Anytime Algorithms for the Longest Common Palindromic Subsequence Problem – Supplementary Material –

Marko Djukanovic[1], Günther R. Raidl[1], and Christian Blum[2]
[1]Institute of Logic and Computation, TU Wien, Vienna, Austria
[2]Artificial Intelligence Research Institute (IIIA-CSIC),
Campus UAB, Bellaterra, Spain
{djukanovic|raidl}@ac.tuwien.ac.at, christian.blum@iiia.csic.es

This document provides the following supplementary information:

- the complete set of graphics concerning the anytime performance of the proposed algorithms, both concerning the solution quality and the gaps
- descriptions of the 2–LCPS algorithms from literature to which we compare in the main paper, with our choices of certain implementation aspects

1 Anytime plots of the algorithms showing the evolution of the obtained solution quality

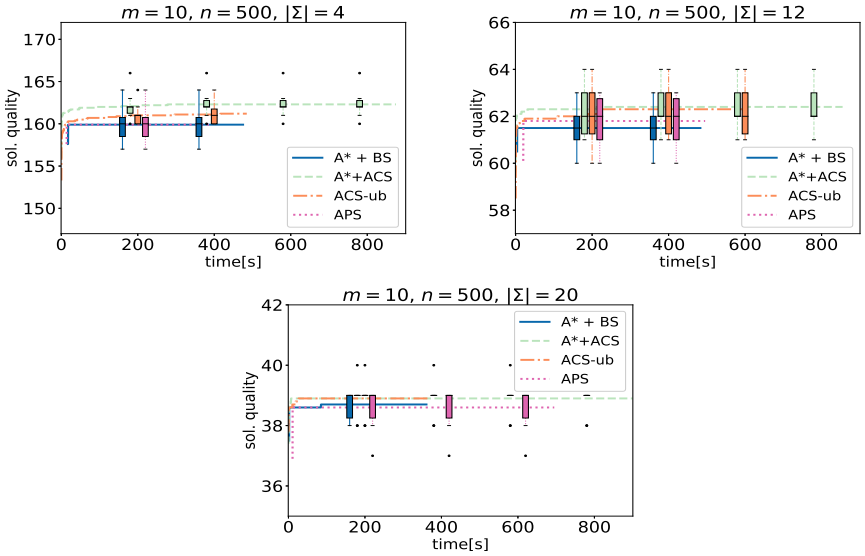

Figure 1: Instances with m =10 and n =500.

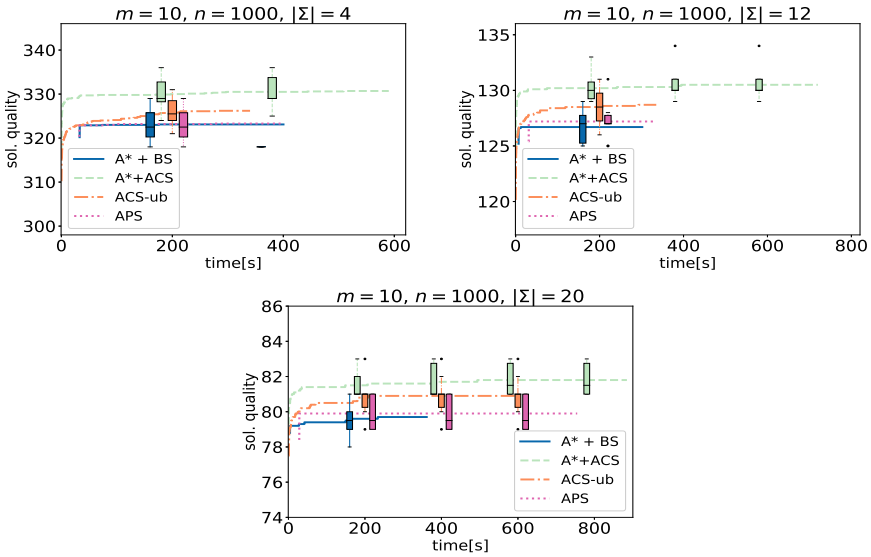

Figure 2: Instances with m =10 and n =1000.

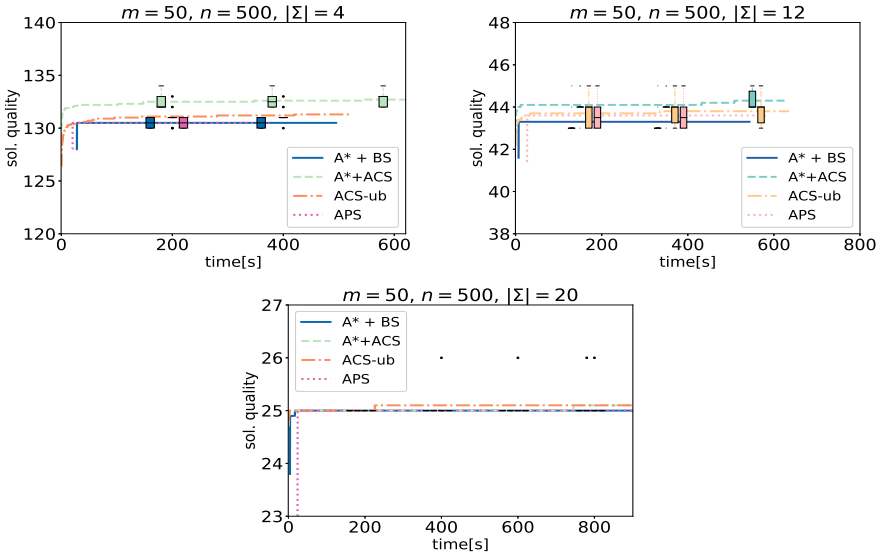

Figure 3: Instances with m =50 and n =500.

