RepFrag: A Graph based Method for Finding Repeats and Transposons from Fragmented Genomes

Nirmalya Bandyopadhyay
CISE Department
University of Florida,
Gainesville, FL 32611, USA
nirmalya@cise.ufl.edu

A. Mark Settles
Plant Molecular and Cellular Biology Program and Horticultural Sciences Department
University of Florida,
Gainesville, FL 32611, USA
settles@ufl.edu

Tamer Kahveci
CISE Department
University of Florida,
Gainesville, FL 32611, USA
tamer@cise.ufl.edu

ABSTRACT

Growing sequencing and assembly efforts have been met by the advances in high throughput machines. However, the presence of massive amounts of repeats and transposons complicates the assembly process. Given a library of possible repeats, this paper considers the problem of identifying repeats and transposons in the fragments (also called reads) generated from sequencing machines. This is a difficult problem as the locations of the fragments on the complete genome are not known. Furthermore, due to insertion, deletion and other evolutionary factors, different copies of repeats can diverge from each other. The presence of transposons, (also called *jumping genes*) makes the problem even harder as they can split other repeats and make them diverge from the actual repeats.

We develop a graph based method named *RepFrag* which can efficiently identify repeats in a given set of fragments. We first align the fragments to the repeats in the given repeat library. We model the alignments as the *vertices* in the graphs. We create *edges* between two vertices if they can jointly express a potential repeat better. We traverse the paths in this graph to find a path that has a high potential of representing a complete repeat. We mask the aligned regions on the fragments corresponding to the vertices on that path. Using the unaligned regions, we create a new fragment and align it with the repeats. We modify the existing graphs based on the new alignments, if there are any. We iterate this process of path selection and modification of graph until no promising path remains.

We compared the performance of our method to that of RepeatMasker on 30 different fragment datasets generated from five different chromosomes of *Arabidopsis Thaliana* for varying coverage values and fragment lengths. On average RepFrag had a 35% better true positive-false positive ratio. RepFrag was 7.3 times faster than RepeatMasker on the average. Thus, our results suggest that, our method improves significantly over RepeatMasker in terms of speed and accuracy.

1. INTRODUCTION

ACM-BCB 2010 Niagara Falls, New York USA

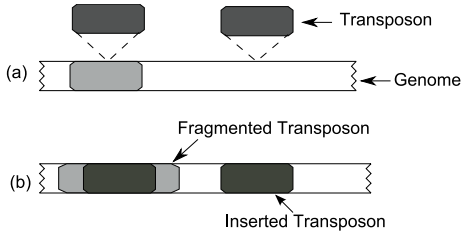


Figure 1: The figures illustrate how transposons penetrate the chromosome. Figure (a) shows that the chromosome already contains one transposon and two other transposons are about to *jump* into the chromosome. Figure (b) depicts the chromosome after that event. Note, that one transposon in dark brown severs the one in light brown.

Availability of high throughput machines has greatly improved the sequencing efforts. Existing sequencing machines can identify the nucleotide sequences of a large number of short fragments, called *reads*. We use the term *fragment* to denote reads in this paper. For example, 3730xl DNA Analyzer can create fragments that are of length roughly 650 bases [1]. Also, 454 sequencing-by-synthesis technology [22] can generate fragments of around 400 bases.

Once the fragments from a genome are obtained, often the next step is to assemble them to produce the complete genome sequence which can be composed of many millions of bases. Assembling the fragments produced by the sequencing machines is a difficult problem due to two reasons. First, a large number of repetitive sequences, also know as repeats, transposons and transposable elements are present in the genome sequence. Second, sequencing machines do not provide information about the location of the fragments on the complete genome. The paper focuses on the first reason.

Repeats and transposons are repetitive sequences. We use the term *repeat* to denote repeats and transposons in this paper. They can occupy a significant portion of the genomes. For example, 45% of the human genome [11] and 50-80% of the maize genome [23] consist of transposable elements. Based on their length and structure they can be classified into different categories. Microsatellites or simple sequence repeats are short repeat sequences consisting

This work was supported partially by NSF under grants CCF-0829867, IIS-0845439 and the UF Research initiatives (Project number: 00072365).

of 1-6 nucleotides. Minisatellites are longer repeat units that are of length 10-60 nucleotides. Another important category is transposons or transposable elements that are dispersed throughout the genome. Transposons are also called the *jumping genes*, as a transposon in one location of the genome can move (cut and paste) or replicate (copy and paste) itself to another location of the genome. A transposon can jump inside another transposon and split it. Figure 1 illustrates the *jumping* of the transposons into the chromosome.

