Fast Arc-Annotated Subsequence Matching in Linear Space

Philip Bille and Inge Li Gørtz

Technical University of Denmark. E-mail: {phbi, ilg}@imm.dtu.dk

Abstract. An arc-annotated string is a string of characters, called bases, augmented with a set of pairs, called arcs, each connecting two bases. Given arc-annotated strings P and Q the arc-preserving subsequence problem is to determine if P can be obtained from Q by deleting bases from Q. Whenever a base is deleted any arc with an endpoint in that base is also deleted. Arc-annotated strings where the arcs are "nested" are a natural model of RNA molecules that captures both the primary and secondary structure of these. The arc-preserving subsequence problem for nested arc-annotated strings is basic primitive for investigating the function of RNA molecules. Gramm et al. [ACM Trans. Algorithms 2006] gave an algorithm for this problem using $O(nm)$ time and space, where m and n are the lengths of P and Q, respectively. In this paper we present a new algorithm using $O(nm)$ time and $O(n+m)$ space, thereby matching the previous time bound while significantly reducing the space from a quadratic term to linear. This is essential to process large RNA molecules where the space is a likely to be a bottleneck. To obtain our result we introduce several novel ideas which may be of independent interest for related problems on arc-annotated strings.

1 Introduction

An *arc-annotated string* S is a string augmented with an *arc set* A_S. Each character in S is called a *base* and the arc set A_S is a set of pairs of positions in S connecting two distinct bases. We say that S is a *nested arc-annotated string* if no two arcs in A_S share an endpoint and no two arcs cross each other, i.e., for all $(i_l, i_r), (i'_l, i'_r) \in A_S$ we have that $i_l < i'_l < i_r$ iff $i_l < i'_r < i_r$. Given arc-annotated strings P and Q we say that P is a *arc-preserving subsequence* (APS) of Q, denoted $P \sqsubseteq Q$, if P can be obtained from Q by deleting 0 or more bases from Q. Whenever a base is deleted any arc with an endpoint in that base is also deleted. The *arc-preserving subsequence problem* (APS) is to determine if $P \sqsubseteq Q$. If P and Q are both nested arc-annotated strings we refer to the problem as the *nested arc-preserving subsequence problem* (NAPS). Fig. 1(a) shows an example of nested arc-annotated strings.

Ribonucleic acid (RNA) molecules are often modeled as nested arc-annotated strings. Here, the string consists of bases from the 4-letter alphabet $\{\mathrm{A}, \mathrm{U}, \mathrm{C}, \mathrm{G}\}$, called the *primary structure*, and an arc set consisting of pairings between bases, called the *secondary structure*. RNA molecules are central for many biological

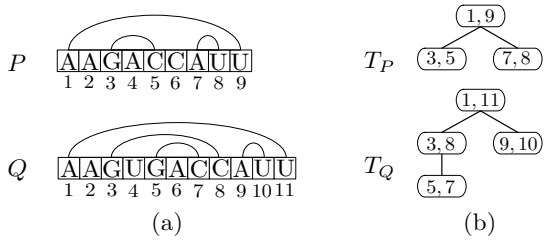

Fig. 1. (a) Nested arc-annotated strings P and Q. Here, P and Q contain arcs connecting their first and last bases. (b) The corresponding trees T_P and T_Q induced by the arcs.

functions and NAPS is a basic primitive for investigating the precise functionality of RNA molecules. The key idea is to model a specific function of RNA molecules as an arc-annotated string F. Given a RNA molecule R we can then determine (to some extent) if R performs the same function by computing if $F \sqsubseteq R$.

Building on earlier work in a related model of RNA molecules by Vialette [16], Gramm et al. [10] introduced and gave an algorithm for NAPS using $O(nm)$ time and space, where m and n are the lengths of P and Q, respectively. Kida [12] presented an experimental study of this algorithm and Damaschke [8] considered a special restricted case of the problem.

Results We assume a standard unit-cost RAM model with word size $\Theta(\log n)$ and a standard instruction set including arithmetic operations, bitwise boolean operations, and shifts. The space complexity is the number of words used by the algorithm. All of the previous results are in same model of computation. Throughout the paper P and Q are nested arc-annotated strings of lengths m and n, respectively. In this paper we present a new algorithm with the following complexities.

Theorem 1. *Given nested arc-annotated strings P and Q of lengths m and n, respectively, we can solve the nested arc-preserving subsequence problem in time $O(nm)$ and space $O(n+m)$.*

Hence, we match the running time of the currently fastest known algorithm and at the same time we improve the space from $O(nm)$ to $O(n+m)$. This space improvement is critical for processing large RNA molecules. In particular, an algorithm using $O(nm)$ space quickly becomes infeasible, even for moderate sizes of RNA molecules, due to costly accesses to external memory. An algorithm using $O(m+n)$ space is much more scalable and allows us to handle significantly larger RNA molecules. Furthermore, we note that obtaining an algorithm using $O(nm)$ time and $o(nm)$ space is mentioned as an open problem in Gramm et al. [10].

