LCNS, Vol 2556, pp. 182–193, 2002

Pattern Matching for Arc-Annotated Sequences

Jens Gramm*, Jiong Guo**, and Rolf Niedermeier

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
{gramm,guo,niedermr}@informatik.uni-tuebingen.de

Abstract. A study of pattern matching for arc-annotated sequences is started. An $O(nm)$ time algorithm is given to determine whether a length m sequence with nested arc annotations is an arc-preserving subsequence of a length n sequence with nested arc annotations, called APS(NESTED,NESTED). Arc-annotated sequences and, in particular, those with nested arc structure are motivated by applications in RNA structure comparison. Our algorithm can be used to accelerate a recent fixed-parameter algorithm for LAPCS(NESTED,NESTED) and generalizes results for ordered tree inclusion problems. In particular, the presented dynamic programming methodology implies a quadratic time algorithm for an open problem posed by Vialette.

1 Introduction

Basic motivation. Pattern matching in strings is a core problem of computer science. It is of foundational importance in several application areas, the most recent one being computational biology. Numerous versions of pattern matching and related problems occur in practice, ranging in difficulty from linear time solvable to NP-hard problems. 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 [3–5, 7, 9]. Herein, we encounter a seemingly sharp border between (practically important) problem versions that we 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 the much more general term. For a sequence S, an *arc annotation* of S is a set of unordered pairs of positions in S. Lin *et al.* [7] argue that the biologically most important special case is the one of *nested* arc annotations. Here, one requires that no two arcs share an endpoint and no two arcs cross each other. We will also

* Supported by the Deutsche Forschungsgemeinschaft (DFG), research project OPAL (optimal solutions for hard problems in computational biology), NI 369/2-1.

** Partially supported by the Deutsche Forschungsgemeinschaft (DFG), Zentrum für Bioinformatik Tübingen (ZBIT).

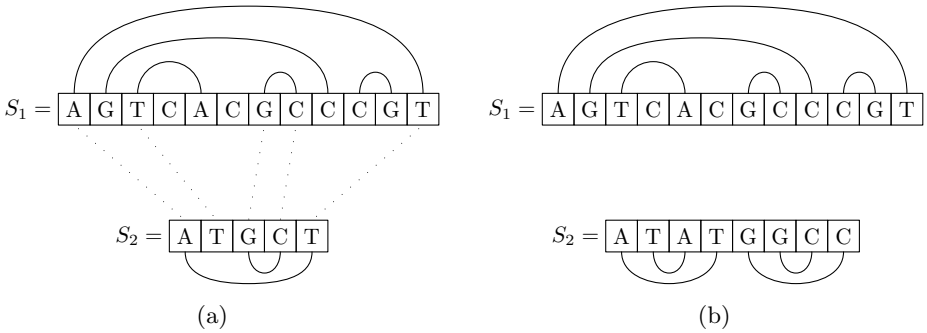

Fig. 1. Input instances of APS(NESTED, NESTED). (a) Yes-instance, S_2 is an aps of S_1. (b) No-instance, S_2 is *not* an aps of S_1

APS(.,.)	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	NP-complete [4]				
CROSSING	—	NP-complete [4]	NP-complete		?
NESTED	—	—	$O(nm)$		

Table 1. Survey of computational complexity for different versions of APS. Most NP-completeness results easily follow from results of Evans [4] for the corresponding LAPCS problems, or, at least, can be proven in a similar way in spirit to there. We omit the details here. The $O(nm)$ time algorithms are described in this paper. The complexity of APS(CROSSING,PLAIN) remains unclassified. We mention in passing that APS(CHAIN,PLAIN) can be solved in $O(n+m)$ time.

consider the less restrictive *crossing* arc structure where the only requirement is that no two arcs share an endpoint. Furthermore, we exhibit the simpler arc structure *chain* which is a nested structure with nesting depth one. Finally, the term *plain* refers to sequences without arcs and the term *unlimited* refers to a completely unrestricted arc structure. Now, we are ready to define the pattern matching problems studied, namely the Arc-Preserving Subsequence problem for various types of arc annotations: APS(TYPE1, TYPE2).

Input: An arc-annotated sequence S_1, $|S_1| = n$, with arc structure TYPE1 and an arc-annotated sequence S_2, $|S_2| = m$, with arc structure TYPE2.