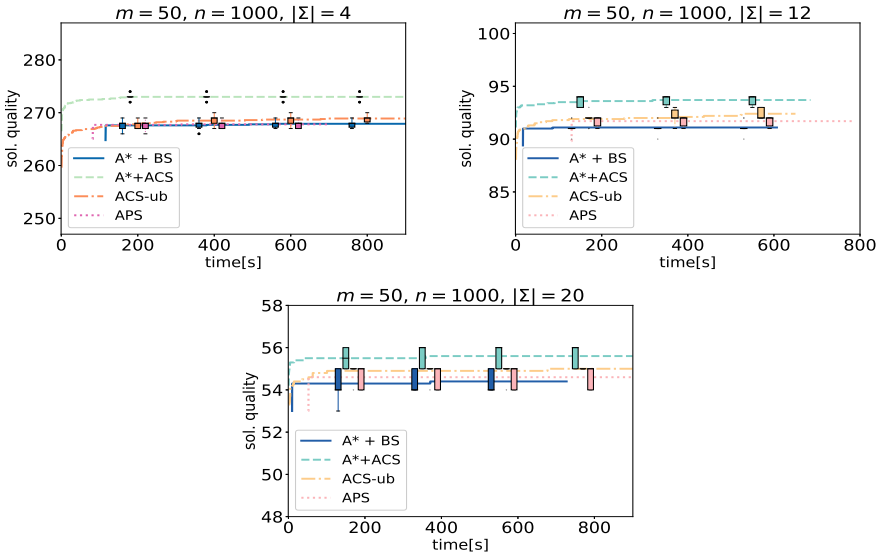

Figure 4: Instances with m =50 and n =1000.

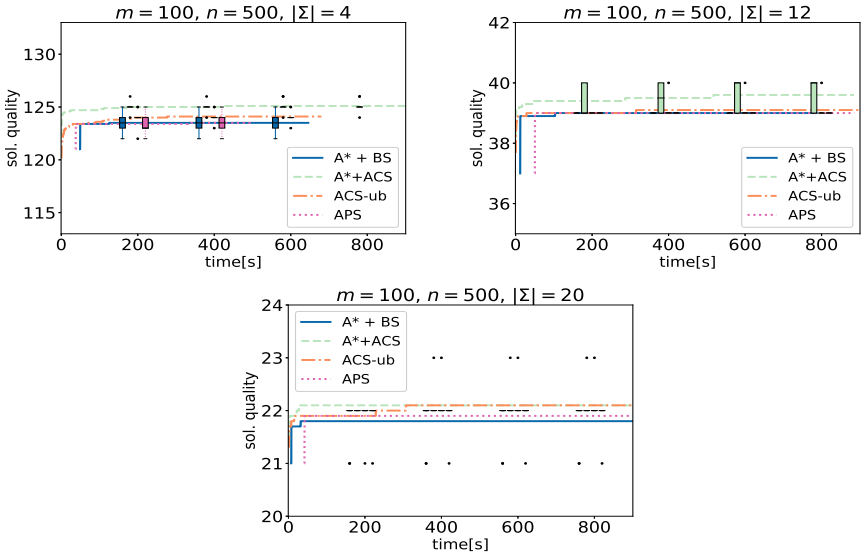

Figure 5: Instances with m =100 and n =500.

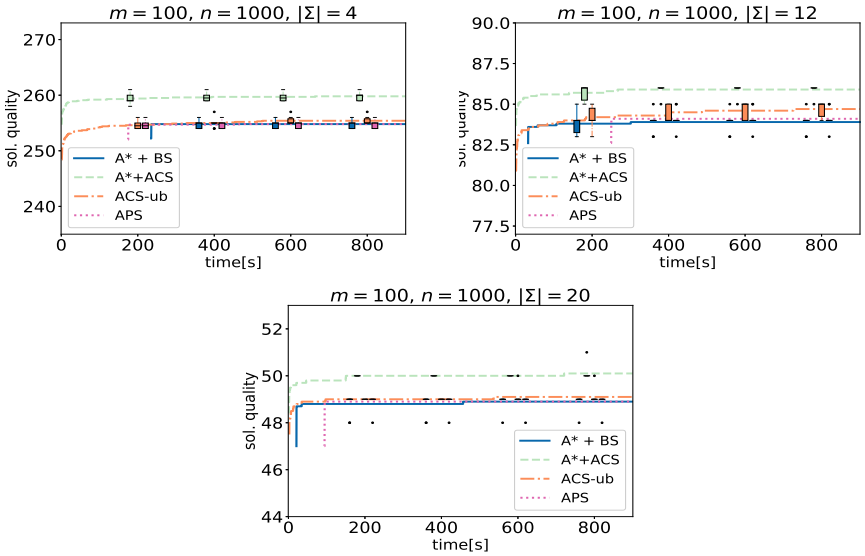

Figure 6: Instances with m =100 and n =1000.

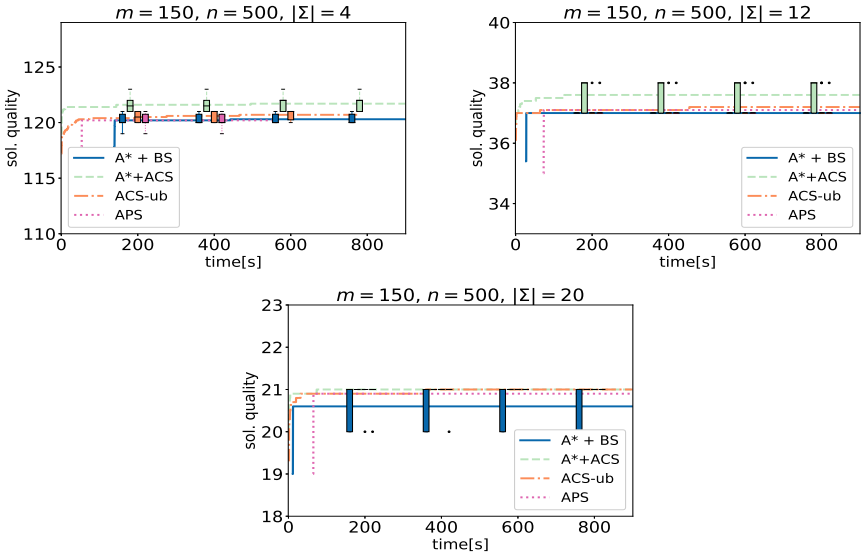

Figure 7: Instances with m =150 and n =500.

