A Remark on the Subsequence Problem for Arc-Annotated Sequences with Pairwise Nested Arcs

Peter Damaschke
Department of Computer Science and Engineering, Chalmers University
41296 Göteborg, Sweden
ptr@cs.chalmers.se

Abstract

The Arc-Preserving Subsequence (APS) problem appears in the comparison of RNA structures in computational biology. Given two arc-annotated sequences of length n and $m < n$, APS asks if the shorter sequence can be obtained from the longer one by deleting certain elements along with their incident arcs. It is known that APS with pairwise nested arcs can be solved in $O(mn)$ time. We give an algorithm running in $O(m^2+n)$ time in the worst case, actually it is even faster in particular if the shorter sequence has many arcs. The result may serve as a buidling block for improved APS algorithms in the general case.

Keywords: RNA structure, pattern matching, dynamic programming

1 Introduction

RNA secondary structure comparison is of vital interest in computational biology, because structure determines to a large extent the function of non-coding RNA molecules. A common model for RNA structure is arc-annotated sequences. Given an alphabet Σ, an arc-annotated sequence (S, P) is a sequence S, that is, a string over Σ, together with a set P of disjoint arcs, i.e., pairs of elements of the sequence. Elements of a pair can be arbitrarily far apart in the sequence. In the RNA case, Σ is the "alphabet" of ribonucleotids A,C,G,U, and arcs represent hydrogen bonds. These bonds are disjoint, therefore we assume that no element is incident to more than one arc.

The Arc-Preserving Subsequence (APS) problem takes as input two arc-annotated sequences (S, P) and (T, Q) of length n and $m < n$, respectively, and asks if (T, Q) is contained in (S, P) in the sense that one can obtain (T, Q) by deleting certain elements of (S, P) along with their incident arcs. Equivalently, we may think of an embedding of (T, Q) in (S, P) that establishes an induced subgraph relation of ordered graphs and preserves the order of elements in T, but is in general not contiguous. Note carefully that we use the term *subsequence* in this non-contiguous sense: T is called a subsequence of S if T is obtained by deleting certain elements from S. In contrast, we say that T is a *substring* of S if T occurs in S without gaps. For any sequence S we use $S[i]$ and $S[i..j]$ to denote the ith element of S and the segment of S from the ith to the jth symbol (inclusively). By convention, $S[i..j]$ is empty if $i > j$. If T is a subsequence of S, we say that an embedding *occupies position* k in S if some element of T is mapped to $S[k]$. Sometimes we may not strictly distinguish formally between a symbol (element of the alphabet) and an occurence of the symbol at a specific position in a sequence, but the correct meaning will be clear from context.

The APS problem appears in pattern searching in RNA databases and as a subproblem in other (harder) RNA comparison tasks [1, 3, 4, 5, 6, 7, 8]. Recently, [2] presented a comprehensive study of APS with several natural restrictions imposed on the structure of arc sets, with the goal to understand the borderline between polynomial-time solvable and NP-complete cases. We refer to [2] for an overview of known complexity results. Given four endpoints $i < j < k < l$ of two arcs, these two arcs are called *nested* if they connect i with l and j with k. The two arcs are *crossing* if they connect i with k and j with l, and *non-crossing* in all other cases. We focus on problem APS with *pairwise nested arcs*, defined as the case when, in both (T, Q) and (S, P), any two arcs are nested.

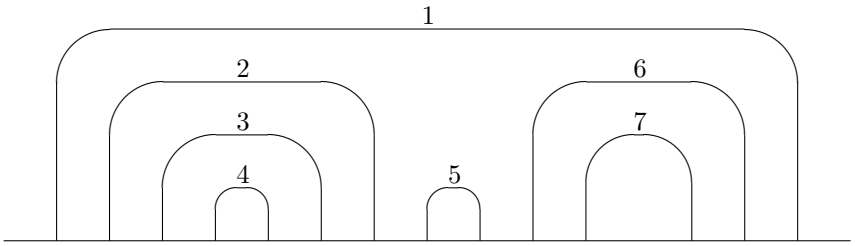


Figure 1. An example of pairwise non-crossing arcs. Arcs in $\{1, 2, 3, 4\}$ or in $\{1, 6, 7\}$ are also pairwise nested.

