Efficient Algorithms and Software for Detection of Full-length LTR Retrotransposons

Anantharaman Kalyanaraman, Srinivas Aluru
Department of Electrical and Computer Engineering,
Iowa State University, Ames, IA 50011
ananthk,aluru@iastate.edu

Abstract

LTR retrotransposons constitute one of the most abundant classes of repetitive elements in eukaryotic genomes. In this paper, we present a new algorithm for detection of full-length LTR retrotransposons in genomic sequences. The algorithm identifies regions in a genomic sequence that show structural characteristics of LTR retrotransposons. Three key components distinguish our algorithm from that of current software — (i) a novel method that preprocesses the entire genomic sequence in linear time and produces high quality pairs of LTR candidates in running time that is constant per pair, (ii) a thorough alignment-based evaluation of candidate pairs to ensure high quality prediction, and (iii) a robust parameter set encompassing both structural constraints and quality controls providing users with a high degree of flexibility. Validation of both our serial and parallel implementations of the algorithm against the yeast genome indicates both superior quality and performance results when compared to existing software.

1. Introduction

Retrotransposons are DNA sequences that reside within cells of a host organism, copying and inserting themselves into the host genome. Studies have revealed their ubiquity in many eukaryotic organisms, both plants and animals — they constitute up to 50% of the maize genome [25, 26], 90% of the wheat genome [5] and 40% of the human genome [27]. *LTR retrotransposons* form a special class of retroelements that are typically characterized by two long terminal near-identical repeat sequences, one at the $5'$ end and the other at the $3'$ end of the inserted retrotransposon; these terminal repeats are referred to as *Long Terminal Repeats* or *LTRs*.

Understanding the behavior of retrotransposons has led to significant advances in molecular genetics and functional genomics [3, 8, 21, 26]. Because of their mobile nature, retrotransposons play a key role in genomic rearrangements and evolution [8, 13]. The transposition mechanism by which retrotransposons copy and relocate involves an RNA-intermediary — a copy of the retrotransposon is made into an RNA molecule, which is then inserted back as a DNA molecule in another location of the host genome, with the aid of a "reverse transcriptase" enzyme. This mechanism being highly similar to the transposition mechanism of retroviruses such as the *HIV* has contributed to a sustained interest in retrotransposon research [2, 4]. Full-length LTR retrotransposons in particular, can be used to provide significant insights into species evolution because of the following property: the two LTRs of a retrotransposon are completely identical when the retrotransposon inserts itself, but can undergo mutations and become increasingly divergent with time [24, 29]. These reasons and others have been a driving force for continued research in LTR retrotransposons. Also with the continued advancement in sequencing technology and with various new large-scale genome sequencing projects of many complex eukaryotic organisms either currently underway or finished, understanding retrotransposons and their biological role in all these newly sequenced genomes has become imperative in furthering research in functional and molecular genomics.

In this paper, we propose an efficient algorithm for *de novo* prediction of full-length LTR retrotransposons with key emphasis on performance and quality. The main contributions of this research are the following:

- an efficient algorithm for quickly generating "high-quality candidates" that drastically reduce the search space. The algorithm has a total run-time complexity that is proportional to the size of the input plus the number of candidates (i.e., constant time per candidate);
- a thorough alignment-based evaluation of candidates using standard dynamic programming techniques [23] that guarantees optimality in the alignment score;

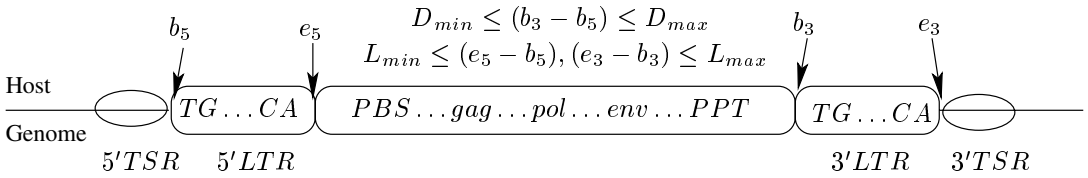

Figure 1. Structural characterization of a typical full-length LTR retrotransposon.

- Support for a robust parameter set encompassing both structural constraints and quality controls; and
- an implementation of our algorithm that can run on both serial and parallel computers.

Preliminary validations indicate that our software produces better quality results than currently available software with significantly faster run-times. For example, our software took 25 minutes on the yeast genome and made better predictions than *LTR_STRUC* [19], which took 210 minutes despite not using rigorous alignments. Furthermore, the parallel implementation of our algorithm can be used to further reduce the run-time proportional to the number of processors used. Our software also provides a flexible framework to incorporate more LTR-specific improvements with minimal changes to the algorithmic core.

