

Pattern Matching for Arc-Annotated Sequences

Jens Gramm

Universität Tübingen

and

Jiong Guo and Rolf Niedermeier

Friedrich-Schiller-Universität Jena

We study pattern matching for arc-annotated sequences. An $O(nm)$ time algorithm is given for the problem to determine whether a length m sequence with nested arc annotation is an arc-preserving subsequence (aps) of a length n sequence with nested arc annotation, called $\text{APS}(\text{NESTED}, \text{NESTED})$. Arc-annotated sequences and, in particular, those with nested arc annotation are motivated by applications in RNA structure comparison. Our algorithm generalizes results for ordered tree inclusion problems and it is useful for recent fixed-parameter algorithms for $\text{LAPCS}(\text{NESTED}, \text{NESTED})$, which is the problem of computing a longest arc-preserving common subsequence of two sequences with nested arc annotations. In particular, the presented dynamic programming methodology implies a quadratic-time algorithm for an open problem posed by Vialette.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.1 [Discrete Mathematics]: Combinatorics; J.3 [Life and Medical Sciences]: Biology and Genetics

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Arc-annotated sequences, dynamic programming, pattern matching, RNA secondary structure.

1. INTRODUCTION

Basic motivation. Pattern matching in strings is a core problem of computer science. It is of foundational importance in several application areas, perhaps the most recent one being computational biology [Gusfield 1997; Sankoff and Kruskal 1983]. Numerous versions of pattern matching and related problems occur in practice, ranging in difficulty from linear-time solvable to NP-hard. In this paper, we study a pattern matching problem that was originally motivated by RNA structure comparison and motif search, a topic that has recently received considerable attention [El-Mabrouk and Raffinot 2002; Evans 1999a; 1999b; Evans and Wareham 2001; Jiang et al. 2004; Lin et al. 2002; Vialette 2002]. Herein, we encounter a seemingly sharp border between (practically important) problem versions that we

An extended abstract of this paper appeared in *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2002)*, volume 2556 in LNCS, pages 182–193, Springer, 2002. The main work was done while all authors were with Universität Tübingen.

Research by J. Gramm was supported by the Deutsche Forschungsgemeinschaft (DFG), research project OPAL (optimal solutions for hard problems in computational biology), NI 369/2.

Research by J. Guo was partially supported by the Deutsche Forschungsgemeinschaft (DFG), Zentrum für Bioinformatik Tübingen (ZBIT), and Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4.

Authors' address: Jens Gramm, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany, e-mail: gramm@informatik.uni-tuebingen.de. Jiong Guo and Rolf Niedermeier, Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany, e-mail: {guo,niedermeier}@minet.uni-jena.de.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

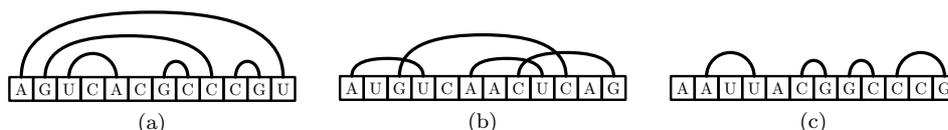
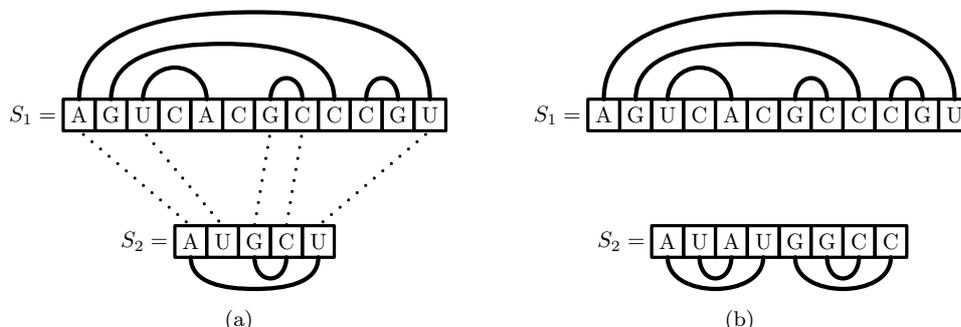


Fig. 1. Examples for an (a) nested, (b) crossing, and (c) chain arc annotation

Fig. 2. Input instances of $\text{APS}(\text{NESTED}, \text{NESTED})$. (a) Yes-instance, S_2 is an aps of S_1 . (b) No-instance, S_2 is *not* an aps of S_1

show to be efficiently solvable in quadratic time and slightly more general versions that turn out to be NP-complete.

Problem definition. We study pattern matching for *arc-annotated* sequences. Due to the problem motivation from computational biology, we use the terms “string” and “sequence” in a synonymous way. Note, however, that we clearly distinguish between the terms “substring” and “subsequence,” the latter being not necessarily contiguous and, thus, being the more general term. For a sequence S , an *arc annotation* of S is a set of unordered pairs of positions in S . Each pair of positions represents an *arc* and the two positions denote the two *endpoints* of the arc. We distinguish five kinds of arc annotations which are informally presented as follows (for a more formal definition, see Sect. 2). An arc annotation is *nested* if no two arcs share an endpoint and no two arcs cross each other (see Fig. 1(a)). Less restrictively, an arc annotation is *crossing* if no two arcs share an endpoint (Fig. 1(b)). A nested arc annotation is of type *chain* if its nesting depth is one (Fig. 1(c)). Finally, the term *plain* refers to sequences without arcs and the term *unlimited* refers to a completely unrestricted arc annotation. Lin et al. [2002] argue that nested arc annotations are the biologically most important of these types. Given two arc-annotated sequences S_1 and S_2 , S_2 is an *arc-preserving subsequence* (*aps*, for short) of S_1 if S_2 (including its arc annotation) can be obtained from S_1 by deleting all but $|S_2|$ letters from S_1 —when deleting a letter at position i then *all* arcs with endpoint i are deleted. Now, we are ready to define the pattern matching problems studied in this work, namely the ARC-PRESERVING SUBSEQUENCE problem for various types of arc annotations:

$\text{APS}(\text{TYPE1}, \text{TYPE2})$

Input: Two arc-annotated sequences S_1 , $|S_1| = n$, and S_2 , $|S_2| = m$ with $m \leq n$, where the arc annotations of S_1 and S_2 are of kind TYPE1 and TYPE2, respectively.

Question: Does S_2 occur as an arc-preserving subsequence in S_1 ?

Fig. 2(a) gives an example for a yes-instance and Fig. 2(b) gives an example for a no-instance of $\text{APS}(\text{NESTED}, \text{NESTED})$. Clearly, the problem specification only makes sense if TYPE1 comprises TYPE2.

Results. Our main result is that $\text{APS}(\text{NESTED}, \text{NESTED})$ can be solved in $O(nm)$ time. Table I surveys known and new results for various types of APS. In addition,

APS(...)	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	NP-complete [Evans 1999a; 1999b]				
CROSSING	—	NP-complete [Evans 1999a; 1999b]	NP-complete	?	
NESTED	—	—	$O(nm)$		

Table I. Survey of computational complexity for different versions of $\text{APS}(\text{TYPE1}, \text{TYPE2})$ where rows and columns correspond to possible choices of TYPE1 and TYPE2 , respectively. Most NP-completeness results easily follow from results of Evans [Evans 1999a; 1999b] for the corresponding LAPCS problems, or, at least, can be proven in a similar way as there; we omit the details here. The $O(nm)$ time algorithms are described in this paper. The complexity of $\text{APS}(\text{CROSSING}, \text{PLAIN})$ remains unclassified. We mention in passing that $\text{APS}(\text{CHAIN}, \text{PLAIN})$ can be solved in $O(n)$ time

we study a *modified* version of $\text{APS}(\text{UNLIMITED}, \text{NESTED})$ where the alphabet is unary, each base has to be endpoint of at least one arc, and we determine whether S_2 is an *arc-substructure* (*ast*) of S_1 . The term arc-substructure will be introduced in Sect. 2. Deriving an $O(nm)$ time algorithm for this case, we answer an open question of Vialette [2002]. Observe that in general $\text{APS}(\text{UNLIMITED}, \text{NESTED})$ is NP-complete (see Table I).

Relations to previous work. There are basically two lines of research our results refer to. The first one is that of similar pattern matching problems and the other one is that of results (mostly NP-completeness, approximation, and fixed-parameter tractability) for the more general LONGEST ARC-PRESERVING COMMON SUBSEQUENCE problem (LAPCS). As to directly related pattern matching problems, the work perhaps most closely connected to ours is that of Vialette [2002]. He studied very similar pattern matching problems also motivated by RNA secondary structure analysis. Although most of our results do not directly compare to his ones (because of the somewhat different models), our approach leads to an answer of one of his open questions, posed in [Vialette 2002], asking for the algorithmic complexity (NP-complete vs. solvable in polynomial time) of the aforementioned modified version of $\text{APS}(\text{UNLIMITED}, \text{NESTED})$.¹ The corresponding question for crossing instead of nested structure patterns was very recently also shown to be solvable in polynomial time [Gramm 2004], illustrating the use of this kind of pattern matching algorithms in protein structure analysis.

Moreover, our work generalizes results achieved in the context of structured text databases for the ordered tree inclusion problem [Kilpeläinen 1992; Kilpeläinen and Mannila 1995]. Kilpeläinen and Mannila already presented quadratic-time algorithms for a strict special case of $\text{APS}(\text{NESTED}, \text{NESTED})$. In their case, sequence information is not taken into account and we can easily derive their problem from ours. Bafna et al. [1995], among other things, considered the corresponding arc-preserving *substring* (instead of subsequence) problems. For instance, they give a quasilinear-time algorithm for the arc-annotated substring problem where both sequences have crossing arc annotations. By way of contrast, $\text{APS}(\text{CROSSING}, \text{CROSSING})$ is NP-complete.