APS is solvable in $O(mn)$ time if both (T, Q) and (S, P) have only pairwise nested arcs. The time bound holds even in the more general case that no crossing arcs appear [4]. (One has to be careful with the terminology. The latter case is usually denoted APS(NESTED, NESTED), i.e., the condition NESTED refers to non-crossing arcs.) It was explicitly mentioned as an open problem in [2] whether APS for pairwise nested arcs can be solved faster. In the present note we give an affirmative answer, proving a worst-case bound $O(m^2+n)$. The idea is a rather simple use of monotonicity properties inside dynamic programming (Section 2). The new worst-case bound is, of course, already an improvement if m is significantly smaller than n. Furthermore, our analysis in Section 3 shows that the actual running time is typically even better, especially if arcs in (T, Q) are dense.

Pairwise nested arcs is a very special case that rarely appears in pure form in RNA strings. However, its study is motivated, observing that our result can be further used as a building block for efficient pattern matching in more general instances, as we briefly sketch in Section 4.

2 An Auxiliary String Embedding Problem

We define an auxiliary problem which we call Conflict-Free Prefix Embedding (CFPE): The input of CFPE consists of four sequences S_1, S_2, T_1, T_2, with $|T_1| = m_1$, $|T_2| = m_2$, and $|S_1| = |S_2| = n$. Assuming that T_i is a subsequence of S_i $(i = 1, 2)$, consider a pair of embeddings. We say that *embeddings are in conflict* if there exists some k such that position k is occupied in both S_1 and S_2. We define a boolean-valued function E by $E(x_1, x_2, y) = 1$ if there exist conflict-free embeddings of $T_1[1..x_1]$ in $S_1[1..y]$ and $T_2[1..x_2]$ in $S_2[1..y]$, and $E(x_1, x_2, y) = 0$ otherwise. With the understanding that $0 < 1$, obviously E is monotone increasing in argument y and monotone decreasing in x_1 and x_2.

Our problem CFPE is to determine, for all $x_1 \leq m_1$ and $x_2 \leq m_2$, the smallest $y \leq n$ with $E(x_1, x_2, y) = 1$, or to report that $E(x_1, x_2, n) = 0$. That is, the output is an $m_1 \times m_2$ matrix with these y (or a special symbol in the negative case) as entries.

Theorem 1 *Problem CFPE is solvable in time $O(m_1 m_2 + k)$, where $k = \min\{y | E(m_1, m_2, y) = 1\}$. The result holds regardless of the alphabet size.*

Proof. For any fixed y we may think of the $E(x_1, x_2, y)$ as a matrix with entries 0 or 1. By the monotonicity properties of E, the 1-entries and 0-entries form an "upper staircase" and "lower staircase" subset, respectively, of the

matrix. We say that two matrix entries at (x_1, x_2) and (x'_1, x'_2) are *adjacent* if either $x_1 = x'_1$ and $|x_2 - x'_2| = 1$, or $x_2 = x'_2$ and $|x_1 - x'_1| = 1$.

We shall compute all $E(x_1, x_2, y)$ from the $E(x_1, x_2, y-1)$. Clearly, it is enough to consider those (x_1, x_2) which satisfy $E(x_1, x_2, y-1) = 0$ but are adjacent to some 1-entries in the $(y-1)$th matrix. (When y grows by 1, a conflict-free embedding can accommodate at most one more element from either T_1 or T_2.) We call these matrix elements *switch candidates*. We get the dynamic programming formula $E(x_1, x_2, y) = E(x_1, x_2, y-1) + 1$ in the following cases:
$E(x_1 - 1, x_2, y-1) = 1$ and $T_1[x_1] = S_1[y]$, or
$E(x_1, x_2 - 1, y-1) = 1$ and $T_2[x_2] = S_2[y]$.