2. Problem Description and Related Work

The typical structure of a full-length LTR retrotransposon has been well characterized in the literature. For computational purposes, these structural attributes can be modeled as follows (see Figure 1 for an illustration):

- **Similarity Constraint:** There are two sequences, $5'$ *LTR* and $3'$ *LTR*, which show a good sequence homology as demonstrated by an alignment between them.
- **Distance Constraint:** The number of nucleotides separating the starting positions of the two LTRs is bounded by a minimum of D_{min} and a maximum of D_{max}.
- **TSR:** On the host genome, there may exist an exact matching repeat of length 5 or 6 nucleotides immediately preceding and following the $5'$ and $3'$ LTRs respectively; these repeats are commonly referred to as the *target site repeats* or *TSRs*.
- **LTR motif:** Most LTR sequences start and end with a conserved motif such as $TG \ldots CA$.
- **Other Signals:** The region between the $5'$ and the $3'$ LTR sequences is characterized by a series of special purpose genes and sequences: *primer binding site* or *PBS*, *gag*, *pol*, *env*, and *poly-purine tract* or *PPT*.

Because of the strong homology between $5'$ and $3'$ LTRs, they are expected to contain long exact matches. Thus, identification of exact matching repeats serves as a good starting point for LTR retrotransposon detection. Repeat detection is a well studied problem and a number of excellent programs are already available. These include *RepeatMasker* [28], *REPuter* [15, 16] and *RECON* [1]. LTR retrotransposons, on the other hand, are uniquely characterized by distance constraints. Therefore, the repeats identified by general purpose repeat identification software must be screened to eliminate repeats that do not satisfy the distance constraint. For instance, the *SMaRTFinder* program [22] designed for retrotransposon detection utilizes *REPuter* for repeat detection prior to screening for additional LTR features. The problem with this approach is the run-time incurred in generating large numbers of repeats which could not be LTRs due to distance constraints. For example, if retrotransposons sharing a common LTR are abundant in a genome, generic repeat finding programs must generate all pairs of these LTRs.

A more efficient solution is to build software that is specifically designed for LTR retrotransposon detection, and *LTR_STRUC* [19] is the only available program that is so designed. It has been used for detection of full-length LTR retrotransposons in *Oryza sativa* [18], *Mus musculus* [20] and *Drosophila melanogaster* [6]. The underlying algorithm, however, is a brute-force approach that results in unnecessarily long run-times, which could be problematic for large genomic sequences. A more efficient algorithm will significantly reduce the cost of identifying potential LTR pairs, and the resulting time savings could be utilized to improve prediction quality.

The underlying algorithm in *LTR_STRUC* can be viewed as a two-step procedure: (i) detect all pairs of genomic locations that both satisfy the distance constraint and are starting positions of two "highly similar" (say 70% identity) substrings (or "seeds") of a particular fixed length ω (say 40 nucleotides) — each such pair can be considered a "candidate pair"; (ii) for each candidate pair generated, extend the seeds in either direction as long as the alignment continues to satisfy the similarity constraint. The resulting aligning regions are reported as a full-length LTR retrotransposon. Alignment of an extension is computed by a simple greedy

strategy that aligns longer exact matches before aligning the remainder of the region with shorter matches. This method does not guarantee a best possible alignment of the predicted LTRs, and therefore has the potential danger of missing some LTR pairs. Ideally, an alignment method that computes a combinatorially optimal alignment score is desirable to ensure that no such genuine LTR pairs are missed.

Candidate pairs are generated by the following brute-force approach: Let s denote the input genomic sequence of length n. Walk along s and for each position i, $1 \leq i \leq n$, scan all positions j, $(i + D_{min}) \leq j \leq (i + D_{max})$. For each (i, j)-pair, compute the percentage identity of the two $\omega-$length substrings starting at i and j. If the identity is above the similarity threshold (say 70%), then the pair (i, j) is reported as a "candidate pair" and is further evaluated for alignment as described above. It can be easily seen that the algorithm has a worst-case run-time complexity of $O(n \times (D_{max} - D_{min}) \times \omega)$. In practice, D_{max} could be as high as 10,000 - 15,000 and D_{min} could be as low as 100. In an attempt to save run-time, the algorithm's implementation resorts to a technique of sampling the search interval i.e., the value of i is incremented by some $\Delta i > 1$ instead of 1. This would reduce the run-time cost by a factor of Δi, but at the expense of prediction accuracy. Moreover, this algorithm will consider many redundant or "duplicate" pairs of locations corresponding to the same matching pair of regions. To see this, note that if a $5'$-$3'$ LTR pair share a long exact match of length $l > \omega$ nucleotides, then there are $(l - \omega + 1)$ pairs of $\omega-$length identical substrings and the algorithm will generate all these pairs of locations even though they correspond to the same longer exact match. Ideally, generation of such"duplicates" pairs should be completely avoided in the interest of run-time. Also note that the run-time complexity is independent of the repetitive nature of the genome, i.e., while at long stretches of the genome that have no LTRs, this algorithm would search for an entire $(D_{max} - D_{min})$-length interval only to result in more wasted effort.

In a pilot study on a Windows machine with 1 GHz Pentium III processor conducted by one of our colleagues [7], *LTR_STRUC* took 3.5 hours on the entire yeast genome (over 12 Mbp) and over 15 hours on chromosome 1 of *Arabidopsis thaliana* genome (over 30 Mbp). These high run-times are likely to be a major limiting factor in the applicability of the *LTR_STRUC* software on larger genomes such as the human, maize, etc., mainly because a biologist would like to run a *de novo* prediction tool such as *LTR_STRUC* multiple times for different parameter settings, before arriving at a high-quality repository of predictions.