As to LAPCS problems, we only briefly mention that most problems in that context become NP-complete [Evans 1999a; 1999b; Lin et al. 2002] as, in particular, $\text{LAPCS}(\text{NESTED}, \text{NESTED})$. That is why researchers focused on approximation (factor 2) [Jiang et al. 2004] and exact fixed-parameter algorithms [Alber et al. 2004]. Notably, our algorithm solving $\text{APS}(\text{NESTED}, \text{NESTED})$ can be used as a subprocedure to handle easy cases in the exact exponential-time algorithm of [Alber et al. 2004]. Thus, a speed-up of this fixed-parameter algorithm can

¹Note that independently of this work, Vialette [2004] has meanwhile also presented a polynomial-time algorithm for the mentioned problem; this algorithm has, however, a much higher running time, namely $O(mn^3 \log n)$, than the $O(n \log n + nm)$ -time algorithm presented here.

be achieved. Moreover, it forms the basis of a second fixed-parameter algorithm for LAPCS(NESTED,NESTED) with the “dual parameterization” (see [Alber et al. 2004] for details). Similar applications seem to be possible in the approximation context. Finally, note that for easier types of arc annotations such as in LAPCS(NESTED,CHAIN) $O(nm^3)$ time dynamic programming algorithms have been developed [Jiang et al. 2004]—for the special case APS(NESTED,CHAIN) beaten by our $O(nm)$ algorithm.

Structure of the paper. The results of the work are developed in a stepwise manner. We begin with an algorithm for APS(NESTED,PLAIN), generalize it to APS(NESTED,CHAIN), and further generalize it to APS(NESTED,NESTED). In this way, more and more new ideas and technical subtleties are introduced. Finally, turning our attention to modified versions of APS(UNLIMITED,NESTED), we apply our dynamic programming techniques elaborated so far to answer an open question of Vialette [2002] in the context of detecting RNA structure motifs.

2. PRELIMINARIES

For a sequence S of length $|S| = n$, we use $S[i]$ with $1 \leq i \leq n$ to refer to the *base* at position i in S and we use $S[i_1, i_2]$ with $1 \leq i_1, i_2 \leq n$ to denote the substring of S starting with $S[i_1]$ and ending with $S[i_2]$ (and the empty string if $i_2 < i_1$). An *arc annotation* (or *arc set*) A of S is a set of pairs of numbers from $\{1, 2, \dots, n\}$. Each pair $(i_l, i_r) \in A$ satisfies $i_l < i_r$ and connects the two bases $S[i_l]$ and $S[i_r]$ at positions i_l and i_r in S by an arc. In most cases, we will require that no two arcs share an endpoint, i.e., $(i_l, i_r), (i'_l, i'_r) \in A$ only if all $i_l, i_r, i'_l,$ and i'_r are pairwise distinct. The *arc-annotated substring* $S[i_1, i_2]$, $1 \leq i_1, i_2 \leq n$, of the arc-annotated sequence (S, A) has as arc set those arcs $(i_l, i_r) \in A$ for which $i_1 \leq i_l < i_r \leq i_2$.

Let S_1 and S_2 be two sequences with arc sets A_1 and A_2 , respectively. If $S_1[i] = S_2[j]$ for $1 \leq i \leq |S_1|$ and $1 \leq j \leq |S_2|$ we refer to this as a *base match*. If S_2 is a subsequence of S_1 then it induces a one-to-one *mapping* M from $\{1, 2, \dots, |S_2|\}$ to a subset of $\{1, 2, \dots, |S_1|\}$, given by

$$M = \{ \langle j, i_j \rangle \mid 1 \leq j \leq |S_2|, 1 \leq i_j \leq |S_1|, S_2[j] = S_1[i_j], \text{ and } i_j < i_{j+1} \}.$$

We say that S_2 is an *arc-preserving subsequence* (*aps*) of S_1 if there is a mapping M that preserves the arcs induced by M , i.e., for all $\langle j, i_j \rangle, \langle j', i_{j'} \rangle \in M$: $(j, j') \in A_2 \iff (i_j, i_{j'}) \in A_1$.

In this paper, we study the ARC-PRESERVING SUBSEQUENCE problem (APS): Given an arc-annotated sequence S_1 and an arc-annotated pattern sequence S_2 , the question is to determine whether S_2 is an aps of S_1 . We use $|S_1| := n$ and $|S_2| := m$. Depending on the arc annotations of S_1 and S_2 , several versions APS(TYPE1, TYPE2) can be defined where the arc annotation of S_1 is TYPE1 and the arc annotation of S_2 is TYPE2. An arc set has a *nested* arc annotation if no two arcs share an endpoint and no two arcs cross each other, i.e., for all $(i_l^1, i_r^1), (i_l^2, i_r^2) \in A$ it holds that $i_l^2 < i_l^1 < i_r^2$ iff $i_l^2 < i_r^1 < i_r^2$. In a *plain* arc annotation, the sequence has no arcs at all, *chain* arc annotations have nested structure with nesting depth one, *crossing* refers to arc annotations where the only requirement is that no two arcs share an endpoint, and *unlimited* refers to completely arbitrary arc annotations.

Given two arc-annotated sequences S_1 and S_2 , and substrings $S_1[i_1, i_2]$ and $S_2[j_1, j_2]$ of S_1 and S_2 , respectively, we define the *best aps-match* of $S_2[j_1, j_2]$ in $S_1[i_1, i_2]$ as the rightmost position j , $j_1 \leq j \leq j_2$, in S_2 such that $S_2[j_1, j]$ is an arc-preserving subsequence of $S_1[i_1, i_2]$ (or $j_1 - 1$ if no such j exists).

Let S be an arc-annotated sequence with arc annotation A . We say arc (i_l, i_r) is *inside* arc (i'_l, i'_r) if $i'_l < i_l < i_r < i'_r$. An *innermost* arc has no other arc inside and an *outermost* arc is not inside any other arc. For each arc $(i_l, i_r) \in A$, we define a set $S^{(i_l, i_r)}$ which contains the positions of the bases that are inside arc (i_l, i_r) but

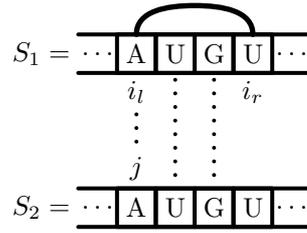


Fig. 3. Computing a best aps-match in an APS(NESTED, PLAIN) instance: Matching both $S_1[i_l]$ and $S_1[i_r]$ to bases in S_2 , e.g., $S_2[j]$ and $S_2[j+3]$, would contradict the arc-preserving property since $(i_l, i_r) \in A_1$ but $(j, j+3) \notin A_2$. Therefore, the best aps-match of $S_2[j, m]$ in $S_1[i_l, i_r]$ is only $j+2$ (dotted lines indicate base matches)

not inside any arcs that are inside (i_l, i_r) :

$$S^{(i_l, i_r)} = \{ i \mid i_l < i < i_r \text{ but not } i_l < i'_l < i < i'_r < i_r \text{ for any } (i'_l, i'_r) \in A \}.$$

If A has a *nested* arc annotation then the sets $S^{(i_l, i_r)}$ for different arcs are disjoint. We define S^0 as the set of endpoints of the outermost arcs in A and of positions of all bases which are not inside any arcs in A . From this definition, we conclude that for nested A the sets S^0 and $S^{(i_l, i_r)}$ for all $(i_l, i_r) \in A$ partition S :

OBSERVATION 2.1. For arc-annotated sequence S with nested arc annotation A , we have $|S| = |S^0| + \sum_{(i_l, i_r) \in A} |S^{(i_l, i_r)}|$. \square

Under the restriction that the alphabet is unary, motivated by considerations of Vialette [2002], we define a new problem which is related to APS. We assume arc-annotated sequences (S_1, A_1) and (S_2, A_2) such that every base in S_1 and every base in S_2 is an endpoint of an arc. We say that S_2 is an *arc-substructure* (*ast*) of S_1 if there is a mapping M such that arcs in A_2 are mapped to arcs in A_1 , i.e., for all $\langle j, i_j \rangle, \langle j', i_{j'} \rangle \in M$: $(j, j') \in A_2 \implies (i_j, i_{j'}) \in A_1$. In comparison with the definition of aps, the bases in S_1 which are matched to some bases in S_2 can be connected by additional arcs not present in S_2 —this is not allowed for an aps. Then, the ARC-SUBSTRUCTURE problem (AST) is to determine whether S_2 is an ast of S_1 . Analogously to APS, we can define $\text{AST}(\text{TYPE1}, \text{TYPE2})$, where TYPE1 and TYPE2 denote the arc annotations of S_1 and S_2 , respectively. In this paper, we consider the AST problem where TYPE1 is unlimited and TYPE2 is nested.

3. APS(NESTED, PLAIN)

An instance of APS(NESTED, PLAIN) is given by (S_1, A_1) and an arc-annotated pattern sequence (S_2, A_2) , where A_1 is a nested arc annotation while there is no arc in S_2 , i.e., $A_2 = \emptyset$. Compared to the pattern matching of two sequences without arc annotation, here, we have to guarantee the arc-preserving property, i.e., we cannot match both endpoints of an arc in A_1 to two bases in S_2 . Consider the small example shown in Fig. 3 where there are two substrings of the input sequences. The two substrings have identical sequence information except that there is an arc in the first substring. Thus, we can match at most three bases of the second substring to the bases of the first substring leaving either the first A or the last U unmatched. Furthermore, we cannot decide which of the two bases should be matched locally without the information about other parts of the both sequences. However, observe that, for a fixed *innermost* arc (i_l, i_r) in A_1 and a fixed position j in S_2 as in the small example in Fig. 3, we can easily determine the maximum length substring of S_2 starting at j which is an aps of $S_1[i_l, i_r]$, i.e., the best aps-match of $S_2[j, m]$ in $S_1[i_l, i_r]$. The best aps-match of this example is then $j+2$. Consider now an arc (i_l, i_r) which has some arcs inside of it. If we have already all information about the best aps-matches corresponding to the arcs inside of it, then, based on this

information, it is also easy to compute the best aps-matches of $S_2[j, m]$ in $S_1[i_l, i_r]$ for arbitrary $1 \leq j \leq m$. For the bases outside all arcs in S_1 , the normal left-to-right approach for pattern match applies. Hence, in the following, we present a dynamic programming algorithm solving APS(NESTED, PLAIN) which is based on the above idea, processing the arcs in A_1 from inside to outside and storing the best aps-matches in a dynamic programming table. The presentation is organized as follows. Firstly, we define the employed dynamic programming table. Secondly, we show that the best aps-match of an S_2 substring in an S_1 substring can be computed efficiently under the assumption that all best aps-matches corresponding to the arcs of the S_1 substring are already computed and stored in the table. Employing this computation of best aps-matches, we, thirdly, show how to fill the entries of the dynamic programming table. Finally, we present the resulting algorithm in an overview.