Compared to the previous work by Gramm et al. [10] our algorithm is not only more space-efficient but also simpler. Our algorithm is based on a single unified dynamic programming recurrence, whereas the algorithm by Gramm et al. requires computing and tabulating auxiliary information in multiple phases

mixed with dynamic programming. Our approach allows us to better expose the features of NAPS and is essential for obtaining a linear space algorithm.

Techniques As mentioned above, our algorithm is based on a new dynamic programming recurrence. Essentially, the recursion expresses for any pair of substrings P' and Q' of P and Q, respectively, the longest prefix of P' which is an arc-preserving subsequence of Q' in term of smaller substrings of P' and Q'. We combine several new ideas with well-known techniques to convert our recurrence into an efficient algorithm.

First, we organize the dynamic programming recurrence into Γ *sequences*. A Γ sequence for a given substring Q' of Q is a simple $O(m)$ space representation of the longest arc-preserving subsequences of each prefix of P in Q'. We show how to efficiently manipulate Γ sequences to get new Γ sequences using a small set of simple operations, called the *primitive operations*. Secondly, we organize the computation of Γ sequences using a recursive algorithm that traverses the tree structure of the arcs in Q. The algorithm computes the Γ sequence for each arc in Q using the primitive operations. To avoid storing too many Γ sequences during the traversal we direct the computation according to the well-known *heavy-path decomposition* of the tree. This leads to an algorithm that stores at most $O(\log |A_Q|)$ Γ sequences. Since each Γ sequence uses $O(m)$ space the total space becomes $O(m \log |A_Q| + n)$.

Finally, to achieve linear space we exploit a structural property of Γ sequences to compress them efficiently. We obtain a new representation of Γ sequences that only requires $O(m)$ *bits*. Plugging in the new representation into our algorithm the total space becomes $O(n + m)$ as desired. However, the resulting algorithm requires many costly compressions and decompressions of Γ sequences at each arc in the traversal. As a practical and more elegant solution we show how to augment the compressed representation of Γ sequences using standard *rank/select indices* to obtain constant time random access to elements in Γ sequences. This allows us to compress each Γ sequence only once and avoid decompression entirely without affecting the complexity of the algorithm.

Related Work Arc-annotated strings are a natural model of RNA molecules that captures both the primary and secondary structure of these. Consequently, a wide range of pattern matching problems for them have been studied, see e.g., [1–3, 6, 9, 10, 14]. Among these, NAPS is one of the most basic and fundamental problems.

The NAPS problem generalizes the *tree inclusion problem* for ordered trees [4, 7,13]. Here, the goal is to determine if a tree can be obtained from another tree by deleting nodes. This is equivalent to NAPS where all bases in both strings have an incident arc. The authors have shown how to solve the tree inclusion problem in time $O(nm/\log n + n \log n)$ and space $O(n+m)$ [4]. Compared to our current result for NAPS the space complexity is the same but the time complexity for tree inclusion is a factor $O(\log n)$ better for most values of m and n. Though our obtained complexities for the tree inclusion problem and NAPS are very similar, the ideas and techniques behind the results differ significantly. While

the definition of the two problems seems very similar it appears that the more general NAPS is significantly more complicated. We leave it as an interesting research direction to determine the precise relationship between NAPS and the tree inclusion problem.

Several generalizations of NAPS have also been studied relaxing the requirement that arcs should be nested [5,9,10]. In nearly all cases the resulting problem becomes NP-complete.

Due to lack of space some of the proof are omitted from this extended abstract. They can be found in the full version of the paper.

2 Preliminaries and Notation

Let S be an arc-annotated string with arc set A_S. The length of S is the number of bases in S and is denoted $|S|$. We will assume that our input strings P and Q have the arcs $(1, |P|)$ and $(1, |Q|)$, respectively. If this is not the case we may always add additional connected bases to the start and end of P and Q without affecting the solution or complexity of the problem. We do this only to ensure that the nesting of the arcs form a tree (rather than a forest) which simplifies the presentation of our algorithm.

The *arc-annotated substring* $S[i_1, i_2]$, $1 \leq i_1, i_2 \leq |S|$, is the string of bases starting at i_1 and ending at i_2. The arc set associated with $S[i_1, i_2]$ is the subset of A_S of arcs with both endpoints in $[i_1, i_2]$. We define $S[i_1] = S[i_1, i_1]$ and $S[i_1, i_2] = \epsilon$ (the empty string) if $i_1 > i_2$. Note the arc set of an arc-annotated string of length ≤ 1 is also empty. A *split* of S is a partition of S into two substrings $S[1, i]$ and $S[i + 1, |S|]$, for some i, $0 \leq i \leq |S|$. The split is an *arc-preserving split* if no arcs in A_S cross i, i.e., all arcs either have both endpoints in $S[1, i]$ or $S[i+1, |S|]$. We say that the index i *induces* a (arc-preserving) split of S.

