Efficient chaining of seeds in ordered trees

Julien Allali[1,2,3,5], Cedric Chauve[3], Pascal Ferraro[1,2,4], and Anne-Laure Gaillard[1]

[1] LaBRI, Université Bordeaux 1, CNRS UMR 5800, Talence, France
[2] Pacific Institute for Mathematical Sciences, CNRS UMI 3069, Vancouver, BC, Canada
[3] Department of Mathematics, Simon Fraser University, Burnaby (BC), Canada.
[4] Department of Computer Science, University of Calgary, Calgary, AB, Canada
[5] Institut Polytechnique de Bordeaux, Bordeaux, France
julien.allali@labri.fr,cedric.chauve@sfu.ca,pascal.ferraro@labri.fr,anne-laure.gaillard@labri.fr

Abstract. We consider the problem of chaining seeds in ordered trees. Seeds are mappings between two trees Q and T and a chain is a subset of non overlapping seeds that is consistent with respect to postfix order and ancestrality. This problem is a natural extension of a similar problem for sequences, and has applications in computational biology, such as mining a database of RNA secondary structures. For the chaining problem with a set S of m seeds of cumulated size $\|S\|$, we describe an algorithm with complexity $O(\|S\| \log(\|S\|) + m\|S\| \log(m))$ in time and $O(m\|S\|)$ in space.
Note. To appear in *Journal of Discrete Algorithms*, special issue for *IWOCA 2010*. Preliminary version in *IWOCA 2010*, *LNCS* 6460, pp. 260273, 2011. Version of December 3, 2011.

1 Introduction

Comparing sequences is a basic task in computational biology. A fundamental application of sequence comparison is to search efficiently in a database a set of sequences that are similar to a query sequence. The exponential increase in the amount of available sequence data still motivates the need for very efficient sequence comparison algorithms. In particular, pairwise comparison based on the computation of an exact edit distance between a query and every sequence of the database is not practical due to the quadratic time complexity of edit distance computation. A typical approach to tackle this issue is to rely on short sequences, called *seeds*, present in the query. Seeds can be detected very quickly in the database using indexing techniques [4]; then an optimal set of seeds, called a *chain*, that are collinear in both the query and a sequence of the database can be computed. Widely used programs such as BLAST [1] and FASTA [11, 14] rely on such an approach. We refer the reader to [2, 7] for surveys of sequence comparison in computational biology. From an algorithmic point of view, given m seeds an optimal chain between two sequences can be computed in $O(m \log(m))$ time and $O(m)$ space [10] (see [13] for a recent survey).

With the recent development of high-throughput genome annotation methods, similar problems appear to be relevant for the analysis of more complex combinatorial objects, used to model biological structures of higher complexity [15, 19]. Our present work, although relatively general, is motivated by such questions, applied to RNA secondary structures. An RNA secondary structure can indeed be represented by a tree or a graph whose nodes are the nucleotides and whose edges are the chemical bonds between them [16]. Mining large RNA secondary structure databases, such as Rfam [6], is an important computational biology problem. An initial approach, adapting the notion of edit distance to ordered trees, was pioneered by Zhang and Shasha [20]. The tree edit approach has been extended in several ways since then, leading either to hard problems, when a comprehensive set of edit operations is considered [9], or to algorithms with a worst-case time complexity at best cubic, even with a minimal set of edit operations [5, 20]. Recently, approaches have been proposed to filter large RNA Heyne *et al.* [8] introduced a chaining problem on an alternative representation of ordered trees called arc annotated sequences, motivated by pairwise RNA secondary structure comparison: once an optimal chain of seeds between two given RNA secondary structures is detected, the regions between successive seeds are processed independently using an edit distance algorithm, which speeds up significantly the comparison process. They considered seeds defined as *exact common patterns* and designed a dynamic programming algorithm to solve the seed chaining problem. To the best of our knowledge, [8] is

the first paper addressing the problem of computing a chain in trees (see also [12]). Our contribution in the present work is a new algorithm for finding the score of an optimal chain between two ordered trees (the Maximal Chaining Problem).

After some preliminaries (Sections 2, 3, 4), that describe combinatorial properties of chains in trees and pre-processing algorithms, we describe in Section 5 our algorithms for solving the Maximal Chaining Problem, followed in Section 6 by a comparison with the algorithm of Heyne *et al.* [8].

2 Preliminaries: background and problem statement

Ordered trees Let T be an ordered rooted tree of size n. Nodes of T are identified with their postfix-order index from 0 to $n-1$. Thus, $n-1$ represents the root of T. T_i is the subtree of T rooted at vertex i. We denote by $T[i,j]$ the forest induced by the nodes that belong to the interval $[i,j]$; if $i > j$, then $T[i,j]$ is empty. The partial order relation "i is an ancestor of j" is denoted by $i \prec j$. A leaf of T is a vertex with no child. For a tree T and a node i of T, the first (resp. last) leaf visited during a postfix traversal of T_i is denoted by $l(i)$ (resp. $r(i)$) and called the *leftmost leaf* (resp. *rightmost leaf*) of the node i. The ordered forest induced by the proper descendants of i is denoted by $\widehat{T_i} = T[l(i), i-1]$.

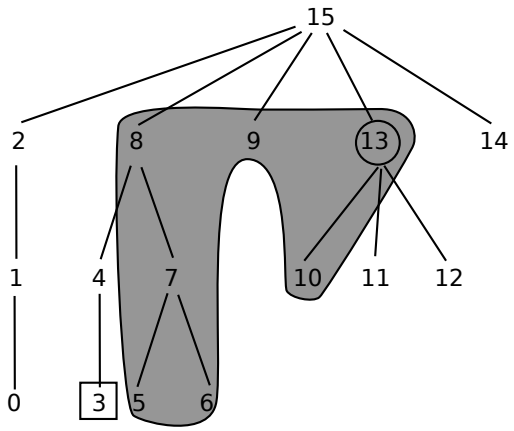

Fig. 1. Example of an internal forest $G = \{5,6,7,8,9,10,13\}$ containing three internal trees $G_1 = \{5,6,7,8\}$, $G_2 = \{9\}$ and $G_3 = \{10,13\}$. $r_G = 13$. $r_{G_1} = 8$, $r_{G_2} = 9$, $r_{G_3} = 13$. $L(G) = \{5,6,9,10\}$. Node 7 is completely inside G. $B(G) = \{5,6,8,9,10,13\}$. $l(G) = 3$.

Seeds, chains and the Maximum Chaining Problem We now introduce the fundamental notions we consider in the present work and define formally the problem we address.

Definition 1. *(see Fig 1 for an illustration) Let T be an ordered rooted tree:*

1. *Let $G = (g_0, \ldots, g_{k-1})$ be a sequence of k distinct nodes of T, with $0 \leq g_j < n$.*
   - *If the subgraph of T induced by G is connected, then G is called an* internal tree *rooted at g_{k-1} also referred to as r_G.*
   - *If there is a partition of G into p internal trees $G_1, \ldots G_p$, respectively rooted at $r_{G_1}, \ldots, r_{G_p}$, and such that for all $1 < i \leq p$, r_{G_i} is the right sibling of $r_{G_{i-1}}$ in T, then G is called an* internal forest*. (Thus any internal tree is also an internal forest).*
   - *r_{G_p} (the root of the rightmost tree of G) is called* the root of G*, denoted by r_G.*
2. *The leftmost leaf of an internal forest G is defined by $l(G) = min(l(x)/x \in G)$. Remark the leftmost leaf of an internal forest doesn't need to be a leaf of this internal forest (as in Fig 1).*

3. *The set of leaves of an internal forest G is denoted by $L(G)$.*
4. *A node g_j of an internal forest G is* completely inside *G if g_j is not a leaf of T and all its children belong to G. The set of nodes of G that are not completely inside G is called the* border *of G and is denoted by $B(G)$.*
5. *Two internal forest G^1 and G^2* overlap *if $G^1 \cap G^2 \neq \emptyset$.*

We now recall the central notion of *valid mapping* between two trees introduced in [17, 16] for the tree edit distance.