Dynamic programming table. We construct a dynamic programming table T of size $|A_1| \cdot m$. Each arc in A_1 corresponds to a row of this table and each position of S_2 corresponds to a column. We refer to the table entries corresponding to an arc $(i_l, i_r) \in A_1$ by $T(i_l, j)$, where j is an arbitrary position in S_2 . Entry $T(i_l, j)$ is defined to contain the best aps-match of $S_2[j, m]$ in $S_1[i_l, i_r]$.

Computing best aps-matches. Assume that for two arc-annotated substrings $S_1[i_1, i_2]$ and $S_2[j_1, j_2]$ of S_1 and S_2 all entries of T corresponding to arcs in the arc-annotation of $S_1[i_1, i_2]$ have already been computed. Then, we show in Fig. 4 how their best aps-match can be computed in form of a function maxaps .² The computation of maxaps is based on an exhaustive case distinction on the first bases in both sequences, namely $S_1[i_1]$ and $S_2[j_1]$. We distinguish whether $S_1[i_1] = S_2[j_1]$ (only in this case they can be matched) and whether $S_1[i_1]$ is an endpoint of an arc (note that if $S_1[i_1]$ is an endpoint of an arc, then it is a left endpoint since i_1 is the first position in the arc-annotated substring $S_1[i_1, i_2]$). In each of the cases, we invoke an appropriate recursive call of maxaps after either $S_1[i_1]$ and $S_2[j_1]$ have been matched or at least one position in S_1 has been skipped. It is essential that, due to the arc-preserving property and since S_2 has no arcs, we can, for $(i_l, i_r) \in A_1$, match only $S_1[i_l]$ or $S_1[i_r]$ with a base in S_2 , but not both; this is illustrated in Fig. 3. The central case of function maxaps applies when $S_1[i_1]$ is the left endpoint of an arc (i_l, i_r) , i.e., $i_1 = i_l$. Then, we can use the best aps-match for $S_2[j_1, j_2]$ in $S_1[i_l, i_r]$ which has already been computed and is stored in $T(i_l, j_1)$: This allows us to skip $S_1[i_l, i_r]$ and $S_2[j_1, T(i_l, j_1)]$. For further details refer to Fig. 4.

Computing entries of the dynamic programming table. Now, we explain how function maxaps is used to fill the entries of Table T and to solve APS(NESTED, PLAIN) for the given instance. As stated before, we process the arcs of A_1 in the order of their right endpoints, i.e., we process the arc set from inside to outside and from left to right. The nested arc annotation of A_1 guarantees that, when processing an arc $(i_l, i_r) \in A_1$, all arcs $(i'_l, i'_r) \in A_1$ inside (i_l, i_r) , i.e., $i_l < i'_l < i'_r < i_r$, have already been processed. An entry of table T , corresponding to $(i_l, i_r) \in A_1$ and $1 \leq j \leq m = |S_2|$, is computed by

$$T(i_l, j) := \max\{\text{maxaps}(S_1[i_l, i_r - 1], S_2[j, m]), \text{maxaps}(S_1[i_l + 1, i_r], S_2[j, m])\}.$$

In computing $T(i_l, j)$, we match a longest possible substring starting at $S_2[j]$ either to $S_1[i_l, i_r - 1]$ or to $S_1[i_l + 1, i_r]$. This way of computing $T(i_l, j)$ is motivated by the arc-preserving property: Since there are no arcs in S_2 , we can match, for an arc in S_1 , either only its left endpoint or only its right endpoint to a base in S_2 (or none of them). Note that when computing $T(i_l, j)$ all arcs inside of (i_l, i_r) have already been computed and, thus, the computation of the function maxaps is well-defined.

²Note that, although not explicitly stated, here and in the following the positions i_1, i_2 and j_1, j_2 also have to be considered as parameters of maxaps .

Function $\text{maxaps}(S_1[i_1, i_2], S_2[j_1, j_2])$
Input: Arc-annotated substrings $S_1[i_1, i_2]$ and $S_2[j_1, j_2]$.
Output: Best aps-match of $S_2[j_1, j_2]$ in $S_1[i_1, i_2]$.
Global: Table entries $T(i_l, j)$ for $(i_l, i_r) \in A_1$ with $i_1 \leq i_l < i_r \leq i_2$ and $j_1 \leq j \leq j_2$ containing the best aps-match of $S_2[j, j_2]$ in $S_1[i_l, i_r]$.

Method:

```

/* Recursion stops... */
if  $i_1 > i_2$  or  $j_1 > j_2$  then /* ...if  $S_1$  or  $S_2$  are empty,... */
    return  $j_1 - 1$ ;
else if  $i_1 = i_2$  then /* ...if  $S_1$  has only one base,... */
    return  $\left\{ \begin{array}{l} j_1 \text{ if } S_1[i_1] = S_2[j_1] \\ j_1 - 1 \text{ otherwise} \end{array} \right\}$ ;
else if  $i_1 < i_2$  and  $j_1 = j_2$  then /* ...or if  $S_2$  has only one base. */
    return  $\left\{ \begin{array}{l} j_1 \text{ if there is } i_1 \leq i \leq i_2 \text{ with } S_1[i] = S_2[j_1] \\ j_1 - 1 \text{ otherwise} \end{array} \right\}$ ;
else /* If  $i_1 < i_2$  and  $j_1 < j_2$  then recursion continues: */
    if  $S_1[i_1]$  is not an endpoint of an arc then
        return  $\left\{ \begin{array}{l} \text{maxaps}(S_1[i_1 + 1, i_2], S_2[j_1 + 1, j_2]) \text{ if } S_1[i_1] = S_2[j_1] \\ \text{maxaps}(S_1[i_1 + 1, i_2], S_2[j_1, j_2]) \text{ otherwise} \end{array} \right\}$ ;
    else /*  $S_1[i_1]$  is left endpoint of arc  $(i_l, i_r) \in A_1$ , i.e.,  $i_1 = i_l$  */
        return  $\text{maxaps}(S_1[i_r + 1, i_2], S_2[T(i_l, j_1) + 1, j_2])$ ;
    end if
end if

```

Fig. 4. Recursive definition of maxaps for arc-annotated sequence (S_1, A_1) with nested A_1 and sequence S_2 with plain arc annotation

Algorithm aps_np
Input: Sequence S_1 with nested arc annotation
and pattern sequence S_2 without arcs;

Method:

```

array of int  $T[n][m]$ ;
/***** Phase 1 *****/
for each  $(i_l, i_r) \in A_1$  (ordered by their right endpoints) do
    for  $j = 1$  to  $m$  do
         $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps}(S_1[i_l, i_r - 1], S_2[j, m]), \\ \text{maxaps}(S_1[i_l + 1, i_r], S_2[j, m]) \end{array} \right\}$ 
    end for
end for

/***** Phase 2 *****/
if  $\text{maxaps}(S_1[1, n], S_2[1, m]) = m$ 
    then print ' $S_2$  is an aps of  $S_1$ ';
    else print ' $S_2$  is not an aps of  $S_1$ ';
end if

```

Fig. 5. Outline in pseudo-code of the algorithm that solves $\text{APS}(\text{NESTED}, \text{PLAIN})$

Resulting algorithm. The resulting algorithm to solve $\text{APS}(\text{NESTED}, \text{PLAIN})$ is divided into two main phases. The first phase computes the table entries corresponding to arcs in A_1 ordered by the arcs' right endpoints. When processing an arc $(i_l, i_r) \in A_1$, we use the table entries corresponding to the arcs (i'_l, i'_r) directly inside (i_l, i_r) , i.e., there is no arc which is inside (i_l, i_r) and which has (i'_l, i'_r) inside of it. The second phase deals with those parts of S_1 which are outside all arcs or

the left endpoints of outermost arcs, namely, it computes $\text{maxaps}(S_1[1, n], S_2[1, m])$. Here, we use the table entries corresponding to the outermost arcs in A_1 . Depending on the return value of the function, we determine whether S_2 is an arc-preserving subsequence of S_1 . An outline of the whole algorithm in pseudo-code is given in Fig. 5.

The correctness of the presented algorithm follows directly from the way the arcs of S_1 are processed, the definition of maxaps , and how we compute a table entry.

Running Time Analysis

In order to determine the running time of algorithm `aps_np` in Fig. 5 itself, we firstly consider the time taken by one call of maxaps . Note that, in Fig. 5, a call of $\text{maxaps}(S_1[i_1, i_2], S_2[j_1, j_2])$ is made only for arc-annotated substrings $S_1[i_1, i_2]$ such that $i_1, i_2 \in S_1^{(i_l, i_r)} \cup \{i_l, i_r\}$ for some $(i_l, i_r) \in A_1$ or for $i_1, i_2 \in S_1^0$.