By monotonicity, every element of the $m_1 \times m_2$ matrix we are working with is switched at most once while y is growing, hence we have $O(m_1 m_2)$ switches. For the time bound, however, we must take into account the time needed to determine the switch candidates that will actually switch in every step, and update the set of switch candidates. This task can be managed in $O(m_1 m_2)$ time in total, by using a suitable data structure: Note that in every step y every switch candidate "knows" by its position (x_1, x_2) in the matrix the alphabet symbol in S_1 or S_2 whose occurrence at position y would make (x_1, x_2) a 1-element. Thus we proceed as follows. For each alphabet symbol in S_1 and S_2, respectively, we initialize a queue for the corresponding switch candidates. (Actually we initialize every queue when the symbol appears the first time, hence the procedure does not rely on a fixed alphabet size.) When this symbol actually appears as $S_1[y]$ or $S_2[y]$, respectively, the queue assigned to it is emptied and the switches are executed. Moreover, after every switch, new switch candidates are created in the obvious way: Take the next element in the direction of the switch and put it in the designated queue. We have to remark that a 0-element in a corner of the staircase may appear in two queues (belonging to S_1 and S_2), and it may remain in one queue even after the switch, but such obsolete switch candidates being already 1-elements are recognized later when the queue is emptied. Clearly, this is yet another constant-time operation for each element and does not affect the time bound. The algorithm stops as soon as $E(m_1, m_2, y) = 1$. Altogether we need $O(k)$ time to read the symbols in S_1 and S_2, and to initialize and call the queues. □

A more tricky implementation and storage of function values (i.e., a concise description of the output) can make the algorithm work even faster, but since this does not improve our worst-case time bound, we discuss it only as a side remark: Let us maintain only the horizontal and vertical lines in the matrix that separate 1-elements from 0-elements. Due to the conditions for a switch, only entire lines are moved in one step. This reduces the total time to the number

of different lines, which in turn equals the number of corners during the course of the algorithm. This number can be significantly smaller than the matrix size $m_1 m_2$. (We wonder if it can be easily computed in advance from the given sequences. This would allow to choose the straightforward implementation or the advanced one that involves some administration overhead. We leave this as an open question.) Note also that a corner is split in two new corners only by a conflict, and that corners can coalesce. Anyway, we will now apply Theorem 1 to APS with pairwise nested arcs.

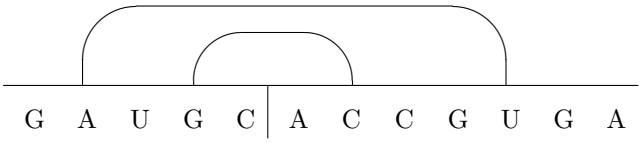
3 The Result for Pairwise Nested Arcs

Given two arc-annotated sequences (S, P) and (T, Q) of length n and m, respectively, both with pairwise nested arcs, we solve APS through a sequence of instances of CFPE. Before the construction of an embedding we have to do some preprocessing.

First we fix in S some arbitrary position between two neighbored elements inside the innermost arc of P. We split S in two sequences S^L, S^R to the left and to the right of the split position. We reverse S^R, and insert occurences of a dummy symbol (which is not in Σ) both in S^L and in the reverse of S^R, in such a way that the following holds:

- any two elements of S connected by an arc of P are aligned in the resulting pair of sequences,
- any symbol that was not involved in an arc of P is aligned to a dummy symbol, and
- the two filled-up sequences have equal lengths.

Trivially, such an alignment exists and can be constructed in $O(n)$ time. We denote the resulting sequences again S^L and S^R.



left	G	+	+	A	U	+	+	G	C	+
right, reversed	+	A	G	U	+	G	C	C	+	A

Figure 2. An example for the transformation of (S, P).

Arcs of Q are totally ordered inwards, in an obvious sense. Let us denote them $q_1, \ldots, q_k$, where q_1 is the outermost and q_k the innermost arc. Let $T[l_i]$ and $T[r_i]$ be the elements connected by q_i. We abbreviate the (possibly empty) substrings between endpoints of arcs as follows. (Remember the convention for empty substrings.)
$T_0^L = T[1..(l_1 - 1)]$ and $T_0^R = T[(r_1 + 1)..m]$,
$T_i^L = T[(l_i + 1)..(l_{i+1} - 1)]$ and $T_i^R = T[(r_{i+1} + 1)..(r_i - 1)]$ for $i = 1, \ldots, k-1$,
$T^M = T[(l_k + 1)..(r_k - 1)]$.
Let $m_i^L = |T_i^L|$, $m_i^R = |T_i^R|$, and $m^M = |T^M|$.

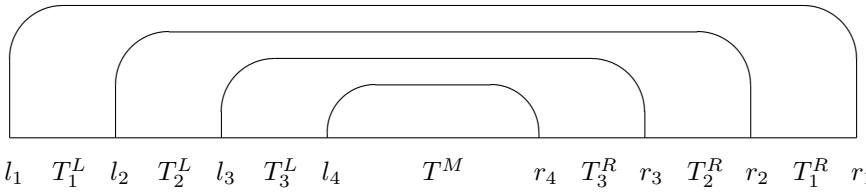


Figure 3. Denotations in (T, Q).