Definition 2. *Given two trees Q and T, a valid mapping P between Q and T is a set of pairs of $Q \times T$ such that, if (q_i, t_i) and (q_j, t_j) belong to P, then*

1. *$q_i = q_j$ if and only if $t_i = t_j$,*
2. *$q_i < q_j$ if and only if $t_i < t_j$,*
3. *$q_i \prec q_j$ if and only if $t_i \prec t_j$.*

From now we use the term *mapping* to refer to a valid mapping. Given a mapping P between Q and T, the smallest internal forest of Q (resp. T) that contains all nodes of Q (resp. T) belonging to a pair of P is denoted by Q_P (resp. T_P). Q_P and T_P are respectively called the internal forests of Q and T induced by P.

The following definition introduces the central notion of *seed* between two ordered trees. Roughly speaking, seeds are internal forests, one in each tree, with similar structure in terms of number of trees and borders.

Definition 3. *Let Q and T be two ordered trees.*

1. *A* seed *P between Q and T is a mapping between Q and T such that:*
   - *Both internal forests Q_P and T_P contain the same number t of internal trees, of respective roots (in increasing postfix order) $r_{Q_{P_1}}, \ldots, r_{Q_{P_t}}$ in Q and $r_{T_{P_1}}, \ldots, r_{T_{P_t}}$ in T.*
   - *For all $1 \leq i \leq t$, $(r_{Q_{P_i}}, r_{T_{P_i}}) \in P$.*
   - *Any node of the border of Q_P (resp. T_P) belongs to a pair of P.*
2. *The border $B(P)$ (resp. leaves $L(P)$) of the seed P is the set of pairs $(x, y) \in P$ such that $x \in B(Q_P)$ and $y \in B(T_P)$ (resp. $x \in L(Q_P)$ and $y \in L(T_P)$).*
3. *The* size *$|P|$ of the seed P is the number of pairs it contains.*
4. *For a set S of seeds, $\|S\|$ is the sum of the sizes of the $|S|$ seeds in S.*

Remark 1. Let (x, y) belong to a seed P. x is a leaf of Q_P if and only if y is a leaf of T_P. However, it is not true that x is in the border of Q_P if and only if y is in the border of P_P. Moreover, if $x \in B(Q_P)$ but $y \notin B(T_P)$, then $(x, y) \notin B(P)$.

Remark 2. Theoretically, the number of seeds between Q and T can be exponential in the size of Q and T, although in applications such as RNA secondary structure comparison, this exponential upper bound is unlikely to be reached (see [8] for example).

Definition 4. *Let Q and T be two ordered trees.*

1. *A pair (P^1, P^2) of seeds between Q and T is* chainable *if Q_{P^1} does not overlap Q_{P^2}, T_{P^1} does not overlap T_{P^2}, and $P^1 \cup P^2$ is a mapping.*
2. *A* chain *is a set $C = \{P^0, P^1, \ldots, P^{\ell-1}\}$ of seeds between Q and T such that any pair (P^i, P^j) of distinct seeds in C is chainable.*
3. *Given a scoring function v for the seeds P^i, the score of a chain C is the sum of the scores of its seeds: $v(C) = \sum_i v(P^i)$.*
4. *Given a set S of possibly overlapping seeds between Q and T, $\mathcal{C}_S(Q, T)$ denotes the set of all possible chains between Q and T in S.*

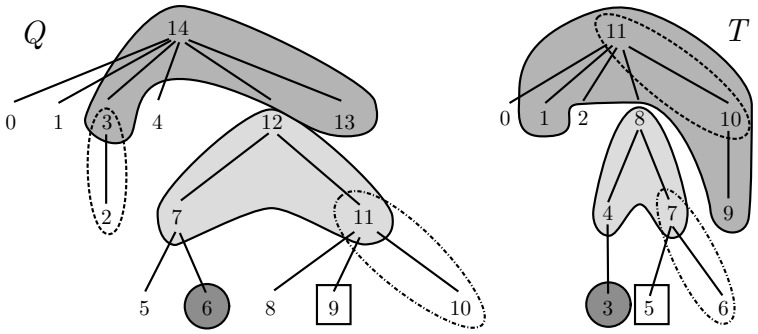

Fig. 2. An instance of the MCP with 6 seeds: $P^0 = \{(2,10),(3,11)\}$, $P^1 = \{(6,3)\}$, $P^2 = \{(9,5)\}$, $P^3 = \{(10,6),(11,7)\}$, $P^4 = \{(7,4),(11,7),(12,8)\}$, $P^5 = \{(3,1),(13,9),(14,11)\}$. If for every seed $v(P^i) = |P^i|$, an optimal chain is composed of $\{P^1, P^2, P^4, P^5\}$ and has a score of 8.

We can now define the Maximum Chaining Problem, illustrated in Fig. 2.

Problem. Maximum Chaining Problem (MCP):
Input: A pair (Q,T) of ordered rooted trees, a set $S = \{P^0, \ldots, P^{m-1}\}$ of m possibly overlapping seeds between Q and T, a scoring function v on the seeds P^i.
Output: The maximum score of a chain C included in S:

$$MCP(Q,T,S) = \max\{v(C); C \in \mathcal{C}_S(Q,T)\}.$$

Remark 3. Without loss of generality, from now we assume that the seeds P^i are sorted increasingly according to the postfix number of their roots in Q, that is: $r_{Q_{P^0}} \leq \cdots \leq r_{Q_{P^i}} \leq \cdots \leq r_{Q_{P^{m-1}}}$. Furthermore, for every (x,y) belonging to a seed P^j, we denote by x^j the unique node y of T associated with x in P^j. We also denote $r_{Q_{P^j}}$ by r_j, and then $r_{T_{P^j}}$ by r_j^j. For a given chain C, the *last* seed of C is then the seed with the highest postfix index in Q.

Remark 4. Note that we consider here only the problem of computing the score of a chain for exposition reasons; standard backtracking techniques allow to compute a maximum-score chain from our dynamic programming algorithms.

Motivation and result statement As far as we know, [8] is the only work that attacks the MCP in tree structures, although the authors describe it in terms of arc-annotated sequences. They proposed a dynamic programming algorithm to solve the maximum chaining problem with some restrictions on the seeds (precisely, seeds are maximal exact patterns common to the considered sequences). This dynamic programming technique is different from the approach used for the currently best known algorithms for Maximum Chaining Problem in sequences [10, 13]. Moreover, when applied to arc-annotated sequences with no arc (*i.e.* sequences) and m seeds, it can be shown this algorithm has a worst-case time complexity in $O(m^2)$ (see Section 6). Our main result is the following:

Theorem 1. *Let S be a set of m seeds between two ordered trees Q and T. After an $O(\|S\|)$ time preprocessing of S, one can solve the Maximum Chaining Problem in $O(\|S\| \log(\|S\|) + m\|S\| \log(m))$ worst-case time and $O(m\|S\|)$ worst-case space.*

Remark 5. The complexities stated in Theorem 1 do not depend on the size of the trees Q and T. This is possible due to assumptions on the representations of these trees that are described precisely in Sections 4 and 5.