LEMMA 3.1. *Let either $S'_1 = S_1^{(i_l, i_r)} \cup \{i_l, i_r\}$ for an arc $(i_l, i_r) \in A_1$ or $S'_1 = S_1^0$. In both cases, if $i_1, i_2 \in S'_1$, then a call of $\text{maxaps}(S_1[i_1, i_2], S_2[j_1, j_2])$ takes $O(|S'_1|)$ time.*

PROOF. According to the recursive definition of maxaps , the recursion stops if one of the input substrings consists of only one base. Otherwise, the recursion for $\text{maxaps}(S_1[i_1, i_2], S_2[j_1, j_2])$ is continued by one of the following recursive calls:

- $\text{maxaps}(S_1[i_1 + 1, i_2], S_2[j_1 + 1, j_2])$, if $S_1[i_1] = S_2[j_1]$ and $S_1[i_1]$ is not an endpoint of an arc;
- $\text{maxaps}(S_1[i_1 + 1, i_2], S_2[j_1, j_2])$, if $S_1[i_1] \neq S_2[j_1]$ and $S_1[i_1]$ is not an endpoint of an arc;
- $\text{maxaps}(S_1[i_r + 1, i_2], S_2[T(i_l, j_1) + 1, j_2])$, if $S_1[i_1]$ is the left endpoint of an arc (i_l, i_r) .

Thus, every call of maxaps decreases the size of S'_1 by at least one. Since at most S'_1 comparisons have to be made till the recursion stops, the running time of $\text{maxaps}(S_1[i_1, i_2], S_2[j_1, j_2])$ is upperbounded by $O(|S'_1|)$. \square

THEOREM 3.2. *APS(NESTED, PLAIN) can be solved in $O(nm)$ time.*

PROOF. With Lemma 3.1, entry $T(i_l, j)$ for arc (i_l, i_r) and $1 \leq j \leq m$, which is equal to the maximum return value of two calls of maxaps , can be computed in $O(|S_1^{(i_l, i_r)}|)$ time. Computing all table entries corresponding to (i_l, i_r) can, thus, be done in $O(|S_1^{(i_l, i_r)}| \cdot m)$ time. Thus, the first phase, i.e., computing the entries for arcs in A_1 , needs $\sum_{(i_l, i_r) \in A_1} O(|S_1^{(i_l, i_r)}| \cdot m)$ time. The second phase takes, analogously as shown in Lemma 3.1, $O(|S_1^0|)$ time. From Observation 2.1 we know that $n = |S_1^0| + \sum_{(i_l, i_r) \in A_1} |S_1^{(i_l, i_r)}|$, and we obtain the claimed time bound. \square

4. APS(NESTED, CHAIN)

In this section, we extend the algorithm `aps_np` to an algorithm that solves APS (NESTED, CHAIN), where the pattern sequence S_2 has a chain arc annotation. Thus, each arc in A_2 has to be matched to an arc in A_1 . To this end, we introduce the notion of an *innermost matching arc* and a new function `maxaps_nc`.

An arc $(i_l, i_r) \in A_1$ is a *matching arc* for an arc $(j_l, j_r) \in A_2$ if the corresponding endpoints of the two arcs are the same, i.e., $S_1[i_l] = S_2[j_l]$ and $S_1[i_r] = S_2[j_r]$, and $S_2[j_l + 1, j_r - 1]$ is an arc-preserving subsequence of $S_1[i_l + 1, i_r - 1]$. An *innermost matching arc* $(i_l, i_r) \in A_1$ for $(j_l, j_r) \in A_2$ is an arc which is a matching arc for (j_l, j_r) such that there is no arc inside (i_l, i_r) that is also a matching arc for (j_l, j_r) . Since A_1 is nested, in particular no two innermost matching arcs for an arc in A_2 are nested, i.e., for two innermost matching arcs (i_l^1, i_r^1) and (i_l^2, i_r^2) for

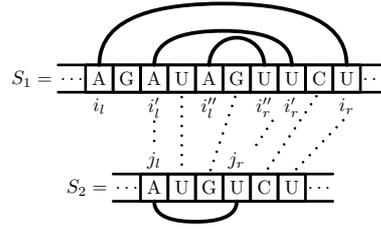


Fig. 6. Computing a best aps-match in an APS(NESTED, CHAIN) instance: $(i'_l, i'_r) \in A_1$ is an innermost matching arc for $(j_l, j_r) \in A_2$. Therefore, the best aps-match of $S_2[j_l, m]$ in $S_1[i'_l, i'_r]$ is $j_l - 1$, in $S_1[i'_l, i'_r]$ it is j_r , and in $S_1[i_l, i_r]$ it is $j_r + 2$ (the respective base matches are indicated by dotted lines)

$(j_l, j_r) \in A_2$, we have either $i_l^1 < i_r^1 < i_l^2 < i_r^2$ or $i_l^2 < i_r^2 < i_l^1 < i_r^1$. Observe that if S_2 is an aps of S_1 then S_2 can be always matched to S_1 by assigning innermost matching arcs to all arcs in A_2 . In the following algorithm, from several matching arcs for an arc in A_2 we choose only the innermost matching arc. By this choice, a smallest possible substring of S_1 is used to match $S_2[j_l, j_r]$, leaving a largest possible portion of S_1 to match the remaining substrings of S_2 . The computation of best aps-matches under this criterion is illustrated in Fig. 6. Preferring the innermost matching arc is essential for our dynamic programming approach.

Dynamic programming table. The dynamic programming table T of size $|A_1| \cdot m$ is defined exactly as in Sect. 3.

Computing best aps-matches. Table T requires the computation of best aps-matches but, in contrast to Sect. 3, now A_2 can also contain arcs. We introduce a new function `maxaps_nc` that extends the function `maxaps` from Sect. 3 by additionally taking the arcs in A_2 into account. The recursive definition of function `maxaps_nc` is given in Fig. 7; given arc-annotated substrings $S_1[i_1, i_2]$ and $S_2[j_1, j_2]$ and given all table entries $T(i_l, j)$ for $(i_l, i_r) \in A_1$ with $i_1 \leq i_l < i_r \leq i_2$ and $j_1 \leq j \leq j_2$, it computes the best aps-match of $S_2[j_1, j_2]$ in $S_1[i_1, i_2]$. Here, the case distinction has also to take into account the cases in which $S_2[j_1]$ is the endpoint of an arc. However, $S_2[j_1]$ can, due to the definition of arc-annotated substrings, not be right endpoint of an arc in the arc annotation of $S_2[j_1, j_2]$. In the case that $S_2[j_1]$ is a left endpoint but $S_1[i_1]$ not, it is not possible to match $S_1[i_1]$ with $S_2[j_1]$ due to the arc-preserving property. We invoke a new call of `maxaps_nc` after $S_1[i_1]$ has been skipped. The case that both $S_1[i_1]$ and $S_2[j_1]$ are arc endpoints is covered by the case that $S_1[i_1]$ is the left endpoint of arc (i_l, i_r) : Independently of whether or not $S_2[j_1]$ is also arc endpoint, we make use of the best aps-match that is, following the preconditions that we made for `maxaps_nc`, already computed and stored in $T(i_l, j_1)$. Hence, `maxaps_nc` treats $S_1[i_l, i_r]$ and $S_2[j_1, T(i_l, j_1)]$ as single bases, skipping both substrings before continuing the recursion. The changes required in `maxaps_nc` in comparison to `maxaps` are highlighted in Fig. 7.

Computing entries of the dynamic programming table. To compute table T , we process the arcs in A_1 in increasing order of their right endpoints. For each arc $(i_l, i_r) \in A_1$, we then compute table entries $T(i_l, j)$ corresponding to bases j in S_2 . In this, our goal is to find, for each arc in A_2 , its innermost matching arcs in A_1 . For this purpose, we divide the computation of table entries corresponding to arc (i_l, i_r) into two phases: The first phase computes those table entries corresponding to bases in S_2 which are either inside an arc of A_2 or left endpoints of arcs in A_2 . The second phase computes those table entries corresponding to bases in S_2 which are outside all arcs of A_2 .

Phase 1. Firstly, we compute those $T(i_l, j)$, where (i_l, i_r) is an arc in A_1 and $S_2[j]$

```

Function maxaps_nc( $S_1[i_1, i_2], S_2[j_1, j_2]$ )
Input: Arc-annotated substrings  $S_1[i_1, i_2]$  and  $S_2[j_1, j_2]$ .
Output: Best aps-match of  $S_2[j_1, j_2]$  in  $S_1[i_1, i_2]$ .
Global: Table entries  $T(i_l, j)$  for  $(i_l, i_r) \in A_1$  with  $i_1 \leq i_l < i_r \leq i_2$  and  $j_1 \leq j \leq j_2$  containing the best aps-match of  $S_2[j, j_2]$  in  $S_1[i_l, i_r]$ .

Method:
                                /* Recursion stops... */
if  $i_1 > i_2$  or  $j_1 > j_2$  then                                /* ...if  $S_1$  or  $S_2$  are empty... */
    return  $j_1 - 1$ ;
else if  $i_1 = i_2$  then                                        /* ...if  $S_1$  has only one base,... */
    return  $\left\{ \begin{array}{l} j_1 \quad \text{if } S_1[i_1] = S_2[j_1] \text{ and } S_2[j_1] \text{ is not an endpoint} \\ j_1 - 1 \text{ otherwise} \end{array} \right\}$ ;
else if  $i_1 < i_2$  and  $j_1 = j_2$  then                    /* ...or if  $S_2$  has only one base. */
    return  $\left\{ \begin{array}{l} j_1 \quad \text{if there is } i_1 \leq i \leq i_2 \text{ with } S_1[i] = S_2[j_1] \\ j_1 - 1 \text{ otherwise} \end{array} \right\}$ ;
else                                                        /* If  $i_1 < i_2$  and  $j_1 < j_2$  then recursion continues: */
    if neither  $S_1[i_1]$  nor  $S_2[j_1]$  is an endpoint then
        return  $\left\{ \begin{array}{l} \text{maxaps\_nc}(S_1[i_1 + 1, i_2], S_2[j_1 + 1, j_2]) \text{ if } S_1[i_1] = S_2[j_1] \\ \text{maxaps\_nc}(S_1[i_1 + 1, i_2], S_2[j_1, j_2]) \text{ otherwise} \end{array} \right\}$ ;
        else if  $S_2[j_1]$  is an endpoint of an arc but  $S_1[i_1]$  is not then
            return  $\text{maxaps\_nc}(S_1[i_1 + 1, i_2], S_2[j_1, j_2])$ ;
        else                                                /*  $S_1[i_1]$  is left endpoint of arc  $(i_l, i_r)$ , i.e.,  $i_1 = i_l$  */
            return  $\text{maxaps\_nc}(S_1[i_r + 1, i_2], S_2[T(i_l, j_1) + 1, j_2])$ ;
    end if
end if

