Beam Search for the Longest Common Subsequence Problem

Christian Blum[1], Maria J. Blesa[1] and Manuel López-Ibáñez[2]

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

[2]School of Engineering and the Built Environment
Napier University, Edinburgh, UK
m.lopez-ibanez@napier.ac.uk

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, in 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 classical approaches as well as recent metaheuristic approaches.

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 $(\mathcal{S}, \Sigma)$, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is a set of n strings over a finite alphabet Σ, the problem consists in finding the longest string t^* that is a subsequence of all the strings in $\mathcal{S}$. Such a string t^* is called the *longest common subsequence* of the strings in $\mathcal{S}$.

Traditional computer science applications of this problem are in data compression [25], syntactic pattern recognition [17], file comparison [1], text edition [20] and query optimization in databases [21]. More recent applications include, for example, computational biology [24, 15] and the production of circuits in field programmable gate arrays [6].

Existing work.   The LCS problem was shown to be NP-hard [18] for an arbitrary number n of input strings. If n is fixed, the problem is polynomially solvable by dynamic programming [12]. Standard dynamic programming requires $O(l^n)$ 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] provide numerous references), dynamic programming becomes quickly impractical when either n or l grow.

An alternative to dynamic programming was proposed by Hsu and Du [13]. Their algorithm finds the longest path from the root to the leaves in a search tree where each node has $|\Sigma|$ children and each child corresponds to adding one letter to the common subsequence. This algorithm was further improved by Singireddy [23] 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 [8], SB outperforms dynamic programming for large n and small l. Singireddy [23] also proposed an integer programming approach based on Branch and Cut, however, this technique is also of complexity $O(l^n)$.

Approximate methods for the LCS problem were first proposed by Chin and Poon [7] and Jiang and Li [16]. The Long Run algorithm (LR) [16] returns the longest common subsequence 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. [5] and the BEST-NEXT heuristic [9, 14] also guarantee a factor of $|\Sigma|$ of the optimal common subsequence in the worst case, however, their results are typically much better than those of LR in terms of solution quality. Guenoche and Vitte [11] described a greedy algorithm that iteratively selects the next letter that minimises a given greedy function. Their algorithm 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. [6].

More recently, Easton and Singireddy [8] proposed an approximate large neighbourhood search technique called Time Horizon Specialized Branching (THSB) that makes internal use of the Specialized Branching algorithm. In addition, they implemented a variant of Guenoche and Vitte's algorithm [11], called G&V, that selects the best solution obtained from running the original algorithm with four different greedy functions proposed by Guenoche [10]. Easton and Singireddy [8] compared their algorithm with G&V, LR and EXPANSION, showing that THSB was able to obtain better results than all of them in terms of solution quality. Their results also showed that G&V outperforms EXPANSION and LR with respect to solution quality, while requiring a shorter time than EXPANSION and a similar computation time as LR. Finally, Shyu and Tsai [22] studied the application of ant colony optimization (ACO) to the LCS problem and concluded that their ACO algorithm dominates both EXPANSION and BEST-NEXT in terms of solution quality, while being much faster than EXPANSION.

Our contribution. In this work we propose the application of beam search (BS) to the LCS problem. Beam search is a classical tree search method that was introduced in the context of scheduling [19]. The central idea behind beam search 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 provide a study of different parameter settings that gives inside into the working of the algorithm. Furthermore, we apply the algorithm to recently published benchmark sets. This enables a comparison to existing methods from the literature.

The paper is organized as follows. In Section 2 we present the beam search approach to the LCS problem. The experimental evaluation of the algorithms is shown in Section 3. Finally, in Section 4 we offer conclusions and an outlook to future work.

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** $|\Sigma_t^{\mathrm{nd}}| > 0$ **do**
4:      $a := \mathsf{Choose\_From}(\Sigma_t^{\mathrm{nd}})$
5:      $t := ta$
6: **end while**
7: **output:** common subsequence t
_____

2 Beam Search

Beam search (BS) is a classical AI technique that was introduced in the context of scheduling [19]. BS performs a heuristic version of branch and bound with a breadth-first search strategy. Only the most promising k_{bw} nodes (that is, partial solutions) at each level of the search tree are selected for further examination. Parameter k_{bw} is refered to as the beam width. In general, the larger is the beam width the better are the results and the slower is the algorithm. Crucial components of BS are the underlying constructive heuristic for extending partial solutions and the upper bound function for evaluating partial solutions. In the following we present our implementation for the LCS problem in detail.

2.1 Constructive Heuristic

The so-called BEST-NEXT heuristic [9, 14] is a fast heuristic for the LCS problem. We will use the construction mechanism of this heuristic for beam search. Given a problem instance $(\mathcal{S}, \Sigma)$, the BEST-NEXT heuristic produces a common subsequence t sequentially from left to right, adding at each construction step exactly one letter to the current subsequence. The algorithm stops when no more letters can be added, that is, each further letter would produce an invalid solution. The pseudo-code of this heuristic is shown in Algorithm 1. Before we explain all the aspects of BEST-NEXT we first need to introduce the following definitions and notations. They all assume a given common subsequence t of the strings in $\mathcal{S}$ (see Figure 1 for an example).

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 $i = 1, \ldots, n$.

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 letter $a \in \Sigma$ does not appear in y_i, p_i^a is set to ∞.

3. 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 $i = 1, \ldots, n$;

4. $\Sigma_t^{\mathrm{nd}} \subseteq \Sigma$ denotes the set of non-dominated letters of the alphabet Σ with respect to t. Obviously it is required that a letter $a \in \Sigma_t^{\mathrm{nd}}$ appears in each string s_i at least once after the position pointer p_i.

Function $\mathsf{Choose\_From}(\Sigma_t^{\mathrm{nd}})$—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

$$\overbrace{\underline{a}\ c\ \underline{b}\ \underline{c}\ a\ \underline{d}}^{x_1}\ \overbrace{b\ b\ d}^{y_1} \qquad \overbrace{c\ \underline{a}\ \underline{b}\ d\ a\ \underline{c}\ \underline{d}}^{x_2}\ \overbrace{c\ d}^{y_2} \qquad \overbrace{b\ \underline{a}\ \underline{b}\ \underline{c}\ \underline{d}}^{x_3}\ \overbrace{d\ a\ a\ b}^{y_3}$$

$$\underset{p_1}{\uparrow}\ \underset{p_1^b}{\uparrow}\ \underset{p_1^d}{\uparrow} \qquad\qquad \underset{p_2}{\uparrow}\ \underset{p_2^c}{\uparrow}\underset{p_2^d}{\uparrow} \qquad\qquad \underset{p_3}{\uparrow}\ \underset{p_3^d}{\uparrow}\underset{p_3^a}{\uparrow}\ \underset{p_3^b}{\uparrow}$$

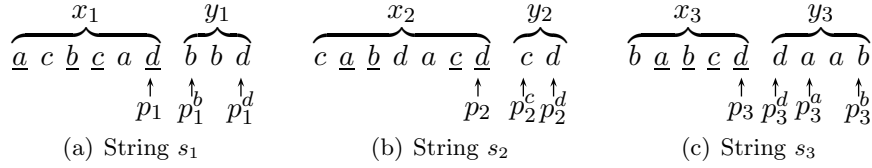(a) String s_1 (b) String s_2 (c) String s_3

Figure 1: This example assumes that a problem instance $(\mathcal{S} = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\})$ is given where $s_1 = acbcadbbd$, $s_2 = cabdacdcd$, and $s_3 = babcddaab$. Moreover, $t = abcd$. Figures (a), (b), and (c) show the corresponding division of s_i into x_i and y_i, as well as the setting of the pointers p_i and the next positions of the 4 letters in y_i. In case a letter does not appear in y_i, the corresponding pointer is set to ∞. For example, as letter a does not appear in y_1, p_1^a is set to ∞.

chosen 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 $\eta_1()$—is known from the literature (see, for example, Fraser [9]). The second one—henceforth denoted by $\eta_2()$—is new. They are defined as follows:

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

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

In function $\mathsf{Choose\_From}(\Sigma_t^{\mathrm{nd}})$ we choose $a \in \Sigma_t^{\mathrm{nd}}$ such that $\eta_1(a) \geq \eta_1(b)$ (respectively, $\eta_2(a) \geq \eta_2(b)$) for all $b \in \Sigma_t^{\mathrm{nd}}$. This completes the description of the BEST-NEXT heuristic.