3. Our Approach

Let s denote the input DNA sequence comprising of n nucleotides. For computational purposes, we view s as a string of n characters in alphabet $\Sigma = \{A, C, G, T, N\}$, where '$N$' may denote either a low-quality or masked base in the input sequence. Let $s[i]$ denote the character at position i in s ($1 \leq i \leq n$). Let $s[i..j]$ denote the substring $s[i]s[i+1]\ldots s[j]$. Let $left(i) = s[i-1]$, if $i > 1$, and 'N' otherwise; similarly, let $right(i) = s[i+1]$, if $i < n$, and 'N' otherwise. Two identical substrings $s[i_1..(i_1 + k)] = s[i_2..(i_2+k)]$ are said to be *left-maximal* (respectively *right-maximal*) if and only if $left(i_1) \neq left(i_2)$ (respectively $right(i_1 + k) \neq right(i_2 + k)$). They are said to be a *maximal matching pair* if they are both right- and left-maximal. We will assume that aligning 'N' with any other character should be treated as a mismatch.

The main idea of our approach is to have an efficient linear time preprocessing of the entire input sequence, followed by an algorithm that provides a direct mechanism (as opposed to a searching mechanism) for generation of "candidate pairs". Our definition of "candidate pairs" is based on maximal matches subject to LTR retrotransposon length constraints. While the generation of maximal matches is well studied in literature using suffix trees [9] our pair generation algorithm follows a related strategy using suffix arrays [17] that takes into account the distance constraints while ensuring that each candidate pair is generated exactly once without any duplicates. Each candidate pair is then subjected to a rigorous alignment test that guarantees an alignment with the combinatorially best score for testing against the similarity constraint.

3.1. The Sequential Algorithm

Let L_{min} (L_{max}) denote the minimum (maximum) allowed length of an LTR (as shown in Figure 1). Let L_{ex} denote a length such that any $5'$-$3'$ pair of LTRs will share at least one exact match of that length. (This user-specified parameter can even be analytically computed as follows: if ψ, the rate of mutation (as a fraction) in the host genome is known, a reasonable value can be $\frac{L_{min}}{(\psi \times L_{min})+1}$.)

Definition 1 Candidate Pair: *A pair of genomic positions (i_1, i_2) ($1 \leq i_1, i_2 \leq n$) is defined as a candidate pair if and only if it satisfies the following properties:*

- *the positions satisfy the distance constraint, i.e.,* $(i_1 + D_{min}) \leq i_2 \leq (i_1 + D_{max})$.
- *the substrings* $(s[i_1..(i_1 + L_{ex} - 1)]$ *and* $s[i_2..(i_2 + L_{ex} - 1)])$ *are left-maximal.*

Note that there is a one-to-one correspondence between the set of maximal matching pairs of minimum length L_{ex} and

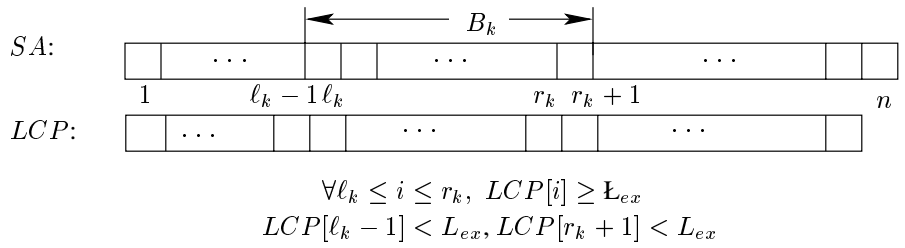

Figure 2. Illustration of the process of creation of buckets in our preprocessing algorithm.

left-maximal pairs of length L_{ex}. Our algorithm comprises of three phases: a preprocessing phase, a candidate pair generation phase, and an alignment phase.

3.1.1 Preprocessing

The goal of preprocessing is to achieve a two-level partition of positions $\{1, 2, \ldots, n\}$ in s — the first level is based on the $L_{ex}-$length substring that starts at each position, and the second level is based on the character that is preceding each position. The preprocessing algorithm is as follows: construct a suffix array (denoted by SA) data structure [17] on s in linear time [10, 12, 14] and also the corresponding longest common prefix array (denoted by LCP) [11]. As a result, $SA[i]$ is the i^{th} lexicographically smallest suffix in s ($\forall 1 \leq i \leq n$), and $LCP[i]$ is the length of the longest common prefix between suffixes $SA[i]$ and $SA[i+1]$ ($\forall 1 \leq i \leq n-1$). Next, a set $B = \{B_1, B_2, \ldots, B_m\}$ of m buckets is generated such that $\forall i, j \in B_k$, $\forall 1 \leq k \leq m$, $s[i..(i + L_{ex} - 1)] = s[j..(j + L_{ex} - 1)]$. This is achieved by linearly scanning the $LCP[1..n-1]$ array and recording all maximal intervals in which the LCP values are all greater than or equal to L_{ex}. The value of m is therefore the number of such maximal intervals. For each maximal interval the set of all suffix entries that it covers in the array $SA[1..n]$ is then assigned to a unique bucket in B. (See Figure 2 for an illustration.) Since every LCP entry covers two consecutive suffix entries in SA, each resulting bucket contains at least two suffix entries, i.e., $0 \leq m \leq \lfloor \frac{n}{2} \rfloor$. Choosing maximal intervals in the LCP array with values $\geq L_{ex}$ ensures that $\forall 1 \leq k \leq m$, $\forall i \in B_k$, all substrings $s[i..(i + L_{ex} - 1)]$ are identical. Next, each bucket is sorted in the ascending order of the position numbers. This is done once for all buckets through a stable integer sort. Each bucket B_k is then further partitioned into $|\Sigma|$ ordered sets called $Lsets$: $\forall c \in \Sigma$, $Lset^k_c = \{i \mid left(i) = c, i \in B_k\}$. It is easy to see that one can partition every B_k into these individual $Lsets$ still maintaining the internal sorted order within each $Lset$. Maintaining the sorted order is critical for efficient generation of candidate pairs, as will soon become evident.