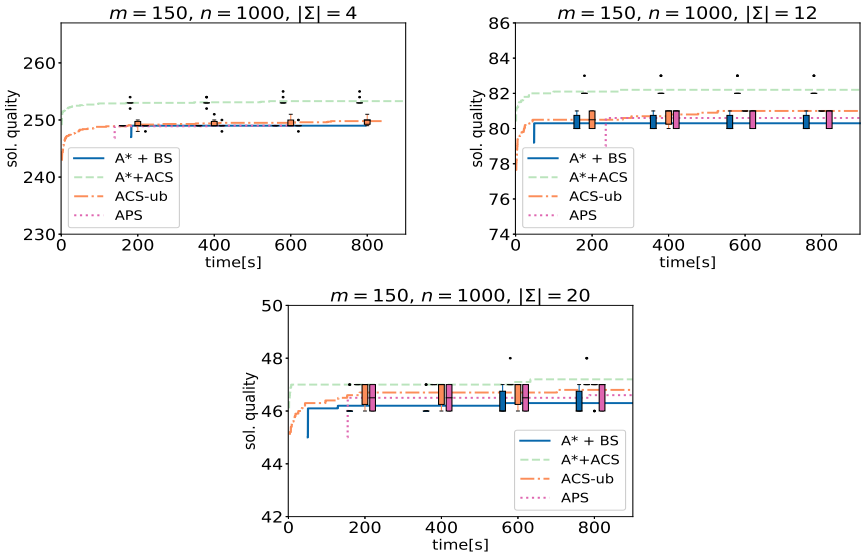

Figure 8: Instances with m =150 and n =1000.

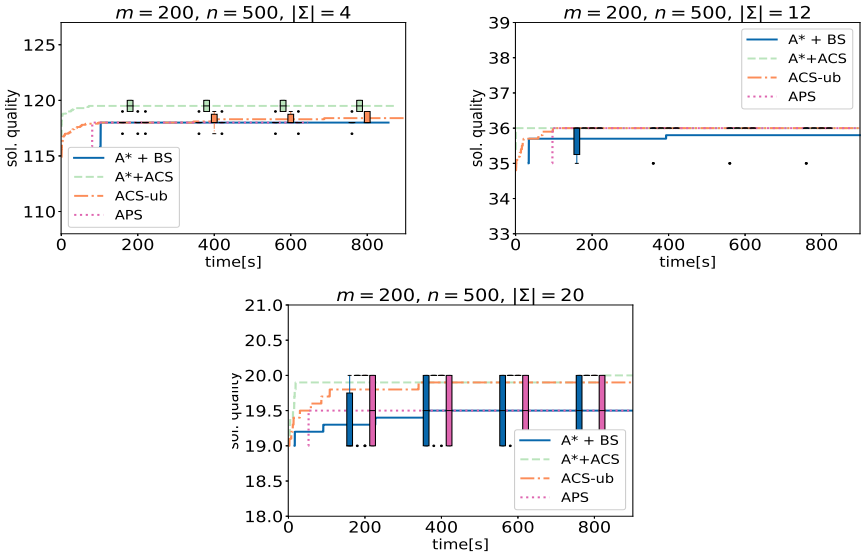

Figure 9: Instances with m =200 and n =500.

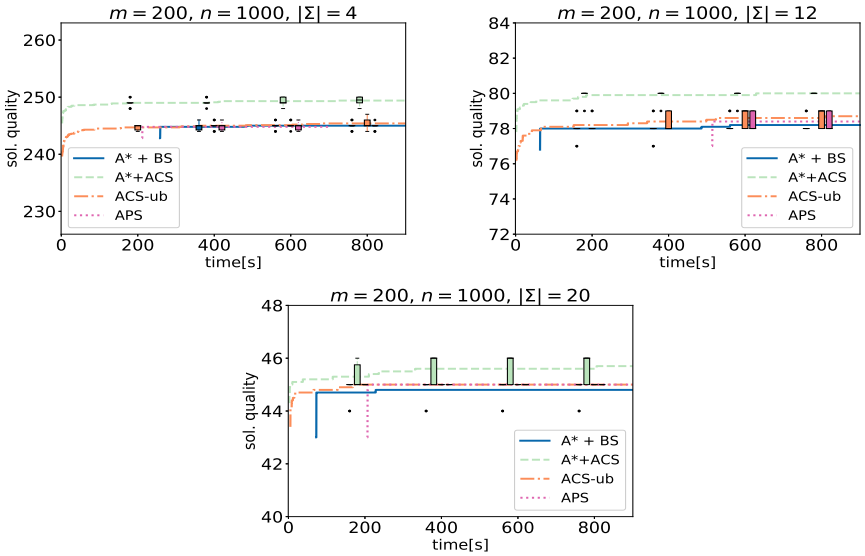

Figure 10: Instances with m =200 and n =1000.

2 Anytime plots of the algorithms showing the evolution of the obtained gaps

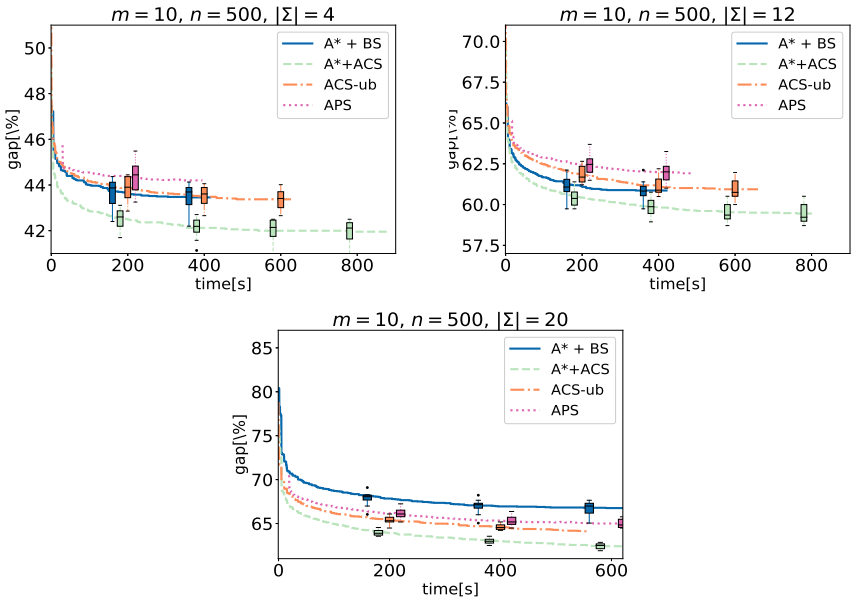

Figure 11: Instances with m =10 and n =500.