2.2 Upper Bound

A second crucial element of Bs is the upper bound function used by the algorithm to evaluate partial solutions. Remember that a given common subsequence t splits each string $s_i \in \mathcal{S}$ into a first part x_i and into a second part y_i, that is, $s_i = x_i \cdot y_i$ (see previous section). 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:

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

In words, for each letter $a \in \Sigma$ the minimum number (over $i = 1, \dots, n$) of its occurences in y_i is taken. Summing up these minima and adding the resulting sum to the length of t results in the upper bound value. 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 guide the search process of Bs well.

2.3 The Beam Search Algorithm

As mentioned before, our Bs algorithm is based on the construction mechanism of the BEST-NEXT heuristic and on the upper bound function outlined in previous sections. The version of Bs that we implemented—see Algorithm 2—works roughly as follows: Apart from a problem instance $(\mathcal{S}, \Sigma)$, the algorithm requires three input parameters: $k_{\mathrm{bw}} \in \mathbb{Z}^+$ is the so-called *beam*

Algorithm 2 Beam search (Bs) for the LCS problem

1: **input:** a problem instance $(\mathcal{S}, \Sigma)$, k_{bw}, μ, $\eta()$
2: $B_{\mathrm{compl}} := \emptyset$, $B := \{\epsilon\}$, $t_{\mathrm{bsf}} := \epsilon$
3: **while** $B \neq \emptyset$ **do**
4:     $C := \mathsf{Produce\_Children}(B)$
5:     $C := \mathsf{Filter\_Children}(C)$ {this function is optional}
6:     $B := \emptyset$
7:     **for** $k = 1, \ldots, \min\{\lfloor \mu k_{\mathrm{bw}} \rfloor, |C|\}$ **do**
8:         $\langle t, a \rangle := \mathsf{Choose\_Best\_Child}(C, \eta())$
9:         $t := ta$
10:        **if** $\mathrm{UB}(t) = |t|$ **then**
11:            $B_{\mathrm{compl}} := B_{\mathrm{compl}} \cup \{t\}$
12:            **if** $|t| > |t_{\mathrm{bsf}}|$ **then** $t_{\mathrm{bsf}} := t$ **end if**
13:        **else**
14:            **if** $\mathrm{UB}(t) \geq |t_{\mathrm{bsf}}|$ **then** $B := B \cup \{t\}$ **end if**
15:        **end if**
16:        $C := C \setminus \{t\}$
17:    **end for**
18:    $B := \mathsf{Reduce}(B, k_{\mathrm{bw}})$
19: **end while**
20: **output:** $\mathrm{argmax}\, \{|t| \mid t \in B_{\mathrm{compl}}\}$

---

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 $\eta()$ is the particular greedy function used, either $\eta_1()$ or $\eta_2()$. At each step of the algorithm is given a set B of subsequences 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 (children) of the subsequences in B.[1] At each step, the best $\lfloor \mu k_{\mathrm{bw}} \rfloor$ partial solutions from C are selected with respect to the greedy function. A chosen partial solution is either stored in set B_{compl} in case it is a complete solution, or—in case its upper bound value $\mathrm{UB}()$ 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} partial solutions. This is done by evaluating the subsequences in B by means of the upper bound function $\mathrm{UB}()$, and by subsequently selecting the k_{bw} subsequences with the greatest upper bound values.

In the following we explain the functions of Algorithm 2 in detail. The algorithm uses four different functions. Given the current beam B as input, function $\mathsf{Produce\_Children}(B)$ produces the set C of non-dominated children (extensions) of all the subsequences in B. More in detail, C is a set of tuples $\langle t, a \rangle$, where $t \in B$ and $a \in \Sigma_t^{\mathrm{nd}}$. The second function— $\mathsf{Filter\_Children}(C)$—extends the non-domination relation, as defined in Section 2.1 for the children of a subsequence, for children of different subsequences of the same length. More specifically, given two children $\langle t, a \rangle \in C$ and $\langle z, b \rangle \in C$ (where $t \neq z$), $\langle t, a \rangle$ dominates $\langle z, b \rangle$ if and only if the position pointers for a appear before the position pointers for b in all n strings.

---

[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}.

The third function—Choose_Best_Child$(C, \eta())$—is used for choosing extensions from C. This function requires one of the two greedy functions outlined before as a parameter. However, note that for the comparison of two children $\langle t, a \rangle \in C$ and $\langle z, b \rangle \in C$ 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 children $\{\langle t, a \rangle \mid a \in \Sigma_t^{\mathrm{nd}}\}$ descending from a subsequence t, the child $\langle t, b \rangle$ with $\eta(\langle t, b \rangle) \geq \eta(\langle t, a \rangle)$ for all $a \in \Sigma_t^{\mathrm{nd}}$ receives rank 1, denoted by $r(\langle t, b \rangle) = 1$. The child with the second highest greedy weight receives rank 2, etc. Note that the notation $\eta(a)$ (as introduced previously) is extended here to the notation $\eta(\langle t, a \rangle)$. 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

$$\nu(\langle t, a \rangle) := r(\langle \epsilon, t_1 \rangle) + \left(\sum_{i=1}^{|t|-1} r(\langle t_1 \cdots t_i, t_{i+1} \rangle) \right) + r(\langle t, a \rangle) \quad , \tag{4}$$

where ϵ is the empty string, and t_i denotes the letter at position i in subsequence t. Morever, $t_1 \cdots t_i$ denotes the substring of t from position 1 to postion i. In contrast to the greedy function weights, these newly defined $\nu()$-values can be used to compare children descending from different subsequences. In fact, a call of function Choose_Best_Child$(C, \eta())$ returns the partial solution in C with maximal $\nu()^{-1}$ value.

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 $|B| = k_{\mathrm{bw}}$.

3   Experiments

Our experimental setup investigates the effect of the parameters k_{bw}, μ and $\eta()$ in terms of both solution quality and computation time. The beam width parameter k_{bw} is the number of subsequences that the Bs algorithm keeps in memory at each iteration. Intuitively, larger values of k_{bw} should produce better results at the cost of higher computation times. Recall that the parameter μ determines the number of children that are chosen from all possible children at each iteration of the search. 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 to be chosen for the next iteration. Finally, $\eta()$ may correspond to greedy function $\eta_1()$ as defined in Eq. 1, or $\eta_2()$ as defined in Eq. 2 on page 4. The effect of both greedy functions on solution quality and computation time is not clear, and thus, it is a subject of our investigation.