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

That is, if one deletes all but m letters from S_1 (thus, we assume $n \geq m$)—when deleting a letter at position i then *all* arcs with endpoint i are deleted—can one obtain S_2? Fig. 1(a) gives an example for a yes-instance and Fig. 1(b) gives an example for a no-instance of APS(NESTED,NESTED). Clearly, the problem specification only makes sense if arc structure TYPE1 comprises TYPE2.

Results. Our main result is that APS(NESTED,NESTED) can be solved in $O(nm)$ time. Table 1 surveys known and new results for various types of APS. In addition, we study a *modified* version of APS(UNLIMITED,NESTED) where the alphabet is unary, each base has to be endpoint of at least one arc, and we determine whether S_2 forms an *arc substructure (ast)* of S_1. Deriving an $O(nm)$ time algorithm for this case, we answer an open question of Vialette [9]. Observe that

in general APS(UNLIMITED,NESTED) is NP-complete (see Table 1). Due to the lack of space, several details had to be omitted.

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 [9]. 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 asking for the algorithmic complexity (NP-complete vs. polynomially solvable) of the aforementioned modified case of APS(UNLIMITED,NESTED). Moreover, our work generalizes results achieved in the context of structured text databases for the so-called ordered tree inclusion problem [8]. Kilpeläinen and Mannila already presented quadratic time algorithms for a strict special case of APS(NESTED,NESTED). In their case, sequence information is not taken into account and we can easily derive their problem from ours. Bafna *et al.* [2], among other things, considered the corresponding arc-preserving *substring* problems.

As to LAPCS problems, we only briefly mention that most problems in that context become NP-complete [4, 7] as, in particular, LAPCS(NESTED,NESTED). That is why researchers focussed on approximation (factor 2) [5] and fixed-parameter algorithms [1]. Notably, our algorithm for APS(NESTED,NESTED) can be used as a subprocedure to handle easy cases in the exact exponential time algorithm of [1]. Thus, a speed-up of this fixed-parameter algorithm can be achieved. Similar applications might be possible in the approximation context. Finally, note that for easier arc structures such as in LAPCS(NESTED,CHAIN) $O(nm^3)$ time algorithms have been developed [5]—for the special case APS(NESTED,CHAIN) beaten by our $O(nm)$ algorithm.

2 Preliminaries and Definitions

For a sequence S of length $|S| = n$, an *arc annotation* (or *arc set*) A of S is a set of pairs of numbers from $\{1, 2, \ldots, n\}$. 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 pairwisely distinct. 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 $\{1, 2, \ldots, |S_2|\}$ to a subset of $\{1, 2, \ldots, |S_1|\}$, given by $M = \{\langle j, i_j \rangle \mid 1 \leq j \leq |S_2|, 1 \leq i_j \leq |S_1|\}$. We say that S_2 is an *arc-preserving subsequence (aps)* of S_1 if the arcs induced by M are preserved, i.e., for all $\langle j_l, i_l \rangle, \langle j_r, i_r \rangle \in M$: $(j_l, j_r) \in A_2 \Longleftrightarrow (i_l, i_r) \in A_1$.

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 $|S_2| := m$ if not mentioned otherwise. Depending on the arc structures of S_1 and S_2, several versions APS(type1, type2) can be defined where the arc structure of S_1 is type1 and the arc structure of S_2 is type2. An arc set has *nested* arc structure 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 *plain* arc structures, the sequence has no arcs at all, *chain* arc structures have nested structure with nesting depth one, *crossing* refers to arc structures where the only requirement is that no two arcs share an endpoint, and *unlimited* refers to completely arbitrary arc structures.

Under the restriction that the alphabet is unary, 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_1 is endpoint of an arc. A sequence $i_1, \ldots, i_{|S_2|} \in \{1, \ldots, |S_1|\}$ with $i_1 < \cdots < i_{|S_2|}$ defines a mapping $M = \{\langle j, i_j \rangle \mid 1 \leq j \leq |S_2|, 1 \leq i_j \leq |S_1|\}$. Then, 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 matched to arcs in A_1, i.e., for all $\langle j_l, i_l \rangle, \langle j_r, i_r \rangle \in M$: $(j_l, j_r) \in A_2 \implies (i_l, i_r) \in A_1$. Then, the Arc-Substructure problem (AST) is to determine whether S_2 is an ast of S_1. Analogously to APS, we can define AST(type1, type2), where the arc structures of S_1 and S_2 are, e.g., unlimited or nested.