3.1.2 Candidate Pair Generation

Once the input sequence is preprocessed, candidate pairs can be generated from within each bucket. The algorithm for generating candidate pairs is presented in Figure 3 and an illustration to help understand the algorithm is provided in Figure 4.

For each bucket B_k, all $Lsets$ are scanned in the ascending order of the position number. A position i in $Lset^k_{c_1}$ is paired with a position j if and only if $j \in Lset^k_{c_2}$ such that $c_2 \neq c_1$ or $c_2 =$'N' (i.e., the substrings $s[i..(i + L_{ex} - 1)]$ and $s[j..(j + L_{ex} - 1)]$ are left-maximal), and $(i + D_{min}) \leq j \leq (i + D_{max})$ (i.e., the pair (i, j) satisfies the distance constraints). This guarantees that a pair (i, j) is generated only if it is a candidate pair by Definition 1. Enumerating all j (and only those j) that should be paired with i is achieved in the following manner. Since each $Lset$ is internally sorted by position numbers, the entries for j for a given value of i are placed consecutively in $Lset^k_{c_2}$, defined by a range say, $[b_i, \ldots, e_i]$. If i is the first entry of the ordered set $Lset^k_{c_1}$, b_i can be located in $Lset^k_{c_2}$ by performing a linear scan until a value that is $\geq (i + D_{min})$ is encountered. Once b_i is located one can continue pairing i with all subsequent elements from b_i in $Lset^k_{c_2}$ until $(i + D_{max})$ is exceeded or the $Lset$ is exhausted. The last element to be paired is e_i. Henceforth, in advancing each i to its next position say i' in $Lset^k_{c_1}$, it is sufficient to start searching for $b_{i'}$ from b_i onwards, since $b_{i'} \geq b_i$ as $i' > i$. Even better, one can record the position of $b_{i'}$ if found before e_i, while generating pairs for i, and directly start from $b_{i'}$ for i'.

Since the algorithm ensures all entries in each bucket are considered for i and that for each such i, all candidates for j from the same bucket are considered, it can be seen that our candidate pair generation does not miss any candidate pair by Definition 1. Moreover, since each entry in a bucket is considered for i exactly once it is also easy to see that each candidate pair is generated exactly once.

Lemma 1 *Let $s[i_1..(i_1 + k - 1)]$ and $s[i_2..(i_2 + k - 1)]$ be two maximal matching substrings, for some $k \geq L_{ex}$, and $(i_1 + D_{min}) \leq i_2 \leq (i_1 + D_{max})$. Then (i_1, i_2) is generated exactly once.*

Algorithm 1 *Candidate Pair Generation*

Input: Bucket B_k
L_1: FOR EACH $c_1 \in \Sigma$ DO
L_2: FOR EACH $i \in Lset^k_{c_1}$ DO
L_3: FOR EACH $c_2 \in \Sigma$ and ($c_1 \neq c_2$ or $c_1 = c_2 =$ 'N') DO
S_1: $b_i \leftarrow min\{j | j \in Lset^k_{c_2}, D_{min} \leq (j-i) \leq D_{max}\}$
S_2: $e_i \leftarrow max\{j | j \in Lset^k_{c_2}, D_{min} \leq (j-i) \leq D_{max}\}$
S_3: Generate pairs $(i, j), \forall j \in Lset^k_{c_2}, b_i \leq j \leq e_i$

Figure 3. Algorithm to generate candidate pairs from a given bucket B_k.