```

Fig. 7. Recursive definition of maxaps_nc for arc-annotated sequences (S_1, A_1) with nested A_1 and (S_2, A_2) with chain A_2 . The differences to function maxaps (Fig. 4) are highlighted

is a base inside an arc (j_l, j_r) from A_2 , in the same way as in Sect. 3:³

$$T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps_nc}(S_1[i_l, i_r - 1], S_2[j, j_r - 1]), \\ \text{maxaps_nc}(S_1[i_l + 1, i_r], S_2[j, j_r - 1]) \end{array} \right\}$$

Secondly, we compute $T(i_l, j_l)$, where $S_2[j_l]$ is the left endpoint of an arc (j_l, j_r) (it is not necessary to compute table entries corresponding to the right endpoints): Following the definition of a best aps-match, $T(i_l, j_l)$ is set to j_r if (i_l, i_r) is an innermost matching arc for (j_l, j_r) and, otherwise, computed by $\text{maxaps_nc}(S_1[i_l + 1, i_r], S_2[j_l, m])$. By computing $T(i_l, j_l)$ in this way, we prefer the *innermost* matching arcs to other matching arcs that would be possible. In the following, we describe how innermost matching arcs are determined. Using the information which we computed for the bases inside (j_l, j_r) and which we saved in table T , it is easy to test whether $\text{maxaps_nc}(S_1[i_l + 1, i_r - 1], S_2[j_l + 1, j_r - 1]) = j_r - 1$, and to determine whether (i_l, i_r) is a matching arc for (j_l, j_r) . To decide whether it is an innermost matching arc, we recall that we process the arcs in A_1 in increasing order by their right endpoints. Therefore, we simply keep track of the so far last found innermost matching arc for each arc in A_2 . If there was none so far or the match involved an arc $(i'_l, i'_r) \in A_1$ left of (i_l, i_r) , i.e., $i'_l < i'_r < i_l < i_r$, then (i_l, i_r) is an innermost matching arc for (j_l, j_r) . Then, $T(i_l, j_l)$ is set to j_r . If (i_l, i_r) is not an innermost matching arc for (j_l, j_r) , a call of $\text{maxaps_nc}(S_1[i_l + 1, i_r], S_2[j_l, m])$ is made to test if there is an innermost matching arc for (j_l, j_r) inside arc (i_l, i_r) . If so, the recursion

³Note that, if $S_2[j]$ is inside an arc (j_l, j_r) , then $S_2[j, j_r - 1]$ has no arc annotation. Then, maxaps and maxaps_nc return the same result with $S_2[j, j_r - 1]$ as the second parameter in the subsequent calls to maxaps_nc.

```

Algorithm aps_nc
Input: Sequence  $S_1$  with nested arc annotation
         and pattern sequence  $S_2$  with chain arc annotation;
Method:
  array of int  $T[n][m]$ ;
  /***** Phase 1 *****/
  for each  $(i_l, i_r) \in A_1$  (ordered by their right endpoints) do
    for each  $(j_l, j_r) \in A_2$  do
      for  $j = j_l + 1$  to  $j_r - 1$  do
         $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps\_nc}(S_1[i_l, i_r - 1], S_2[j, j_r - 1]), \\ \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j, j_r - 1]) \end{array} \right\}$ 
      end for
       $T(i_l, j_l) := \begin{cases} j_r & \text{if } (i_l, i_r) \text{ is an innermost} \\ & \text{matching arc for } (j_l, j_r), \\ \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j_l, m]) & \text{otherwise.} \end{cases}$ 
    end for
  end for

  /***** Phase 2 *****/
  for each  $(i_l, i_r) \in A_1$  (ordered by their right endpoints) do
    for each  $j \in S_2$  such that  $S_2[j]$  is outside of all arcs do
       $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps\_nc}(S_1[i_l, i_r - 1], S_2[j, m]), \\ \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j, m]) \end{array} \right\}$ 
    end for
  end for

  /***** Phase 3 *****/
  if  $(\text{maxaps\_nc}(S_1[1, n], S_2[1, m]) = m)$ 
    then print ' $S_2$  is an aps of  $S_1$ ';
    else print ' $S_2$  is not an aps of  $S_1$ ';
  end if

```

Fig. 8. Outline in pseudo-code of the algorithm that solves APS(NESTED, CHAIN)

will continue to try to match as many of the bases of S_2 following arc (j_l, j_r) as possible to substring $S_1[i_l + 1, i_r]$. The returned value is the maximum index of these bases. If there is no innermost matching arc inside (i_l, i_r) for (j_l, j_r) , then it is not difficult to verify that none of the calls $\text{maxaps_nc}(S_1[i, i_r], S_2[j_l, m])$, for all $i_l < i \leq i_r$, can return a value greater than $j_l - 1$.

Phase 2. We complete the table entries corresponding to an arc $(i_l, i_r) \in A_1$ by processing those bases in S_2 which are outside all arcs in S_2 ; this is done as computing the table entries in Sect. 3, only employing function `maxaps_nc` instead of `maxaps`.

Resulting algorithm. The algorithm to solve APS(NESTED, CHAIN) has three phases. The first two phases, as described above, deal with computing table entries. Phase 1 computes those entries of Table T corresponding to the positions in S_2 which are either inside an arc of A_2 or which are left endpoints of arcs of A_2 . Phase 2 completes table T by processing those bases in S_2 which are outside all arcs using function `maxaps_nc`. In the third phase, if $\text{maxaps_nc}(S_1[1, n], S_2[1, m])$ returns m , then S_2 is an arc-preserving subsequence of S_1 . The algorithm is outlined in Fig. 8. In summary, this yields the following result.

THEOREM 4.1. APS(NESTED, CHAIN) can be solved in $O(nm)$ time.

PROOF. The correctness of the algorithm follows from the correctness of Algorithm `aps_np` and the observation that function `maxaps_nc` treats the arcs of S_2 as single bases and skips them by matching them to an innermost matching arc.

In the following, we estimate the running time for the three phases individually:

Phase 1. Analogously to Sect. 3, we consider, firstly, the running time of a call of `maxaps_nc`. Function `maxaps_nc` works almost in the same way as `maxaps` except that there is an additional case involving arcs in S_2 . However, in this case, the size of the S_1 substring under consideration is decreased by at least one in the recursive call of `maxaps_nc`. Hence, a call of `maxaps_nc` has the same upper bound on the running time as a call of `maxaps` (this can be shown in analogy to Lemma 3.1). Therefore, the innermost loop in the first phase of our algorithm takes at most $O(|S_1^{(i_l, i_r)}| \cdot |S_2^{(j_l, j_r)}|)$ time.

Concerning the endpoints of (j_l, j_r) , we make at first a call of `maxaps_nc` ($S_1[i_l + 1, i_l - 1], S_2[j_l + 1, j_r - 1]$), which needs at most $O(|S_1^{(i_l, i_r)}|)$ time, to determine whether (i_l, i_r) is an innermost matching arc for (j_l, j_r) . If the question is answered positively, the second loop is finished; if not, another call of `maxaps_nc` is made, which takes also at most $O(|S_1^{(i_l, i_r)}|)$ time. Thus, the middle loop over arcs in A_2 can be done in time $\sum_{(j_l, j_r) \in A_2} O(|S_1^{(i_l, i_r)}| \cdot |S_2^{(j_l, j_r)}|)$. The time required for all three loops of the first phase sums up to

$$\sum_{(i_l, i_r) \in A_1} \sum_{(j_l, j_r) \in A_2} O(|S_1^{(i_l, i_r)}| \cdot |S_2^{(j_l, j_r)}|) = \sum_{(j_l, j_r) \in A_2} O(n \cdot |S_2^{(j_l, j_r)}|),$$

where equality holds since with Observation 2.1 we have $\sum_{(i_l, i_r) \in A_1} |S_1^{(i_l, i_r)}| = O(n)$.

Phase 2. In this phase, we complete the table for the bases of S_2 outside all arcs. As shown above, a call to `maxaps_nc` corresponding to $(i_l, i_r) \in A_1$ takes $O(|S_1^{(i_l, i_r)}|)$ time. Thus, the second phase can be done in time

$$\sum_{(i_l, i_r) \in A_1} O(|S_1^{(i_l, i_r)}|) \cdot |S_2^0| = O(n \cdot |S_2^0|),$$

where S_2^0 contains the endpoints of arcs in S_2 and the bases outside all arcs in S_2 . In summary, Phases 1 and 2, take

$$O(n \cdot |S_2^0|) + \sum_{(j_l, j_r) \in A_2} O(n \cdot |S_2^{(j_l, j_r)}|) = O(nm),$$

since, due to the chain arc annotation of S_2 , it follows from Observation 2.1 in particular that $S_2^0 + \sum_{(j_l, j_r) \in A_2} |S_2^{(j_l, j_r)}| = O(m)$.

Phase 3. The third phase is the same as the second phase in Algorithm `aps_np` in Sect. 3 and, thus, can be done in $O(n)$ time.