Experiments were run on a Intel Core2 1.66 GHz with 2 MB of cache size. Algorithms were implemented in ANSI C++ and compiled with GCC 4.1.2 in GNU/Linux 2.6.20.

3.1   Benchmark Instances

We consider three different sets of instances in our experiments. The first set, henceforth denoted by BLU, was generated by the following procedure. First, for each instance of alphabet

Σ, number of strings n and length l, a *base string* of length l is created by randomly generating letters from Σ. Then, each of the n strings is produced by traversing the base string and deleting each letter with a probability of 0.1. We generated 10 instances for each of the 80 combinations of $|\Sigma| \in \{2, 4, 8, 24\}$, $n \in \{10, 100\}$ and $l \in \{100, 200, 300, \ldots, 1000\}$.

In addition, we applied the Bs algorithm to two sets of benchmark instances from the literature. One set comes from Easton and Singireddy [8]. 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 second set of instances comes from Shyu and Tsai [22]. Their instances are biologically inspired, and thus, they considered alphabet sizes $|\Sigma| = 4$, corresponding to DNA sequences, and $|\Sigma| = 20$, corresponding to protein sequences. They studied three different types of instances. One was randomly generated, presumably in the same way as Easton and Singireddy's instances. The other two sets consist of real DNA and protein sequences of rats and viruses.

3.2 Experimental results for set **BLU**

We applied Bs with different parameters to each instance in set **BLU**. Results are shown graphically in Fig. 4. We only show the results for $l = 1000$. The results are similar for other string lengths but the effect of the different parameters is less clear. Each plot in Fig. 4 shows the effect of the combination of different values of k_{bw}, μ and $\eta()$, where each point summarises the results of one configuration of parameters of Bs. The top part of each plot gives the computation time in milliseconds for each configuration of parameters of Bs, while the bottom part gives the length of the LCS found by the same configuration of Bs. Each point corresponds to the mean result for the 10 instances with the same size, and error bars give an interval of plus/minus standard deviation around the mean. (Standard deviation is often too small for error bars to be visible).

Two main observations can be made from the results. First, as expected the solution quality steadily improves with higher beam widths (k_{bw}) at the expense of longer computation times, specially for small alphabets and few strings. Nonetheless, Bs with a beam width of $k_{\text{bw}} = 10$ is able to obtain good approximations within ten seconds even for the largest instances. The second main conclusion is that $\eta_2()$ performs clearly worse than $\eta_1()$ in terms of solution quality for a high number of strings n, while there is still little difference with respect to computation time. This is particularly true for moderate values of k_{bw}. Moreover, $\eta_2()$ performs specially worse for low values of μ, while the results obtained with $\eta_1()$ are still good even for $\mu = 1$. Even though for high values of k_{bw} and μ, the differences between the two greedy functions $\eta_1()$ and $\eta_2()$ are small, they are still always in favour of $\eta_1()$. As for μ, a value of 1.5 is clearly better for an alphabet size of $|\Sigma| = 2$, while for higher values of $|\Sigma|$ a value of $\mu > 2$ gives better results.

Following these findings, we select two configurations of Bs, one resulting in good solutions generated fast and another configuration that requires more time but generates the best solutions. The "low time" configuration corresponds to parameters $\eta_1()$, $k_{\text{bw}} = 10$, 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 $k_{\text{bw}} = 100$. These two configurations of Bs are compared in Table 1 with the results obtained by the classical EXPANSION and BEXT-NEXT algorithms. The results are overwhelmingly in favour of Bs. First, the best results are always obtained by the "high quality" configuration of Bs, which is always noticeably faster than EXPANSION. Therefore, this configuration of Bs completely outperforms EXPANSION. Second, although BEXT-NEXT

is evidently the fastest approach, the quality of its solutions is very poor compared to those obtained by "low time" Bs. According to the plots in Fig. 4, we could employ a much smaller beam width, e.g. $k_{\mathrm{bw}} = 2$, such that the run time of Bs is always under a second, and still Bs would generate better solutions than BEXT-NEXT.

So far, all runs of Bs made use of the function Filter_Children() in Algorithm 2 on page 5. The use of filtering typically produces better quality solutions for instances with a small number of strings ($n = 10$), as shown graphically in plots (a), (c) and (e) of Fig. 2. The benefit is even larger for higher values of k_{bw} —compare plot (a) versus plot (c). Interestingly, the longer the length of the strings, the larger is the difference in the solution quality between using filtering or not. In the same way, the computation time overhead of using filtering increases with the length of the strings, as shown in plots (a) and (c), and with the size of the alphabet, as shown in plot (e). On the other hand, for large number of strings ($n = 100$), filtering may actually reduce computation time, as shown in plots (b), (d) and (f) of Fig. 2. In this group of instances, however, filtering is not always beneficial in terms of solution quality. Nonetheless, the median difference (as denoted by the line within each boxplot) 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   Comparison with THSB and G&V

As for the instances provided by Easton and Singireddy [8], we again investigated the best parameters of Bs. The complete results of fine-tuning Bs are shown in Figures 5 and 6 by means of plots similar to those used to describe the fine-tuning of Bs in the previous section. The only difference is that here each point and error bar correspond to the mean and standard deviation over 50 instances (instead of 10). In these figures, each row of plots corresponds to instances with the same number of strings, namely $n = \{10, 50, 100\}$, and each column corresponds to instances with the same alphabet size: $|\Sigma| = \{2, 10\}$ in Fig. 5 and $|\Sigma| = \{25, 100\}$ in Fig. 6.

Interestingly, the best parameters are somewhat different from the ones for the previous set of benchmark instances. As before, increasing the beam width results in better solutions at the expense of longer computation times. However, in contrast to the instances of set BLU, here the greedy function $\eta_2()$ performs clearly better than $\eta_1()$, specially for large n (number of strings), and even for the highest beam width and μ there are notable differences in solution quality between $\eta_1()$ and $\eta_2()$. Moreover, for both greedy functions the use of $\mu > 1$ seems fundamental for achieving good results. These differences clearly indicate that the method used to generate the instances can have an influence on the performance of the greedy functions.

Taking into account these results, we select two configurations of Bs as the ones giving the best results for this set of instances. The configuration using $k_{\mathrm{bw}} = 10$, $\eta_2()$ and $\mu = 1.5$ gives good solutions for all instances in a relatively short time. On the other hand, both $\eta_2()$, $k_{\mathrm{bw}} = 100$ and $\mu = 1.5$ for $|\Sigma| = 2$, and $\eta_2()$, $k_{\mathrm{bw}} = 50$, $\mu = 3$ for $|\Sigma| > 2$, give the best solution quality but require much longer computation times. Using these parameter settings, Table 2 compares the performance of Bs with the results of THSB (for both high quality and low time configurations) and G&V provided by Easton and Singireddy [8]. Easton and Sigireddy performed their experiments on a 1.5GHz Pentium IV, which should be slightly slower than

Table 1: Comparison of EXPANSION, BEST-NEXT and Bs for instances of set BLU. $|t^*|$ is the solution quality and "Time" is the computation time in seconds. Each values corresponds to the mean and, in parenthesis, the standard deviation of the results of 10 instances.