An *embedding* of P in Q is an injective function $f : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ such that

1. for all $j \in \{1, \ldots, m\}$, $P[j] = Q[f(j)]$. (base match condition)
2. for all indices $j_l, j_r \in \{1, \ldots, m\}$, $(j_l, j_r) \in A_P \Leftrightarrow (f(j_l), f(j_r)) \in A_Q$. (arc match condition)
3. for all $j \in \{1, \ldots, m\}$, $i < j \Leftrightarrow f(i) < f(j)$. (order condition)

If $f(j) = i$ we say that j is *matched* to i in the embedding. From the definition of arc-preserving subsequences we have that $P \sqsubseteq Q$ iff there is an embedding of P in Q.

3 The Dynamic Programming Recurrence

In this section we give our dynamic programming recurrence for the NAPS problem. Essentially, the recursion expresses for any pair of substrings P' and Q' of P and Q, respectively, the longest prefix of P' which is an arc-preserving subsequence of Q' in terms of smaller substrings of P' and Q'.

We show the following key properties of arc-preserving splits.

Lemma 1 (Splitting Lemma). *Let P' and Q' be arc-annotated substrings of P and Q, respectively, and let (Q_1, Q_2) be any arc-preserving split of Q'.*

(i) If $P' \sqsubseteq Q'$ then there exists an arc-preserving split (P_1, P_2) of P' such that $P_1 \sqsubseteq Q_1$ and $P_2 \sqsubseteq Q_2$.
(ii) Let (P_1, P_2) be an arc-preserving split of P'. Then $P_1 \sqsubseteq Q_1$ and $P_2 \sqsubseteq Q_2 \Rightarrow P' \sqsubseteq Q'$.

For $1 \leq j_l \leq m$, $l \in \{1, 2\}$ and $1 \leq i_1 \leq i_2 \leq n$ define $\gamma(j_1, j_2, i_1, i_2)$ to be the largest integer k such that $P[j_1, k] \sqsubseteq Q[i_1, i_2]$ and k induces an arc-preserving split of $P[j_1, j_2]$. It follows that $\gamma(1, m, 1, n) = m$ if and only if $P \sqsubseteq Q$.

The Splitting Lemma gives us a very useful property of γ: The requirement that k induces an arc-preserving split of $P[j_1, j_2]$ in the definition of γ implies that if there exists an embedding f of $P[k+1, j_2]$ in $Q[i_2, i]$ for some i then by the Splitting Lemma the embedding of $P[j_1, k]$ in $Q[i_1, i_2]$ (which exists by the definition of γ) can be extended with f to get an embedding of $P[j_1, j_2]$ in $Q[i_1, i]$. This would not be true if we dropped the requirement that k induces an arc-preserving split of $P[j_1, j_2]$. Formally,

Corollary 1. *Let i be an index inducing an arc-preserving split of $Q[i_1, i_2]$. Then, $\gamma(j_1, j_2, i_1, i_2) = \gamma(\gamma(j_1, j_2, i_1, i) + 1, j_2, i + 1, i_2)$.*

Intuitively, the corollary says that to compute the largest prefix of P that can be embedded in Q we can greedily match the bases and right endpoints of arcs of P as much to the left in Q as possible. The dynamic programming recurrence for γ is as follows.

Base cases. $\gamma(j_1, j_2, i_1, i_2)$ is equal to

$$\begin{cases} j_1 - 1 & \text{if } j_1 > j_2, \quad (1) \\ j_1 & \text{if } i_1 = i_2 \text{ and } P[j_1] = Q[i_1] \text{ and} \\ & \quad (j_1, j_r) \notin A_P \text{ for all } j_r \leq j_2, \quad (2a) \\ j_1 - 1 & \text{if } i_1 = i_2 \text{ and } (P[j_1] \neq Q[i_1] \text{ or} \\ & \quad (j_1, j_r) \in A_P \text{ for some } j_r \leq j_2). \quad (2b) \end{cases}$$

Recursive cases. $i_1 < i_2$ and $j_1 \leq j_2$.

If $(i_1, i_r) \notin A_Q$ for all $i_r \leq i_2$ then $\gamma(j_1, j_2, i_1, i_2)$ is equal to

$$\begin{cases} \gamma(j_1 + 1, j_2, i_1 + 1, i_2) & \text{if } (j_1, j_r) \notin A_P \text{ for all } j_r \leq j_2 \text{ and } P[j_1] = Q[i_1], \quad (3) \\ \gamma(j_1, j_2, i_1 + 1, i_2) & \text{if } (j_1, j_r) \in A_P \text{ for some } j_r \leq j_2 \text{ or } P[j_1] \neq Q[i_1], \quad (4) \end{cases}$$