We prove Theorem 1 in Section 5 and we compare our algorithm to Heyne *et al.* [8] algorithm in Section 6. In particular, we show that, when applied to unary trees, equivalent to sequences (see Remark 6 below), our algorithm solves the maximum chaining problem in $O(m\log(m))$ time and $O(m)$ space, the best known complexity to chain seeds in sequences.

Remark 6. To compare with chaining algorithms for sequences, we represent a sequence $u = (u_0, \ldots, u_{n-1})$ by a unary tree, rooted at a node labeled by u_{n-1}, where every internal node has a single child and u_0 is the unique leaf: the sequence of nodes visited by the postfix-order traversal of this tree is exactly u.

3 Combinatorial properties of seeds and chains

We first describe combinatorial properties of seeds and chains, that naturally lead to a recursive scheme to compute a maximum chain. More precisely, we show that given a chain C and its last seed P, the roots and border of P define a partition of both $Q - Q_P$ and $T - T_P$ into a set of pairs of forests, defined in terms of the key notion of *chainable areas* (Definitions 5 and 6 below), that contain the seeds $C - \{P\}$ and form sub-chains of C.

Definition 5. *Let P be a seed between two trees Q and T and $(a, b; c, d)$ be a quadruple such that $l(Q_P) \leq a < b < r_{Q_P}$, $l(T_P) \leq c < d < r_{T_P}$, $Q_P \cap Q[a, b] = \emptyset$ and $T_P \cap T[c, d] = \emptyset$. $(a, b; c, d)$ is a* chainable area for *P if, for all $i \in [a, b]$ and all $j \in [c, d]$, $P \cup (i, j)$ is a valid mapping; it is a* maximal chainable area for *P if neither $(a - 1, b; c, d)$ or $(a, b + 1; c, d)$ or $(a, b; c - 1, d)$ or $(a, b; c, d + 1)$ are chainable areas for P.*

For example, in Fig. 2, if $P = P^5$, then $(4, 12; 2, 8)$ is a maximal chainable area. See also Figure 3 below.

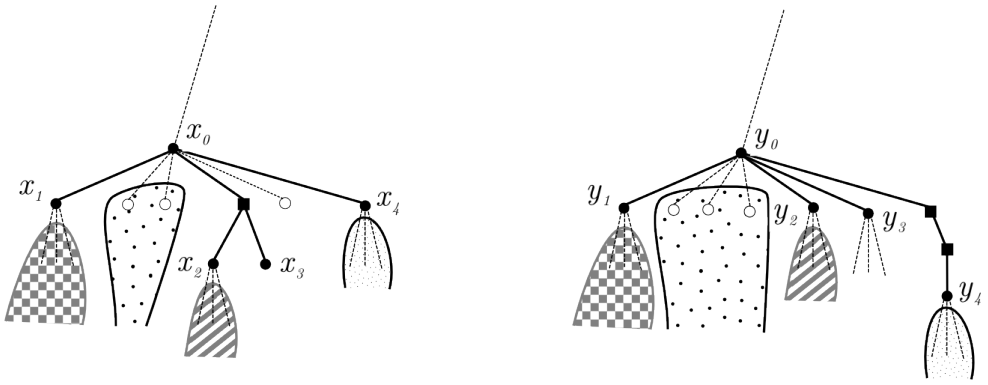

Fig. 3. Illustration of the notion of chainable areas of a seed of size 5: $P = \{(x_0, y_0), \ldots, (x_4, y_4)\}$ and there are 4 maximal chainable areas for P each indicated by a different filling pattern.

We now refine the definition of chainable area to define precisely the areas associated to the border nodes of a seed. Such areas are fundamental as mapped nodes of Q and T that belong to a chain have to belong to such chainable areas.

Definition 6. *1. Let $(x, y) \in B(P)$ for a seed P between Q and T. We define by $F(x, y) = \{(a_i, b_i; c_i, d_i)\}$ the set of all maximal chainable areas for P included in $(Q_x; T_y)$ such that there is no border node of P in Q (resp. T) on the path from b_i to x (resp. d_i to y). We call this set the* chainable areas of (x, y).

2. The chainable areas *of a seed P, denoted by $CA(P)$, is the union of the sets of quadruples $F(x, y)$ for all pairs $(x, y) \in B(P)$.*

For example, let us consider a pair (x, y) in $L(P)$ such that x and y are not leaves of respectively Q and T, then $F(x, y)$ represents the couple of forests $\widehat{Q_x}$ and $\widehat{T_y}$, $F(x, y) = \{(l(x), x-1; l(y), y-1)\}$. In Fig. 2, with $P = P^4$ and $(x, y) = (11, 7)$, $F(x, y) = \{(8, 10; 5, 6)\}$; with $P = P^5$, if $(x, y) = (14, 11) \in B(P^5) - L(P^5)$, $F(x, y) = \{(0, 1; 0, 0), (4, 12; 2, 8)\}$.

The following property is a relatively straightforward consequence of the definitions of seeds and chainable areas.

Property 1. Given a seed P between trees Q and T, $|CA(P)| \leq 2 \times |B(P)| - 1$.

Notation. For a seed P and any chainable area $(a, b; c, d)$, we say that $P \subset (a, b; c, d)$ if $a \leq l(Q_P) \leq r_{Q_P} \leq b$ and $c \leq l(T_P) \leq r_{T_P} \leq d$; for a chain C, $C \subset (a, b; c, d)$ if all its seeds are included in $(a, b; c, d)$. We finally denote by $F_j(x)$ the set of quadruples $F(x, x^j)$ for the pair of nodes $(x, x^j) \in B(P^j)$.

The next property describes the structure of any chain, between two forests $Q[a, b]$ and $T[c, d]$, included in a set of m seeds $S = \{P^0, \ldots, P^{m-1}\}$. It is a direct consequence of the constraints that define a valid mapping and the fact that seeds are non-overlapping in a chain.

Property 2. Let P^j be the last seed of a chain C between two forests $Q[a, b]$ and $T[c, d]$.

1. C can be decomposed into $|CA(P^j)| + 2$ (possibly empty) distinct sub-chains:
   - P^j itself,
   - for each $(e, f; g, h) \in CA(P^j)$ a (possibly empty) chain between $Q[e, f]$ and $T[g, h]$,
   - and a chain between $Q[a, l(Q_{P^j}) - 1]$ and $T[c, l(T_{P^j}) - 1]$.
2. C is a chain of maximum score among all chains in $Q[a, b]$ and $T[c, d]$ that contain P^j if and only if each of its sub-chains described above are chains of maximum score in the corresponding forests defined by P^j.