Instance			Expansion		Bext-Next		Bs (low time)		Bs (high quality)	
$|\Sigma|$	n	l	$|t^*|$	Time	$|t^*|$	Time	$|t^*|$	Time	$|t^*|$	Time
2	10	1000	515.4 (16.2)	43.9 (3.3)	556.9 (14.4)	0.0 (0.0)	613.2 (14.6)	0.6 (0.0)	648.0 (15.0)	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)	6.3 (0.2)	541.0 (8.0)	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)	0.9 (0.0)	534.7 (12.6)	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)	9.3 (0.2)	369.3 (4.6)	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)	0.7 (0.0)	462.3 (12.9)	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)	10.6 (0.3)	258.7 (4.7)	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)	1.3 (0.0)	385.6 (7.0)	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)	13.5 (0.2)	147.7 (4.6)	268.3 (8.0)

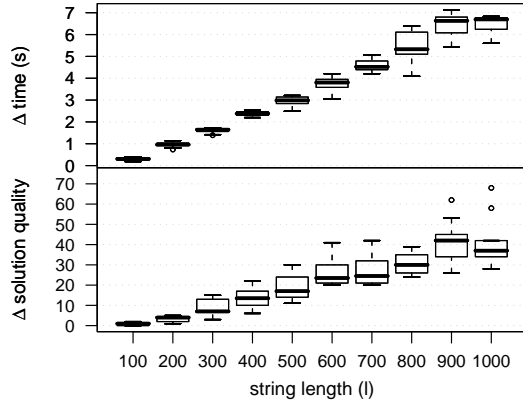**Bs (low time)**: $\eta_1()$, $k_{bw} = 10$, and $\mu = 1.5$ for $|\Sigma| = 2$ or $\mu = 3$ for $|\Sigma| > 2$.
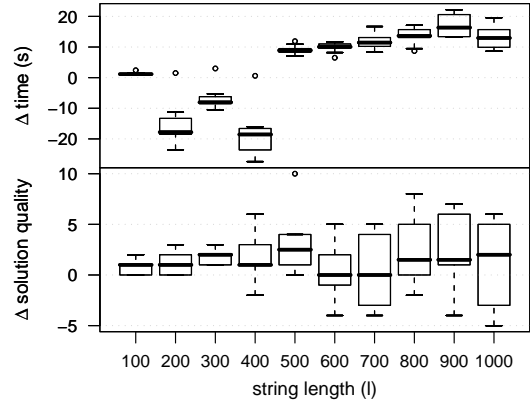**Bs (high quality)**: $\eta_1()$, $k_{bw} = 100$, and $\mu = 1.5$ for $|\Sigma| = 2$ or $\mu = 3$ for $|\Sigma| > 2$.
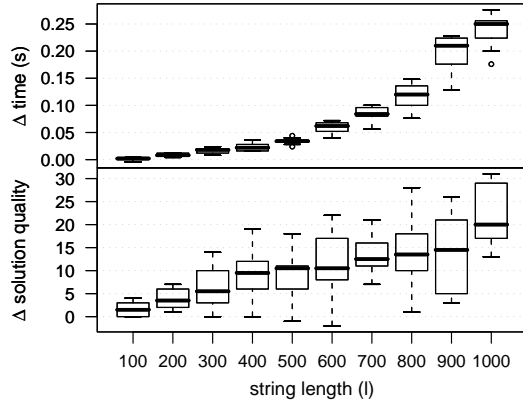
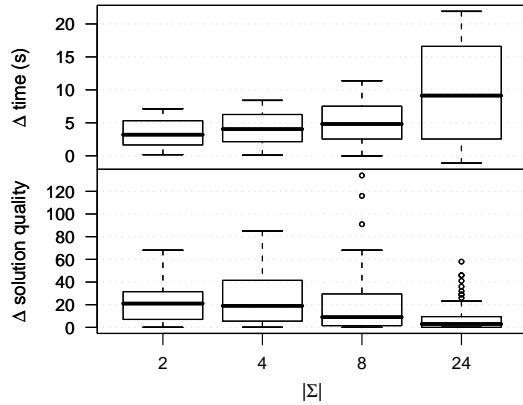(a) $|\Sigma| = 2$, $n = 10$, Bs: $k_{\mathrm{bw}} = 10$, $\mu = 1.5$, $\eta_1()$
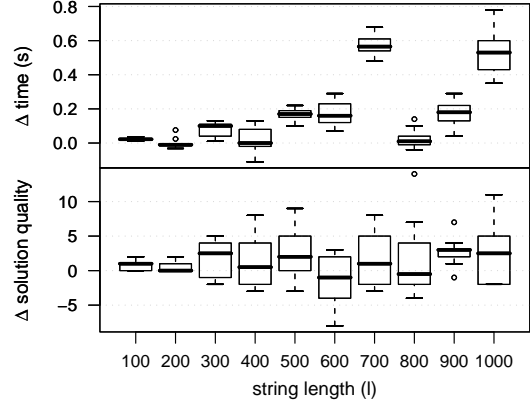
(b) $|\Sigma| = 2$, $n = 100$, Bs: $k_{\mathrm{bw}} = 10$, $\mu = 1.5$, $\eta_1()$

(c) $|\Sigma| = 2$, $n = 10$, Bs: $k_{\mathrm{bw}} = 100$, $\mu = 1.5$, $\eta_1()$

(d) $|\Sigma| = 2$, $n = 100$, Bs: $k_{\mathrm{bw}} = 100$, $\mu = 1.5$, $\eta_1()$

(e) $n = 10$, $l = \{100, 200, \ldots, 1000\}$, Bs: $k_{\mathrm{bw}} = 100$, $\mu = \{1.5(\text{in case } |\Sigma| = 2), 3(\text{in case } |\Sigma| > 2)\}$, $\eta_1()$
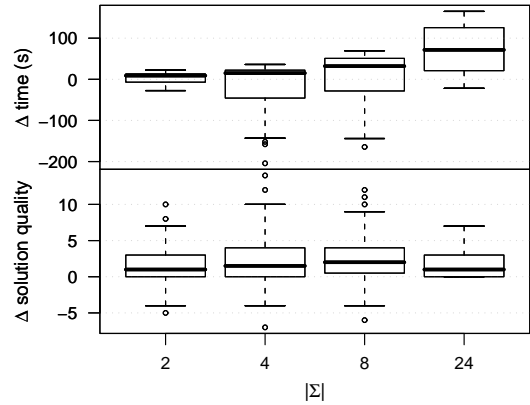
(f) $n = 100$, $l = \{100, 200, \ldots, 1000\}$, Bs: $k_{\mathrm{bw}} = 100$, $\mu = \{1.5(\text{in case } |\Sigma| = 2), 3(\text{in case } |\Sigma| > 2)\}$, $\eta_1()$

Figure 2: Comparison between using filtering or not on instances of set BLU.

our machine.

The results in Table 2 show that the "low time" configuration of Bs completely dominates the "high quality" and much slower configuration of THSB. That is, Bs is able to obtain better solutions in a much shorter time. The "high quality" configuration of Bs further improves the solution quality while still being in most cases faster than the "high quality" configuration of THSB. As for the "low time" configuration of THSB, it is typically faster than Bs, however, the resulting sequences are much worse than those obtained by Bs. In fact, G&V is able to obtain larger sequences in a much shorter time, hence the results of the "low time" configuration of THSB bear little significance.

Comparing G&V and Bs, the former is definitely the fastest of the three algorithms, specially for the largest instances, but at a significant loss of solution quality with respect to Bs. In fact, considering Figs. 5 and 6, the worst results of Bs when using $\eta_2()$, $k_{\mathrm{bw}} = 2$ and $\mu = 1$ are still better than those obtained by G&V.