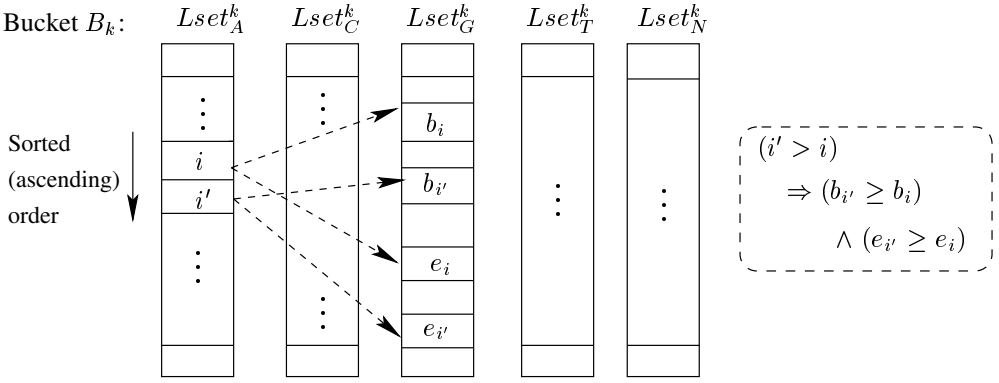

Figure 4. Illustration of the candidate pair generation algorithm. Shown are the five $Lsets$ for a given bucket B_k. The entry $i \in Lset^k_A$ is paired with all entries satisfying the distance constraint, denoted by the interval $[b_i \ldots e_i]$, in $Lset^k_G$. For the next entry $i' \in Lset^k_A$, the corresponding interval $[b_{i'} \ldots e_{i'}]$ is such that $b_{i'} \geq b_i$ and $e_{i'} \geq e_i$.

Proof: If $s[i_1..(i_1 + k - 1)]$ and $s[i_2..(i_2 + k - 1)]$ are two maximal matching substrings of length $\geq L_{ex}$ then $s[i_1..(i_1 + L_{ex} - 1)] = s[i_2..(i_2 + L_{ex} - 1)]$ and they are left-maximal. Therefore (i_1, i_2) is a candidate pair by Definition 1. Also since we only generate left-maximal pairs based on fixed-length (L_{ex}) matches, only one candidate pair per maximal pair gets generated. ■

3.1.3 Run-time Analysis

For the preprocessing phase, the construction of suffix array [10, 12, 14] and LCP array [11] takes $O(n)$ time. Generating the set of buckets also takes $O(n)$ time because the algorithm does a linear scan of the arrays. Ordering within each bucket by position numbers and then generating all $Lsets$ for all buckets are integer sorting operations. The outermost loops, L_1 and L_2 of Algorithm 1, over all iterations visits each position $\{1, \ldots, n\}$ at most once, although in an arbitrary order. Step S_3 coupled with Lemma 1 imply a run-time directly proportional to the number of candidate pairs generated. For steps S_1 and S_2, note that at worst case, locating b_i may take $O(n)$. However, the amortized worst case is still $O(n)$ because each entry is considered exactly once for choice of i and at most $2 \times |\Sigma|$ times for j, implying a run-time cost of $O((2 \times |\Sigma| + 1) \times n) = O(n)$, (taking $|\Sigma| = 5$ to be a negligible constant). Thus the candidate pair generation algorithm has an optimal run-time, i.e., $O(n)$ plus the number of candidates pairs in s.

3.1.4 Alignment and LTR Prediction

Once a candidate pair is reported, the regions flanking the corresponding match are aligned and evaluated to check if the aggregate region indeed has an expected LTR structure. For a candidate pair (i, j), two pairs of substrings all of length $(L_{max} - L_{ex})$ are extracted — the "pair-to-the-left" is $s[i-(L_{max}-L_{ex})..i-1]$ and $s[j-(L_{max}-L_{ex})..j-1]$, and the "pair-to-the-right" is $s[i + L_{ex}..i + L_{max} - 1]$ and

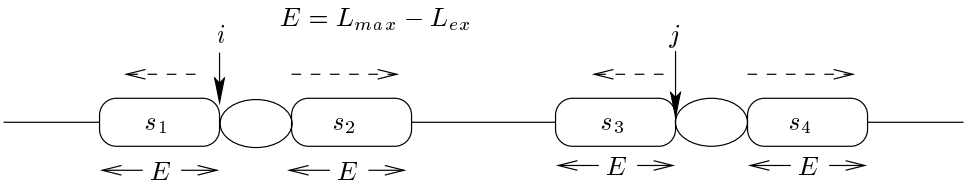

Figure 5. Two alignments are performed for each candidate pair (i, j): (s_1, s_3) and (s_2, s_4). The alignment directions are denoted by dotted arrows.

$s[j + L_{ex}..j + L_{max} - 1]$. An optimal alignment algorithm with both affine- and end-gap penalties using banded dynamic programming techniques is computed for each of these two pairs. While the strings of pair-to-the-right are directly aligned, the strings of pair-to-the-left are first reversed and then aligned. An aggregate score is then computed by adding the scores of the best aligning prefixes of the pair-to-the-right and that of the reversed pair-to-the-left, and the matching score of the anchored match in the middle. Special care is taken that the length of the overall aligning region respects L_{max} and L_{min}, and that the new two starting positions of the two aligning regionsstill satisfy the distance constraint. If the score satisfies the similarity constraint, the boundaries of the two aligning regions is reported as a *predicted pair of LTRs*. For an illustration of the alignment step refer to Figure 5.

