subsequences and supersequences. *Journal of the ACM* 25, 322–336 (1978)
2. Gusfield, D.: *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
3. Aho, A., Hopcroft, J., Ullman, J.: *Data structures and algorithms*. Addison-Wesley, Reading, MA (1983)
4. Smith, T., Waterman, M.: Identification of common molecular subsequences. *Journal of Molecular Biology* 147(1), 195–197 (1981)

5. Bergeroth, 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, pp. 39–48. IEEE press, Los Alamitos (2000)
6. Bonizzoni, P., Della Vedova, G., Mauri, G.: Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics* 110, 13–24 (2001)
7. Blum, C., Cotta, C., Fernández, A.J., Gallardo, J.E.: A probabilistic beam search algorithm for the shortest common supersequence problem. In: C.C., et al. (eds.) *Proceedings of EvoCOP 2007 – Seventh European Conference on Evolutionary Computation in Combinatorial Optimisation*. LNCS, vol. 4446, Springer, Heidelberg (2007) (In press)
8. Fraser, C.B.: Subsequences and supersequences of strings. PhD thesis, University of Glasgow (1995)
9. Huang, K., Yang, C., Tseng, K.: Fast algorithms for finding the common subsequences of multiple sequences. In: *Proceedings of the International Computer Symposium*, pp. 1006–1011. IEEE press, Los Alamitos (2004)
10. Ow, P.S., Morton, T.E.: Filtered beam search in scheduling. *International Journal of Production Research* 26, 297–307 (1988)
11. Blum, C.: Beam-ACO—Hybridizing ant colony optimization with beam search: An application to open shop scheduling. *Computers & Operations Research* 32(6), 1565–1591 (2005)