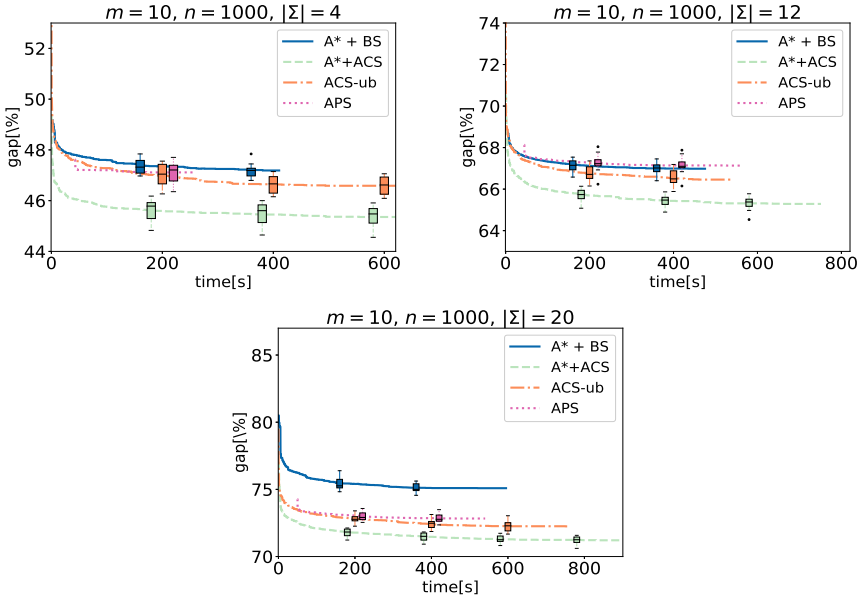

Figure 12: Instances with m =10 and n =1000.

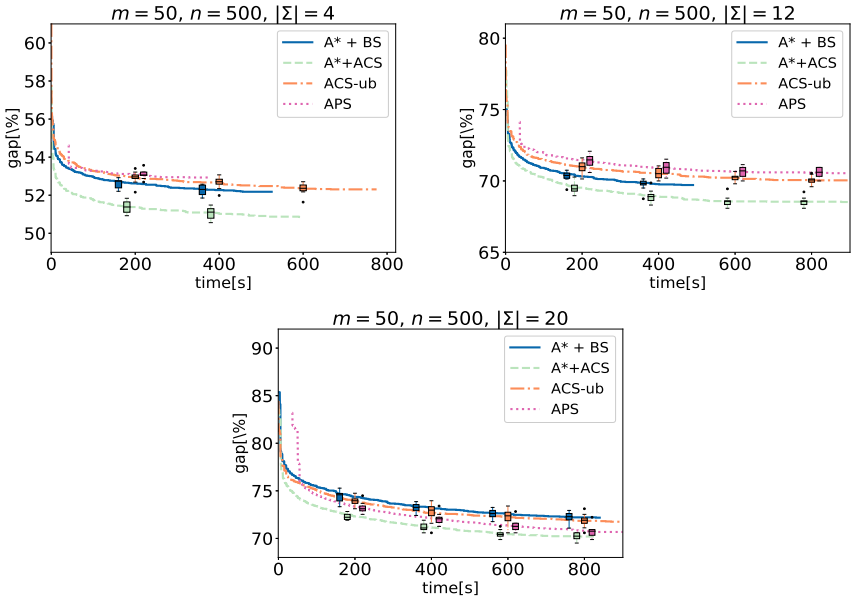

Figure 13: Instances with m =50 and n =500.

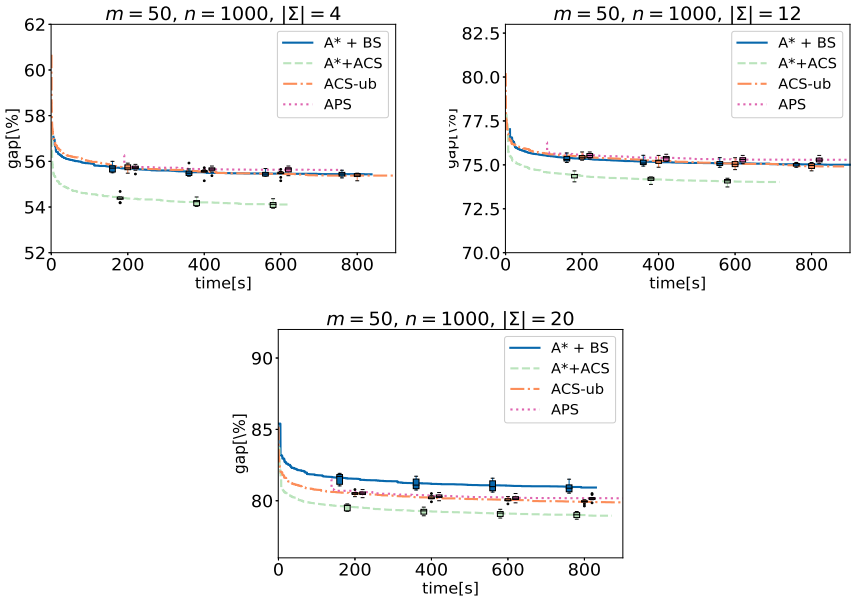

Figure 14: Instances with m =50 and n =1000.