The above outlined alignment method is extended to accommodate for other structural LTR attributes such as the TSRs, LTR motifs, and other signals. TSRs are detected by looking for an exact match of length 5-6 nucleotides immediately left and right of the alignment boundaries and adjusting the boundaries if necessary. Similarly, boundaries are adjusted depending on the presence of motifs as well. Depending on whether TSRs and/or LTR motifs were found, a "confidence level" is computed and reported as part of the predicted LTR region. The confidence level is computed as follows: $W_{TSR} * TSR_{code} + W_{motif} * Motif_{code}$, where $0 \leq W_{TSR}, W_{motif} \leq 1$ are weights assigned by the user as a means to specify how important finding a TSR and the motifs are. $TSR_{code} = 1$, if the two predicted TSRs are identical, and 0 otherwise. $Motif_{code} = 1$, if both $5'$ and $3'$ LTRs start and end with TG and CA respectively, 0.5 if only one motif is found, and 0 otherwise. Other structural attributes such as PBS, *gag*, *pol*, *env*, PPT, etc., can also be incorporated into the algorithm framework. Our current implementation accounts only for TSRs and motifs, and these other attributes are planned future work.

3.2. Parallelization

Our algorithm can be parallelized as follows: The input sequence can be distributed evenly across processors; to ensure that no pairs are missed, the last $D_{max} - 1$ characters in each processor are duplicated in the next processor, resulting in roughly $\frac{n}{p} + D_{max}$ characters per processor. Once distributed, the serial algorithm can be run in each processor and results reported without needing any further communication. The speedup of the preprocessing phase is proportional to $\frac{n}{\frac{n}{p}+D_{max}}$, implying that as long as $D_{max} << \frac{n}{p}$, a linear speedup can be achieved. The speedup of the candidate pair generation and alignment phases are dependent on the input, and the distribution of the repetitive elements among processors. Although one can think of dynamic load-balancing strategies during the alignment phase, our current implementation does not provide such features.

4. Results and Discussion

Validation of our software was performed by running the program on the entire yeast genome and comparing the results against a "benchmark" of known LTR retrotransposon locations ([13], see the website http://www.public.iastate.edu/~voytas for more details). The list of parameters and values input to our software is shown in Table 1. The yeast genome has 16 chromosomes, and the benchmark has a total of 50 known full-length LTR retrotransposons.

Our software predicted 46 out of the 50 LTR pairs. Of the remaining 4, three were predicted on decreasing the similarity threshold from 70% to 60%. The remaining one LTR pair was missed out from the prediction set because $5'$ and $3'$ LTRs were significantly different in their lengths (140 bp and 334 bp). Of the 46 predicted LTR pairs, 41 were predicted accurately — with 38 showing a confidence level of 1 as both the $TG \ldots CA$ motif and identical TSRs, and the other 3 LTR pairs were reported with a lower confidence because all of them lacked identical $5'$ and $3'$ TSRs (consistent with the benchmark). There were a total of 5 LTR

Table 1. Parameter set for our program with default values.

Parameter Name	Default Value	Comment
D_{min},D_{max}	100, 15,000	Distance constraint for $5'$-$3'$ LTR pair
L_{min}, L_{max}	100, 1,000	Length constraints for $5'$-$3'$ LTR pair
L_{ex}	20	Exact match length requirement for $5'$-$3'$ LTR pair
τ	70%	Similarity threshold of a $5'$-$3'$ LTR pair
match	2	Match score
mismatch	-5	Mismatch score
open_gap	6	Gap opening penalty
continuation_gap	1	Gap continuation penalty
w_{TSR}	0.5	Weight for presence/absence of TSR
w_{TGCA}	0.5	Weight for presence/absence of the motif $TG \ldots CA$

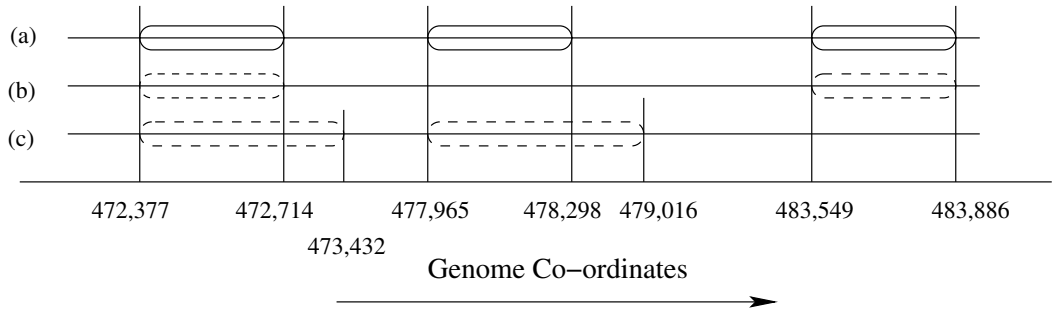

Figure 6. A case of nested retrotransposons in chromosome 10 of *S. cerevisiae* with 3 LTRs. The bottom-most line indicates the genome (not to scale). Part (a) shows the benchmark co-ordinates for the LTRs. Parts (b) and (c) show the two predictions made by our software.

pairs that were predicted correctly, but with boundary mis-predictions - the exact boundaries of the $5'$ and $3'$ LTRs were off by a maximum of 27 bases in one of the cases. We found that these boundary mis-predictions were because the scores of the corresponding optimal alignments were better than that of the "biologically-preferred" alignments in the benchmark.