Adding up the time costs of the three phases, we obtain the total running time of the algorithm, $O(nm)$. \square

5. APS(NESTED, NESTED)

The basic idea how the algorithm for `APS(NESTED, NESTED)` builds on the algorithms in Sect. 3 and 4 is as follows. We employ a dynamic programming table which is defined exactly as in Sect. 3. We can compute the table entries corresponding to the bases inside of the innermost arcs in A_2 in the same way as in the algorithm for `APS(NESTED, PLAIN)`. The information saved in the entries enables us to find the innermost matching arcs for the innermost arcs in A_2 . When processing an arc of A_2 which is not innermost, i.e., there are arcs inside it, we observe that the arcs which are directly inside this arc form a chain structure. More precisely, when processing the arcs in A_2 in the order of their right endpoints, these inner arcs are already processed at this point. Therefore, we can process these arcs in the same way as in the first phase of the algorithm for `APS(NESTED, CHAIN)`, i.e., we treat them as single bases and skip them by matching them to their innermost matching arcs.

```

Algorithm aps_nn
Input: Sequences  $S_1$  and  $S_2$  both with nested arc annotation;
Method:
  array of int  $T[n][m]$ ;
  /***** Phase 1 *****/
  for each  $(i_l, i_r) \in A_1$  (ordered by their right endpoints) do
    for each  $(j_l, j_r) \in A_2$  (ordered by their right endpoints) do
      for each  $j \in S_2^{(j_l, j_r)}$  do
         $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps\_nc}(S_1[i_l, i_r - 1], S_2[j, j_r - 1]), \\ \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j, j_r - 1]) \end{array} \right\}$ 
      end for
       $T(i_l, j_l) := \begin{cases} j_r & \text{if } (i_l, i_r) \text{ is an innermost} \\ & \text{matching arc for } (j_l, j_r) \\ \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j_l, m]) & \text{otherwise.} \end{cases}$ 
    end for
  end for

  /***** Phase 2 *****/
  for each  $(i_l, i_r) \in A_1$  (ordered by their right endpoints) do
    for each  $j \in S_2$  such that  $S_2[j]$  is outside of all arcs do
       $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps\_nc}(S_1[i_l + 1, i_r], S_2[j, m]), \\ \text{maxaps\_nc}(S_1[i_l, i_r - 1], S_2[j, m]) \end{array} \right\}$ 
    end for
  end for

  /***** Phase 3 *****/
  if  $(\text{maxaps\_nc}(S_1[1, n], S_2[1, m]) = m)$ 
    then print ' $S_2$  is an aps of  $S_1$ ';
  else print ' $S_2$  is not an aps of  $S_1$ ';
  end if

```

Fig. 9. Outline in pseudo-code of the algorithm that solves APS(NESTED, NESTED)

The algorithm solving APS(NESTED, NESTED) is outlined in Fig. 9. In contrast to the algorithm for APS(NESTED, CHAIN), the first phase is extended to process all arcs in A_2 : To fill the dynamic programming table T as already used in the previous sections, we process the arcs in A_1 from inner to outer arcs and process, for every arc in A_1 , the arcs in A_2 from inner to outer arcs, using the function `maxaps_nc` from the previous section. The second phase is, then, the same as in the algorithm for APS(NESTED, CHAIN): We complete the table for the bases in S_2 that are outside all arcs. Finally, we compute `maxaps_nc`($S_1[1, n]$, $S_2[1, m]$). If it returns m , then S_2 is an aps of S_1 . Similar to the running time analysis for APS(NESTED, CHAIN), we obtain:

THEOREM 5.1. APS(NESTED, NESTED) can be solved in $O(nm)$ time. \square

6. MODIFIED VERSIONS OF APS(UNLIMITED, NESTED)

This section is devoted to modified versions of APS problems, motivated by the search for RNA structure motifs. Our central point of reference is an open problem posed by Vialette [2002] which can be answered using the dynamic programming techniques developed in the previous sections. For the sake of an easier comprehensibility, we present our approach in two steps: Firstly, we show how to solve AST(UNLIMITED, NESTED), which is “half way” between the APS problems and the problem stated in [Vialette 2002]. Secondly, we show how to extend the derived algorithm to solve the problem which, in [Vialette 2002], is called PATTERN MATCHING OVER 2-INTERVAL SET restricted to $\{<, \square\}$ -structured patterns. The

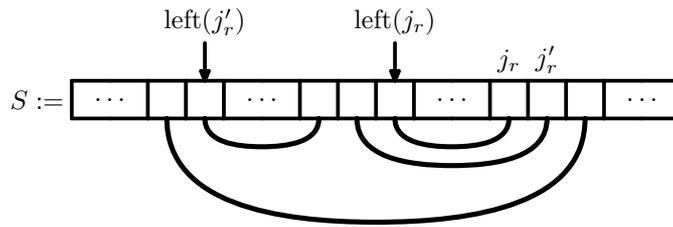


Fig. 10. Example illustrating the definition of $\text{left}(j)$ with $1 \leq j \leq |S|$ for an arc-annotated sequence S with nested arc annotation

difference of this problem to $\text{AST}(\text{UNLIMITED}, \text{NESTED})$ is that the endpoints of arcs are given as intervals instead of single bases. The resulting algorithm runs in quadratic time.

6.1 $\text{AST}(\text{UNLIMITED}, \text{NESTED})$

We use S_1 to denote the sequence with unlimited arc annotation and, since, in contrast to the previous sections, there can be up to $O(|S_1|^2)$ many arcs, we use n to denote $|A_1|$. We use S_2 to denote the pattern sequence with nested arc annotation and $m := |S_2|$.

$\text{AST}(\text{UNLIMITED}, \text{NESTED})$ differs from $\text{APS}(\text{NESTED}, \text{NESTED})$, see Sect. 5, in the following ways. On the one hand, it is more general since S_1 can have unlimited arc annotation. This makes it impossible to inspect the inside of the arcs by functions maxaps (Sect. 3) and $\text{maxaps}_{\text{nc}}$ (Sect. 4) which heavily relied on the nested arc annotation of S_1 . On the other hand, we have the additional restriction that *every* base, both in S_1 and in S_2 , is an endpoint of at least one arc. This makes “partial” arc matches impossible, where we match only one but not the other endpoint of an arc in A_1 with a base in S_2 which is not an endpoint; this scenario contributed much to the computational difficulty of the problems in the previous sections. Moreover, for a simpler setting, we restrict ourselves in AST , as opposed to APS , to a unary alphabet. In the following, we outline how to adapt our dynamic programming techniques from the previous sections to the new problem.

Dynamic programming table. Again, we build a dynamic programming table T of size $(|S_1| + 1) \times |A_2|$ and compute its entries by processing the arcs of the nested arc annotation of S_2 in the order of their right endpoints. Since only S_2 has a nested arc annotation, we change the meaning of the entries in T . Now, table T has entries for every position in S_1 and every arc in A_2 ; we refer to the table entry corresponding to position $1 \leq i \leq |S_1|$ and arc $(j_l, j_r) \in A_2$ by $T(i, j_r)$. We define the meaning of a T entry in a “dual” way compared to the previous sections: There, $T(i_l, j)$ corresponded to an arc $(i_l, i_r) \in A_1$ and a position j in S_2 and it specified the *maximum* length substring *starting* in $S_2[j]$ which is an aps of $S_1[i_l, i_r]$. Here, in contrast, $T(i, j_r)$ corresponds to a position i in S_1 and an arc $(j_l, j_r) \in A_2$ and it specifies the *minimum*-length substring which *ends* in $S_1[i]$ and of which $S_2[j_l, j_r]$ is an ast .

To specify the content of a T entry more precisely, we introduce two additional definitions.

DEFINITION 6.1. *Given an arc-annotated sequence S with nested arc annotation A , and integer j , $1 \leq j \leq |S|$, $\text{left}(j)$ denotes the minimum index $j' \leq j$ such that each of $S_2[j']$, $S_2[j' + 1]$, \dots , $S_2[j]$ is an endpoint of an arc $(j_l, j_r) \in A$ with $j_r \leq j$.*

Due to the order of processing the arcs in A_2 , $S_2[\text{left}(j), j]$ is, intuitively speaking, the longest substring ending in $S_2[j]$ in which all bases have been examined when processing $S_2[j]$. An illustrating example is displayed in Fig. 10.

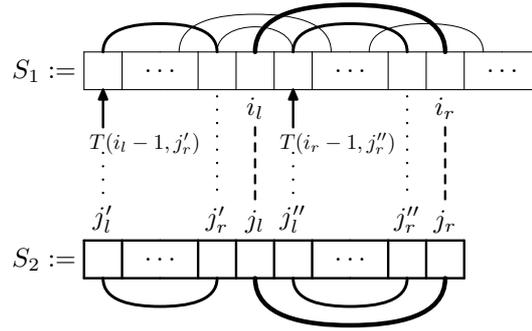


Fig. 11. Example illustrating how we determine whether $(j_l, j_r) \in A_2$ can be matched with $(i_l, i_r) \in A_1$ (match is indicated by the dashed lines) such that $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[1, i_r]$. Here, $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[1, i_r]$: Note that $\text{left}(j_r) = j'_l$ and (1) since $i_l < T(i_r - 1, j''_r)$, $S_2[j''_l, j''_r]$ is an ast of $S_1[T(i_r - 1, j''_r), i_r - 1]$, and (2) since $T(i_l - 1, j'_r) \neq 0$, $S_2[j'_l, j'_r]$ is an ast of $S_1[T(i_l - 1, j'_r), i_l - 1]$ (these matches are indicated by the dotted lines)

DEFINITION 6.2. Given an instance of $\text{AST}(\text{UNLIMITED}, \text{NESTED})$ consisting of arc-annotated sequences S_1 and S_2 , and $1 \leq j_1 < j_2 \leq |S_2|$ and $1 \leq i \leq |S_1|$, integer i' , $1 \leq i' < i$, is called an *ast-match* of $S_2[j_1, j_2]$ at $S_1[i]$ if $S_2[j_1, j_2]$ is an ast of $S_1[i', i]$. The best ast-match of $S_2[j_1, j_2]$ at $S_1[i]$ is the maximum integer i' that is an ast-match of $S_2[j_1, j_2]$ at $S_1[i]$ (and 0 if no such index exists).