Point 2 of Property 2 naturally leads to a recursive scheme to compute an optimal chain between two forests $Q[a, b]$ and $T[c, d]$ that ends by the last seed of a set. If we denote by $MCP'(Q[a, b], T[c, d], \{P^0 \ldots P^j\})$ the score of a maximum chain between $Q[a, b]$ and $T[c, d]$ that contains P^j as last seed, then:

$$MCP'(Q[a, b], T[c, d], \{P^0 \ldots P^j\}) = \tag{1}$$

$$\begin{cases} 0 & \text{if } P^j \not\subset (a, b; c, d), \\ v(P^j) + \sum\limits_{(e,f;g,h) \in CA(P^j)} MCP(Q[e, f], T[g, h], \{P^0 \ldots P^{j-1}\}) & \\ \quad + MCP(Q[a, l(Q_{P^j}) - 1], T[c, l(T_{P^j}) - 1], \{P^0 \ldots P^{j-1}\}) & \text{otherwise.} \end{cases}$$

and, assuming that the seeds sorted incrementally (see Remark 3), $MCP(Q, T, S)$ can be computed using MCP' as follows:

$$MCP(Q[a, b], T[c, d], \{P^0 \ldots P^j\}) = \max_{i=0\ldots j} MCP'(Q[a, b], T[c, d], \{P^0 \ldots P^i\}) \tag{2}$$

$$MCP(Q, T, S) = MCP(Q[0, r_Q], T[0, r_T], S) \tag{3}$$

The main challenge in designing an algorithm for the MCP is to implement efficiently this recursive formula, that was already central in the dynamic programming algorithm of [8]. Next section details the preprocessing phase for computing the chainable areas of all seeds. The reader can also directly read the Section 5, where we assume the chainable areas are already computed and we focus on the chaining algorithm.

4 An algorithm for computing chainable areas of seeds

We describe now a fundamental pre-processing step to solve the MCP, that computes the chainable areas for the border nodes of a given set of seeds. We consider here a seed P between two trees Q and T, and $B(P)$ its border, and we describe an efficient algorithm (Algorithm 1) that computes $F(x, y)$ for any given pair (x, y) of nodes belonging to $B(P)$.

Preliminaries We start by assuming some properties of the encoding of seeds and trees that will be key in the design of an efficient algorithm to compute chainable areas. These properties rely on a simple representation using arrays.

For a node i in Q (resp. T) that belongs to the border of P, we assume that we can access the following informations in constant time: $l(i)$, the leftmost leaf of i, and $u(i)$, the node with the highest postfix index such that $r(u(i)) = r(i)$ (where $r(i)$ is the rightmost leaf of i). We refer to nodes $u(i)$ as *rightmost roots*.

If $B(P)$ contains k pairs of nodes belonging to $Q \times T$ (called *pairs* from now), we assume that $B(P)$ is available in an array B of k pairs of integers. For $0 \leq i < k$, $B[i]$ represents the $(i+1)^{\text{th}}$ pair of B, ordered in increasing postfix order, $B[i]_Q$ and $B[i]_T$ are respectively the nodes of $B[i]$ that belongs to Q and T.

Algorithm 1 makes use of a stack of pairs of nodes called *Stack*. The last element inserted in *Stack* is referred to as $top(Stack)$, and $top(Stack)_Q$ and $top(Stack)_T$ represent respectively the node of $top(Stack)$ that belongs to Q and T. We use $push(Stack, (x, y))$ to add (x, y) into *Stack* and $pop(Stack)$ to remove the last element of the stack.

Description of the algorithm Before we give the pseudo-code of the algorithm computing the chainable areas of P, we describe the main points underlying it. First, pairs are traversed incrementally according to their postfix index, ensuring descendants are visited before their parents. Each time a pair is visited (the *current pair*), it is inserted into *Stack* (hence *Stack* contains pairs sorted increasingly by their postfix index). For a given $B[i]$, the chainable areas $F(B[i])$ are stored into a list, sorted in increasing postfix order. Then only two cases need to be considered:

1. the current pair $B[i]$ is a pair of leaves in $Q_P \times T_P$ (*i.e.* $B[i] \in L(P)$)[6], or
2. the current pair $B[i]$ is not a pair of leaves in $Q_P \times T_P$ (*i.e.* $B[i] \notin L(P)$).

Algorithm 1 $F(x, y)$: compute the $F(x, y)$ for a seed P

```
1  sort B incrementally
2  foreach pair B[i] of B do
3      F(B[i]) = ∅
4  for i from 0 to k − 1 do
5      if i = 0 or B[i − 1]_Q < l(B[i]_Q) then # B[i] ∈ L(P)
6          if l(B[i]_Q) ≤ B[i]_Q − 1 and l(B[i]_T) ≤ B[i]_T − 1 then
7              # B[i]_Q is not a leaf in Q and B[i]_T is not a leaf in T
8              F(B[i]) = (l(B[i]_Q), B[i]_Q − 1; l(B[i]_T), B[i]_T − 1)
9      else # F(B[i]) is empty
10         (x, y) = top(Stack); pop(Stack) #˜x is a direct descendant of B[i]_Q
11         # computing the rightmost chainable area of B[i]:
12         if u(x) + 1 ≤ B[i]_Q − 1 and u(y) + 1 ≤ B[i]_T − 1 then
13             F(B[i])+ = (u(x) + 1, B[i]_Q − 1; u(y) + 1, B[i]_T − 1)
14         # computing the intermediate chainable areas of B[i]:
15         while Stack is not empty and top(Stack)_Q ≥ l(B[i]_Q) do
16             (s, t) = top(Stack); pop(Stack)
17             if u(s) + 1 ≤ l(x) − 1 and u(t) + 1 ≤ l(y) − 1 then
18                 F(B[i])+ = (u(s) + 1, l(x) − 1; u(t) + 1, l(y) − 1)
19             (x, y) = (s, t)
20         # computing the leftmost chainable area of B[i]
21         if l(x) − 1 ≥ l(B[i]_Q) and l(y) − 1 ≥ l(B[i]_T) then
22             F(B[i])+ = (l(B[i]_Q), l(x) − 1; l(B[i]_T), l(y) − 1)
23     push(Stack, B[i])
```

[6] Since a seed is a valid mapping, ancestral and order relations between border nodes are respected. Thus, if a border node is a leaf in Q_P, it is also a leaf in T_P.

Lines 5–8 of Algorithm 1 correspond to the first case and requires a single additional explanation: if $B[i]_Q$ (resp. $B[i]_T$) is a leaf of Q (resp. T), then the chainable area of $B[i]$ is empty.

In the second case, there can be several chainable areas associated to $B[i]$, separated by the descendants of $B[i]$ in Q and T. A fundamental point is that the current pair $B[i]$ has necessarily descendants in $B(P)$, and that these descendants have already been inserted into *Stack*. In particular, it is easy to see that *Stack* contains only entries (x, y) such that x is a *direct descendant* of $B[i]_Q$, and the roots of previous (already processed) internal trees of P: no vertex on the path from x to $B[i]_Q$ belongs to B followed by the roots of the already processed internal trees (their postfix index are less than $l(B[i]_Q)$, line 15).

The (possibly empty) chainable area $(a, b; c, d)$ to the right of the rightmost direct descendant (x, y) of $B[i]$ ($a > x$ and $c > y$) and the (possibly empty) chainable area $(a, b; c, d)$ to the left of its leftmost direct descendant (x, y) ($b < x$ and $d < y$) require a particular treatment (lines 12–13 and lines 21–22 respectively).