If $(i_1, i_r) \in A_Q$ for some $i_r < i_2$, then $\gamma(j_1, j_2, i_1, i_2)$ is equal to

$$\gamma(\gamma(j_1, j_2, i_1, i_r) + 1, j_2, i_r + 1, i_2) \quad (5)$$

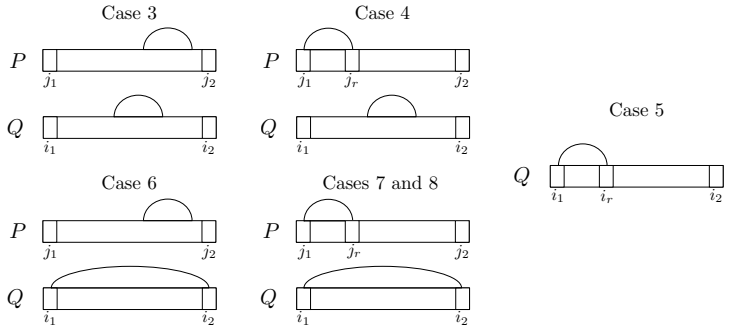

Fig. 2. The main cases from the recurrence relation. Case (3): Neither P or Q starts with an arc. Case (4): P starts with an arc, Q does not. Case (5): Q starts with an arc not spanning Q. We split Q after the arc and compute γ first in the first half and then continue the computation in the other. Case (6): Q starts with an arc, P does not. Case (7)-(8): Both P and Q starts with an arc.

If $(i_1, i_2) \in A_Q$ then $\gamma(j_1, j_2, i_1, i_2)$ is equal to

$$\begin{cases} \max\{\gamma(j_1, j_2, i_1 + 1, i_2), & \\ \qquad \gamma(j_1, j_2, i_1, i_2 - 1)\} & \text{if } (j_1, j_r) \notin A_P \text{ for all } j_r \leq j_2, \quad (6) \\ \gamma(j_1, j_2, i_1 + 1, i_2) & \text{if } (j_1, j_r) \in A_P \text{ for some } j_r \leq j_2, \quad (7) \\ & \qquad \text{and } P[j_1] \neq Q[i_1] \text{ or } P[j_r] \neq Q[i_2], \\ \max\{\phi, \gamma(j_1, j_2, i_1 + 1, i_2)\} & \text{if } (j_1, j_r) \in A_P \text{ for some } j_r \leq j_2, \quad (8) \\ & \qquad P[j_1] = Q[i_1] \text{ and } P[j_r] = Q[i_2], \end{cases}$$

where

$$\phi = \begin{cases} j_r & \text{if } \gamma(j_1 + 1, j_r - 1, i_1 + 1, i_2 - 1) = j_r - 1 \\ j_1 - 1 & \text{otherwise.} \end{cases}$$

The cases are visualized in Fig. 2.

The base cases $(1) - (2)$ cover the cases where $P[j_1, j_2]$ is the empty string ($j_2 > j_1$) or $Q[i_1, i_2]$ is a single base ($i_1 = i_2$). Let $k = \gamma(j_1, j_2, i_1, i_2)$. Case (3) and (5) follows directly from Corollary 1. In case (4) and (7) the base $Q[i_1]$ cannot be part of an embedding of $P[j_1, k]$ in $Q[i_1, i_2]$ and thus $\gamma(j_1, j_2, i_1, i_2) = \gamma(j_1, j_2, i_1 + 1, i_2)$. In case (6) either $Q[i_1]$ or $Q[i_2]$, but not both, can be part of an embedding of $P[j_1, k]$ in $Q[i_1, i_2]$. Thus, $\gamma(j_1, j_2, i_1, i_2) = \max\{\gamma(j_1, j_2, i_1, i_2 - 1), \gamma(j_1, j_2, i_1 + 1, i_2)\}$. Case (8) is the most complicated one. Both $Q[i_1, i_2]$ and $P[j_1, j_2]$ starts with an arc and the bases of the arcs match. An embedding of $P[j_1, k]$ into $Q[i_1, i_2]$ either (i) matches the two arcs, (ii) matches the arc (j_1, j_r) and the rest of $P[j_1, k]$ in $Q[i_1+1, i_2]$ or (iii) matches nothing ($k = j_1-1$). In case (ii) $\gamma(j_1, j_2, i_1, i_2) = \gamma(j_1, j_2, i_1 + 1, i_2)$. Case (i) requires that $P[j_1 + 1, j_r - 1] \sqsubseteq Q[i_1 + 1, i_2 - 1]$. We express this in the recurrence by using an auxiliary function ϕ which is j_r if $\gamma(j_1 + 1, j_r - 1, i_1 + 1, i_2 - 1) = j_r - 1$ and $j_1 - 1$ otherwise, since

in the last case the arc (j_1, j_r) cannot be matched to the arc (i_1, i_2). Since we want the largest match we take the maximum of the two cases (i) and (ii) (case (iii) is covered by these two).

In the next sections we show how to transform the recurrence into a space efficient algorithm for NAPS.

4 The Algorithm

We now present an algorithm to solve NAPS in $O(nm)$ time and $O(m \log |A_Q| + n)$ space. In the next section we show how to further reduce the space to $O(n+m)$ to get Theorem 1. The result relies on a well-known path decomposition for trees applied to arc-annotated strings combined with a new idea to organize the dynamic programming recurrence computation.

