Procrastination Leads to Efficient Filtration for Local Multiple Alignment

Aaron E. Darling[1], Todd J. Treangen[2], Louxin Zhang[4], Carla Kuiken[5], Xavier Messeguer[2], and Nicole T. Perna[3]

[1] Department of Computer Science, University of Wisconsin, USA
darling@cs.wisc.edu
[2] Department of Computer Science, Technical University of Catalonia, Barcelona, Spain
treangen@lsi.upc.edu
[3] Department of Animal Health and Biomedical Sciences, Genome Center, University of Wisconsin, USA
[4] Department of Mathematics, National University of Singapore, Singapore
[5] T-10 Theoretical Biology Division, Los Alamos National Laboratory, USA

Abstract. We describe an efficient local multiple alignment filtration heuristic for identification of conserved regions in one or more DNA sequences. The method incorporates several novel ideas: (1) palindromic spaced seed patterns to match both DNA strands simultaneously, (2) seed extension (chaining) in order of decreasing multiplicity, and (3) procrastination when low multiplicity matches are encountered. The resulting local multiple alignments may have nucleotide substitutions and internal gaps as large as w characters in any occurrence of the motif. The algorithm consumes $\mathcal{O}(wN)$ memory and $\mathcal{O}(wN \log wN)$ time where N is the sequence length. We score the significance of multiple alignments using entropy-based motif scoring methods. We demonstrate the performance of our filtration method on Alu-repeat rich segments of the human genome and a large set of Hepatitis C virus genomes. The GPL implementation of our algorithm in C++ is called `procrastAligner` and is freely available from `http://gel.ahabs.wisc.edu/procrastination`

1 Introduction

Pairwise local sequence alignment has a long and fruitful history in computational biology and new approaches continue to be proposed [1,2,3,4]. Advanced filtration methods based on spaced-seeds have greatly improved the sensitivity, specificity, and efficiency of many local alignment methods [5,6,7,8,9]. Common applications of local alignment can range from orthology mapping [10] to genome assembly [11] to information engineering tasks such as data compression [12]. Recent advances in sequence data acquisition technology [13] provide low-cost sequencing and will continue to fuel the growth of molecular sequence databases. To cope with advances in data volume, corresponding advances in computational methods are necessary; thus we present an efficient method for local multiple alignment of DNA sequence.

P. Bücher and B.M.E. Moret (Eds.): WABI 2006, LNBI 4175, pp. 126–137, 2006.

Unlike pairwise alignment, local multiple alignment constructs a single multiple alignment for all occurrences of a motif in one or more sequences. The motif occurrences may be identical or have degeneracy in the form of mismatches and indels. As such, local multiple alignments identify the basic repeating units in one or more sequences and can serve as a basis for downstream analysis tasks such as multiple genome alignment [14,15,16,17], global alignment with repeats [18,19], or repeat classification and analysis [20]. Local multiple alignment differs from traditional pairwise methods for repeat analysis which either identify repeat families *de novo* [21] or using a database of known repeat motifs [22].

Previous work on local multiple alignment includes an Eulerian path approach proposed by Zhang and Waterman [23]. Their method uses a *de Bruijn* graph based on exactly matching k-mers as a filtration heuristic. Our method can be seen as a generalization of the *de Bruijn* filtration to arbitrary spaced seeds or seed families. However, our method employs a different approach to seed extension that can identify long, low-copy number repeats.

The local multiple alignment filtration method we present has been designed to efficiently process large amounts of sequence data. It is not designed to detect subtle motifs such as transcription factor binding sites in small, targeted sequence regions–stochastic methods are better suited for such tasks [24].

2 Overview of the Method

Our local multiple alignment filtration method begins by generating a set of candidate multi-matches using *palindromic* spaced seed patterns, listed in Table 1. The seed pattern is evaluated at every position of the input sequence, and the lexicographically-lesser of the forward and reverse complement subsequence induced by the seed pattern is hashed to identify seed matches—see Figure 1. The use of *palindromic* seed patterns offers computational savings by allowing both strands of DNA to be processed simultaneously.