For each arc $(i_l, i_r) \in A_1$, we define a set $I_1^{(i_l, i_r)}$ (analogously $I_2^{(j_l, j_r)}$ for $(j_l, j_r) \in A_2$) which contains positions of the bases that are inside arc (i_l, i_r) but not inside any arcs that are inside (i_l, i_r),

$$I_1^{(i_l, i_r)} = \{\, i \mid i_l < i < i_r \} \setminus \bigcup_{\substack{(i'_l, i'_r) \in A_1 \\ \wedge\ i_l < i'_l < i'_r < i_r}} \{\, i' \mid i'_l < i' < i'_r \}.$$

If A_1 has a *nested* arc structure then the sets $I_1^{(i_l, i_r)}$ for different arcs are disjoint. We define I_1 as the set of positions of endpoints of the outermost arcs in A_1 and of positions of all bases which are not inside any arcs in A_1. 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).

3 APS(nested,plain)

An instance of APS(nested, plain) is given by (S_1, A_1) and (S_2, A_2), where S_1 has a nested arc structure A_1 while there is no arc in S_2, i.e., $A_2 = \emptyset$. We assume that $n \geq m$ since, otherwise, S_2 cannot be a subsequence of S_1. To solve the problem, we construct a dynamic programming table T of size $O(|A_1|m)$. Each

Computation of maxaps$(S_1[i_1, i_2], S_2[j_1, j_2])$:

- If $i_1 > i_2$ or $j_1 > j_2$ then maxaps$(S_1[i_1, i_2], S_2[j_1, j_2]) := j_1 - 1$.
- If $i_1 = i_2$ then maxaps$(S_1[i_1, i_1], S_2[j_1, j_2]) :=$
  - j_1, if $S_1[i_1] = S_2[j_1]$;
  - $j_1 - 1$, otherwise.
- If $j_1 = j_2$ and $i_1 < i_2$ then maxaps$(S_1[i_1, i_2], S_2[j_1, j_1]) :=$
  - j_1, if $S_1[i_1] = S_2[j_1]$;
  - maxaps$(S_1[i_1 + 1, i_2], S_2[j_1, j_1])$, otherwise.
- If $i_1 < i_2$ and $j_1 < j_2$ then maxaps$(S_1[i_1, i_2], S_2[j_1, j_2]) :=$
  - 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;
  - 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;
  - 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 $(i_l, i_r) \in A_1$, i.e., $i_1 = i_l$.

Fig. 2. Recursive definition of maxaps

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 rightmost position $j' \geq j$ in S_2 such that $S_2[j, j']$ is an arc-preserving subsequence of $S_1[i_l, i_r]$ (or $j - 1$ if no such j' exists). In order to compute table T, we process the arcs of A_1 in the order of their right endpoints. The nested arc structure of A_1 implies 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. Thus, we process the arcs from inside to outside and from left to right.

We divide the algorithm 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 directly inside (i_l, i_r). The second phase deals with those parts of S_1 which are outside all arcs. Here, we use the table entries corresponding to the outermost arcs in A_1. Thereby, we determine whether S_2 is an arc-preserving subsequence of S_1. In both phases, we make use of function maxaps, where maxaps$(S_1[i_1, i_2], S_2[j_1, j_2])$ returns the largest j', $j_1 \leq j' \leq j_2$, 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). Fig. 2 shows how to compute maxaps. Note that maxaps is well-defined since when we compute maxaps$(S_1[i_1, i_2], S_2[j_1, j_2])$ then all entries in T corresponding to arcs in A_1 with both endpoints inside $S_1[i_1, i_2]$ have already been computed before. An outline of the whole algorithm in pseudo-code is given in Fig. 3. An entry of table T, corresponding to $(i_l, i_r) \in A_1$, is computed by