3.4 Comparison with ACO

With regard to the set of instances provided by Shyu and Tsai [22], we repeat the same analysis. That is, we fine-tune the parameters of Bs and graphically show the results in Fig. 7. Plots in the same row correspond to the same instance type (either Random, Rat or Virus). Plots in the same column have the same alphabet size $|\Sigma|$, number of strings n and string length l. We only show the results for $n = 200$ because the conclusions are similar for other values of n. Since there is only one instance available for each instance type and combination of $|\Sigma|$, n and l, each point in the plots corresponds to the result of just one run of Bs. This fact may explain the high variability of the results. Despite this variability, some trends can be identified. As in all previous instances, higher values of k_{bw} typically produce improved results. For the Random instances, plots (a) and (b) in Fig. 7 are somehow similar to the ones concerning the instances of Easton and Singireddy from the previous section (compare plot (c) in Fig. 5 and plot (i) in Fig. 6). In particular, the combination of $\eta_2()$ and values of $\mu > 1.5$ is clearly better than the alternatives. On the other hand, for Rat and Virus there are striking differences between the different alphabet sizes. This may be explained by structural differences, since instances with $|\Sigma| = 4$ correspond to DNA sequences, while instances with $|\Sigma| = 20$ correspond to protein sequences. The behaviour of Bs for the DNA sequences resembles the behaviour shown in plot (d) of Fig. 4, corresponding to the BLU set, although no clear pattern can be identified for the instances derived from protein sequences. Nonetheless, the combination of $\eta_2()$ and values of $\mu > 1.5$ results in longer sequences more frequently than other combinations of parameters. Therefore, we choose $\eta_2()$, $k_{\mathrm{bw}} = 100$, with $\mu = 3$ for $|\Sigma| = 4$ and $\mu = 5$ for $|\Sigma| = 20$ as a "high quality" configuration. The chosen "low time" configuration is $\eta_2()$, $k_{\mathrm{bw}} = 10$, with $\mu = 3$ for $|\Sigma| = 4$ and $\mu = 5$ for $|\Sigma| = 20$.

We then compare these two configurations of Bs with the results obtained by Shyu and Tsai with their ACO algorithm [22]. Their algorithm was implemented in C++ and their experiments were run on a AMD Athlon 2100+ CPU, which should be slightly faster than our machine. Tables 3, 4 and 5 show this comparison for Random, Rat and Virus instances, respectively. These tables are graphically summarised in Fig. 7, where each point corresponds to the result of one run of Bs or to the mean of 10 runs of ACO in one instance. Error bars delimit plus/minus one standard deviation around the mean 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

Table 2: Comparison between G&V, THSB and Bs. $|t^*|$ is the solution quality and "Time" is the computation time in seconds, both averaged over 50 instances. Results for G&V and THSB are taken from Easton and Singireddy [8]. We provide the standard deviation in parenthesis for our results.

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

Bs (low time): $\eta_2()$, $k_{\mathrm{bw}} = 10$, $\mu = 1.5$.
Bs (high quality): $\eta_2()$, $k_{\mathrm{bw}} = 100$, $\mu = 1.5$ for $|\Sigma| = 2$; $\eta_2()$, $k_{\mathrm{bw}} = 50$, $\mu = 3$ for $|\Sigma| > 2$.

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 $|\Sigma| = 20$, "low time" Bs also 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, "high quality" Bs is clearly slower than ACO. Nonetheless, "high quality" Bs outperforms both ACO and "low time" Bs, being the difference particularly large for instances with few strings (small value of n). The overall conclusion is that Bs obtains as good solutions as ACO, and often even better ones, in a shorter time, while higher beam widths lead to clearly better solutions at the expense of possibly longer running times than ACO.

Table 3: Comparison of ACO and Bs for Random instances. $|t^*|$ is the solution quality and "Time" is the computation time in seconds. Results for ACO are taken from Shyu and Tsai [22] and show the mean and, in parenthesis, the standard deviation of 10 independent runs for a single instance. Since Bs is deterministic, the result shown is the one obtained for each single instance.

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

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

Table 4: Comparison of ACO and Bs for Rat instances. $|t^*|$ is the solution quality and "Time" is the computation time in seconds. Results for ACO are taken from Shyu and Tsai [22] and show the mean and, in parenthesis, the standard deviation of 10 independent runs for a single instance. Since Bs is deterministic, the result shown is the one obtained for each single instance.

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

Table 5: Comparison of ACO and Bs for Virus instances. $|t^*|$ is the solution quality and "Time" is the computation time in seconds. Results for ACO are taken from Shyu and Tsai [22] and show the mean and, in parenthesis, the standard deviation of 10 independent runs for a single instance. Since Bs is deterministic, the result shown is the one obtained for each single instance.

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

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

(a) Random data: $|\Sigma| = 4$, $l = 600$ (b) Rat data: $|\Sigma| = 4$, $l = 600$ (c) Virus data: $|\Sigma| = 4$, $l = 600$

(d) Random data: $|\Sigma| = 20$, $l = 600$ (e) Rat data: $|\Sigma| = 20$, $l = 600$ (f) Virus data: $|\Sigma| = 20$, $l = 600$
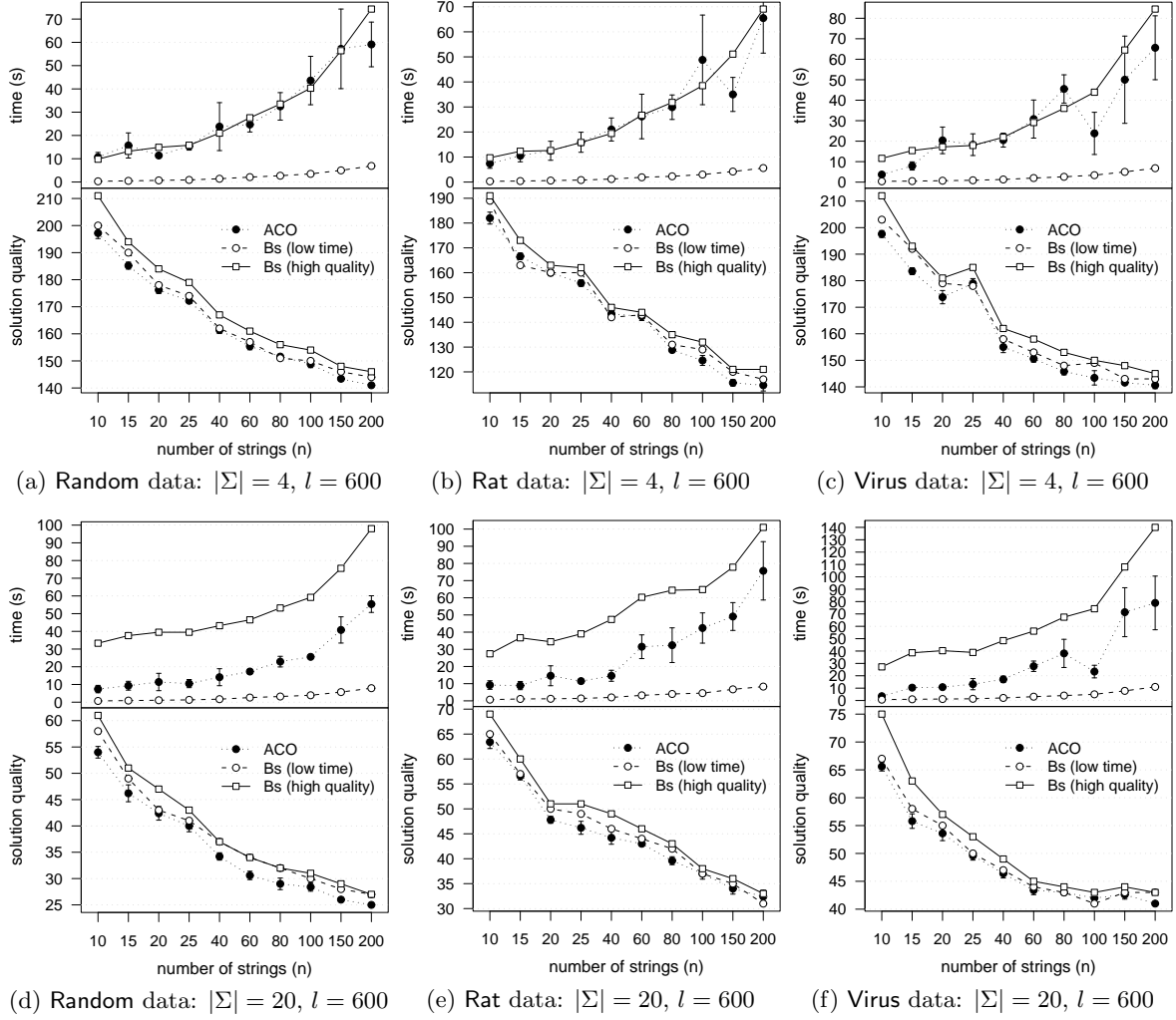
Figure 3: Graphical comparison of ACO and Bs on the instances of Shyu and Tsai [22].