Given an initial set of matching sequence regions, our algorithm then maximally extends each match to cover the entire surrounding region of sequence identity. A visual example of maximal extension is given by the black match

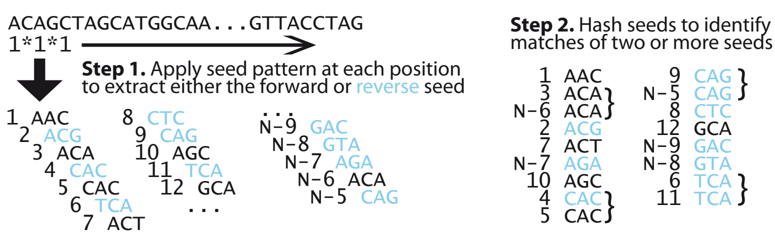

Fig. 1. Application of the palindromic seed pattern `1*1*1` to identify degenerate matching subsequences in a nucleotide sequence of length N. The lexicographically-lesser of the forward and reverse complement subsequence induced by the seed pattern is used at each sequence position.

Table 1. Palindromic spaced seeds used by `procrastAligner`. The sensitivity ranking of a seed at various levels of sequence identity is given in the columns at right. A seed with rank 1 is the most sensitive seed pattern for a given weight and percent sequence identity. The default seeds used by `procrastAligner` are listed here, while the full list of high-ranking seeds appears on the website.

Weight	Pattern	Seed Rank by Sequence Identity					
		65%	70%	75%	80%	85%	90%
5	11*1*11	1	1	1	1	1	1
6	1*11***11*1	1	1	1	1	1	1
7	11**1*1*1**11	1	1	1	1	1	1
8	111**1**1**111	1	1	1	1	1	1
9	111*1**1**1*111	1	1	1	1	1	1
10	111*1**1*1**1*111	1	1	1	1	1	1
11	1111**1*1*1**1111	1	1	1	1	1	2
12	1111**1*1*1*1**1111	5	3	1	1	1	1
13	1111**1**1*1*1**1**1111	> 10	5	1	1	1	1
14	1111**11*1*1*11**1111	2	2	1	1	1	1
15	1111*1*11**1**11*1*1111	1	1	1	1	1	1
16	1111*1*11**11**11*1*1111	2	1	1	1	1	1
18	11111**11*1*11*1*11**11111	1	1	1	1	1	1
19	1111*111**1*111*1**111*1111	5	2	1	1	1	1
20	11111*1*11**11*11**11*1*11111	> 10	> 10	3	1	1	1
21	11111*111*11*1*11*111*11111	1	1	1	3	3	2

in Figure 2. In order to extend over each region of sequence $\mathcal{O}(1)$ times, our method extends matches in order of decreasing multiplicity–we extend the highest multiplicity matches first. When a match can no longer be extended without including a gap larger than w characters, our method identifies the neighboring *subset* matches within w characters, i.e., the light gray seed in Figure 2. We then *link* each neighboring subset match to the extended match. We refer to the

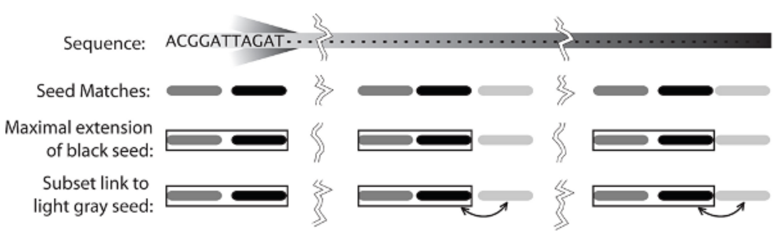

Fig. 2. Seed match extension. Three seed matches are depicted as black, gray, and light gray regions of the sequence. Black and gray have multiplicity 3, while light gray has multiplicity 2. We maximally extend the black seed to the left and right and in doing so, the black seed chains with the gray seed to the left. The light gray seed is adjacent to only two out of three components in the extended black seed. We *procrastinate* and extend the light gray seed later. We create a link between light gray and the extended black seed match.

extended match as a *superset* match. Rather than immediately extend the subset match(es), we *procrastinate* and extend the subset match later when it has the highest multiplicity of any match waiting to be extended. When extending a match with a linked superset (light gray in Figure 2), we immediately include the entire region covered by the linked superset match–obviating the need to re-examine sequence already covered by a previous match extension.

We score alignments generated by our method using the entropy equation and exact p-value method in [25]. Our method may produce many hundreds or thousands of local multiple alignments for a given genome sequence, thus it is important to rank them by significance. When computing column entropy, we treat gap characters as missing data.