With these denotations we prove:

Theorem 2 *APS for arc-annotated sequences with pairwise nested arcs is solvable in time $O(m^2 + n)$ in the worst case.*

Proof. First we embed T_0^L and the reverse of T_0^R conflict-free in S^L and in the reverse of S^R, respectively. Using Theorem 1 we determine the smallest y_0 such that the prefixes of length y_0 of S^L and of the reversed S^R can accommodate T_0^L and the reversed T_0^R, respectively, in $O(m_0^L m_0^R + y_0)$ time. Then, we find the outermost arc of P which does not touch these prefixes and matches q_1. Since this requires only symbol comparisons, the time for this search is proportional to the number of arcs checked in P. From S^L and the reverse of S^R we cut away the prefixes until the (aligned) endpoints of the matching arc. This finishes the phase that embeds arc q_1 and everything of T outside this arc.

We continue in the same way with T_i^L, T_i^R, q_{i+1}, for $i = 1, \ldots, k-1$, always with the *remaining* S^L, S^R. The time for the ith phase is $O(m_i^L m_i^R + y_i)$ (where y_i has the similar meaning as y_0 above), plus the number of arcs checked in P.

The last phase is slightly different. Using Theorem 1 we embed T^M and its reverse conflict-free in S^L and in the reverse of S^R, in time $O((m^M)^2 + y_k)$. If this succeeds for some pair x_1, x_2 with $x_1 + x_2 \geq m^M$, we have found an embedding of (T, Q) in (S, P). If not, or if S^L, S^R was already emptied earlier, we report that no embedding can exist.

Proving correctness is straightforward. We have the following induction hypothesis: The algorithm finds the outermost arc p_i in P such that $T[1..l_i]T[r_i..m]$

can be embedded in $S[1..a_i]S[b_i..n]$, where $S[a_i]$ and $S[b_i]$ are the elements connected by p_i. For $i = 1$ this is obvious from the specification of the first phase. Assume that the hypothesis holds for i. By definition of p_{i+1} there exists an embedding of $T[1..l_{i+1}]T[r_{i+1}..m]$ in $S[1..a_{i+1}]S[b_{i+1}..n]$. If we denote by $S[a'_i], S[b'_i]$ the elements connected by the arc of P that matches q_i in this embedding, it induces an embedding of $T[l_i..l_{i+1}]T[r_{i+1}..r_i]$ in $S[a'_i..a_{i+1}]S[b_{i+1}..b'_i]$. Furthermore, since $S[a_i], S[b_i]$ are the ends of the outermost possible arc that matches q_i in a complete embedding, we have $a_i \leq a'_i$ and $b'_i \leq b_i$, hence $T[l_i..l_{i+1}]T[r_{i+1}..r_i]$ can also be embedded in the longer $S[a_i..a_{i+1}]S[b_{i+1}..b_i]$. Since phase $i + 1$ finds the shortest possible pair of embeddings inwards, and starts by inducton hypothesis from both ends of p_i, it does not pass arc p_{i+1}, which completes the inductive step. Finally we use the induction hypothesis for k to complete the correctness argument. Suppose that an embedding exists. Since our algorithm has already found an embedding of $T[1..l_k]T[r_k..m]$ in $S[1..a_k]S[b_k..n]$ with q_k matched onto the *outermost* possible arc p_k, any complete embedding must include an embedding of T^M in $S[a_k + 1..b_k - 1]$. This can be viewed as a conflict-free pair of embeddings of some prefix and some reversed suffix of T^M in the prefix and reversed suffix of $S[a_k + 1..b_k - 1]$ to the left and right, respectively, of the split position, where this prefix and suffix of T^M together have to build the whole T^M. Now, obviously, existence of a solution with $x_1 + x_2 \geq m^M$ is equivalent to this condition.