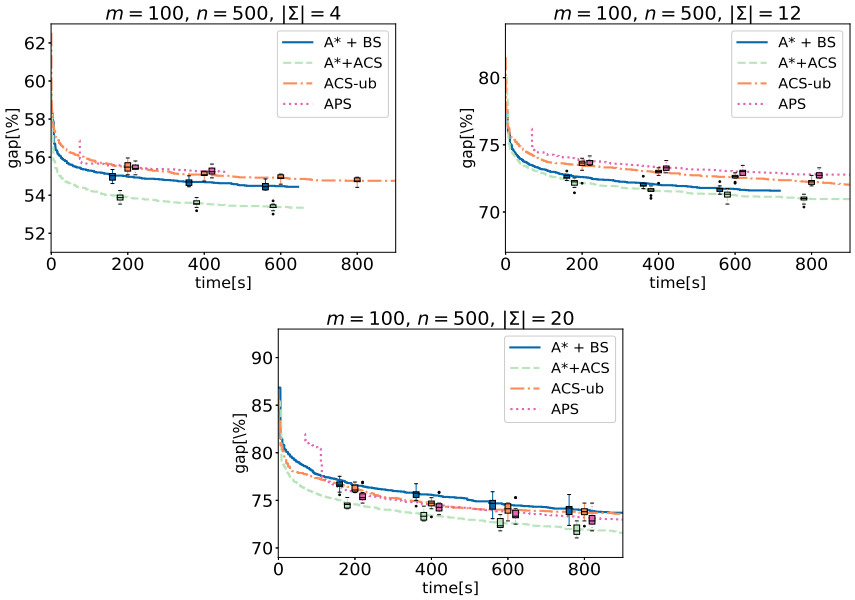

Figure 15: Instances with m =100 and n =500.

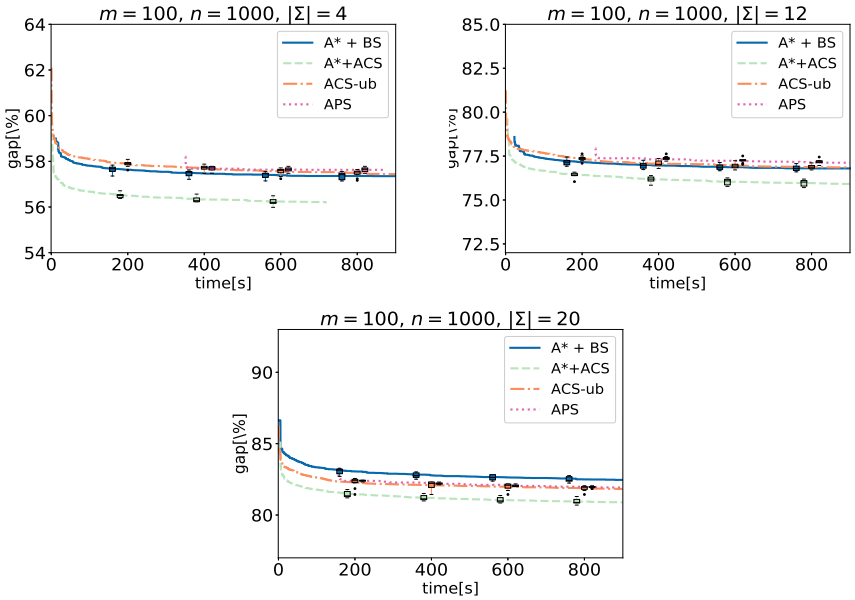

Figure 16: Instances with m =100 and n =1000.

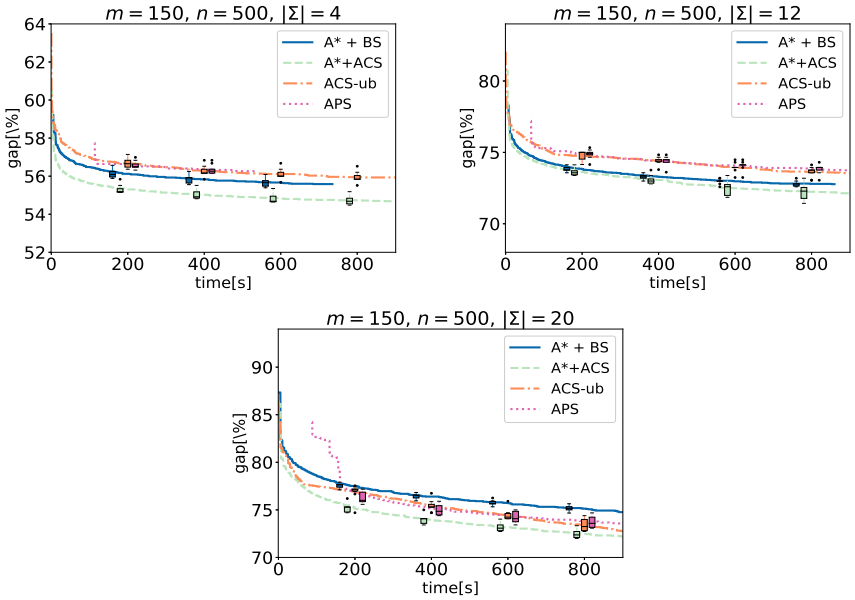

Figure 17: Instances with m =150 and n =500.

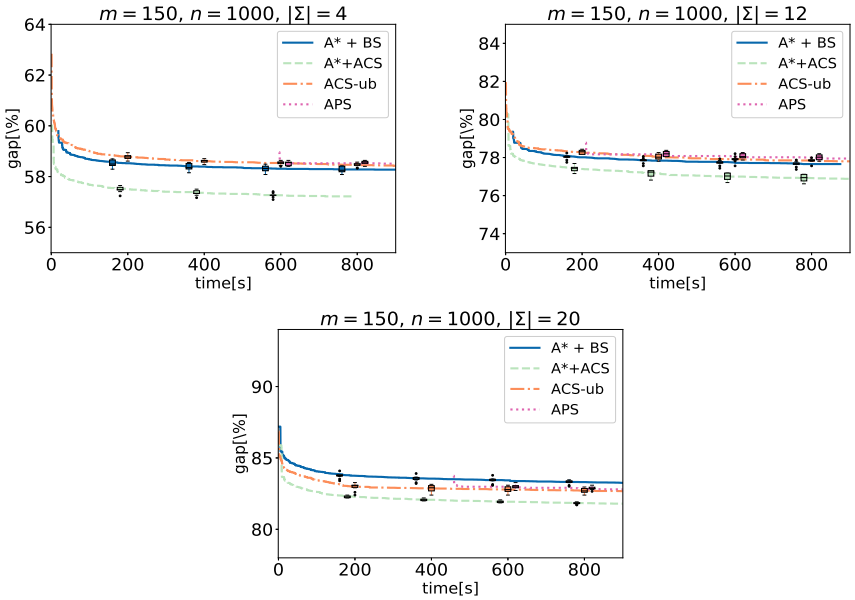

Figure 18: Instances with m =150 and n =1000.