3 Algorithm

3.1 Notation and Assumptions

Given a sequence $\mathcal{S} = s_1, s_2, \ldots, s_N$ of length N defined over an alphabet $\{A, C, G, T\}$, our goal is to identify local multiple alignments on subsequences of $\mathcal{S}$. Our filtration method first generates candidate chains of ungapped alignments, which are later scored and possibly re-aligned. Denote an ungapped alignment, or match, among subsequences in $\mathcal{S}$ as an object M. We assume as input a set of ungapped alignments $\mathbf{M}$. We refer the number of regions in $\mathcal{S}$ matched by a given match $M_i \in \mathbf{M}$ as the *multiplicity* of M_i, denoted as $|M_i|$. We refer to each matching region of M_i as a *component* of M_i. Note that $|M_i| \geq 2 \; \forall \; M \in \mathbf{M}$. We denote the left-end coordinates in $\mathcal{S}$ of each component of M_i as $M_i.L_1, M_i.L_2, \ldots, M_i.L_{|M_i|}$, and similarly we denote the right-end coordinates as $M_i.R_x$. When aligning DNA sequences, matches may occur on the forward or reverse complement strands. To account for this phenomenon we add an orientation value to each matching region: $M_i.O_x \in \{1, -1\}$, where 1 indicates a forward strand match and -1 for reverse.

Our algorithm has an important limitation on the matches in $\mathbf{M}$: no two matches M_i and M_j may have the same left-end coordinate, e.g. $M_i.L_x \neq M_j.L_y \; \forall \; i, j, x, y$ except for the identity case when $i = j$ and $x = y$. This constraint has been referred to by others as *consistency* and *transitivity* [26] of matches. In the present work we only require consistency and transitivity of matches longer than the seed length, e.g. seed matches may overlap.

3.2 Data Structures

Our algorithm begins with an initialization phase that creates three data structures. The first data structure is a set of *Match Records* for each match $M \in \mathbf{M}$. The *Match Record* stores M, a unique identifier for M, and two items which will be described later in Section 3.3: a set of linked match records, and a *subsuming match pointer*. The linked match records are further subdivided into four classes: a left and right *superset link*, and left and right *subset links*. The *subsuming match pointer* is initially set to a *NULL* value. Figure 3 shows a schematic

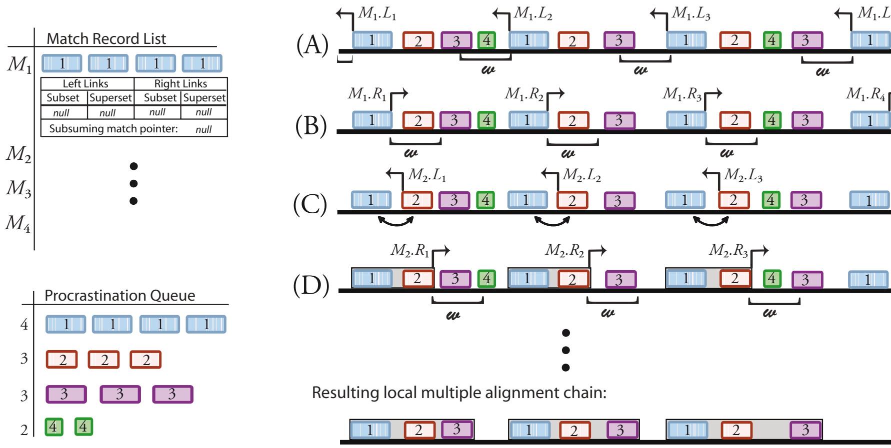

Fig. 3. The match extension process and associated data structures. **(A)** First we pop the match at the front of the procrastination queue: M_1 and begin its leftward extension. Starting with the leftmost position of M_1, we use the *Match Position Lookup Table* to enumerate every match with a left-end within some distance w. Only $M_4.L_1$ is within w of M_1, so it forms a singleton *neighborhood group* which we discard. **(B)** M_1 has no *neighborhood groups* to the left, so we begin extending M_1 to the right. We enumerate all matches within w to the right of M_1. M_2 lies to the right of 3 of 4 components of M_1 and so is not subsumed, but instead gets linked as a right-subset of M_1. We add a left-superset link from M_2 to M_1. **(C)** Once finished with M_1 we pop M_2 from the front of the procrastination queue and begin leftward extension. We find the left-superset link from M_2 to M_1, so we extend the left-end coordinates of M_2 to cover M_1 accordingly. No further leftward extension of M_2 is possible because M_1 has no left-subset links. **(D)** Beginning rightward extension on M_2 we construct a neighborhood list and find a chainable match M_3, and a subset M_4. We extend M_2 to include M_3 and mark M_4 as inconsistent and hence not extendable. Upon completion of the chaining process we have generated a list of local multiple alignments.