Comparisons of the *LTR_STRUC* predictions on the yeast genome data set against the benchmark were done by one of our colleagues [7]. (A similarity threshold of 70% was used for direct comparison with our results.) The results are as follows: of the 50 LTR pairs, only 40 were predicted by *LTR_STRUC*. The program missed three cases where the $5'$ and $3'$ TSRs are not identical. (Our software detected these cases with an appropriate lower confidence.) We could not ascertain the reason(s) for missing of the remaining 7 LTR pairs by *LTR_STRUC*. Our speculations are that the program either failed to generate candidate pairs because of jumping by Δi characters as a means to save run-time or that an alignment that was inferior to a best alignment was computed on aligning them.

There was also a case of "nested" retrotransposon in the benchmark data set. This is present in chromosome 10, elements labeled *YJRWdelta11 (Ty1-1), YJRWdelta12 (Ty1-1/Ty1-2) and JRWdelta13 (Ty1-2)* in the webpage http://www.public.iastate.edu/~voytas. This is a case where one LTR is shared between two full-length retrotransposons (See Figure 6). Our software also predicted two retrotransposons for this case, one of which with consistent boundary and TSR predictions as well.

Besides the yeast genome, we also ran our program on a collection of randomly selected 9 rice BAC sequences, the results for which have already been published by McCarthy *et al.* [18] using *LTR_STRUC*. Both the programs detected 8 full-length LTR retrotransposons in common. However, our software detected 4 more confident predictions which were absent in the *LTR_STRUC* prediction. On the other hand, *LTR_STRUC* predicted 2 LTRs which were not predicted by our software that look like solo-LTR elements (and not full-length).

As for run-time on the yeast genome (over 12 Mbp), *LTR_STRUC* took about 210 minutes on a Windows 1 GHz

Table 2. Run-time results of our software.

Organism	Genome Size (in bp)	Number of processors	Total Time (in minutes)
Saccharomyces cerevisiae	12,070,811	8	3.5
Arabidopsis thaliana	119,186,497	32	67
Drosophila melanogaster	118,357,599	32	41

machine, while our software took 25 minutes on a Linux single processor 1.1 GHz machine. While our software spends much less time for candidate pair generation algorithm than *LTR_STRUC*, it spends more time in the alignment phase simply because it does more work to guarantee optimality. For example, on the yeast genome, our software spent only 8% of the time in preprocessing and generating pairs, while the remaining 92% was spent in aligning the LTR candidates. This extra effort spent in a ensuring a thorough alignment is what has resulted in a better prediction accuracy of our software when compared to *LTR_STRUC*, as was seen in the above validation studies. More run-time analysis were performed on our software on a Linux cluster of 16 nodes, each with 2 Intel Xeon 3.06 GHz processors and 2 GB RAM, the results of which are shown in Table 2.

The results of validating our software are encouraging. The prediction accuracy over the validated set of yeast genome is better than that of the *LTR_STRUC* because our software can more efficiently accommodate for mutations in LTR and TSR regions. Moreover the software offers better flexibility and provides user with a better control — the user can assign weights to the presence/absence of TSRs and $TG\ldots CA$ motifs, and the software can output its predictions with associated confidence levels reflecting the weights specified by the user. Also, if a user is searching a newly sequenced genome for LTR retroelements, the user can try different combinations of weights and scoring parameters and observe changes in the output before deciding on a correct set of parameters. The speed of our software plays a critical role in facilitating multiple runs and experiments using different parameter settings. Cases that correspond to multiple nested LTR retrotransposon insertions can be detected by running our software iteratively on the genome, and excising out all full-length elements from the genome found in a previous iteration.

In its current state, our software provides an efficient mechanism to predict pairs of genomic regions that bear structural semblance to be part of a full-length LTR retrotransposon. There are, however, various planned functional improvements that are essential to ensure a high prediction accuracy. The genomic region between a pair of LTR sequences typically contains other special signals such as PPT, PBS, *gag*, *pol*, and *env*, and detection of these signals is important in confirming the biological identity of each prediction. PPT can be detected by searching for a purine rich region immediately $5'$ of the predicted $3'$ LTR boundary. For detection of PBS, a short sequence or a collection of short sequences corresponding to the tRNA priming sequence can be input by the user, and the software can search for a complementary sequence immediately $3'$ of the $5'$ LTR predicted sequence. The genes *gag* and *pol* can be detected by looking for corresponding coding sequences. Incorporating these features into the software and validating it for false predictions against known LTR retroelements is essential before applying it to discover previously unknown full-length retroelements.

5. Conclusion

In this paper, we provided efficient algorithms and software towards detection of full-length LTR transposons. The salient features of our method includes: a quick and efficient method of generating candidate LTR pairs, which allows for a rigorous method to align the candidates as a means to guarantee high quality LTR predictions. The software has been designed with the intent of giving a high degree of flexibility to the user. There are numerous planned functional improvements to the software, such as incorporation of detection strategies for PPT, PBS, *gag, pol* genes in the structure finding procedure, detection of nested retrotransposons, etc. Due to the ubiquity of LTR retroelements in large scale genomes, the utility of a highly accurate and yet fast and scalable LTR discovery tool is key to the advancement of biological understanding of these genomes.