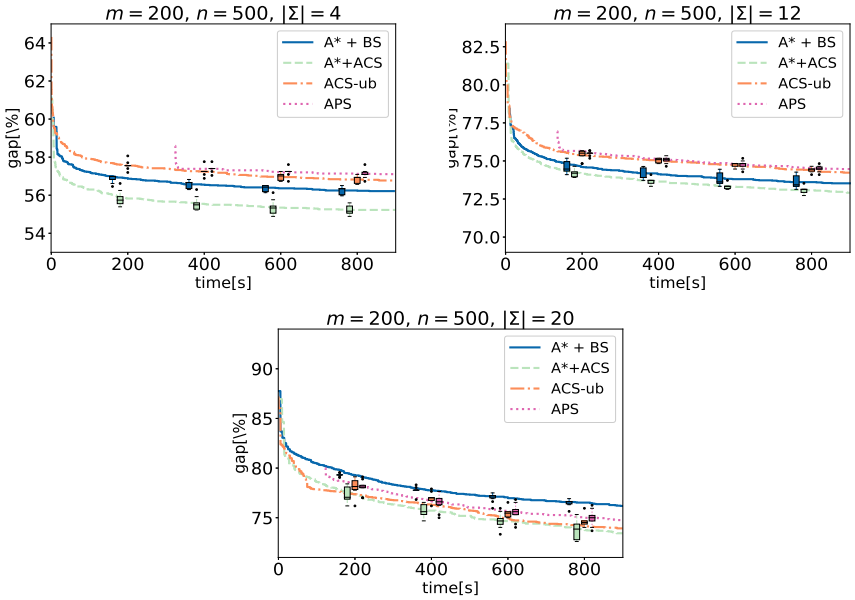

Figure 19: Instances with m =200 and n =500.

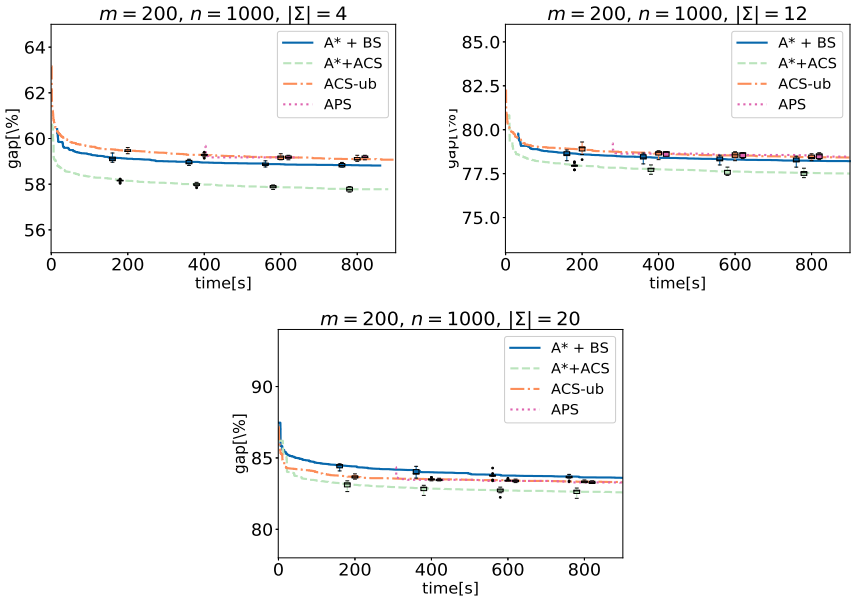

Figure 20: Instances with m =200 and n =1000.

3 The 2-LCPS Algorithms

The existing literature on approaches for solving the 2–LCPS problem is more of theoretical nature. In other words, a computational study comparing the different approaches has not been performed so far. In the following, we sketch the existing approaches, putting emphasis on the data structures we chose in order to obtain efficient implementations. Note that a straightforward *Constraint Programming* (CP) model has already been described in the appendix of the main paper. In the following we focus, therefore, on the description of the remaining three approaches: (1) a Dynamic Programming (DP) approach, (2) an algorithm which solves the *Maximum Nesting Depth Rectangle Structures* (MNDRS) problem from computational geometry to which the 2–LCPS problem can be reduced, and (3) the so-called CPSA approach which uses an automaton to solve the 2–LCPS problem.

3.1 Dynamic Programming Approach

Chowdhury et al. [1] presented a dynamic programming approach for solving the 2–LCPS problem. The idea is as follows. Let s_1 and s_2 be the input strings of equal length.[1] Let $lcps(i, j, k, l)$, $1 \leq i, j, k, l \leq n$, store the optimal length of the LCPS for substrings $s_1[i, j]$ and $s_2[k, l]$, that is, $lcps(i, j, k, l) := |\text{LCPS}(s_1[i, j], s_2[k, l])|$. Chowdhury proved that the following recursion leads to an optimal solution to the 2–LCPS problem:

$$lcps(i,j,k,l) = \begin{cases} 0, & \text{for } i > j \ \vee \ k > l \\ 1, & \text{for } i = j \ \wedge \ k \leq l \\ & \wedge \ s_1[i] = s_2[k] \\ 2 + lcps(i+1, j-1, k+1, l-1), & \text{for } i < j \ \wedge \ k < l \\ & \wedge \ s_1[i] = s_1[j] \\ & = s_2[k] = s_2[l] \\ \max(lcps(i+1,j,k,l), lcps(i,j-1,k,l), & \\ \quad lcps(i,j,k+1,l), lcps(i,j,k,l-1)), & \text{otherwise.} \end{cases}$$

The first one of the two main cases is obtained when $i < j$, $k < l$ and $s_1[i] = s_1[j] = s_2[k] = s_2[l]$. In this case, the value of $lcps(i, j, k, l)$ can easily be obtained as $lcps(i, j, k, l) := 2 + lcps(i + 1, j - 1, k + 1, l - 1)$. Otherwise, the value of $lcps(i, j, k, l)$ can be recursively calculated by solving the four smaller subproblems corresponding to the values of $lcps(i + 1, j, k, l)$, $lcps(i, j - 1, k, l)$, $lcps(i, j, k + 1, l)$, and $lcps(i, j, k, l - 1)$. The maximum of these values actually corresponds to $lcps(i, j, k, l)$.