of the match record. We refer to the second data structure as a *Match Position Lookup Table*, or **P**. The table has N entries $p_1, p_2, \ldots, p_N$, one per character of $\mathcal{S}$. The entry for p_t stores the unique identifier of the match M_i and x for which $M_i.L_x = t$ or the *NULL* identifier if no match has t as a left-end coordinate. We call the third data structure a *Match extension procrastination queue*, or simply the *procrastination queue*. Again, we denote the multiplicity of a match M by $|M|$. The *procrastination queue* is a binary heap of matches ordered on $|M|$ with higher values of $|M|$ appearing near the top of the heap. The heap is initially populated with all $M \in \mathbf{M}$. This queue dictates the order in which matches will be considered for extension.

3.3 Extending Matches

Armed with the three aforementioned data structures, our algorithm begins the chaining process with the match at the front of the *procrastination queue*. For a match M_i that has not been subsumed, the algorithm first attempts extension to the left, then to the right. Extension in each direction is done separately in an identical manner and we arbitrarily choose to describe leftward extension first. The first step in leftward match extension for M_i is to check whether it has a left superset link. If so, we perform a *link extension* as described later. For extension of M_i without a superset link, we use the *Match Position Lookup Table* $\mathbf{P}$ to enumerate all matches within a fixed distance w of M_i. For each component $x = 1, 2, \ldots, |M_i|$ and distance $d = 1, 2, \ldots, w$ we evaluate first whether $p_{M_i.L_x-(d \cdot M_i.O_x)}$ is not *NULL*. If not then $p_{M_i.L_x-(d \cdot M_i.O_x)}$ stores an entry $\langle M_j, y\rangle$ which is a pointer to neighboring match M_j and the matching component y of M_j.

In order to consider matches on both forward and reverse strands, we must evaluate whether $M_i.O_x$ and $M_j.O_y$ are consistent with each other. We define the relative orientation of $M_i.O_x$ and $M_j.O_y$ as $o_{i,j,x,y} = M_i.O_x \cdot M_j.O_y$ which causes $o_{i,j,x,y} = 1$ if both $M_i.O_x$ and $M_j.O_y$ match the same strand and -1 otherwise. We create a tuple of the form $\langle M_j, o_{i,j,x,y}, x, d, y\rangle$ and add it to a list called the *neighborhood list*. In other words, the tuple stores (1) the unique match ID of the match with a left-end at sequence coordinate $M_i.L_x - (d \cdot M_i.O_x)$, (2) the relative orientation of $M_i.O_x$ and $M_j.O_y$, (3) the matching component x of M_i, (4) the distance d between M_i and M_j, and (5) the matching component y of M_j. If $M_j = M_i$ for a given value of d, we stop adding *neighborhood list* entries after processing that one. The *neighborhood list* is then scanned to identify groups of entries with the same match ID M_j and relative orientation $o_{i,j,x,y}$. We refer to such groups as *neighborhood groups*. Entries in the same *neighborhood group* that have identical x or y values are considered "ties" and need to be broken. Ties are resolved by discarding the entry with the larger value of d in the fourth tuple element: we prefer to chain over shorter distances. After tiebraking, each *neighborhood group* falls into one of several categories:

- **Superset:** The *neighborhood group* contains $|M_i|$ separate entries. M_j has higher multiplicity than M_i, e.g. $|M_j| > |M_i|$. We refer to M_j as a superset of M_i.
- **Chainable:** The *neighborhood group* contains $|M_i|$ separate entries. M_j and M_i have equal multiplicity, e.g. $|M_j| = |M_i|$. We can chain M_j and M_i.
- **Subset:** The *neighborhood group* contains $|M_j|$ separate entries such that $|M_j| < |M_i|$. We refer to M_j as a subset of M_i.
- **Novel Subset:** The *neighborhood group* contains r separate entries such that $r < |M_i| \wedge r < |M_j|$. We refer to the portion of M_j in the list as a *novel subset* of M_i and M_j because this combination of matching positions does not exist as a match in the initial set of matches $\mathbf{M}$.