Heavy-Path Decomposition of Arc-Annotated Sequences Let S be a nested arc-annotated string containing the arc $(1, |S|)$ (recall that we assume that both P and Q have this arc). The arcs in A_S induce a rooted and ordered tree T_S rooted at the arc $(1, |S|)$ as shown in Fig. 1(b). We use standard tree terminology for the relationship between arcs in T_S. Let (i_l, i_r) be an arc in A_S. The *depth* of (i_l, i_r) is the number of edges on the path from (i_l, i_r) to the root in T_S. An arc with no children is a leaf arc and otherwise an internal arc. Define $T_S(i_l, i_r)$ to be the subtree of T_S rooted at (i_l, i_r) and let size(i_l, i_r) be the number of arcs in $T_S(i_l, i_r)$. Note that size$(1, |S|) = |A_S|$. If (i'_l, i'_r) is an arc in $T_S(i_l, i_r)$ then (i_l, i_r) is an ancestor of (i'_l, i'_r) and if also $(i'_l, i'_r) \neq (i_l, i_r)$ then (i_l, i_r) is a proper ancestor of (i'_l, i'_r). If (i_l, i_r) is a (proper) ancestor of (i'_l, i'_r) then (i'_l, i'_r) is a (proper) descendant of (i_l, i_r).

As in [11] we partition T_S into disjoint paths. We classify each arc as either *heavy* or *light*. The root is light. For each internal arc (i_l, i_r) we pick a child (i^h_l, i^h_r) of maximum size and classify it as heavy. The remaining children are light. An edge to a light child is a *light edge* and an edge to a heavy child is a *heavy edge*. Let lightdepth(i_l, i_r) denote the number of light edges on the path from (i_l, i_r) to the root of T_S. We use the following well-known bound for trees restated for nested arc-annotated sequences.

Lemma 2 (Harel and Tarjan [11]). *Let S be a nested arc-annotated string containing the arc $(1, |S|)$. For any arc $(i_l, i_r) \in A_S$,* lightdepth$(i_l, i_r) \leq \log |A_S| + O(1)$.

Removing the light edges we partition T_S into *heavy paths*.

Manipulating Γ Sequences For positions i_1 and i_2 in Q, $i_1 \leq i_2$, define the Γ *sequence* for i_1 and i_2 as

$$\Gamma(i_1, i_2) = \gamma(m, m, i_1, i_2), \gamma(m-1, m, i_1, i_2), \ldots, \gamma(1, m, i_1, i_2).$$

Thus, $\Gamma(i_1, i_2)$ is the sequence of endpoints of the longest prefixes of each suffix of P that is an arc-preserving subsequence of $Q[i_1, i_2]$. We can efficiently manipulate Γ sequences as suggested by the following lemma.

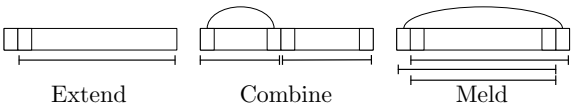

Fig. 3. The extend, combine, and meld operations, respectively. For each operation the substring range(s) below the string indicate the endpoints of the input Γ sequence(s) needed in the operation to compute the Γ sequence for the entire string.

Lemma 3. *For any positions i_1 and i_2 in Q, $i_1 \leq i_2$, we can compute in $O(m)$ time*

(i) $\Gamma(i_2, i_2)$.
(ii) $\Gamma(i_1, i_2)$ *from* $\Gamma(i_1 + 1, i_2)$ *if* $(i_1, i_r) \notin A_Q$ *for any* $i_r \leq i_2$.
(iii) $\Gamma(i_1, i_2)$ *from* $\Gamma(i_1, i_r)$ *and* $\Gamma(i_r + 1, i_2)$ *if* $(i_1, i_r) \in A_Q$ *for some* $i_r < i_2$.
(iv) $\Gamma(i_1, i_2)$ *from* $\Gamma(i_1, i_2 - 1)$, $\Gamma(i_1 + 1, i_2)$, *and* $\Gamma(i_1 + 1, i_2 - 1)$ *if* $(i_1, i_2) \in A_Q$.

Proof. All the cases follow directly from the dynamic programming recurrence. Case (i) follows from case (2) of the recurrence, Case (ii) from case (3) and (4) of the recurrence, Case (iii) from case (5) of the recurrence and Case (iv) from case (6)–(8) of the recurrence. □

We will use each of 4 cases in Lemma 3 as primitive operations in our algorithm and we refer to (i), (ii), (iii), and (iv) as an *initialize*, an *extend*, a *combine*, and a *meld* operation, respectively. Fig. 3 illustrates the extend, combine, and meld operations. An extend operation from $\Gamma(i_1 + k, i_2)$ to $\Gamma(i_1, i_2)$, for some $k > 1$, is defined to be the sequence of k extend operations needed to compute $\Gamma(i_1, i_2)$ from $\Gamma(i_1 + k, i_2)$.

The Algorithm We now present our main algorithm. Initially, we construct T_Q with a heavy path decomposition in $O(n)$ time and space. Then, we recursively compute Γ sequences for each arc $(i_l, i_r) \in A_Q$ in a top-down traversal of T_Q. The Γ sequence for the root contains the value $\gamma(1, m, 1, n)$ and hence this suffices to solve NAPS. At an arc $(i_l, i_r) \in A_Q$ in the traversal there are two cases to consider:

Case 1: (i_l, i_r) *is a leaf arc.* We compute $\Gamma(i_l, i_r)$ as follows.