The possible chainable areas between two direct descendants are considered by the loop defined by lines 15–19. To compute these chainable areas, we rely on the following simple properties for a pair $(x, x^j) \in B(P)$ and a chainable area $(a, b; c, d)$ of (x, x^j):

1. $(a, b; c, d)$ is such that b and d are children of x and x^j, and a and c are the leftmost leafs of children of x and x^j.
2. If $(a, b; c, d)$ is not the rightmost chainable area, b and d are such that $b + 1$ and $d + 1$ are the leftmost leafs of a direct descendant of x and x^j.
3. If $(a, b; c, d)$ is not the leftmost chainable, a and c are such that $a - 1$ and $c - 1$ are children of x and x^j and are either border nodes or ancestors of border nodes.

As *Stack* is sorted increasingly, the top contains the rightmost direct descendants of the current pair $B[i]$. The loop in lines 15–19 computes the area between the direct descendants using the above properties. Note that any direct descendant is now out of the *Stack* and is replaced by the current pair. At the end of the algorithm, *Stack* contains the roots of the internal trees of P.

Finally, before each insertion of a new chainable area into $F(x, y)$, we test whether the area is empty or not (*cf.* lines 6, 12, 17, 21).

Property 3. The time complexity of this algorithm is in $O(|B(P)|)$.

This proposition is obvious as we iterate on any pair and each pair is inserted only once into *Stack*.

Remark 7. From now, we then assume that the chainable areas of all seeds of a set $S = \{P^0, \ldots, P^m\}$ of seeds have been computed in time $O(\|S\|)$, and that, for every seed P^j and vertex i of Q belonging to the border of P^j in Q, $F_j(i)$ is encoded by a list ordered by increasing postfix order (i.e. the order in which they have been computed by Algorithm 1).

5 Algorithms for the Maximum Chaining Problem

We first describe a simple implementation of the recursive formula given in Section 3 which does not require any special data structure algorithm. In a second step, we introduce our main result, a more intricate algorithm proving Theorem 1.

From now, we consider two ordered trees Q and T, a set $S = \{P^0, \ldots, P^{m-1}\}$ of m seeds and a scoring function v on S. We assume that the score $v(j)$ of a seed P^j can be accessed in constant time. Moreover, for a node i in Q belonging to a seed P^j, we assume that the corresponding node in T, i^j (or more precisely its postfix number in T) can be accessed in constant time. Finally, for every node i in Q and T, its leftmost leaf $l(i)$ is also supposed to be accessed in constant time. We also assume that the seeds P^i are sorted increasingly according to the postfix number of their roots in Q (Remark 3).

5.1 A simple algorithm

We assume that the seeds of S are given as a list I of triples (i, f, j) such that:

1. i is the postfix number of either the root of Q_{P^j} or a border node of Q_{P^j} (*i.e.* $i \in B(Q_{P^j}) \cup \{r_j\}$) and,
2. f is a flag indicating if i is either border ($f = 0$) or root ($f = 1$) for Q_{P^j}.

Hence, we do not require as input the whole seed mappings but just borders and roots of the seeds, as it is usual when chaining seeds in sequences for example.

During a preprocessing phase, I is sorted in lexicographic order: $(i, r, j) < (i', r', j')$ if $i > i'$ or $i = i',\ r < r'$ or $i = i',\ r = r',\ j < j'$. Thus, if a node is both in the border and root of P^j, it first appears in I in two occurrences: first as a border, then as a root.

In our algorithm, we visit successively the elements of I in increasing order, and a seed P^j is said to be *processed* after its root has been processed (*i.e.* the current element of I is greater than $(r_j, 1, j)$ for the order defined above).

In order to compute in constant time the partial MCP for any pair of forests in $CA(P^j)$ as described in equation (1), we introduce a data structure M indexed by quadruples of integers $(a, b; c, d)$ defining the forests $Q[a, b]$ and $T[c, d]$. These quadruples $(a, b; c, d)$ belong to a set $Y = Y_1 \cup Y_2 \cup Y_3$ defined as follows:

$$Y_1 = \bigcup_{j=0}^{m-1} CA(P^j), Y_2 = \{(0, r_Q, 0, r_T)\},$$

$$Y_3 = \{(a, l(r_j) - 1; c, l(r_j^j) - 1) \mid \exists (b, d); (a, b; c, d) \in Y_1 \cup Y_2 \text{ and } P^j \subset (a, b; c, d)\}$$

In order to understand intuitively Y_3, note that, for each chainable area $(a, b; c, d)$ and each seed P^j included in this chainable area, there is an entry of Y_3 indicating the parts of the trees Q and T that can contain a chain having P^j as last seed.

We also use an array V of m integers to store the intermediate values of the table MCP' (equation (1)). Finally, in Algorithm 2, the function `Update` replaces the value of $M[a, b, c, d]$ by a given real number w if w is greater than $M[a, b, c, d]$.

Algorithm 2 MCP_1: compute the score of a maximum chain.

```
1  for j from 0 to m − 1 do V[j] = v(j)
2  foreach (a, b; c, d) ∈ Y do M[a, b, c, d] = 0
3  foreach (i, f, j) in I do
4    if f = 0 then # i.e. (i, i^j) ∈ B(P^j)
5      foreach (a, b; c, d) ∈ F_j(i) do V[j] = V[j] + M[a, b, c, d]
6    else # i.e. f = 1 and i is the root of Q_{P^j}, i = r_j
7      foreach (a, b; c, d) ∈ Y_1 ∪ Y_2 s.t. P^j ⊂ (a, b; c, d) do
8        Update M[a, b, c, d] with w = V[j] + M[a, l(Q_{P^j}) − 1, c, l(T_{P^j}) − 1]
9        foreach P^g ⊂ (r_j + 1, b; r_j^j + 1, d) do
10         Update M[a, l(Q_{P^g}) − 1, c, l(T_{P^g}) − 1] with w
11     V[j] = V[j] + M[0, l(Q_{P^j}) − 1, 0, l(T_{P^j}) − 1]
12 return max_j V[j]
```

Correctness of the algorithm The correctness of the algorithm relies on the following invariants for the two data structures V and M:

M1. After the root of P^j has been processed, then for every $(a, b; c, d) \in Y$, $M[a, b, c, d] = MCP(Q[a, b], T[c, d], \{P^0, \ldots, P^j\})$.

V1. After the root of P^j has been processed, then $V[j] = MCP'(Q, T, \{P^0, \ldots, P^j\})$.

Obviously, V1 implies that $\max_j V[j]$ contains the score of the maximum chain (equations (2) and (3)). Let us assume now that M1 is satisfied. If the seed P^j has been processed, then $V[j]$ contains the sum of $v(j)$ (line 1), the MCP scores of the chainable areas of all its border nodes (line 5) and the MCP score between forests $Q(0, l(Q_{P^j}) - 1)$ and $T(0, l(T_{P^j}) - 1)$ (line 11). From Property 2 and (1), $V[j] = MCP'(Q, T, \{P^0, \ldots, P^j\})$ and V1 is satisfied.

We prove M1 by induction. Initially, since no seed has been processed, line 2 ensures that M1 is satisfied. Now let us assume that M1 is satisfied for all processed seeds $\{P^0, \ldots, P^{j-1}\}$ and the input $(i, 1, j)$ is being processed. If $P^j \not\subset (a, b; c, d)$, then by induction, M1 is satisfied for $M[a, b, c, d]$. Otherwise, the loop in lines 7 and 8 ensures that M1 is satisfied for all entries $M[a, b, c, d]$ such that $(a, b; c, d) \in Y_1 \cup Y_2$, as $(a, l(Q_{P^j}) - 1; c, l(T_{P^j}) - 1)$ does not contain P^j ; thus by induction M1 is satisfied for this index. Finally, the loop in lines 9–9 update all $(a, b; c, d) \in Y_3$ containing P^j, and M1 is satisfied for all entries of M.