The algorithm considers each *neighborhood group* for chaining in the order given above: chainable, subset, and finally, novel subset. Superset groups are ignored, as any superset links would have already been created when processing the superset match.

Chainable matches. To chain match M_i with *chainable* match M_j we first update the left-end coordinates of M_i by assigning $M_i.L_x \leftarrow \min(M_i.L_x, M_j.L_y)$ for each $\langle i,j,x,y\rangle$ in the *neighborhood group* entries. Similarly, we update the right-end coordinates: $M_i.R_x \leftarrow \max(M_i.R_x, M_j.R_y)$ for each $\langle i,j,x,y\rangle$ in the group. If any of the coordinates in M_i change we make note that a *chainable* match has been chained. We then update the *Match Record* for M_j by setting its *subsuming match pointer* to M_i, indicating that M_j is now invalid and is subsumed by M_i. Any references to M_j in the *Match Position Lookup Table* and elsewhere may be lazily updated to point to M_i as they are encountered. If M_j has a left superset link, the link is inherited by M_i and any remaining neighborhood groups with *chainable* matches are ignored. *Chainable* groups are processed in order of increasing d value so that the nearest *chainable* match with a superset link will be encountered first. A special case exists when $M_i = M_j$. This occurs when M_i represents an inverted repeat within w nucleotides. We never allow M_i to chain with itself.

Subset matches. We defer subset match processing until no more chainable matches exist in the neighborhood of M_i. A subset match M_j is considered to be completely contained by M_i when for all x, y pairs in the *neighborhood group*, $M_i.L_x \leq M_j.L_y \wedge M_j.R_y \leq M_i.R_x$. When subset match M_j is completely contained by M_i, we set the *subsuming match pointer* of M_j to M_i. If the subset match is not contained we create a *link* from M_i to M_j. The subset link is a tuple of the form $\langle M_i, M_j, x_1, x_2, \ldots, x_{|M_j|}\rangle$ where the variables $x_1 \ldots x_{|M_j|}$ are the x values associated with the $y = 1 \ldots |M_j|$ from the *neighborhood list* group entries. The link is added to the left subset links of M_i and we remove any pre-existing right superset link in M_j and replace it with the new link.

Novel subset matches. A novel subset may only be formed when both M_i and M_j have already been maximally extended, otherwise we discard any novel subset matches. When a novel subset exists matches we create a new match record M_{novel} with left- and right-ends equal to the outward boundaries of M_i and M_j. Rather than extend the novel subset match immediately, we *procrastinate* and place the novel subset in the *procrastination queue*. Recall that the novel subset match contains r matching components of M_i and M_j. In constructing M_{novel}, we create links between M_{novel} and each of M_i and M_j such that M_{novel} is a left and a right subset of M_i and M_j, respectively. The links are tuples of the form outlined in the previous section on subset matches.

Occasionally a *neighborhood group* representing a novel subset match may have $M_i = M_j$. This can occur when M_i has two or more components that form a tandem or overlapping repeat. If $M_i.L_x$ has $M_i.L_y$ in its neighborhood, and $M_i.L_y$ has $M_i.L_z$ in its neighborhood, then we refer to $\{x, y, z\}$ as a tandem unit of M_i. A given tandem unit contains between one and $|M_i|$ components of M_i, and the set of tandem units forms a partition on the components of M_i. In this situation we construct a novel subset match record with one component for each tandem unit of M_i. If M_i has only a single tandem unit then we continue without creating a novel subset match record. Figure 4 illustrates how we process tandem repeats.

Fig. 4. Interplay between tandem repeats and novel subset matches. There are two initial seed matches, one black, one gray. The black match has components labelled 1-7, and the neighborhood size w is shown with respect to component 7. As we attempt leftward extension of the black match we discover the gray match in the neighborhood of components 2 and 5 of black. A subset link is created. We also discover that some components of the black match are within each others' neighborhood. We classify the black match as a tandem repeat and construct a novel subset match with one component for each of the four tandem repeat units: $\{1\}, \{2, 3, 4\}, \{5, 6\}, \{7\}$.

After the first round of chaining. If the *neighborhood list* contained one or more chainable groups we enter another round of extending M_i. The extension process repeats starting with either *link extension* or by construction of a new *neighborhood list*. When the boundaries of M_i no longer change, we classify any subset matches as either subsumed or outside of M_i and treat them accordingly. We process novel subsets. Finally, we may begin extension in the opposite (rightward) direction. The rightward extension is accomplished in a similar manner, except that the neighborhood is constructed from $M_i.R_x$ instead of $M_i.L_x$ and d ranges from $-1, -2, \ldots, -w$ and ties are broken in favor of the largest d value. Where left links were previously used, right links are now used and vice-versa.