16

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, two of them already used in the literature. These sets were generated using different procedures and the results of fine-tuning Bs for each set of instances show important differences between them. In particular, the well-known greedy function $\eta_1()$ [9], defined in Eq. 1, performs better for instances generated by deleting letters from a base string (set BLU), while the proposed greedy function $\eta_2()$, defined in Eq. 2, obtains the best results for instances generated by concatenating random letters (set of Easton and Singireddy [8]).

As for the other parameters of Bs, namely the beam width and μ, the use of a larger beam width improves the quality of solutions obtained by Bs, although it also increases the computation time required to reach a solution. Values of $\mu = 1.5$ generally produce better results for instances with alphabet size $|\Sigma| = 2$. For higher alphabet sizes, values of μ slightly higher than 2 generate high quality solutions if the appropriate greedy function is used. Values higher than $\mu > 5$ result in an increase of computation time that does not pay off in terms of solution quality.

Following these findings, we selected for each set of instances two configurations of parameters of Bs, one characterised by a low run time and another generating very high quality solution. These "low time" and "high quality" configurations were compared with four other approaches: the "low time" and "high quality" configurations of THSB, as defined by its authors, and the G&V and ACO algorithms. The results show that "low time" Bs completely outperforms "high quality" THSB in both quality and time. On the other hand, the "low time" configuration of THSB is outperformed by G&V with respect to both quality and computation time. Although G&V is faster than Bs, the latter finds much better solutions in a relatively short time. As for the ACO algorithm, Bs produces equally good solutions in a much shorter time, specially for high number of strings. If longer runs are allowed, Bs using a larger beam width is able to consistently find better solutions than ACO.

In summary, our experimental analysis shows that the proposed beam search is the best approximation algorithm for solving the LCS problem among the ones considered in this paper. In fact, by using the appropriate greedy function for each instance type, beam search obtains the best results for several different types of instances. With respect to computation time, G&V is faster than beam search but it generates much worse solutions. In all other cases, beam search requires less computation times for matching, and typically improving, the quality of the other algorithms.

Acknowledgements

References

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

[2] L. Bergroth, H. Hakonen, and T. Raita. New approximation algorithms for longest common subsequences. In *String Processing and Information Retrieval: A South American Symposium, 1998. Proceedings*, pages 32–40, 1998.

[3] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of SPIRE 2000 – 7th International Symposium on String Processing and Information Retrieval*, pages 39–48. IEEE press, 2000.

[4] C. Blum and M. Blesa. Probabilistic beam search for the longest common subsequence problem. In T. Stützle, M. Birattari, and H. H. Hoos, editors, *Proceedings of SLS 2007 – Engineering Stochastic Local Search Algorithms*, volume 4638 of *Lecture Notes in Computer Science*, pages 150–161. Springer-Verlag, Berlin, Germany, 2007.

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

[6] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In *Proceedings of the 41st Design Automation Conference*, pages 395–400. IEEE press, 2004.

[7] F. Chin and C. K. Poon. Performance analysis of some simple heuristics for computing longest common subsequences. *Algorithmica*, 12(4–5):293–311, 1994.

[8] T. Easton and A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 2007. In press.

[9] C. B. Fraser. *Subsequences and supersequences of strings.* PhD thesis, University of Glasgow, 1995.

[10] A. Guenoche. Supersequence of masks for oligo-chips. *Journal of Bioinformatics and Computational Biology*, 2(3):459–469, 2004.

[11] A. Guenoche and P. Vitte. Longest common subsequence with many strings: exact and approximate methods. *Technique et science informatiques*, 14(7):897–915, 1995. In French.

[12] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.

[13] W. J. Hsu and M. W. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59, 1984.

[14] K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of the International Computer Symposium*, pages 1006–1011. IEEE press, 2004.

[15] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.

[16] Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[17] S. Y. Lu and K. S. Fu. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 8(5):381–389, 1978.

[18] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, 1978.

[19] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:297–307, 1988.

[20] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, UK, 1983.

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

[22] S. J. Shyu and C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers & Operations Research*, 2007. In Press.

[23] A. Singireddy. Solving the longest common subsequence problem in bioinformatics. Master's thesis, Industrial and Manufacturing Systems Engineering, Kansas State University, Manhattan, KS, 2007.