1. Initialize $\Gamma(i_r, i_r)$ and $\Gamma(i_r - 1, i_r - 1)$.
2. Extend $\Gamma(i_r, i_r)$ and $\Gamma(i_r - 1, i_r - 1)$ to get $\Gamma(i_l + 1, i_r)$, $\Gamma(i_l, i_r - 1)$, and $\Gamma(i_l + 1, i_r - 1)$.
3. Meld $\Gamma(i_l + 1, i_r)$, $\Gamma(i_l, i_r - 1)$, and $\Gamma(i_l + 1, i_r - 1)$ to get $\Gamma(i_l, i_r)$.

Case 2: (i_l, i_r) *is an internal arc.* Let $(i_l^1, i_r^1), \ldots, (i_l^s, i_r^s)$ be the childen arcs of (i_l, i_r) in left-to-right order. To simplify the algorithm we set $i_r^0 = i_l$. We compute $\Gamma(i_l, i_r)$ as follows.

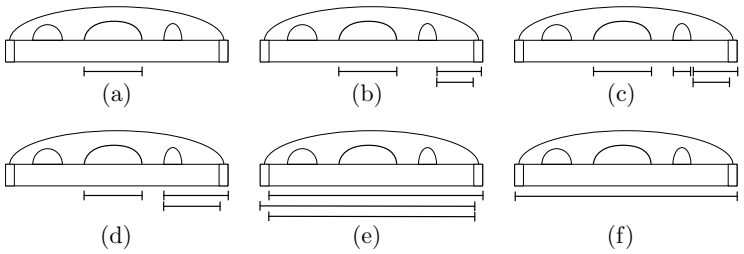

Fig. 4. Snapshot of the Γ sequences computed at an internal arc. The ranges below the arc-annotated sequences represent Γ sequence endpoints. (a) After the recursive call to the heavy child in line 1. (b) After the extend operations in line 3. (c) After the recursive call in line 4(a) (d) After the combine operations in line 4(b). (e) Before the meld operation in line 6. (f) After the meld operation.

1. Recursively compute $R_h := \Gamma(i_l^h, i_r^h)$, where (i_l^h, i_r^h) is the heavy child arc of (i_l, i_r).
2. Initialize $\Gamma(i_r, i_r)$ and $\Gamma(i_r - 1, i_r - 1)$.
3. Extend $\Gamma(i_r, i_r)$ and $\Gamma(i_r - 1, i_r - 1)$ to get $\Gamma(i_r^s + 1, i_r)$ and $\Gamma(i_r^s + 1, i_r - 1)$.
4. For $k := s$ down to 1 do:
   (a) If $k \neq h$ recursively compute $R_k := \Gamma(i_l^k, i_r^k)$.
   (b) Combine R_k with $\Gamma(i_r^k + 1, i_r)$ and with $\Gamma(i_r^k + 1, i_r - 1)$ to get $\Gamma(i_l^k, i_r)$ and $\Gamma(i_l^k, i_r - 1)$.
   (c) Extend $\Gamma(i_l^k, i_r)$ and $\Gamma(i_l^k, i_r - 1)$ to get $\Gamma(i_r^{k-1} + 1, i_r)$ and $\Gamma(i_r^{k-1} + 1, i_r - 1)$.
5. Extend $\Gamma(i_l + 1, i_r - 1)$ to get $\Gamma(i_l, i_r - 1)$.
6. Meld $\Gamma(i_l + 1, i_r)$, $\Gamma(i_l, i_r - 1)$, and $\Gamma(i_l + 1, i_r - 1)$ to get $\Gamma(i_l, i_r)$.

The computation in case 2 is illustrated in Fig. 4. Note that when $k = 1$ in the loop in line 4, line 4(c) computes $\Gamma(i_r^0 + 1, i_r) = \Gamma(i_l + 1, i_r)$ and $\Gamma(i_r^0 + 1, i_r - 1) = \Gamma(i_l + 1, i_r - 1)$. In both cases above the algorithm computes several *local* Γ *sequences* of the form $\Gamma(i, i_r)$ and $\Gamma(i, i_r - 1)$, for some $i \leq i_r$. These sequences are computed in order of decreasing values of i and each sequence only depends on the previous one and recursively computed Γ sequences. Hence, we only need to store a constant number of local sequences during the computation at (i_l, i_r).