Chaining the next match. When the first match popped from the *procrastination queue* has been maximally extended, we pop the next match from the *procrastination queue* and consider it for extension. The process repeats until the *procrastination queue* is empty. Prior to extending any match removed from the *procrastination queue*, we check the match's *subsuming match pointer*. If the match has been subsumed extension is unnecessary.

3.4 Link Extension

To be considered for leftward link extension, M_i must have a left superset link to another match, M_j. We first extend the boundaries of M_i to include the region covered by M_j and unlink M_i from M_j. Then each of the left subset links in M_j are examined in turn to identify links that M_i may use for further extension. Recall that the link from M_i to M_j is of the form $\langle M_j, M_i, x_1, \ldots, x_{|M_i|}\rangle$. Likewise, a left subset link from M_j to another match M_k is of the form $\langle M_j, M_k, z_1, \ldots, z_{|M_k|}\rangle$. To evaluate whether M_i may follow a given link in the left subsets of M_j, we take the set intersection of the x and z values for each M_k that is a left subset of M_j. We can classify the results of the set intersection as:

- **Superset:** $\{x_1, \ldots, x_{|M_i|}\} \subset \{z_1, \ldots, z_{|M_k|}\}$ Here M_k links to every component of M_j that is linked by M_i, in addition to others.
- **Chainable:** $\{x_1, \ldots, x_{|M_i|}\} = \{z_1, \ldots, z_{|M_k|}\}$ Here M_k links to the same set of components of M_j that M_i links.

- **Subset:** $\{x_1, \ldots, x_{|M_i|}\} \supset \{z_1, \ldots, z_{|M_k|}\}$ Here M_i links to every component of M_j that is linked by M_k, in addition to others.
- **Novel Subset:** $\{x_1, \ldots, x_{|M_i|}\} \cap \{z_1, \ldots, z_{|M_k|}\} \neq \emptyset$ Here M_k is neither a superset, chainable, nor subset relative to M_i, but the intersection of their components in M_j is non-empty. M_k and M_i form a novel subset.

Left subset links in M_j are processed in the order given above. Supersets are never observed, because M_k would have already unlinked itself from M_j when it was processed (as described momentarily). When M_k is a chainable match, we extend M_i to include the region covered by M_k and set the subsuming match pointer in M_k to point to M_i. We unlink M_k from M_j, and M_i inherits any left superset link that M_k may have. When M_k is a subset of M_i we unlink M_k from M_j and add it to the *deferred subset list* to be processed once M_i has been fully extended. Finally, we never create novel subset matches during link extension because M_k will never be a fully extended match.

If a chainable match was found during leftward link extension, we continue for another round of leftward extension. If not, we switch directions and begin rightward extension.

3.5 Time Complexity

A *neighborhood list* may be constructed at most w times per character of $\mathcal{S}$, and construction uses sorting by key comparison, giving $\mathcal{O}(wN \log wN)$ time and space. Similarly, we spend $\mathcal{O}(wN \log wN)$ time performing link extension. The upper bound on the total number of components in the final set of matches is $\mathcal{O}(wN)$. Thus, the overall time complexity for our filtration algorithm is $\mathcal{O}(wN \log wN)$.

4 Results

We have created a program called `procrastAligner` for Linux, Windows, and Mac OS X that implements the described algorithm. Our open-source implementation is available as C++ source code licensed under the GPL.

We compare the performance of our method in finding Alu repeats in the human genome to an Eulerian path method for local multiple alignment [23]. The focus of our algorithm is efficient filtration, thus we use a scoring metric that evaluates the filtration sensitivity and specificity of the ungapped alignment chains produced by our method. We compute sensitivity as the number of Alu elements hit by a match, out of the total number of Alu elements. We compute specificity as the ratio of match components that hit an Alu to the sum of match multiplicity for all matches that hit an Alu. Thus, we do not penalize our method for finding legitimate repeats that are not in the Alu family.