Complexity analysis From Property 1, the space required to encode the entries of M indexed by Y_1 is in $O(\|S\|)$. The space required to encode the entries of M indexed by Y_3 is in $O(m^2)$, as for every pair of seeds P^i and P^j, there is at most one chainable area of $CA(P^i)$ that contains P^j.

We now address the worst-case time complexity. First, we discuss the complexity of accessing the chainable areas: we assume that for every entry (i, f, j) of I, $F_j(i)$ has been computed by Algorithm 1 and that there is a constant time access to the first element of $F_j(i)$ (leftmost chainable area), and that the following chainable areas are accessed in constant time as $F_j(i)$ is encoded by a list (Remark 7). The cost of lines 4–5 is $O(\|S\|)$, as each chainable area is considered once, there are $O(\|S\|)$ such areas, and we assumed we can access them in constant time. A naive implementation of lines 6–11 would require $O(m^2\|S\|)$ operations: indeed, there are m iterations of the loop in line 5, the loop in line 7 considers only entries indexed by $Y_1 \cup Y_2$ (there are $O(\|S\|)$ such entries) and the loop on line 9 iterates $O(m)$ times. However, we can notice that there are $O(m)$ entries $(a, b; c, d) \in Y_1 \cup Y_2$ such that $P^j \subset (a, b; c, d)$, and it is possible to pre-process I in time and space $O(m\|S\|)$ in such a way that the loop in line 7 can be implemented to perform $O(m)$ iterations (as Algorithm 2 is not our most efficient algorithm, we do not detail here this pre-processing), leading to a total time complexity of $O(\|S\| \log(\|S\|) + m\|S\| + m^3)$ (respectively for sorting the input, preprocessing of I and then the main algorithm).

5.2 A more efficient algorithm

The key idea to improve the time complexity is to access less entries from M (while maintaining property M1 on the remaining entries though) and to complement M with a data structure R that can be queried in time $O(\log(m))$ instead of $O(1)$, but whose maintenance does not require a loop with $O(m^2)$ iterations.

Formally, let $X = \{(a, c) \text{ s.t. } \exists (a, b; c, d) \in Y_1 \cup Y_2\}$ and R be a data structure indexed by X such that, for a given index $(a, c) \in X$, $R[a, c]$ is a set of pairs (j, s) where j is the index of the seed P^j and s is the maximum score of chains in $(Q[a, r_j], T[c, r_j^j])$ that ends with P^j. Intuitively,

- M is used to access, still in $O(1)$ time, the values $MCP(a, l(Q_{P^j})-1, c, l(T_{P^j})-1, \{P^0 \ldots P^{j-1}\})$ required to compute MCP' in equation (1),
- $R[a, c]$ is used to access, in time $O(\log(m))$, the scores of the best chains included in $(Q[a, r_Q], T[c, r_T])$ (the values $MCP(Q[e, f], T[g, h], \{P^0 \ldots P^{j-1}\})$ in equation (1)) and replace the entries $M[a, b, c, d]$ with $(a, b; c, d) \in Y_1 \cup Y_2$, which were used in the previous algorithm.

The algorithm iterates on a list of triples $J = I \bigcup \left(\cup_{j=0}^{m-1} (l(Q_{P^j}), -1, j)\right)$, sorted using the lexicographic order used in the previous section, with the following modification: if we have two seeds P^j and P^g with $g > j$ such that $(l(Q_{P^j}), l(T_{P^j})) = (l(Q_{P^g}), l(T_{P^g}))$ then only $(l(Q_{P^j}), -1, j)$ occurs in J. Computing J can be done in $O(||S|| \log(||S||))$ time.

Algorithm 3 $MCP_2(Q, T, S, v)$: compute a maximum chaining from S.

```
1  for j from 0 to m − 1 do V[j] = v(j)
2  foreach (a, b; c, d) ∈ Y_3 do M[a, b, c, d] = 0
3  foreach (a, c) ∈ X do R[a, c] = ∅
4  foreach (i, f, j) in J do
5    if f = −1 then # i = l(Q_{P^j})
6      foreach (a, c) ∈ X s.t. a, c < l(Q_{P^j}), l(T_{P^j}) do
7        M[a, l(Q_{P^j}) − 1, c, l(T_{P^j}) − 1]= value s of the last (y, s) of R[a, c] s.t. r^y_y < l(T_{P^j})
8    else if f = 0 then # (i, i^j) ∈ B(P^j)
9      foreach (a, b; c, d) ∈ F_j(i) do
10       Add to V[j] the value s of the last entry (y, s) of R[a, c] s.t. r^y_y ≤ d
11   else # f = 1 and i is the root of Q_{P^j}, i = r_j
12     foreach (a, c) ∈ X s.t. a, c ≤ l(Q_{P^j}), l(T_{P^j}) do
13       w = V[j] + M[a, l(Q_{P^j}) − 1, c, l(T_{P^j}) − 1]
14       Insert entry (j, w) into R[a, c] and update R[a, c] as follow:
15             Find the last entry (y, s) s.t. r^y_y < r^j_j
16             if s < w then
17                   Insert (j, w) just after (y, s) in R[a, c]
18                   Remove from R[a, c] all entries (z, t) s.t. r^j_j ≤ r^z_z and t < w
19     V[j] = V[j] + M[0, l(Q_{P^j}) − 1, 0, l(T_{P^j}) − 1]
20 return max_j V[j]
```

Correctness of the algorithm We consider the following invariants.

M2. After the root of P^j has been processed, then $M[a, b, c, d] = MCP(Q[a, b], T[c, d], \{P^0, \ldots, P^j\})$ for every $(a, b; c, d) \in Y_3$.

V1. After the root of P^j has been processed, then $V[j] = MCP'(Q, T, \{P^0, \ldots, P^j\})$.

R1. After the root of P^j has been processed, then for all $(a, c) \in X$, $R[a, c]$ contains all (y, s) satisfying
   a. $y \leq j$ and $s = MCP'(Q[a, r_y], T[c, r^y_y], \{P^0, \ldots, P^y\})$.
   b. $\forall (z, t) \in R[a, c],\ r^z_z < r^y_y \Rightarrow t < s$.

R2. $\forall (a, c) \in X$, $R[a, c]$ is totally ordered as follows: $(y, s) < (z, t)$ iff $r^y_y < r^z_z$.

We first assume that R1 and R2 are satisfied. As previously, if V1 is satisfied, then the algorithm computes $MCP(Q, T, S)$. Initialization line 1 ensures that $V[j]$ contains $v(j)$. Next, to prove V1, we only need to show that, when we process a border i of a seed P^j (line 10), we add to $V[j]$ the best chain of each chainable area $(a, b; c, d)$ of the border; it follows from three facts:

1. every seed P^{j+e} with $e > 0$ does not belong to the forest $Q[a, b]$ (because $b < i \leq r_{j+e}$) and thus cannot belong to a chain in the $(a, b; c, d)$ area,
2. the score of this chain is present in $R[a, c]$ (from R1) and,
3. it is the last entry (y, s) such that $r^y_y \leq d$ (from R2).