[24] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

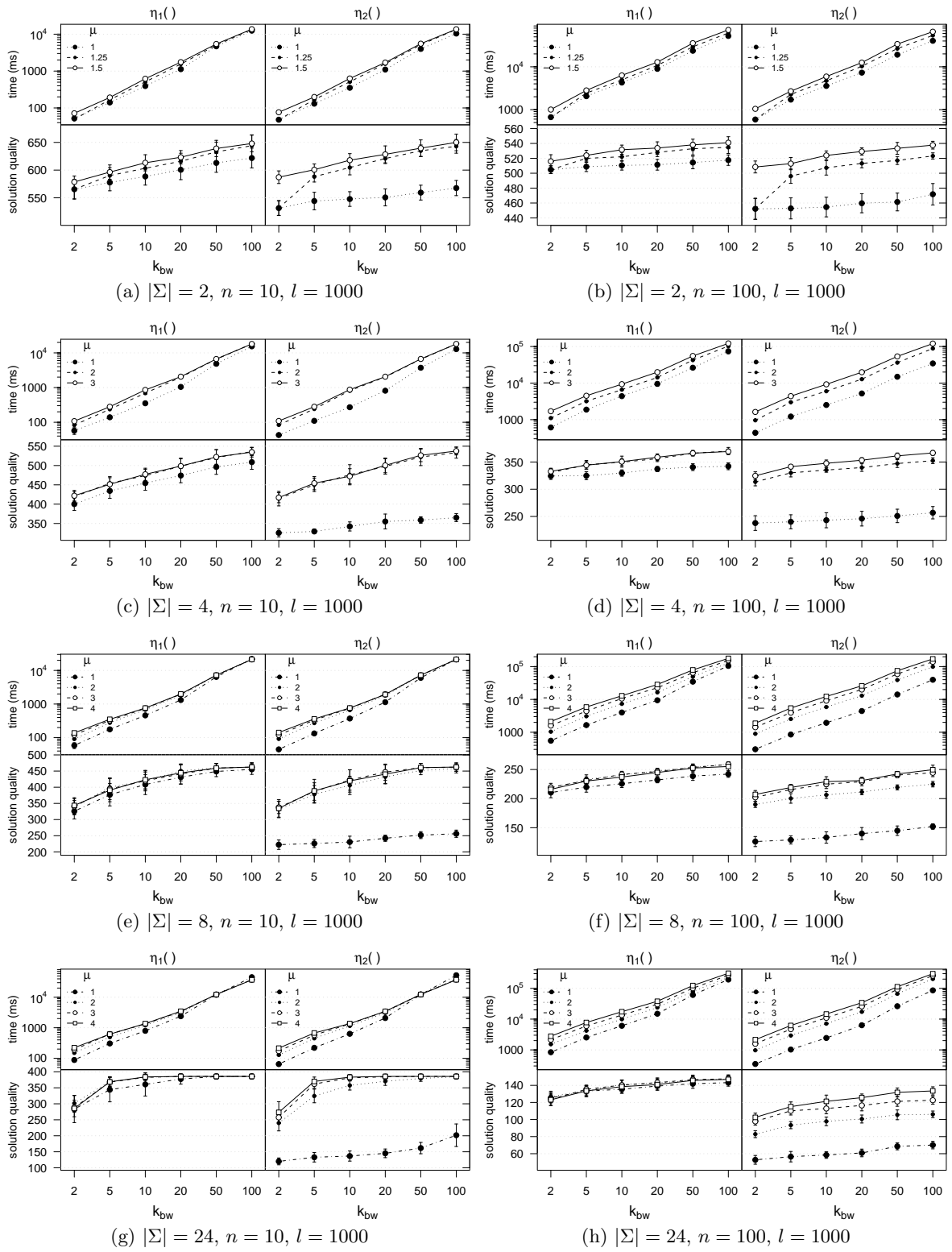[25] J. Storer. *Data Compression: Methods and Theory.* Computer Science Press, MD, 1988.

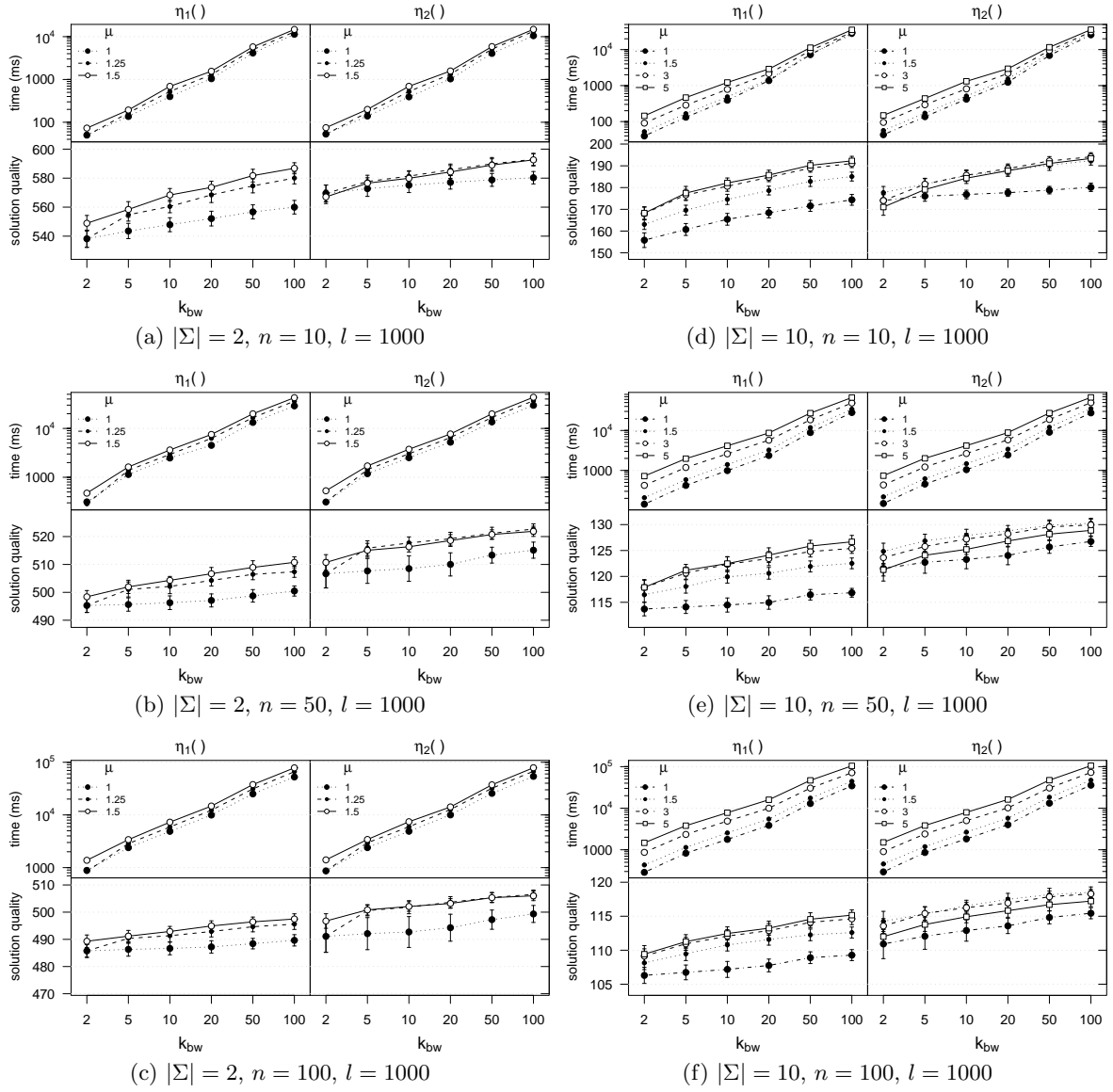Figure 4: Results of fine-tuning Bs for the instances of set BLU.

Figure 5: Results of fine-tuning Bs for the instances of Easton and Singireddy [8] (continues in the next page).