The comparison between `procrastAligner` and the Eulerian method is necessarily indirect, as each method was designed to solve different (but related) problems. The Eulerian method uses a *de Bruijn* graph for filtration, but goes

Table 2. Performance of `procrastAlign` and the Eulerian path approach on Alu repeats. Rep: total number of Alu elements; Family: number of Alu families; Alu: average Alu length in bp (S.D.); Div: average Alu divergence (S.D.); Sn: sensitivity; Sp: specificity; T: compute time; Sw: palindromic seed weight; w: max gap size. Alus were identified by RepeatMasker [22]. We report data for the fast version of the Eulerian path method as given by Table 1 of [23]. Sensitivity and specificity of `procrastAlign` was computed as described in the text.

Accession	Length	Rep	Family	Alu (bp)	Div, %	Method	Sn %	Sp %	T (s)	Sw	w
AF435921	22 Kb	28	10	261 (69)	15.0 (6.4)	Eulerian	96.3	99.4	1	-	-
						procrast	100	95.9	1	9	27
Z15025	38 Kb	52	13	245 (85)	15.7 (5.7)	Eulerian	98.6	96.7	4	-	-
						procrast	100	82.5	2	9	27
AC034110	167 Kb	87	18	261 (72)	12.2 (5.9)	Eulerian	93.5	95.2	14	-	-
						procrast	100	97.9	3	15	45
AC010145	199 Kb	118	13	277 (55)	15.0 (5.6)	Eulerian	85.2	93.7	32	-	-
						procrast	99.1	99.2	3	15	45
Hs Chr 22	1 Mbp	404	32	252 (79)	15.2 (6.1)	Eulerian	72.4	99.4	85	-	-
						procrast	98.3	97.3	20	15	45

beyond filtration to compute gapped alignments using banded dynamic programming. We report scores for a version of the Eulerian method that computes alignments only on regions identified by its *de Bruijn* filter. The results suggest that by using our filtration method, the sensitivity of the Eulerian path local multiple aligner could be significantly improved. A second important distinction is that our method reports *all* local multiple alignment chains in its allotted runtime, whereas the Eulerian method identifies only a single alignment.

We also test the ability of our method to provide accurate anchors for genome alignment. Using a manually curated alignment of 144 Hepatitis C virus genome sequences [27], we measure the anchoring sensitivity of our method as the fraction of pairwise positions aligned in the correct alignment that are also present in `procrastAligner` chains. We measure positive predictive value as the number of match component pairs that contain correctly aligned positions out of the total number of match component pairs. `procrastAligner` may generate legitimate matches in the repeat regions of a single genome. The PPV score penalizes `procrastAligner` for identifying such legitimate repeats, which subsequent genome alignment would have to disambiguate. Using a seed size of 9 and $w = 27$, `procrastAligner` has a sensitivity of 63% and PPV of 67%.

5 Discussion

We have described an efficient method for local multiple alignment filtration. The chains of ungapped alignments that our filter outputs may serve as direct input to multiple genome alignment algorithms. The test results of our prototype implementation on Alu sequences demonstrate improved sensitivity over *de Bruijn* filtration. A promising avenue of further research will be to couple our filtration method with subsequent refinement using banded dynamic programming.

The alignment scoring scheme we use can rank alignments by information content, however a biological interpretation of the score remains difficult. If a phylogeny and model of evolution for the sequences were known *a priori* then a biologically relevant scoring scheme could be used [28]. Unfortunately, the phylogenetic relationship for arbitrary local alignments is rarely known, especially among repetitive elements or gene families within a single genome and across genomes. It may be possible to use simulation and MCMC methods to score alignments where the phylogeny and model of evolution is unknown *a priori*, but doing so would be computationally prohibitive for our application.

Acknowledgements

AED was supported by NLM Training Grant 5T15LM007359-05. TJT was supported by Spanish Ministry MECD Grant TIN2004-03382 and AGAUR Training Grant FI-IQUC-2005. LZ was supported by AFT Grant 146-000-068-112.