M2 can be proved in a similar way than M1, but restricted to entries $M[a, b, c, d]$ such that $(a, b; c, d) \in Y_3$. To check M2 is satisfied, we only need to focus on line 7, as it is the only line that updates M. For entries $M[a, b, c, d]$ such that $a \geq l(Q_{P^j})$ or $c \geq l(T_{P^j})$, $M[a, b, c, d] = 0$ due to the initialization in line 1. For all other entries, M2 follows immediately from R1 and R2, using arguments similar to the previous ones.

Finally, we need to check whether R1 and R2 are satisfied. First, as previously, in the case where $a \geq l(Q_{P^j})$ or $c \geq l(T_{P^j})$, $R[a, c] = \emptyset$ which is ensured by the initialization in line 3. So we need only to consider the case where $a, c < l(Q_{P^j}), l(T_{P^j})$, which is handled in lines 11 to 18. Every seed P^y such that $y < j$ has already been processed and $s = MCP'(Q[a, r_y], T[c, r^y_y], \{P^0, \ldots, P^y\})$ cannot be modified after P^y has been processed, so lines 12 and 13, together with M2, ensure that (y, s) has been inserted into $R[a, c]$ previously, and the same argument applies if $y = j$. Entries (z, t) removed at line 18 do not belong to any of these (y, s), which implies that R1.a and R1.b, and so R1, are satisfied. R2 is obviously satisfied from the position where (j, w) is inserted into $R[a, c]$ in line 17.

Complexity analysis: proof of Theorem 1 The space complexity is given by the space required for structures M and R. M requires a space in $O(m^2)$ as it is indexed by Y_3. R requires a space in $O(m\|S\|)$, as $|Y_1 \cup Y_2| \in O(\|S\|)$ and for each seed P^j, an entry (j, s) is inserted at most once in each $R[a, c]$. All together, the space complexity is then $O(m^2 + m\|S\|) = O(m\|S\|)$.

We now describe the time complexity. First, note that following the technique used for computing maximum chains in sequences [7, 10, 13], the structures $R[l_Q, l_T]$ can be implemented using classical data structures such as AVL or concatenable queues supporting query requests, insertions and deletions, successor and predecessor, in a set of n totally ordered elements in $O(\log(n))$ worst-case time (also called Range Minimum Query).

Now, we analyze the complexity of lines 5 to 7. The loop of line 6 is performed at most $O(m\|S\|)$ times and each iteration requires $O(\log(m))$ time (line 7), which gives a total time complexity of $O(m\|S\| \log(m))$.

Line 10 is applied at most once for each of the $O(\|S\|)$ chainable area $F_j(i)$ (Property 1), and each iteration requires $O(\log(m))$ time, which gives an $O(\|S\| \log(m))$ total time complexity.

Finally, we analyze the complexity of lines 11 to 19. First, we do not consider the operation in line 18. The loop starting in line 12 is performed in $O(m)$ time, and the complexity of each loop is in $O(\|S\|)$ time. The cost of the operations performed during each iteration is $O(\log(\|S\|))$ (lines 13 and 16 are both performed in $O(1)$ and lines 14 and 15 in time $O(\log(\|S\|))$). The total time complexity of this part, without considering line 18, is thus $O(m\|S\| \log(\|S\|))$. To complete the time complexity analysis, we show that the complexity of line 18 is in $O(m\|S\|)$. Indeed, it follows from R2 that all entries removed in one step are consecutive in the total order on $R[a, c]$ defined in R2. Hence, if one call to line 18 removes k elements from $R[a, c]$, it can be done in $O((k + 2)\log(m))$ time, as the successor of a given element can be retrieved in $O(\log(m))$ time. Since every element of R is removed at most once during the whole algorithm, this leads to an amortized complexity of $O(m\|S\| \log(m))$ for line 18. Altogether, our algorithm computes $MCP(Q, T, S)$ in time $O(m\|S\| \log(m))$, using standard data structures and after a preprocessing in time $O(\|S\| \log(\|S\|))$ to compute the chainable areas and to sort J. This proves Theorem 1.

Remark 8. If we consider that Q and T are sequences, or, as described in Remark 6, unary trees, then each of the two trees has a single leaf and each seed is unambiguously defined by its root and border, which implies that$\|S\| = m$. There is only one $R[a, c]$, as $a = c = 0$, that contains $O(m)$ entries. Hence, all loops that were iterating on R have now a single iteration, which reduces the time complexity by a factor m to $O(\|S\| \log(m)) = O(m \log(m))$.

6 Comparison with Heyne *et al.* algorithm [8]

Heyne *et al.* [8] also introduced a chaining problem on an alternative representation of ordered trees. In this section, we first describe their work, together with a detailed analysis of its complexity, followed by a comparison with our algorithm.

Heyne et al. *dynamic programming algorithm* The dynamic programming algorithm described in [8] is based on a recursive traversal of the chainable areas in Q and T. Given two nodes a and c, a dynamic programming matrix $D^{a,c}[b, d]$ contains the score of the best chain in the area $(a, b; c, d)$. Furthermore, an array W is used to store in $W[j]$ the values:

$$MCP'(Q[l(Q_{P^j}), r_j], T[l(T_{P^j}), r_j^j], \{P^0 \dots P^j\}).$$

The dynamic programming recurrence is the following:

$$D^{a,c}[b, d] = \max \begin{cases} 0 & \text{if } b < a \text{ or } d < b \\ D^{a,c}[b-1, d] & \\ D^{a,c}[b, d-1] & \\ \forall P^j \in S \text{ s.t. } r_j = b,\ r_j^j = d,\ l(Q_{P^j}) \geq a \text{ and } l(T_{P^j}) \geq c\text{:} & \\ \quad D^{a,c}[l(Q_{P^j}) - 1, l(T_{P^j}) - 1] + W[j] & \end{cases} \tag{4}$$

where

$$W[j] = v(P^j) + \sum_{(a',b';c',d') \in CA(P^j)} D^{a',c'}[b',d'].$$

To account for the dependence between the computation of the matrices $D^{a,c}$ and of the array W, seeds are processed in increasing postfix order (in Q) of their roots. Finally, the MCP score is stored in $D^{0,0}[r_Q, r_T]$ (note this presentation differs slightly from the one given in [8] as we use the concept of chainable area).

Let n_1 denote the size of Q and n_2 the size of T. Without loss of generality, we assume that $n_2 \leq n_1$. The algorithm described in [8] can be implemented with an $O(n_1 n_2)$ space complexity by discarding from the memory the matrix $D^{a,c}$ after the computation of each chainable area. Details are given in [8] and we now consider the worst-case time complexity. In a preliminary step, the algorithm requires to sort all the seeds and all the chainable areas, which can obviously be done in $O(\|S\| \log \|S\|)$ worst-case time as the number of chainable areas is at most linear in the sum of the size of the borders, *i.e.* $O(\|S\|)$. Then, each area $(a, b; c, d)$ (*i.e.* the computation of the entry $D^{a,c}[b, d]$) requires a processing time in $O((b-a)(d-c)+m)$. (assuming the required entries of W have already been computed). This time complexity is actually bounded by $O(n_1 n_2 + m)$. Updating $W[j]$ can be done incrementally when entries $D^{a',c'}[b',d']$ for $(a',b';c',d') \in CA(P^j)$ are computed, with no additional asymptotic cost. Since a dynamic programming matrix is computed for each chainable area, and there are at most $n_1^2 n_2^2$ such areas, the overall time complexity is then:

$$O(\|S\| \log \|S\| + \min(\|S\|, n_1^2 n_2^2)(n_1 n_2 + m)). \tag{5}$$

With the model of seeds considered in [8], the number of chainable areas (*i.e.* $\|S\|$), and so the number of seeds, can be bounded by $O(n_1 n_2)$ which results in the worst-case time complexity of $O(n_1^2 n_2^2)$ and space complexity of $O(n_1 n_2)$ stated in [8].