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

By this way of computing $T(i_l, j)$, we match a longest possible substring starting at $S_2[j]$ to $S_1[i_l, i_r - 1]$ or $S_1[i_l + 1, i_r]$ and, by this means, we make sure that the

```
Procedure aps_np
Input: Sequence S_1 with nested arc structure
       and pattern sequence S_2 with no arcs;
Global variable: Array of int T[n][m];
    /* (1) Processing the arcs in A_1 */
    for each (i_l, i_r) ∈ A_1 (ordered by their right endpoints) do
       for j = 1 to m do
          T(i_l, j) := max { maxaps(S_1[i_l, i_r − 1], S_2[j, m]),
                          { maxaps(S_1[i_l + 1, i_r], S_2[j, m]) }
       end for
    end for
    /* (2) Processing, in particular, those parts of S_1
     * which are outside all arcs */
    if (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. 3. Outline in pseudo-code of the algorithm that solves APS(NESTED, PLAIN)

arc-preserving property is maintained: Since there are no arcs in S_2 we can match for an arc $(i_l, i_r) \in A_1$ either only its left endpoint or only its right endpoint to a base in S_2 (or none of them) but not both.
To determine the time needed to compute all entries in table T we, firstly, consider the time one call of maxaps takes, and, then the running time of Algorithm aps_np in Fig. 3 itself (proofs are omitted):

Lemma 1 *A call of maxaps*$(S_1[i_1, i_2], S_2[j_1, j_2])$ *takes* $O(|I'_1|)$ *time if* $S_1[i_1] \in I'_1$ *and* $S_1[i_2] \in I'_1$*, where either* $I'_1 = I_1^{(i_l, i_r)}$ *for an arc* $(i_l, i_r) \in A_1$ *or* $I'_1 = I_1$. □

Theorem 1 APS(NESTED, PLAIN) *can be solved in* $O(nm)$ *time.* □

4 APS(NESTED,CHAIN)

An instance of APS (NESTED, CHAIN) consists of an arc-annotated sequence S_1 with nested arc structure and a pattern S_2 with chain arc structure. As in Section 3, we use a dynamic programming table T of size $|A_1|m$.
We define, for $(j_l, j_r) \in A_2$, a *matching arc set* $\mathrm{MA}^{(j_l, j_r)}$ of all innermost matching arcs $(i_l, i_r) \in A_1$. Observe that no two arcs in $\mathrm{MA}^{(j_l, j_r)}$ are nested, i.e., for $(i_l^1, i_r^1), (i_l^2, i_r^2) \in \mathrm{MA}^{(j_l, j_r)}$, we have either $i_l^1 < i_r^1 < i_l^2 < i_r^2$ or $i_l^2 < i_r^2 < i_l^1 < i_r^1$.
It is decisive for our algorithm that, if several arc matches for an arc in A_2 are possible, we choose an innermost arc match: If S_2 is an aps of S_1 then S_2 can be matched to S_1 by assigning innermost arc matches to all arcs in A_2.
We start with a brief overview on our algorithm which, in essence, consists of two stages. In the first stage, we compute those table entries $T(i_l, j)$, where (i_l, i_r) is an arc in A_1 and $S_2[j]$ is a base inside an arc from A_2 or an endpoint of such an arc, analogously as in Section 3: we process the arcs in A_1 in increasing

Computation of maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_2])$:

- If $i_1 > i_2$ or $j_1 > j_2$ then maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_2]) := j_1 - 1$.
- If $i_1 = i_2$ then maxaps_nc$(S_1[i_1, i_1], S_2[j_1, j_2]) :=$
  - j_1, if $S_1[i_1] = S_2[j_1]$ and $S_2[j_1]$ is not an endpoint;
  - $j_1 - 1$, otherwise.
- If $i_1 < i_2$ and $j_1 = j_2$ then maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_1]) :=$
  - j_1, if $S_1[i_1] = S_2[j_1]$ and $S_2[j_1]$ is not an endpoint;
  - maxaps_nc$(S_1[i_1 + 1, i_2], S_2[j_1, j_1])$, if $S_1[i_1] \neq S_2[j_1]$ and $S_2[j_1]$ is not an endpoint;
  - $j_1 - 1$, if $S_2[j_1]$ is an endpoint.
- If $i_1 < i_2$ and $j_1 < j_2$ and neither $S_1[i_1]$ nor $S_2[j_1]$ is an endpoint then maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_2]) :=$
  - maxaps_nc$(S_1[i_1 + 1, i_2], S_2[j_1 + 1, i_2])$, if $S_1[i_1] = S_2[j_1]$;
  - maxaps_nc$(S_1[i_1 + 1, i_2], S_2[j_1, i_2])$, otherwise.
- If $i_1 < i_2$ and $j_1 < j_2$ and $S_2[j_1]$ is an endpoint of an arc but $S_1[i_1]$ is not then maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_2]) :=$ maxaps_nc$(S_1[i_1 + 1, i_2], S_2[j_1, j_2])$.
- If $i_1 < i_2$ and $j_1 < j_2$ and $S_1[i_1]$ is the left endpoint of an arc (i_l, i_r) then maxaps_nc$(S_1[i_1, i_2], S_2[j_1, j_2]) :=$ maxaps_nc$(S_1[i_r + 1, i_2], S_2[T[i_l, j_1] + 1, j_2])$.

Fig. 4. Recursive definition of maxaps_*nc*