References

1. Ma, B., Tromp, J., and Li, M.: PatternHunter: faster and more sensitive homology search. Bioinformatics **18** (2002) 440–445
2. Brudno, M., and Morgenstern, B.: Fast and sensitive alignment of large genomic sequences. Proc. IEEE CSB'02 (2002) 138–147
3. Noé, L., and Kucherov, G.: Improved hit criteria for DNA local alignment. BMC Bioinformatics **5** (2004)
4. Kahveci, T., Ljosa, V., and Singh, A.K.: Speeding up whole-genome alignment by indexing frequency vectors. Bioinformatics **20** (2004) 2122–2134
5. Choi, P, K., Zeng, F., and Zhang, L.: Good spaced seeds for homology search. Bioinformatics **20** (2004) 1053–1059
6. Li, M., Ma, B., and Zhang, L.: Superiority and complexity of the spaced seeds. Proc. SODA'06. (2006) 444–453
7. Sun, Y., and Buhler, J.: Designing multiple simultaneous seeds for DNA similarity search. J. Comput. Biol. **12** (2005) 847–861
8. Xu, J., Brown, D.G., Li, M., and Ma, B.: Optimizing multiple spaced seeds for homology search. Proc. CPM'04 (2004) 47–58
9. Flannick, J., and Batzoglou, S.: Using multiple alignments to improve seeded local alignment algorithms. Nucleic Acids Res. **33** (2005) 4563–4577
10. Li, L., Stoeckert, C.J., and Roos, D.S.: OrthoMCL: identification of ortholog groups for eukaryotic genomes. Genome Res. **13** (2003) 2178–2189
11. Jaffe, D.B., Butler, J., Gnerre, S., Mauceli, E., Lindblad-Toh, K., Mesirov, J.P., Zody, M.C., and Lander, E.S.: Whole-genome sequence assembly for mammalian genomes: Arachne 2. Genome Res. **13** (2003) 91–96
12. Ane, C., and Sanderson, M.: Missing the forest for the trees: phylogenetic compression and its implications for inferring complex evolutionary histories. Syst. Biol. **54** (2005) I311–I317

13. Margulies, M., et al. 55 other authors: Genome sequencing in microfabricated high-density picolitre reactors. Nature **437** (2005) 376–380
14. Darling, A.C.E., Mau, B., Blattner, F.R., and Perna, N.T.: Mauve: multiple alignment of conserved genomic sequence with rearrangements. Genome Res. **14(7)** (2004) 1394–403.
15. Hohl, M., Kurtz, S., and Ohlebusch, E.: Efficient multiple genome alignment. Bioinformatics **18 Suppl 1** (2002) S312–20.
16. Treangen, T., and Messeguer, X.: M-GCAT: Multiple Genome Comparison and Alignment Tool. Submitted (2006)
17. Dewey, C.N., and Pachter, L.: Evolution at the nucleotide level: the problem of multiple whole-genome alignment. Hum. Mol. Genet. **15 Suppl 1** (2006)
18. Sammeth, M., and Heringa, J.: Global multiple-sequence alignment with repeats. Proteins (2006)
19. Raphael, B., Zhi, D., Tang, H., and Pevzner, P.: A novel method for multiple alignment of sequences with repeated and shuffled elements. Genome Res. **14(11)** (2004) 2336–46.
20. Edgar, R.C., and Myers, E.W.: PILER: identification and classification of genomic repeats. Bioinformatics **21 Suppl 1** (2005)
21. Kurtz, S., Ohlebusch, E., Schleiermacher, C., Stoye, J., and Giegerich, R.: Computation and visualization of degenerate repeats in complete genomes. Proc. 8th Intell. Syst. Mol. Biol. ISMB'00 (2000) 228–38.
22. Jurka, J., Kapitonov, V.V., Pavlicek, A., Klonowski, P., Kohany, O., and Walichiewicz, J.: Repbase Update, a database of eukaryotic repetitive elements. Cytogenet. Genome Res. **110** (2005) 462–467
23. Zhang, Y., and Waterman, M.S.: An Eulerian path approach to local multiple alignment for DNA sequences. PNAS **102** (2005) 1285–90.
24. Siddharthan, R., Siggia, E.D., and van Nimwegen, E.: PhyloGibbs: a Gibbs sampling motif finder that incorporates phylogeny. PLoS Comput. Biol. **1** (2005)
25. Nagarajan, N., Jones, N., and Keich, U.: Computing the P-value of the information content from an alignment of multiple sequences. Bioinformatics **21 Suppl 1** (2005)
26. Szklarczyk, R., and Heringa, J.: Tracking repeats using significance and transitivity. Bioinformatics **20 Suppl 1** (2004) I311–I317
27. Kuiken, C., Yusim, K., Boykin, L., and Richardson, R.: The Los Alamos hepatitis C sequence database. Bioinformatics **21** (2005) 379–84
28. Prakash, A., and Tompa, M.: Statistics of local multiple alignments. Bioinformatics **21** (2005) i344–i350