Analysis We first consider the time complexity of the algorithm. To do so we bound the total number of primitive operations. For each arc in A_Q there is 1 initialize and 1 meld operation and for each internal arc there is 1 combine operation. Hence, the total number of initialize, meld, and combine operations is $O(|A_Q|)$. To count the number of extend operations we first define for any arc $(i_l, i_r) \in A_Q$ the set spaces(i_l, i_r) as the set of positions inside (i_l, i_r) but not inside any child arc of (i_l, i_r), that is,

$$\text{spaces}(i_l, i_r) = \{i \mid i_l \leq i \leq i_r \text{ but not } i_l^k \leq i \leq i_r^k \text{ for any child } (i_l^k, i_r^k) \text{ of } (i_l, i_r)\}.$$

For example, spaces$(1, 11)$ for Q in Fig. 1(a) is $\{1, 2, 11\}$. The spaces sets for all arcs is a partition of the positions in Q and thus $\sum_{(i_l,i_r)\in A_Q} \text{spaces}(i_l, i_r) = n$. At an arc (i_l, i_r) the algorithm performs $O(\text{spaces}(i_l, i_r))$ extend operations and hence the total number of extend operations is $O(n)$. By Lemma 3 each primitive operation takes $O(m)$ time and therefore the total running time of the algorithm is $O(|A_Q|m + nm) = O(nm)$.

For the space complexity we bound the number of Γ sequences stored by the algorithm. When the algorithm visits an arc (i_l, i_r) we are currently processing a nested sequence of recursive calls corresponding to a path p in T_Q from the root to (i_l, i_r). The number of Γ sequences stored at each of these recursive calls is the total number of Γ sequences stored. Consider an edge e in p from a parent (i'_l, i'_r) to a child (i''_l, i''_r). If e is heavy the recursive call to (i''_l, i''_r) is done in line 1 of case 2 in the algorithm immediately at the start of the visit to (i'_l, i'_r). Therefore, no Γ sequence at (i'_l, i'_r) is stored. If e is light the recursive call to (i''_l, i''_r) is done in line 4(a). The algorithm stores at most 3 Γ sequences, namely $\Gamma(i''_r+1, i'_r)$, $\Gamma(i''_r+1, i'_r-1)$, and $\Gamma(i_l^{h'}, i_r^{h'})$, where $(i_l^{h'}, i_r^{h'})$ is the heavy child of (i'_l, i'_r). By Lemma 2 there are at most $\log |A_Q| + O(1)$ light ancestors of (i_l, i_r) in T_Q and therefore the total space for stored Γ sequences is $O(m \log |A_Q|)$. The additional space used by the algorithm is $O(n)$. We have,

Lemma 4. *Given nested arc-annotated strings P and Q of lengths m and n, respectively, we can solve the nested arc-preserving subsequence problem in time $O(nm)$ and space $O(m \log |A_Q| + n)$.*

5 Squeezing into Linear Space

We now show how to compress Γ sequence into a compact representation using $O(m)$ *bits*. Plugging the new representation into our algorithm the total space becomes $O(n + m)$ as desired for Theorem 1.

Our compression scheme for Γ sequences relies on the following key property of the values of γ.

Lemma 5. *For any integers j_1, j_2, i_1, i_2, $1 \le j_1 \le j_2 \le m$, $1 \le i_1 \le i_2 \le n$,*

$$j_1 - 1 \le \gamma(j_1, j_2, i_1, i_2) \le \gamma(j_1 + 1, j_2, i_1, i_2) \le m$$

Proof. Adding another base in front of the substring $P[j_1+1, j_2]$ cannot increase the endpoint of an embedding of $P[j_1+1, j_2]$ in Q and therefore $\gamma(j_1, j_2, i_1, i_2) \le \gamma(j_1+1, j_2, i_1, i_2)$. Furthermore, for any substring $P[j_1, j_2]$ we can embed at most $j_1 - j_2$ bases and at least 0 bases in Q implying the remaining inequalities. □

Let i_1, i_2 be indices in Q such that $i_1 \le i_2$ and consider the sequence

$$\Gamma(i_1, i_2) = \gamma(m, m, i_1, i_2), \ldots, \gamma(1, m, i_1, i_2) = \gamma_m, \ldots, \gamma_1$$

By Lemma 5 we have that $\gamma_m, \ldots, \gamma_1$ is a non-increasing and non-negative sequence where γ_m is either m or $m - 1$. We encode the sequence efficiently using

two bit strings V and U defined as follows. The string V is formed by the concatenation of m bit strings $s_m, \ldots, s_1$, that is, $V = s_m \cdot s_{m-1} \cdots s_1$, where $\cdot$ denotes concatenation. The string s_m is the single bit $s_m = m - \gamma_m$ and s_k, $1 \leq k < m$, is given by

$$s_k = \begin{cases} 0 & \text{if } \gamma_{k+1} - \gamma_k = 0 \\ \underbrace{1 \cdots 1}_{\gamma_{k+1} - \gamma_k \text{ times}} & \text{if } \gamma_{k+1} - \gamma_k > 0 \end{cases}$$

Let D_k denote the sum of bits in string $s_m \cdots s_k$. We have that $m - D_m = m - s_m = \gamma_m$ and inductively $m - D_k = \gamma_k$. The string U is the bit string of length $|V|$ consisting of a 1 in each position where a substring in V ends. Given V and U we can therefore uniquely recover $\gamma_m, \ldots, \gamma_1$. Since $\gamma_m, \ldots, \gamma_1$ can decrease by at most $m + 1$ the total number of 1s in V is at most $m + 1$. The total number of 0s is at most m and therefore $|V| \leq 2m + 1$. Hence, our representation uses $O(m)$ bits. We can compress $\gamma_m, \ldots, \gamma_1$ into V and U in a single scan in $O(m)$ time. Reversing the process we can also decompress in $O(m)$ time. Hence, we have the following result.