We have $\sum_{i=0}^{k} y_i = O(n)$, since the y_i refer to mutually disjoint segments of S. Similarly, all m_i^L, m_i^R are lengths of mutually disjoint segments of T. Hence we need time $O((\sum_{i=0}^{k-1} m_i^L m_i^R + y_i) + (m^M)^2 + y_k + |P|) = O(m^2 + n)$. □

Actually, the worst case $O(m^2)$ appears only when Q has few arcs (but in this case the better implementation of dynamic programming sketched after Theorem 1 may take effect), otherwise the quadratic term is much smaller. Note that the algorithm works inwards on T and S in a greedy fashion. Dynamic programming is used only to determine the shortest embeddings of the pairs of substrings of T between the endpoints of neighbored arcs in Q.

In [2] it was shown that APS with pairwise crossing arcs (in both P and Q) can be solved by m-fold application of any algorithm for APS with pairwise nested arcs, resulting in the time bound $O(m^2 n)$. With our Theorem 2 we can immediately improve this result:

Corollary 3 *APS for arc-annotated sequences with pairwise crossing arcs is solvable in time $O(m^3 + mn)$ in the worst case.*

4 Outlook

We suggest extensions of the presented result along the following lines. Arc structures (S, P) without crossing arcs are widely considered an important case of RNA structure. Obviously they can be represented by trees (or forests) where every node v stands for a set N_v of pairwise nested arcs so that all other arcs are inside and outside the innermost and outermost arc of N_v, respectively. (For instance, in Figure 1, root $\{1\}$ has three children representing $\{2, 3, 4\}$, $\{5\}$, $\{6, 7\}$.) Often the tree has only a few nodes, due to abundant pairwise nested arcs. Let us informally discuss the case of APS where pattern (T, Q) still has pairwise nested arcs, and (S, P) is free of crossings. Since the algorithm in Theorem 2 works inwards, we can readily apply it top-down in the tree. An argument as in Theorem 2 shows that a greedy embedding into N_v is optimal at every node v. The new feature is that we must branch, i.e., pass the remainder of (T, Q) to the children of a finished node v, and continue recursively. In every branching phase we can get embedding conflicts only if a unique child of v remains for continued embedding of (T, Q) (that is, if only the subtree rooted at v can accommodate the next arc of Q). Similarly as in Theorem 2, the time *on every path* of the computation is a sum of squares of lengths that sum up to m, and the second term in the time bound remains $O(n)$, since the segments of S to be processed in the tree nodes are disjoint. The overall time depends on the size and shape of the tree. Alternatively, one may also combine the scheme from [4] with a bottom-up variant of the algorithm in Theorem 2 (where the innermost matching arc in every tree path would first be determined in a binary search procedure). More elaboration on the sketched ideas, in order to identify cases still solvable in $o(mn)$ time, is left for further research.

Acknowledgments

This work has been partially supported by the Swedish Research Council (Vetenskapsrådet) through the project "Algorithms for searching and inference in genetics", grant no. 621-2002-4574. The author would like to thank Ferdinando Cicalese for enlightening discussions during a stay at the University of Bielefeld, and the anonymous referees for careful reading and valuable comments.

References

[1] J. Alber, J. Gramm, J. Guo, R. Niedermeier, Computing the similarity of two sequences with nested arc annotations, *Theoretical Computer Science* 312 (2004), 337-358

[2] G. Blin, G. Fertin, R. Rizzi, S. Vialette, What makes the Arc-Preserving Subsequence problem hard?, *1st International Workshop on Bioinformatics Research and Applications IWBRA 2005*, *LNCS* 3515, 860-868

[3] P. Evans, Algorithms and complexity for annotated sequence analysis, PhD thesis, Univ. Victoria, 1999

[4] J. Gramm, J. Guo, R. Niedermeier, Pattern matching for arc-annotated sequences, *22nd Conference on Foundations of Software Technology and Theoretical Computer Science FSTTCS 2002*, *LNCS* 2556, 182-193

[5] J. Guo, Exact algorithms for the longest common subsequence problem for arc-annotated sequences, Master's thesis, Univ. Tübingen, 2002

[6] T. Jiang, G.H. Lin, B. Ma, K. Zhang, The longest common subsequence problem for arc-annotated sequences, *11th Symposium on Combinatorial Pattern Matching CPM 2000*, *LNCS* 1848, 154-165

[7] G. Lin, Z.Z. Chen, T. Jiang, J. Wen, The longest common subsequence problem for sequences with nested arc annotations, *Journal of Computer and System Sciences* 65 (2002), 465-480

[8] K. Zhang, L. Wang, B. Ma, Computing the similarity between RNA structures, *Theoretical Computer Science* 276 (2002), 111-132