The DP approach stores the solution of all possible subproblems in a 4-dimensional table of size $n \times n \times n \times n$, which implies a space complexity of $O(n^4)$. The DP recursion generates $O(n^4)$ distinct subproblems, in $O(1)$ time each, in a bottom-up manner, implying

[1] The case of input strings of different length can easily be transformed to the case of input strings with equal length.

a time complexity of $O(n^4)$ for the approach. Note that the value of an optimal solution is stored in $lcps(1, n, 1, n)$.

3.2 The MNDRS Approach

The *Maximum Nesting Depth Rectangle Structures* (MNDRS) problem is known from computational geometry and can be described as follows. Given a set of rectangles in the Euclidean plane, find a maximum-length sequence of these rectangles such that each rectangle in the sequence contains all following rectangles of the sequence. Chowdhury [1] mapped the 2–LCPS problem to the MNDRS problem by introducing for each pair $((i, k), (j, l))$ of index couples—such that $s_1[i] = s_1[j] = s_2[k] = s_2[l]$—a rectangle in $\mathbb{N}^2$ whose lower left corner is (i, k) and whose upper right corner is (j, l). Moreover, Chowdhury provided a sparse DP approach for solving the MNDRS problem by making use of 3-dimensional balanced range search trees.

Subsequently, Inenaga and Hyyrö [4] proposed an algorithm for solving the MNDRS problem which makes use of two simple but clever data structures. However, since their work is theoretically oriented, they did not deal with the question of how to implement the proposed algorithm in an efficient way. Therefore, the following description of this algorithm presents our own implementation.
The first one of the two data structures mentioned above is used to find—for each combination of a letter $c \in \Sigma$ and a rectangle R—a sub-rectangle R_c of maximum area which is strongly contained in R and whose indexes of the lower left and upper right vertices correspond to letter c. Finding such a rectangle can be done in constant time $O(1)$ by making use of two predecessor and two successor tables (which basically correspond to the Pred and Succ data structures used in the preprocessing of our A*, as described in the main paper). The second data structure is a space-efficient 4D–table used for checking whether or not a rectangle R is processed in the main recursion, as explained in the following. Our implementation uses hash maps to realize this data structure and thus answers a query in $O(1)$ expected time. Note that $O(\mathcal{R}^2)$ memory is needed to store for these data structures.
A recursion is used in order to calculate, for each rectangle, its so-called *nesting weight number*, which denotes the maximum length of a sequence of rectangles nested in R (including the rectangle R itself). The initial call of the recursion is applied to the virtual rectangle $R_V = (0, 0, n + 1, n + 1)$ and its final nesting number actually corresponds to the length of an optimal solution to the 2–LCPS problem. Moreover, all rectangles are initially marked as non-processed. Furthermore, a rectangle R will be marked as processed when its nesting weight number is calculated. The information about the nesting numbers is stored in a 4D-hash table. If an already processed rectangle is encountered in the recursion, its nesting number—as obtained from the 4D-hash table—is immediately returned. Let us suppose a rectangle R is currently being processed. For each letter c, the maximum sub-rectangle R_c is being determined (by making use of the Pred and Succ structures) and the recursion is repeatedly called for each R_c until one of the following conditions is encountered: (1) R_c is *empty*, returning value 0; (2) R_c is a *point* or a *line*,

returning value 1; or (3) R_c is already processed, returning the stored nesting number. Note that a point or a line correspond to a middle letter in the respective solution to the 2–LCPS problem. It can be shown that—concerning the expected asymptotic runtime—the MNDRS approach of Inenaga and Hyyrö for solving the 2–LCPS problem is by a factor of $|\Sigma|$ faster than the DP approach.

The approach of Inenaga and Hyyrö and the approach of Chowdhury et al. have the same memory complexity [4]. Moreover, Chowdhury's MNDRS approach is—with respect to the expected asymptotic runtime—slower than the DP approach in the case of input strings that were generated uniformly at random, where $\mathcal{R} = O(n^2)$. If the number of matchings is significantly lower, for example $\mathcal{R} = O(n^{1.5})$, the MNDRS approach of Chowdhury shows an advantage over the DP in terms of its running times [1]. Additionally, the data structures used in the approach of Inenaga and Hyyrö [4] are simpler—with respect to the ease of implementation—than the sophisticated data structures from computational geometry used in Chowdhury's MNDRS approach. In general, it is not expected that the MNDRS approach of Chowdhury has a significant advantage over the MNDRS approach of Inenaga and Hyyrö in the context of instances generated uniformly at random. For these reasons we decided to implement the algorithm from [4] for comparison purposes.

3.3 A Palindromic Subsequence Automaton Approach

In the following we describe the so-called *Common Palindromic Subsequence Automaton* (CPSA) approach from [2], highlighting the major adaptation we applied in order to obtain an efficient algorithm for solving the 2–LCPS problem. The algorithm is based on a so-called *Palindromic Subsequence Automaton* (PSA) for each input string, that is, a PSA M_1 for input string s_1 and a PSA M_2 for input string s_2. Each of these PSAs works on the space of the first halves of all palindromic subsequences of the corresponding input strings. The major idea for solving the 2–LCPS problem is based on the construction of a so-called intersecting automaton that connects M_1 and M_2.