Lemma 6. *We represent any Γ sequence using $O(m)$ bits. Compression and decompression takes $O(m)$ time.*

We modify our algorithm from Section 4 to take advantage of Lemma 6. Let (i_l, i_r) be an internal arc in A_Q. Immediately before a recursive call to a light child (i_l^k, i_r^k) of (i_l, i_r) we compress the at most 3 Γ sequences maintained at (i_l, i_r), namely $\Gamma(i_l^h, i_r^h)$, where (i_l^h, i_r^h) is the heavy child, $\Gamma(i_r^k+1, i_r)$, and $\Gamma(i_r^k+1, i_r - 1))$. Immediately after returning from the recursive call we decompress the sequences again.

The total number of compressions and decompressions is $O(n)$. Hence, by Lemma 6 the additional time used is $O(nm)$ and therefore the total running time of the algorithm remains $O(nm)$. The space for storing the $O(\log |A_Q|)$ Γ sequences becomes $O(m \log |A_Q|) = O(m \log n)$ *bits*. Hence, the total space is $O(n + m)$. In conclusion, we have shown Theorem 1.

Avoiding Decompression The above algorithm requires $O(n)$ decompressions. We briefly describe how one can these by augmenting the representation of Γ sequences slightly. A *rank/select index* for a bit string B supports the operations $\textsc{rank}(B, k)$ that returns the number of 1 in $B[1, k]$ and $\textsc{select}(B, k)$ that returns the position of the kth 1 in S. We can construct a rank/select index in $O(|B|)$ time that uses $o(|B|)$ bits and supports both operations in constant time [15]. We add a rank/select index to the bit strings V and U in our compressed representation. Since these use $o(m)$ bits this does not affect the space complexity. Let $\gamma_m, \ldots, \gamma_1$ be a Γ sequence compressed into bit strings V and U augmented with a rank/select index. For any k, $1 \leq k \leq m$ we can compute the element γ_k in constant time as

$$m - \textsc{rank}(V, \textsc{select}(U, m + 1 - k))$$

To see the correctness, first note that $\text{SELECT}(U, m+1-k)$ is end position of the $m+1-k$th substring in V. Therefore, $\text{RANK}(V, \text{SELECT}(U, m+1-k))$ is the sum of the bits in the first $m+1-k$ substrings of V. This is D_k and since $\gamma_k = m - D_k$ the computation returns γ_k. In summary, we have the following result.

Lemma 7. *We can represent any Γ sequence in $O(m)$ bits while allowing constant time access to any element.*

The algorithm now only needs to compress Γ sequences once. Whenever, we need an element of a compressed Γ sequence we extract it in constant time as above. Hence, the asymptotic complexities of the algorithm remains the same.

References

1. J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Computing the similarity of two sequences with nested arc annotations. *Theor. Comput. Sci.*, 312(2-3):337–358, 2004.
2. R. Backofen, G. M. Landau, M. Möhl, D. Tsur, and O. Weimann. Fast RNA structure alignment for crossing input structures. In *Proc. 20th CPM*, 2009.
3. V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. In *Proc. 6th CPM, LNCS*, volume 937, pages 1–16, 1995.
4. P. Bille and I. L. Gørtz. The tree inclusion problem: In optimal space and faster. In *Proc. 32nd ICALP, LNCS*, volume 3580, pages 66–77, 2005.
5. G. Blin, G. Fertin, R. Rizzi, and S. Vialette. What makes the Arc-Preserving Subsequence Problem hard? In *Proc. 5th ICCS*, pages 860–868, 2005.
6. G. Blin and H. Touzet. How to compare arc-annotated sequences: The alignment hierarchy. In *Proc. 13th SPIRE, LNCS*, pages 291–303, 2006.
7. W. Chen. More efficient algorithm for ordered tree inclusion. *J. Algorithms*, 26:370–385, 1998.
8. P. Damaschke. A remark on the subsequence problem for arc-annotated sequences with pairwise nested arcs. *Inf. Process. Lett.*, 100(2):64–68, 2006.
9. P. Evans. *Algorithms and Complexity for Annotated Sequence Analysis*. PhD thesis, University of Victoria, 1999.
10. J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. *ACM Trans. Algorithms*, 2(1):44–65, 2006. Announced at FSTTCS 2002.
11. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
12. T. Kida. Faster pattern matching algorithm for arc-annotated sequences. In *Federation over the Web, LNCS*, volume 3847, pages 25–39, 2006.
13. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24:340–356, 1995.
14. G. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *J. Comput. Syst. Sci.*, 65(3):465–480, 2002.
15. I. Munro. Tables. In *Proc. 16th FSTTCS, LNCS*, volume 1180, pages 37–42, 1996.
16. S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theor. Comput. Sci.*, 312(2-3):223–249, 2004. Announced at CPM 2002.