order of their right endpoints. For every arc $(j_l, j_r) \in A_2$, we can, in the same way, also determine its innermost matching arcs $\mathrm{MA}^{(j_l, j_r)}$. In the second stage, we compute the table entries for bases in S_2 which are not inside an arc, using the function maxaps_nc which will be introduced in this section. Table T then is complete and we can compute maxaps_nc$(S_1[1, n], S_1[1, m])$ to decide whether S_2 is an arc-preserving subsequence of S_1.

The new function maxaps_nc extends the function maxaps by additionally taking the arc matching possibilities for arcs in A_2 into account: When processing those parts of S_2 which are not inside an arc, it allows to make use of the precomputed results in table T; thus, it treats the arcs of S_2 like single bases and skips them by matching them to an innermost arc match. The recursive definition of function maxaps_nc is given in Fig. 4. The two stages of the algorithm to solve APS(NESTED, CHAIN) are sketched in the following:

Stage 1: For every arc $(j_l, j_r) \in A_2$, we, firstly, compute $T(i_l, j)$, where $(i_l, i_r) \in A_1$ and $j_l < j < j_r$, i.e., j is inside (j_l, j_r):

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

Secondly, we compute $T(i_l, j_l)$, corresponding to the endpoints of the arc (j_l, j_r): If (i_l, i_r) is an innermost matching arc for (j_l, j_r), then we set $T(i_l, j_l) := j_r$, otherwise we reject the possibility of matching (i_l, i_r) with (j_l, j_r). By computing $T(i_l, j_l)$ in this way, we prefer the *innermost* arc matches in $\mathrm{MA}^{(j_l, j_r)}$ to other arc matches that would be possible. The test whether $(i_l, i_r) \in \mathrm{MA}^{(j_l, j_r)}$ is done as follows: We have an arc match if $\mathrm{maxaps}(S_1[i_l+1, i_r-1], S_2[j_l+1, j_r-1]) = j_r-1$, $S_1[i_l] = S_2[j_l]$, and $S_1[i_r] = S_2[j_r]$. To decide whether it is an *innermost* arc

match, we recall that we process the arcs in A_1 in increasing order by their right endpoints. Therefore, we simply keep track of the previously found innermost arc match. 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 arc match.
Stage 2: After we have all matching possibilities for the arcs in A_2, we can now complete table T by processing those bases in S_2, which are outside all arcs, using function maxaps_nc which guarantees that *every* arc in S_2 has a matching arc in S_1. In the final step of our algorithm, if maxaps_nc$(S_1[1,n], S_2[1,m])$ returns m then S_2 is an arc-preserving subsequence of S_1. In summary, this yields the following result.

Theorem 2 APS(NESTED, CHAIN) *can be solved in* $O(nm)$ *time.* □

5 APS(NESTED,NESTED)

The basic idea how the algorithm for APS(NESTED,NESTED) builds on the algorithms in Sections 3 and 4 is as follows. We can process the bases inside of the innermost arcs in A_2 in the same way as in the algorithm for APS(NESTED,PLAIN). When processing an arc which is not innermost, i.e., there are arcs inside of it, we observe that the arcs which are directly inside this arc form a chain structure. Moreover, when processing the arcs in the order of their right endpoints, they are already processed at this point. Therefore, we can process these arcs in the same way as in the first stage of the algorithm for APS(NESTED,CHAIN).
The algorithm solving APS(NESTED,NESTED) is outlined in Fig. 5. In contrast to the algorithm for APS(NESTED, CHAIN), the first stage is extended to process all arcs in the nested arc structure of A_2: To fill the dynamic programming table as already used in the previous sections, we process the arcs in A_2 from inner to outer arcs and process, for every arc in A_2, the arcs in A_1 from inner to outer arcs. The second stage 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 and, 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.

Theorem 3 APS(NESTED, NESTED) *can be solved in* $O(nm)$ *time.* □

6 AST(UNLIMITED,NESTED)

We use S_1 to denote the sequence with unlimited arc structure 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 structure and $m := |S_2|$.
In contrast to Section 5, the problem is, on the one hand, more general since S_1 can have unlimited arc structure. This makes it impossible to inspect the inside of the arcs by functions like maxaps (Section 3) or maxaps_nc (Section 4) which heavily relied on the nested arc structure of S_1. On the other hand, we

Procedure `aps_nn`
Input: `Sequences` S_1 `and` S_2 `both with nested arc structures;`
Global variable: **Array of int** $T[n][m]$;