Now, we define the contents of table T (to be computed) more formally: Entry $T(i, j_r)$, for $(j_l, j_r) \in A_2$ and $1 \leq i \leq |S_1|$, contains the best ast-match of $S_2[\text{left}(j_r), j_r]$ at $S_1[i]$. If, after all entries are computed, we have $T(|S_1|, m) > 0$ then S_2 is an ast of S_1 .

Computing entries of the dynamic programming table. We initialize $T(0, j_r) := 0$ for every $(j_l, j_r) \in A_2$. To compute other table entries, we process arcs $(j_l, j_r) \in A_2$ by the order of their right endpoints and, for every arc in A_2 , the bases i in S_1 in increasing order, i.e., in a loop $i = 1, \dots, |S_1|$. Thus, when computing $T(i, j_r)$, we can assume that all entries in T that correspond to $(j'_l, j'_r) \in A_2$ with $j'_r < j_r$ have already been computed.

Computing best ast-matches. During the i -loop for one $(j_l, j_r) \in A_2$, we will maintain two variables i' and i'' , where the current value of i' is used as an auxiliary variable to compute the current value of i'' which will be used to determine entry $T(i, j_r)$:

- Variable i' is, before the start of the i -loop, initialized by $i' := 0$ and denotes the currently best ast-match of $S_2[j_l, j_r]$ at $S_1[i]$.
- Variable i'' is, before the start of the i -loop, also initialized by $i'' := 0$ and denotes the currently best ast-match of $S_2[\text{left}(j_r), j_r]$ at $S_1[i]$.

Now, during the i -loop, we process, for every value of i , all arcs $(i_l, i_r) \in A_1$ having i as their right endpoint $i_r = i$. For each of these arcs, we test whether, assuming that we match $(i_l, i_r) \in A_1$ with $(j_l, j_r) \in A_2$, we have to update the value of i' and i'' . After all arcs ending in $S_1[i]$ have been processed, the value of i'' is then the best ast-match of $S_2[\text{left}(j_r), j_r]$ at $S_1[i]$ and $T(i, j_r) := i''$.

The crucial point above is to determine whether the value of variables i' and i'' has to be updated, i.e., given $(i_l, i_r) \in A_1$ and $(j_l, j_r) \in A_2$, whether matching (i_l, i_r) with (j_l, j_r) gives rise to better ast-matches than the ones stored in i' and i'' . This question can be answered in two steps, the first step tests whether we have to update the value of i' , and the second step considers i'' . We use the example shown in Fig. 11 to explain these two steps.

Step 1: Here, the question is whether matching arcs (i_l, i_r) and (j_l, j_r) gives rise to

a better ast-match i'_{new} of $S_2[j_l, j_r]$ at $S_1[i_l]$ than the one stored in i' , i.e., $i' < i'_{new}$. To answer this question, we test as follows whether $S_2[j_l + 1, j_r - 1]$ is an ast of $S_1[i_l + 1, i_r - 1]$ (otherwise, we cannot match (j_l, j_r) with (i_l, i_r)). In Fig. 11, $S_2[j_r - 1]$ is the right endpoint of an arc $(j'_l, j'_r) \in A_2$. The table entries $T(i, j'_r)$, for all $1 \leq i \leq |S_1|$ and (j'_l, j'_r) , have already been computed since $j'_r = j_r - 1 < j_r$. Then, $S_2[j_l + 1, j_r - 1]$ is an ast of $S_1[i_l + 1, i_r - 1]$ iff $i_l < T(i_r - 1, j'_r)$.

Note that, if (j_l, j_r) is not an innermost arc, then $S_2[j_r - 1]$ is always the right endpoint of an arc and $\text{left}(j_r - 1) = j_l + 1$. If, however, (j_l, j_r) is an innermost arc (which implies $j_r - 1 = j_l$) then (j_l, j_r) can always be matched with (i_l, i_r) .

Now, if Step 1 is answered positively, i.e., $S_2[j_l + 1, j_r - 1]$ is an ast of $S_1[i_l + 1, i_r - 1]$, then $i'_{new} := i_l$ is an ast-match of $S_2[j_l, j_r]$ at $S_1[i_r]$. Further, if $i' < i'_{new} = i_l$, then we update $i' := i_l$ since now i_l is the currently maximum index i' that is an ast-match of $S_2[j_l, j_r]$ at $S_1[i_r]$.

Step 2: Here, the question is whether a possible change in the value of i' gives rise to a better ast-match i''_{new} than the one stored in i'' , i.e., $i'' < i''_{new}$. In our example, we have updated i' to i_l as shown in Step 1. Now, we compute as follows the best ast-match i''_{new} , $1 \leq i''_{new} \leq i_l$ of $S_2[\text{left}(j_r), j_l - 1]$ at $S_1[i_l - 1]$ (only if such an $i''_{new} \geq 1$ exists then $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[1, i_r]$ while matching (j_l, j_r) to (i_l, i_r)). Note that, in Fig. 11, $S_2[j_l - 1]$ is the right endpoint of an arc $(j'_l, j'_r) \in A_2$. The table entries $T(i, j'_l - 1)$, for all $1 \leq i \leq |S_1|$, have already been computed since $j_l - 1 = j'_r < j_r$. Then, i''_{new} can be found in $T(i_l - 1, j'_r)$: If $T(i_l - 1, j'_r) \neq 0$, then $T(i_l - 1, j'_r)$ contains the best ast-match of $S_2[\text{left}(j'_r), j'_l]$ at $S_1[i_l - 1]$. If, however, $T(i_l - 1, j'_r) = 0$ then $S_2[j'_l, j'_r]$ is not an ast of $S_1[1, i_l - 1]$ and no i''_{new} exists.

We can proceed as for Fig. 11 whenever $S_2[j_l - 1]$ is right endpoint of an arc. Otherwise, we have $\text{left}(j_r) = j_l$ and $i''_{new} := i_l$.

Now, if we find $i''_{new} \geq 1$ as described and $i'' < i''_{new}$, then we update $i'' := i''_{new}$ since i''_{new} is the currently maximum index such that $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[i''_{new}, i_r]$.

Summarizing, if Step 1 is answered positively, i.e., $S_2[j_l + 1, j_r - 1]$ is an ast of $S_1[i_l + 1, i_r - 1]$, and if we find an $i''_{new} \geq 1$ in Step 2, i.e., $S_2[\text{left}(j_r), j_l - 1]$ is an ast of $S_1[i''_{new}, i_l - 1]$, then i''_{new} is a possible ast-match for $S_2[\text{left}(j_r), j_r]$ at $S_1[i]$. During the i -loop described above we keep, in variable i'' , track of the currently best ast-match.

Resulting algorithm. The algorithm computing the table entries of T in this way is outlined in Fig. 12. Regarding the running time, note that, for every arc in A_2 , we inspect every arc in A_1 once. Thus, we obtain the following result:

THEOREM 6.3. $\text{AST}(\text{UNLIMITED}, \text{NESTED})$ can be solved in $O(nm)$ time. \square

For an easier presentation we focused here on the case of unary alphabets. It is straightforward to generalize the algorithm to non-unary alphabets: Then, two arcs, one in A_1 and one in A_2 , are considered as matches only if the bases at their left endpoints and the bases at their right endpoints coincide. Note that $\text{AST}(\text{UNLIMITED}, \text{CROSSING})$ was shown to be solvable in $O(n^6 m^2)$ time [Gramm 2004] where the problem is referred to as CONTACT MAP PATTERN MATCHING restricted to $\{<, \bar{\cup}\}$ -patterns.

6.2 Arcs over Intervals

In comparison to $\text{AST}(\text{UNLIMITED}, \text{NESTED})$, the PATTERN MATCHING OVER 2-INTERVAL SET problem as described by Vialette [2002] has as input two arc-annotated sets of intervals—we call them I_1 and I_2 —instead of two arc-annotated sequences. Regarding the intervals in I_1 and I_2 we have the following rules:

—Two intervals connected by an arc do not overlap.

Algorithm ast_un

Input: Sequence S_1 with unlimited arc annotation and sequence S_2 with nested arc annotation, over a unary alphabet. In S_1 every base being an endpoint of at least one arc; in S_2 every base being an endpoint of exactly one arc;

Method:

```

array of int  $T[|S_1|][m]$ ;
/** Loop 1 */
for each  $(j_l, j_r) \in A_2$  (ordered by their right endpoints) do
   $i'' := 0$ ; /* currently best ast-match for  $S_2[\text{left}(j_r), j_r]$  */
   $i' := 0$ ; /* currently best ast-match for  $S_2[j_l, j_r]$  */
  /** Loop 2 */
  for  $i = 1$  to  $|S_1|$  do
    for each  $(i_l, i_r) \in A_1$  such that  $i = i_r$  do
      /** (1) Compute the currently best ast-match  $i'$  such that
      **  $S_2[j_l, j_r]$  is ast of  $S_1[i', i]$ .
      **/
      if  $(j_r - j_l = 1)$  then /*  $(j_l, j_r)$  is innermost arc */
         $i' := \max(i_l, i')$ ;
      else /* there is an arc  $(j'_l, j_r - 1)$  */
        if  $(i_l < T(i - 1, j_r - 1))$  then  $i' := \max(i_l, i')$ ;
      end if

      /** (2) Compute the currently best ast-match  $i''$  such that
      **  $S_2[\text{left}(j_r), j_r]$  is ast of  $S_1[i'', i]$ .
      **/
      if  $S_2[j_l - 1]$  is the right endpoint of an arc then
         $i'' := \max(i'', T(i' - 1, j_l - 1))$ ; /*  $T(0, j) := 0$  for all  $j$  */
      else
         $i'' := \max(i'', i')$ ;
      end if
    end for
     $T(i, j_r) := i''$ ;
  end for /* end Loop 2 */
end for /* end Loop 1 */
if  $(T(|S_1|, m) \neq 0)$ 
  then print ' $S_2$  is an ast of  $S_1$ .';
  else print ' $S_2$  is not an ast of  $S_1$ .';
end if

```

Fig. 12. Outline in pseudo-code of the algorithm that solves $\text{AST}(\text{UNLIMITED}, \text{NESTED})$

—No two intervals in I_2 , i.e., of the pattern, do overlap, whereas intervals in I_1 can overlap.