In the following we describe the structure of a PSA for a string s. The automaton is denoted by $M(Q, \Sigma, \tau, w, F)$, where Q is a set of states, $\tau : Q \times \Sigma \mapsto Q$ is a transition function, $w : Q \times Q \mapsto \mathbb{N}_0$ is a weight function, and F is a set of final states. M has a state $q \in Q$ associated with each pair (i, j), $i, j \geq 1$, such that $s[i] = s^{\text{rev}}[j]$. The initial state of the automaton is defined by $q_0 = (0, 0)$. A transition $\tau(q_1, a) = q_2$ between two states $q_1 = (i', j')$ and $q_2 = (i'', j'')$ is possible if and only if $s[i''] = s^{\text{rev}}[j''] = a$ and there exist no positions k and l, $i' < k < i'' \wedge j' < l < j''$, matching the letter a. Note that the states can be seen as the nodes of a weighted directed acyclic graph, with q_0 as the root node. Moreover, existing transitions between states can be seen as the directed edges of this graph. A state q is therefore a partial LCPS solution that corresponds to a directed path from the root node to q. Note that a transition corresponds to the extension of a partial solution, either by one or by two letters. More specifically, if $i'' + j'' = |s| + 1$, the corresponding transition is an extension by a single letter and otherwise an extension by

two letters. The weight function of the PSA is defined accordingly:

$$w(q_1, q_2) = \begin{cases} 2, & \text{if } i'' + j'' < |s| + 1 \\ 1, & \text{if } i'' + j'' = |s| + 1 \\ 0, & \text{else.} \end{cases}$$

Each state can be considered as a final (accepted) state of the automaton, that is, $F = Q$. Any path p from the initial state q_0 to any final state—that is, $p = q_0 \cdots q_{r_i} \cdots q_{r_k}$ with $k \geq 0$—corresponds to the following palindromic subsequence s^p of s:

$$s^p = \begin{cases} s[i_{r_1}] \cdots s[i_{r_{k-1}}] s[i_{r_k}] \cdot (s[i_{r_1}] \cdots s[i_{r_{k-1}}] s[i_{r_k}])^{\text{rev}} & \text{if } i_k + j_k < |s| + 1, \\ s[i_{r_1}] \cdots s[i_{r_{k-1}}] \cdot s[i_{r_k}] \cdot (s[i_{r_1}] \cdots s[i_{r_{k-1}}])^{\text{rev}} & \text{if } i_k + j_k = |s| + 1. \end{cases}$$

Let $M_1 = (Q_1, \Sigma, \tau_1, w_1, F_1)$ be the PSA of s_1 and $M_2 = (Q_2, \Sigma, \tau_2, w_2, F_2)$ be the PSA for s_2, respectively. The intersecting automaton $M_{\text{isec}}(Q, \Sigma, \tau, w, F) = M_1 \cap M_2$, called *Common Palindromic Subsequence Automaton* (CPSA), is defined as follows. It has an initial state (root node) denoted by $q_{M_{\text{isec}}} = (q_0', q_0'')$, where $q_0' \in Q_1$ and $q_0'' \in Q_2$ are the root nodes of M_1 and M_2, respectively. In general, if $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ are the languages accepted by M_1 and M_2, the intersecting automaton M_{isec} will accept all words common to both languages, i.e., $\mathcal{L}(M_{\text{isec}}) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. A maximum path in the directed acyclic graph defined by M_{isec} corresponds to an optimal solution to the 2–LCPS problem.

The transition function τ of M_{isec} is defined as follows. If $q' = (q_1', q_2')$, $q'' = (q_1'', q_2'') \in Q \subseteq Q_1 \times Q_2$, then a transition between the nodes—that is, $\tau(q', a) = q''$ for some $a \in \Sigma$—exists if and only if $\tau_1(q_1', a) = q_1''$ and $\tau_2(q_2', a) = q_2''$. The weight corresponding to this edge is calculated as follows:

$$w(q', q'') = \begin{cases} 2 & \text{if } w_1(q_1', q_1'', a) = w_2(q_2', q_2'', a) = 2 \\ 1 & \text{if } w_1(q_1', q_1'', a) = 1 \vee w_2(q_2', q_2'', a) = 1 \\ 0, & \text{else.} \end{cases} \tag{1}$$

The final states $q \in F$ of M_{isec} are all states for which $q_1' \in F_1$ or $q_2' \in F_2$. In order to construct the intersection automaton M_{isec}, we start by adding the root node $q_{M_{\text{isec}}}$ to a queue Q' and Q. At each step, the top node $q = (q', q'')$ is taken from Q' and the outgoing edges E_1 of q' in M_1 and outgoing edges E_2 of q'' in M_2 are considered. All edges $e_1 = q'r_1' \in E_1$ and $e_2 = q''r_2' \in E_2$ for which $\tau_1(q', a_1) = r_1' \wedge \tau_2(q'', a_2) = r_2' \wedge a_1 = a_2$ will create a new state $r = (r_1', r_2')$ which is then added to Q' and to the set of states Q (if not already there). We implemented Q by means of a hash table in order to be able to efficiently check whether or not a state is already in Q. If state r is added to Q, an extension of functions τ and w of M_{isec} is generated for r by determining the corresponding weights for the newly created edge qr, as defined in (1). Afterwards, q is removed from the top of Q'. The procedure stops once Q' is empty. A detailed description of this process is provided in [2].

As mentioned above, M_{isec} also defines a directed acyclic graph, and for solving the corresponding 2–LCPS problem it is actually sufficient to find a maximum-length path in M_{isec}. This takes time $O(|Q|) = O(|Q_1| \cdot |Q_2|)$ when applying a topological sort to all nodes of Q. For this purpose, the authors of [2] construct a maximum-length automaton [3], which accepts all the subsequences of maximum length among the subsequences from $\mathcal{L}(M_{\text{isec}})$. The automaton is constructed in $O(|Q|)$ time by using a topological sort of the nodes in M_{isec} followed by removing all the transitions and states which are not part of any longest path from the initial state to a final state.

Since a main effort of the algorithm is actually to construct the CPSA, and as we are only interested in finding one optimal solution of possibly several ones, and as constructing the maximum-length automaton followed by solving the 2–LCPS problem is more time consuming than a direct application of the maximum-path algorithm to M_{isec}, we decided to simply use the maximum-path algorithm based on the topological sort of the states of the CPSA in order to solve the 2–LCPS problem, without the construction of a maximum-length automaton. This approach yields more directly one optimal solution of the 2–LCPS problem.

References

[1] S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, 129(4):329–340, 2014.

[2] M. M. Hasan, A. S. M. S. Islam, M. S. Rahman, and A. Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11(2):219–232, 2017.

[3] C. Iliopoulos, M. S. Rahman, M. Voráček, and L. Vagner. Finite automata based algorithms on subsequences and supersequences of degenerate strings. *Journal of Discrete Algorithms*, 8(2):117–130, 2010.

[4] S. Inenaga and H. Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129(C):11–15, 2018.