```
/************************* Stage 1 ******************************/
for each (j_l, j_r) ∈ A_2 (ordered by their right endpoints) do
   for each j ∈ I_2^(j_l, j_r) do
      for each (i_l, i_r) ∈ A_1 (ordered by their right endpoints) do
         if (j_l, j_r) is an innermost arc then
            T(i_l, j) := max{ maxaps(S_1[i_l, i_r − 1], S_2[j, j_r − 1]),
                              maxaps(S_1[i_l + 1, i_r], S_2[j, j_r − 1]) }
         else
            T(i_l, j) := max{ maxaps_nc(S_1[i_l, i_r − 1], S_2[j, j_r − 1]),
                              maxaps_nc(S_1[i_l + 1, i_r], S_2[j, j_r − 1]) }
         end if
      end for
   end for
   for each (i_l, i_r) ∈ A_1 (ordered by their right endpoints) do
      T(i_l, j_l) := { j_r                                  if (i_l, i_r) ∈ MA^(j_l, j_r),
                     { maxaps_nc(S_1[i_l + 1, i_r], S_2[j_l, m]) otherwise.
   end for
end for
/************************* Stage 2 ******************************/
for each j ∈ I_2 such that S_2[j] is not an endpoint do
   for each (i_l, i_r) ∈ A_1 (ordered by their right endpoints) do
      T(i_l, j) := max{ maxaps_nc(S_1[i_l + 1, i_r], S_2[j, m]),
                        maxaps_nc(S_1[i_l, i_r − 1], S_2[j, m]) }
   end for
end for
if (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. 5. Outline in pseudo-code of the algorithm that solves APS(NESTED, NESTED)

have the additional restriction that every base is 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 constituted much of the computational difficulty of the problems in the previous sections. In the following, we outline how to adapt our dynamic programming techniques from the previous sections to the new problem.

Again, we build a dynamic programming table called T and compute its entries by processing the arcs of a nested arc structure, this time the one of S_2, in the order of their right endpoints. In contrast to the previous sections, only S_2 has a nested arc structure; therefore, 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 $1 \leq i \leq |S_1|$ and arc $(j_l, j_r) \in A_2$ by $T(i, j_r)$.

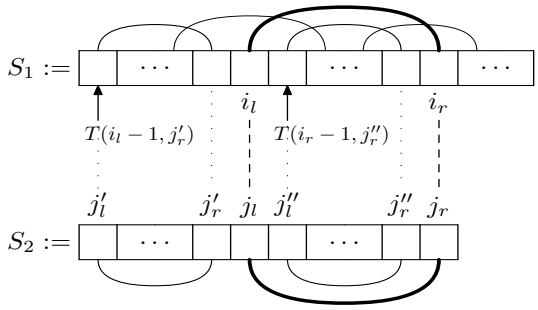

Fig. 6. Example illustrating how we determine whether $(j_l, j_r) \in A_2$ can be matched with $(i_l, i_r) \in A_1$ (match is indicated by the dashed lines) such that $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[1, i_r]$. Here, $S_2[\text{left}(j_r), j_r]$ *is* an ast of $S_1[1, i_r]$: Note that $\text{left}(j_r) = j'_l$ and (1) since $i_l < T(i_r - 1, j''_r)$ we can match $S_2[j''_l, j''_r]$ with $S_1[T(i_r - 1, j''_r), i_r - 1]$, and (2) since $T(i_l - 1, j'_r) \neq -1$, we can match $S_2[j'_l, j'_r]$ with $S_1[T(i_l - 1, j'_r), i_l - 1]$ (these matches are indicated by the dotted lines).