—In the interval setting, S_2 is an *arc-substructure* (*ast*) of S_1 if there is a mapping M of all intervals in I_2 to a subset of *non-overlapping* intervals of I_1 such that the order of intervals is preserved and arcs in A_2 are mapped to arcs in A_1 .

Viallette [2002] states it as an open question whether there is a polynomial time algorithm for $\text{AST}(\text{UNLIMITED}, \text{NESTED})$ over arc-annotated intervals. In the following, we outline how to extend the algorithm from Sect. 6.1 to solve this problem, showing that the problem is solvable in quadratic time. Instead of giving an algorithm with all details, we restrict ourselves to describing the main modifications which are necessary compared to Algorithm `ast_un` from Fig. 12.

In Sect. 6.1, we specified that the arcs in A_1 are processed in the order of their right endpoints. We have to be more precise now since the endpoints of an arc are intervals which again, have two endpoints. We use E_1 and E_2 to denote the

sets of all endpoints of intervals in I_1 and I_2 , respectively. Further, we use, for $i \in I_1$, $l(i) \in E_1$ to denote the left endpoint of interval i and $r(i) \in E_1$ to denote its right endpoint (analogously $l(j), r(j) \in E_2$ for $j \in I_2$). The arcs $(j_l, j_r) \in A_2$ (and $(i_l, i_r) \in A_1$) now have to be processed by the order of $r(j_r)$ (and $r(i_r)$, respectively). This implies three important changes to Algorithm `ast_un` from Fig. 12:

- Since in `AST(UNLIMITED, NESTED)` every base is endpoint of at least one arc, we could, there, assume that the arcs are already ordered by their right endpoints (if they are not, they can be ordered in linear time). Here, instead, the endpoints are from some unbounded range. Therefore, we employ a preprocessing step which orders the arcs in A_1 and A_2 according to the right ends of the intervals at their right endpoints; this preprocessing can be done by sorting in time $O(n \log n)$, where n denotes $|A_1|$. After the preprocessing, we can assume that the elements in E_1 are labeled by $1, 2, \dots, |E_1|$ (analogously for the elements in E_2).
- We have to modify Loop 1 in Fig. 12 such that the arcs $(j_l, j_r) \in A_2$ are processed by the order of $r(j_r)$.
- We have to modify Loop 2 in Fig. 12 such that the arcs $(i_l, i_r) \in A_1$ are processed by the order of $r(i_r)$ (among the arcs with the same $r(i_r)$, the order can be arbitrarily chosen).

We also have to adjust the definition of a best ast-match to our new setting. In Sect. 6.1, we specified that a best ast-match of $S_2[j_1, j_2]$ at $S_1[i]$ denotes the maximum index i'' , $1 \leq i'' \leq i$, such that $S_2[j_1, j_2]$ is an ast of $S_1[i'', i]$. To make this more precise now, let, for $e_l, e_r \in E_1$, $I_1[e_l, e_r]$ denote the arcs $(i_l, i_r) \in A_1$ for which $l(i_l) \geq e_l$ and $r(i_r) \leq e_r$ (analogously for E_2). Then, for $j_1, j_2 \in I_2$, the best ast-match of $I_2[l(j_1), r(j_2)]$ at $e \in E_1$ denotes the maximum $e' \in E_1$ such that $I_2[l(j_1), r(j_2)]$ is an ast of $I_1[e', e]$. This implies the following changes to the algorithm shown in Fig. 12:

- Table T has now entries $T(e, j_r)$ for $e \in E_1$ and j_r corresponds to an arc $(j_l, j_r) \in A_2$. Entry $T(e, j_r)$ denotes the best ast-match of $I_2[l(\text{left}(j_r)), r(j_r)]$ at $e \in E_1$.
- The variables e' and e'' denote elements of E_1 ; then, the computation of e' in Step (1) of Algorithm `ast_un` has to be altered as follows ($(j_l, j_r) \in A_2$ is the arc currently processed in Loop 1 and $(i_l, i_r) \in A_1$ is the arc currently processed in Loop 2):

```

if  $(j_l, j_r)$  is an innermost arc then
     $e' := \max(l(i_l), e')$ ;
else
    if  $(r(i_l) < T(e - 1, l(j_r) - 1))$  then  $e' := \max(l(i_l), e')$  end if
end if

```

The computation of e'' in Step (2) of Algorithm `ast_un` is altered as follows:

```

if  $l(j_l) - 1 = r(j'_r)$  for some  $(j'_l, j'_r) \in A_2$  then
     $e'' := \max(e'', T(e' - 1, l(j_l) - 1))$ ;
else
     $e'' := \max(e'', e')$ ;
end if

```

We summarize our explanations by the following theorem:

THEOREM 6.4. `AST(UNLIMITED, NESTED)` where the endpoints of arcs are intervals is solvable in $O(n \log n + nm)$ time. \square

Note that the problem in Theorem 6.4 corresponds exactly to `PATTERN MATCHING OVER 2-INTERVAL SET` restricted to $\{<, \square\}$ -structured patterns as defined in [Viallette 2002].

7. CONCLUSION

In this work, we started a theoretical study of natural and practically motivated pattern matching problems for arc-annotated sequences. Our main result was to show that pattern matching for sequences with both nested arc structure can be solved in $O(nm)$ time. This generalizes results of Kilpeläinen and Mannila [Kilpeläinen 1992; Kilpeläinen and Mannila 1995] who presented a quadratic time algorithm for a special case of APS(NESTED,NESTED) in which, among others, every base in the pattern is endpoint of an arc and in which, therefore, partial arc matches are not possible. The main applications of our results, however, might lie in the field of RNA structure comparison, an important subject in computational biology. In this context, extending and modifying our result, we gave an efficient algorithm for the PATTERN MATCHING OVER 2-INTERVAL SET problem with $\{<, \sqsubset\}$ -structured patterns, answering an open question of Vialette [2002].

From a theoretical point of view, it would be interesting to determine the complexity of APS(CROSSING,PLAIN) which remained open so far (see Table I for a summary of results now known). In the spirit of [Alber et al. 2004], one might also consider whether there exist useful parameterizations of the NP-complete versions of APS (see Table I) that lead to practically useful fixed-parameter algorithms (cf. [Downey 2003; Fellows 2003; Niedermeier 2006] for surveys on fixed-parameter algorithms). An interesting line of research might also be to investigate whether the compacting techniques (concerning the dynamic programming tables) employed by Fu et al. [2003] may lead to improvements concerning space consumption or running time of our algorithms. In particular, this seems to be prospective in case of “sparse” arc annotations. Concerning computational biology, it remains to advance the application of this kind of pattern matching algorithms in RNA or protein structure analysis as started, e.g., in [Gramm 2004].

ACKNOWLEDGMENT

We thank two anonymous referees for helpful comments improving the presentation.

REFERENCES

- ALBER, J., GRAMM, J., GUO, J., AND NIEDERMEIER, R. 2004. Computing the similarity of two sequences with nested arc annotations. *Theoretical Computer Science* 312, 2-3, 337–358.
- BAFNA, V., MUTHUKRISHNAN, S., AND RAVI, R. 1995. Computing similarity between RNA strings. In *Proc. of 6th CPM*. LNCS, vol. 937. Springer, 1–16.
- DOWNEY, R. G. 2003. Parameterized complexity for the skeptic. In *Proc. of 18th IEEE CCC*. 147–169.
- EL-MABROUK, N. AND RAFFINOT, M. 2002. Approximate matching of secondary structures. In *Proc. of 6th ACM RECOMB*. ACM Press, 156–164.
- EVANS, P. A. 1999a. Algorithms and complexity for annotated sequence analysis. Ph.D. thesis, University of Victoria, Canada.
- EVANS, P. A. 1999b. Finding common subsequences with arcs and pseudoknots. In *Proc. of 10th CPM*. LNCS, vol. 1645. Springer, 270–280.
- EVANS, P. A. AND WAREHAM, H. T. 2001. Exact algorithms for computing pairwise alignments and 3-medians from structure-annotated sequences. In *Proc. of Pacific Symposium on Biocomputing*. 559–570.
- FELLOWS, M. R. 2003. New directions and new challenges in algorithm design and complexity, parameterized. In *Proc. of 8th WADS*. LNCS, vol. 2748. Springer, 505–520.
- FU, W., HON, W. K., AND SUNG, W. K. 2003. On all-substrings alignment problems. In *Proc. of 9th COCOON*. LNCS, vol. 2697. Springer, 80–89.
- GRAMM, J. 2004. A polynomial-time algorithm for the matching of crossing contact-map patterns. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 4, 1, 171–180.
- GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
- JIANG, T., LIN, G.-H., MA, B., AND ZHANG, K. 2004. The longest common subsequence problem for arc-annotated sequences. *Journal of Discrete Algorithms* 2, 2, 257–270.

- KILPELÄINEN, P. 1992. Tree matching problems with applications to structured text data bases. Ph.D. thesis, University of Helsinki, Finland. 1992.
- KILPELÄINEN, P. AND MANNILA, H. 1995. Ordered and unordered tree inclusion. *SIAM Journal on Computing* 24, 2, 340–356.
- LIN, G.-H., CHEN, Z.-Z., JIANG, T., AND WEN, J. 2002. The longest common subsequence problem for sequences with nested arc annotations. *Journal of Computer and System Sciences* 63, 3, 465–480.
- NIEDERMEIER, R. 2006. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press.
- SANKOFF, D. AND KRUSKAL, J., Eds. 1983. *Time Warps, String Edits, and Macromolecules*. Addison-Wesley. Reprinted in 1999 by CSLI Publications.
- VIALETTE, S. 2002. Pattern matching problems over 2-interval sets. In *Proc. of 13th CPM*. LNCS, vol. 2373. Springer, 53–63.
- VIALETTE, S. 2004. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science* 312, 2-3, 223–249.