Chaining seeds in sequences The problem of chaining a set of m seeds over two sequences Q and T of respective lengths n_1 and n_2 can be solved in $O(m \log m)$ [7]. This complexity is the best known, up to now. The algorithm consists in sorting the extremities of the seeds and then considering them incrementally. For each extremity, a balanced search tree (BST) is used to find the best compatible chain or to insert a newly computed chain [13]. Since the size of the BST is bounded by n_1, this algorithm has a worst-case complexity of $O(m \log min(m, n_1))$ in time and $O(m + min(m, n_1))$ in space.

For sequences, Heyne *et al.* algorithm [8] requires a single dynamic programming matrix C such that $C[a, b]$ contains the best chain included in $Q[0 \dots a]$ and $T[0 \dots b]$. Moreover, it does not require that the seeds are sorted and thus computes the maximal chaining in $O(n_1 n_2 + m)$ in time and $O(n_1 n_2)$ in space (or, even better, linear space if no backtracking is performed to extract an optimal chain). One major reason for this complexity difference is that the Heyne's algorithm [8] factorizes some computations that are repeated in the classical seeds chaining algorithm.

So, if the number of seeds is greater than $\frac{n_1 n_2}{\log n_1}$, Heyne's algorithm [8] has a better asymptotic complexity than the usually considered algorithm, that relies on querying BST, since the cost for querying the BST overtakes the cost of scanning the whole search space. As far as we know, this property of dynamic programming algorithm for chaining seeds in sequences was never noticed.

In practice, chaining seeds is often used to avoid a costly pairwise comparison between Q and T. In genomics, for example, FASTA [11] makes use of seeds to avoid a direct quadratic time alignment between the two sequences. The present comparison exhibits the limit on the number of seeds so that the filter remains efficient. In practice, if the seeds are well designed (and in particular have a good specificity), we can assume that m is in $O(min(n_1, n_2))$ in which case the classical algorithm [13, 7] is better, as it does not depend on the size of Q and T.

Chaining seeds in ordered trees The analysis we just outlined for chaining seeds in sequences holds with minor modifications for our problem. Indeed, Algorithm 3 is an extension of the classical algorithm for chaining seeds in sequences, and its worst-case time complexity does not depend on the size of Q and T, but it requires BST to avoid exhaustive traversal of chainable areas. Hence, Heyne's algorithm can have a better worst-case time complexity than Algorithm 3 depending on the structure of the set of seeds to consider.

For example, it is straightforward from our complexity analysis that, if $\|S\| \leq n_1^2 n_2^2$ and $m \log(m) \leq n_1 n_2$, then our algorithm has a better asymptotic worst-case time complexity. Indeed, Heyne *et al.*'s algorithm has a $O(\|S\|(\log(\|S\|) + n_1 n_2 + m))$ time complexity, while our algorithm has a $O(\|S\|(\log(\|S\|) + m \log(m))$ time complexity. Also, if the seeds are of constant size (*i.e.* for every seed P, $|B(P)| < k$ for some given constant k) – which is often the case in computational biology where k-mers are a widely used model of seeds – we have $\|S\| = O(m)$ and, in this case, our algorithm is more efficient if $m \leq \frac{n_1 n_2}{\log n_1}$, which is a realistic assumption.

7 Conclusion

The current paper describes algorithms to solve chaining problems in ordered trees. With respect to similar problems in sequences, these methods exhibit a linear factor increase both in time and space. Chains so obtained can be used to speed-up RNA structure comparisons, as illustrated in [8, 12].

The comparison between our algorithm and Heyne *et al.*'s algorithm [8] raises interesting questions on the impact of the number of seeds in chaining problems. In particular, algorithmic approaches depends both on the number of seeds and the structure of chainable areas. Thus, defining a unified approach that would perform optimally in any case still remains an open problem.

Another natural question related to chaining problems, that, as far as we know, has not been considered in the case of sequences, is to decide whether a given seed P of a set of seeds S belongs to *any* optimal chains or not. However a trade-off between quality and speed needs to be found. Indeed, identifying these *always optimal* seeds would probably ensure good quality chains, whereas the high complexity of these identifications might slow down the detection of similar structures in a large database.

Finally, the problem of finding chains of overlapping seeds has been investigated only recently, and deserves attention [18].

Acknowledgments. Pacific Institute for Mathematical Sciences (PIMS, UMI CNRS 3069), Agence Nationale pour la Recherche project BRASERO (ANR-06-BLAN-0045), Natural Sciences and Engineering Research Council of Canada (NSERC), Multiscale Modeling of Plants associated team (INRIA).

References

1. S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990.
2. S. Aluru, editor. *Handbook of Computational Molecular Biology.* CRC Press, 2005.
3. R. Backofen and S. Will. Local sequence-structure motifs in RNA. *J. Bioinform. Comput. Biol.*, 2(4):681–698, 2004.
4. D.G. Brown. A survey of seeding for sequence alignment. In (I. Mandoiu, A. Zelikovsky, ed.) *Bioinformatics Algorithms: Techniques and Applications.* Wiley-Interscience, 2008.
5. E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):Article 2, 2009.
6. P.P. Gardner, J. Daub, J.G. Tate, *et al.* Rfam: updates to the RNA families database. *Nucleic Acids Res.*, 37(Database issue):D136–D140, 2009.
7. D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.
8. S. Heyne, S. Will, M. Beckstette, and R. Backofen. Lightweight comparison of RNAs based on exact sequence-structure matches. *Bioinformatics*, 25(16):2095–2102, 2009.
9. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *J. Comput. Biol.*, 9(2):371–388, 2002.
10. D. Joseph, J. Meidanis, and P. Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In *SWAT 1992*, volume 621 of *LNCS*, pp 326–337. 1992.
11. D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.
12. A. Lozano, R.Y. Pinter, O. Rokhlenko, G. Valiente, and M. Ziv-Ukelson. Seeded tree alignment. *IEEE/ACM TCBB*, 5(4):503–513, 2008.

13. E. Ohlebusch and M.I. Abouelhoda. Chaining Algorithms and Applications in Comparative Genomics. In (S. Aluru, ed.) *Handbook of Computational Molecular Biology*. CRC Press, 2005.
14. W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *PNAS*, 85(8):2444–2448, 1988.
15. J.S. Pedersen *et al.* Identification and classification of conserved RNA secondary structures in the human genome. *PLoS Comput. Biol.*, 2(4):e33, 2006.
16. B.A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *CABIOS*, 6:309–318, 1990.
17. K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.
18. R. Uricaru, A. Mancheron and E. Rivals. Novel Definition and Algorithm for Chaining Fragments with Proportional Overlaps. *RECOMB-CG 2010*, volume 6398 of *LNCS*, pp 161–172, 2010.
19. Y. Wan, M. Kertesz, R.C. Spitale, E. Segal and H.Y. Chang. Understanding the transcriptome through RNA structure. *Nature Reviews Genetics* 12:641–655, 2011.
20. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.