To specify the entries of T, we need two additional definitions. Given j, $1 \leq j \leq m$, we use $\text{left}(j)$ to denote the minimum index $j' \leq j$ such that each of $S_2[j']$, $S_2[j'+1], \ldots, S_2[j]$ is endpoint of an arc $(j_l, j_r) \in A_2$ 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 largest substring ending in $S_2[j]$ in which all bases are or have been examined when processing $S_2[j]$. Given $1 \leq j_1 < j_2 \leq m$ and $1 \leq i \leq |S_1|$, the *best 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 $S_1[i', i]$ (and -1 if no such index exists). Now, we define the meaning of a T entry in a "dual" way compared to the previous sections: There, $T(i_l, j)$ corresponded to some arc $(i_l, i_r) \in A_1$ and $1 \leq j \leq m$ and it specified the *largest* possible 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 $1 \leq i \leq |S_1|$ and an arc $(j_l, j_r) \in A_2$ and it specifies the *smallest* possible substring which *ends* in $S_1[i]$ and of which $S_2[\text{left}(j_l), j_r]$ is an ast, i.e., it contains the best match of $S_2[\text{left}(j_l), j_r]$ at $S_1[i]$.

The decisive idea of the algorithm is to compute, for every arc $(j_l, j_r) \in A_2$ and every $1 \leq i \leq |S_1|$, the best match of $S_2[\text{left}(j_r), j_r]$ at $S_1[i]$. This value is stored in table entry $T(i, j_r)$. If, after all entries are computed, we have $T(|S_1|, m) \geq 1$ then S_2 is an ast of S_1. To compute the table entries, we process the arcs in A_2 by the order of their right endpoints. For each $(j_l, j_r) \in A_2$, we compute entries $T(i, j_r)$, for all $1 \leq i \leq |S_1|$, as follows: We loop through $i = 1, \ldots, |S_1|$, and process, for every i, all arcs ending at $S_1[i]$, $i_r = i$. For each of these arcs $(i_l, i_r) \in A_1$, we, firstly, set $i'' := i_l$ if $S_2[j_l, j_r]$ is an ast of $S_1[i_l, i_r]$ and i_l is larger than the current i'', and, secondly, we compute, the maximum index i', $1 \leq i' \leq i$, such that $S_2[\text{left}(j_r), j_l - 1]$ is an ast of $S_1[i', i'' - 1]$. How to compute i' is explained more precisely further below for an example situation. Now, during the loop $i = 1, \ldots, |S_1|$, we simply keep track of the currently maximum i'' and the currently maximum i' found so far. The currently best i' is stored in $T(i, j_r)$ after all arcs ending in $S_1[i]$ have been processed.

The crucial point above is to determine, given $(i_l, i_r) \in A_1$ and $(j_l, j_r) \in A_2$, the maximum i' such that $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[i', i_r]$ while (i_l, i_r) is matched with (j_l, j_r). We explain how this question is divided into two separate parts using the example shown in Fig. 6:

Part 1: The inside of arc (j_l, j_r). We test 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. 6, $S_2[j_r - 1]$ is right endpoint of an arc $(j_l'', j_r'') \in A_2$. The table entries $T(i, j_r - 1)$, for all $1 \leq i \leq |S_1|$, have already been computed since $j_r - 1 = j_r'' < j_r$. Then, $S_2[j_l + 1, j_r - 1]$ is an ast of $S_1[i_l + 1, i_r - 1]$ iff $i_l < T(i_r - 1, j_r'')$.