6. Acknowledgments

We are grateful to Xiaowu Gai for introducing us to the retrotransposon identification problem, for providing some useful biological insights during the validation of our software, and for sharing the results of running *LTR_STRUC* on yeast and *Arabidopsis*. We thank Daniel Voytas and Scott Emrich for comments on earlier drafts of our paper. We also thank the *LTR_STRUC* team for providing the tool. Kalyanaraman was supported by an IBM Ph.D. Research Fellow-

ship. This research was partially supported by CREST grant HRD-0420407.

References

[1] Z. Bao and S. Eddy. Automated *de novo* identification of repeat sequence families in sequenced genomes. *Genome Research*, 12:1269–1279, 2002.

[2] F. Bushman. Targeting survival: integration site selection by retroviruses and LTR-retrotransposons. *Cell*, 115:135–138, 2003.

[3] B. Charlesworth, P. Sniegowski, and W. Stephan. The evolutionary dynamics of repetitive DNA in eukaryotes. *Nature*, 371:215–220, 1994.

[4] J. Coffin, S. Hughes, and H. Varmus. Retroviruses. *Plantview*, 1997.

[5] R. Flavell. Repetitive DNA and chromosome evolution in plants. *Philosophical Transactions of the Royal Society of London. B.*, 312:227–242, 1986.

[6] L. Franchini, E. Ganko, and J. McDonald. Retrotransposon-gene associations are wide-spread among *D.melanogaster* populations. *Molecular Biology and Evolution*, 21:1323–1331, 2004.

[7] X. Gai. *Personal Communication*, 2005.

[8] E. Ganko, V. Bhattacharjee, P. Schliekelman, and J. McDonald. Evidence for the contribution of LTR retrotransposons to *C. elegans* gene evolution. *Molecular Biology and Genetics*, 20:1925–1931, 2003.

[9] D. Gusfield. *Algorithms on strings, trees and sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, London, 1997.

[10] J. Karkkanen and P. Sander. Simpler linear work suffix array construction. In *Proc. International Colloquium on Automata, Languages and Programming*, 2003.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Combinatorial Pattern Matching*, pages 181–192, 2001.

[12] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, pages 186–199, 2003.

[13] J. Kim, S. Vanguri, J. Boeke, A. Gabriel, and D. Voytas. Transposable Elements and Genome Organization: A Comprehensive Survey of Retrotransposons Revealed by the Complete *Saccharomyces cerevisiae* Genome Sequence. *Genome Research*, 8:464–478, 1998.

[14] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, pages 200–210, 2003.

[15] S. Kurtz, J. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29:4633–4642, 2001.

[16] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15:426–427, 1999.

[17] U. Manber and G. Myers. Suffix arrays: A new method for on-line search. *SIAM Journal of Computing*, 22:935–948, 1993.

[18] E. McCarthy, L. Liu, G. Lizhi, and J. McDonald. Long terminal repeat retrotransposons of *oryza sativa*. *Genome Biology*, 3:0053.1–0053.11, 2002.

[19] E. McCarthy and J. McDonald. LTR_STRUC: a novel search and identification program for LTR retrotransposons. *Bioinformatics*, 19:362–367, 2003.

[20] E. McCarthy and J. McDonald. LTR Retrotransposons of *Mus musculus*. *Genome Biology*, 5:R14, 2004.

[21] J. Miller, F. Dong, S. Jackson, J. Song, and J. Jiang. Retrotransposon-related DNA sequences in the centromeres of grass chromosomes. *Genetics*, 150:1615–1623, 1998.

[22] M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo. Automated search for LTR retrotransposons. *http://citeseer.ist.psu.edu/644336.html*, 2002.

[23] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[24] B. Peterson-Burch, D. Nettleton, and D. Voytas. Genomic Neighbourhoods for *Arabidopsis* retrotransposons: a role for targeted integration in the distribution of the Metaviridae. *Genome Biology*, 5:R78, 2004.

[25] P. SanMiguel, B. Gaut, A. Tikhonov, Y. Nakajima, and J. Bennetzen. The paleontology of intergene retrotransposons of maize. *Nature Genetics*, 20:43–45, 1998.

[26] P. SanMiguel, A. Tikhonov, Y. Jin, N. Motchoulskaia, D. Zakharov, A. Melake-Berhan, P. Springer, K. Edwards, M. Lee, Z. Avramova, and J. Bennetzen. Nested retrotransposons in the intergenic regions of the maize genome. *Science*, 274:765–768, 1996.

[27] A. Smit. Interspersed repeats and other mementos of transposable elements in mammalian genomes. *Current Opinions in Genetics and Development*, 9:657–663, 1999.

[28] A. Smit and P. Green. RepeatMasker. *http://ftp.genome.washington.edu/RM/RepeatMask er.html*, 1999.

[29] Y. Xiong and T. Eickbush. Origin and evolution of retroelements based upon their reverse transcriptase sequences. *EMBO Journal*, 9:3353–3362, 1990.