(g) $|\Sigma| = 25$, $n = 10$, $l = 2500$

(j) $|\Sigma| = 100$, $n = 10$, $l = 5000$

(h) $|\Sigma| = 25$, $n = 50$, $l = 2500$

(k) $|\Sigma| = 100$, $n = 50$, $l = 5000$

(i) $|\Sigma| = 25$, $n = 100$, $l = 2500$
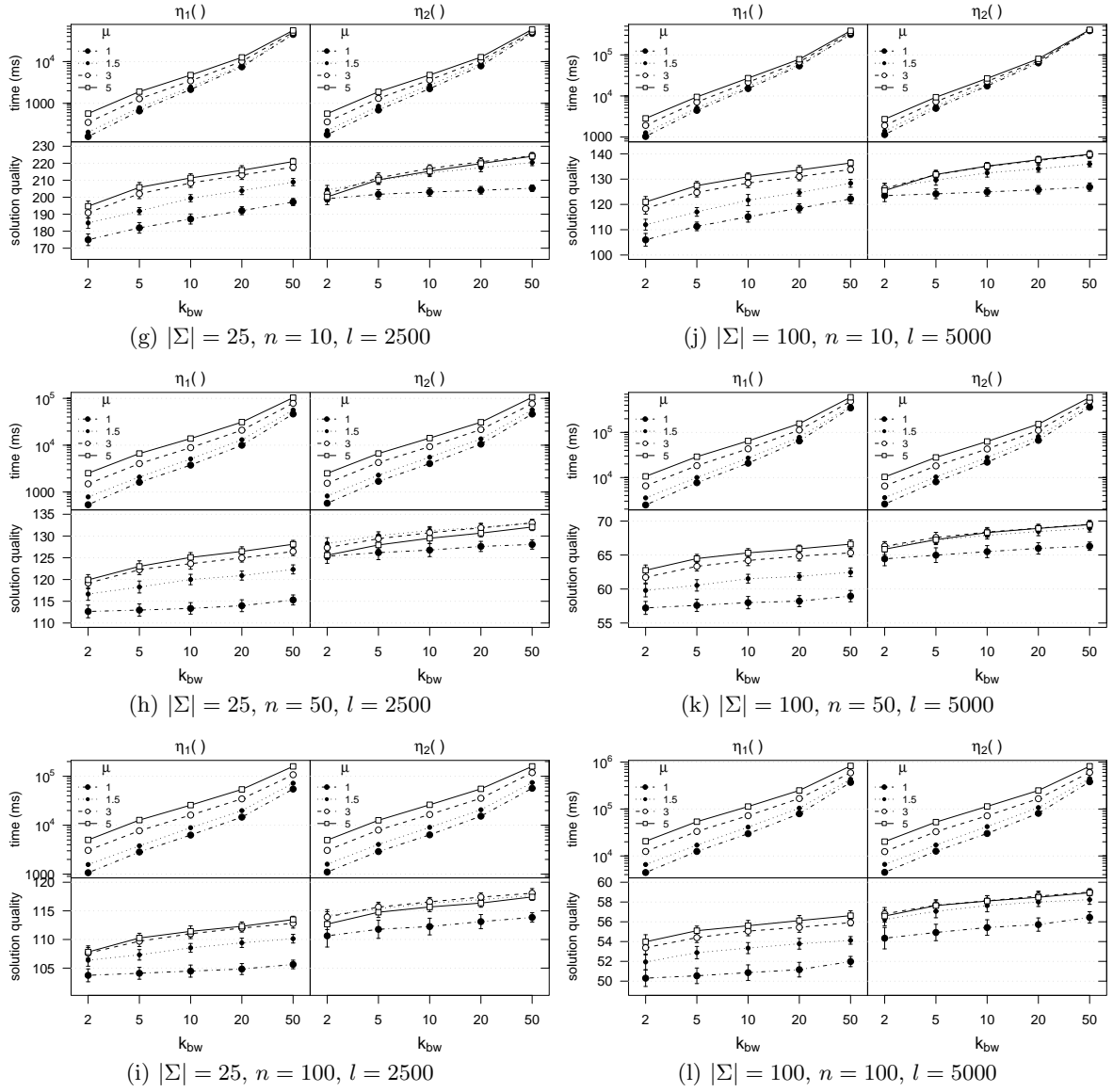
(l) $|\Sigma| = 100$, $n = 100$, $l = 5000$

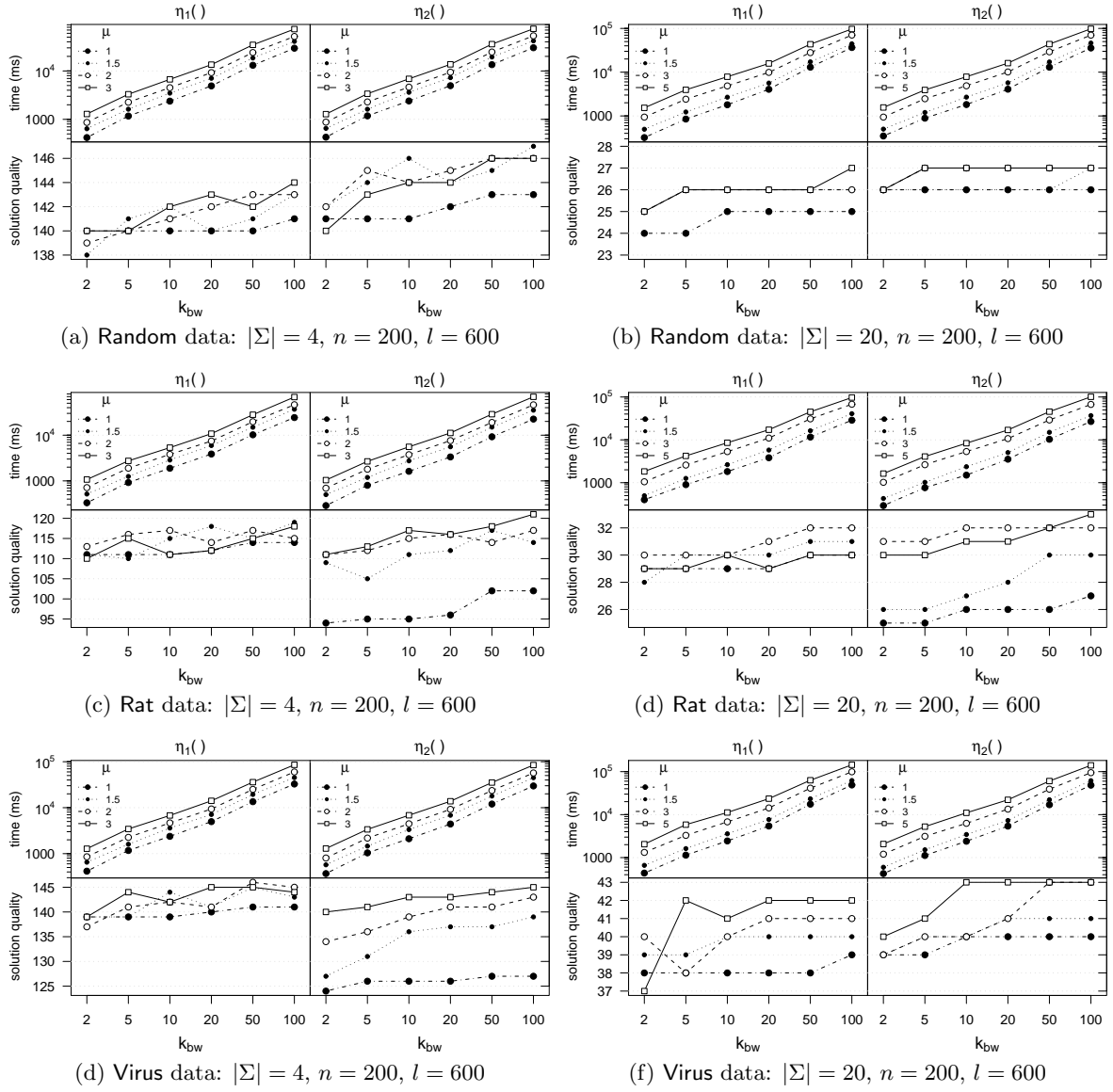Figure 6: Results of fine-tuning Bs for the instances of Easton and Singireddy [8] (continued from the previous page).

Figure 7: Results of fine-tuning Bs for the instances of Shyu and Tsai [22].