Part 2: Left of arc (j_l, j_r). We compute the best match i', $1 \leq i' \leq i_l$, if existent, of $S_2[\text{left}(j_r), j_l - 1]$ at $S_1[i_l - 1]$ (only if such an $i' \geq 1$ exists then $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[1, i_r]$ while matching (i_l, i_r) with (j_l, j_r)). In Fig. 6, $S_2[j_l - 1]$ is a 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, the best i' to be determined can be found in $T(i_l - 1, j_r')$: If $T(i_l - 1, j_r') \neq -1$, then $T(i_l - 1, j_r')$ contains the best match of $S_2[j_l', j_r']$ at $S_1[i_l - 1]$. If, however, $T(i_l - 1, j_r') = -1$ then $S_2[j_l', j_r']$ is not an ast of $S_1[1, i_l - 1]$ and no i' as we search for exists.

Summarizing, if Part 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 $i' \geq 1$ in Part 2, i.e., $S_2[\text{left}(j_r), j_l - 1]$ is an ast of $S_1[i', i_l - 1]$ for $i' \geq 1$ then this i' is the maximum i' such that $S_2[\text{left}(j_r), j_r]$ is an ast of $S_1[i', i_r]$ while (i_l, i_r) is matched with (j_l, j_r). The algorithm computing the table entries of T in this way is outlined in Fig. 7. Regarding the running time, note that, for every arc in A_2, we inspect every arc in A_1 once.

Theorem 4 AST(unlimited, nested) *can be solved in time $O(nm)$.* □

For an easier presentation we focused here on the case of unary alphabet and the restriction that the endpoints of an arc are single bases. It is conceivable that the algorithm can be extended to the case of non-unary alphabet as well as to the problem as stated by Vialette [9] (in this case the endpoints of arcs are given by intervals), showing that his Pattern Matching over 2-Interval Set restricted to $\{<, \sqsubset\}$ structured patterns is solvable in $O(n \log n + nm)$ time (details omitted).

References

1. J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Towards optimally solving the Longest Common Subsequence problem for sequences with nested arc annotations in linear time. In *Proc. of 13th CPM*, number 2373 in LNCS, pages 99–114, 2002. Springer.
2. V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. In *Proc. of 6th CPM*, number 937 in LNCS, pages 1–16, 1995. Springer.
3. N. El-Mabrouk and M. Raffinot. Approximate matching of secondary structures. In *Proc. of 6th ACM RECOMB*, pages 156–164, 2002. ACM Press.
4. P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proc. of 10th CPM*, number 1645 in LNCS, pages 270–280, 1999. Springer.

Procedure ast_un

```
Input: Sequence S_1 with unlimited arc structure and sequence S_2
       with nested arc structure, over unary alphabet, in both
       sequences every base being endpoint of at least one arc;
Global variable: Array of int T[|S_1|][m];

   for each (j_l, j_r) ∈ A_2 (ordered by their right endpoints) do
      i' := −1; /* currently best match for S_2[left(j_r), j_r] */
      i'' := −1; /* currently best match for S_2[j_l, j_r] */
      for i = 1 to |S_1| do
         for each (i_l, i_r) ∈ A_1 such that i = i_r do
            /* (1) Compute the best 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 best match i' such that
             * S_2[left(j_r), j_r] is ast of S_1[i', i]. */
            if S_2[j_l − 1] is right endpoint of an arc then
               i' := max(i', T(i'' − 1, j_l − 1));    /* T(-1,j):=-1 for all j */
            else
               i' := max(i', i'');
            end if;
         end for;
         T(i, j_r) := i';
      end for;
   end for;
   if (T(|S_1|, m) ≠ −1)
      then print 'S_2 is arc substructure of S_1.';
      else print 'S_2 is not arc substructure of S_1.';
   end if;
```

Fig. 7. Outline in pseudo-code of the algorithm that solves AST(UNLIMITED, NESTED).

5. T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In *Proc. of 11th CPM*, number 1848 in LNCS, pages 154–165, 2000. Springer. To appear in *Journal of Discrete Algorithms*.
6. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology* 9(2): 371–388, 2002.
7. G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. In *Proc. of 28th ICALP*, number 2076 in LNCS, pages 444–455, 2001. Springer. To appear in *Journal of Computer and System Sciences*.
8. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing* 24(2): 340–356, 1995.
9. S. Vialette. Pattern matching problems over 2-interval sets. In *Proc. of 13th CPM*, number 2373 in LNCS, pages 53–63, 2002. Springer.