The repetitive nature of the transposable elements complicates sequence assembly. So, most of the assembly methods require a pre-processing of the fragments to identify the repeat sequences. Identifying repeats is a computationally challenging problem because of the following reasons. First, when a transposon moves inside another repeat sequence, it can split that repeat into smaller pieces that diverge from the original repeat. Figure 1 illustrates this process. Further mutations, such as insertion, deletion and replacement can also alter the sequences of a repeat. In this way, the repeat sequence in the library and that on the genome sequence can vary greatly. Thus, it may not be possible to identify them by simply comparing the fragments to a repeat library with a sequence alignment tool. Second, a fragment may not contain an entire repeat. A repeat can spread across several fragments, which makes the identification of the complete unit complicated. Finally, fragments can be very short and numerous. As a result of these challenges, identification of repeats in the fragmented genomes is a complex problem.

In our previous work, we developed a method called *Greedier* that identifies repeats from complete genome sequences [16]. However, with fragmented genomes, which is the more common case, the problem is more difficult as discussed above.

Contributions: *In this paper, we consider the problem of identifying repeats in a given set of fragments.* Assume that we are given a set of fragments $\mathcal{F} = \{f_1, f_2, \ldots, f_N\}$ created from a complete genome sequence. Also, assume that we are given a repeat library $\mathcal{R} = \{r_1, r_2, \ldots, r_M\}$. The problem is to identify the repeat sequences in all $f_i \in \mathcal{F}$ with minimum possible false positives.

We develop a new method called *RepFrag*, to identify repeats in fragments. Briefly, RepFrag works as follows. First, we align all the fragments in $\mathcal{F}$ with all the repeats in $\mathcal{R}$ respectively using Blast [4]. The alignments provide us the seeds which will help us locate the repeats. We, then, build a *graph* from the alignments for each repeat in $\mathcal{R}$. We model the alignments as the *vertices* of the graphs. We create *edges* between two vertices if they can jointly express a potential repeat better (we discuss this in detail in the following sections). We traverse the paths in this graph. Each path represents a complete repeat sequence on the non-fragmented genome. Once we find a path, we mask the aligned regions on the fragments corresponding to the vertices on that path. We also, remove the vertices and edges associated with those fragments. Then we create a new fragment from the non-aligned regions of the fragments on the path and align it with the repeats in $\mathcal{R}$. We modify the existing graphs based on the new alignments, if there are any. This iterative process of path selection and modification of the graphs continues as long as there is a single promising path.

We compared our method to RepeatMasker [25], which became a gold standard among the existing methods due to its accuracy [14]. We have compared the two methods on the datasets we generated from 5 chromosomes of *Arabidopsis Thaliana* (*A.thaliana*) with different set of fragment length and coverage. On average RepFrag had a 35% better true positive-false positive ratio. RepFrag is more than 7 times faster than RepeatMasker on the average. Thus, our method improves significantly over RepeatMasker in terms of speed and true positive- false positive ratio.

Section 2 summarizes the related work. Section 3 describes our method. Section 4 presents the results. Section 5 ends with a brief conclusion.

2. RELATED WORK

Identification of repeats is a well researched problem. We can categorize the existing methods based on the their approaches and the class of repeat they identify. Here, we provide a brief overview of them. A detailed discussion is available in Lerat [14].

Library based approaches: These methods collate a library of curated consensus repeat sequences. They compare the chromosome sequences with the repeats in that library. *The method that we develop in this paper also falls in this category*. RepeatMasker, in this category, has become a gold standard in terms of its accuracy. It performs a similarity search based on the local alignment with Crossmatch or AB-BLAST [2]. In has been extensively used to search repeat for several organisms such as *A.Thaliana* [5] and *Homo Sapiens* [12]. PLOTREP, also a library based tool, can merge the fragmented repeats, that arise in the output due to insertion and deletion between the query and repeat sequences [28]. Greedier detects both fragmented and nested repeats in the chromosome [16]. These methods, apart from Greedier, do not consider the fact that transposons can move into other repeats and split them. As a result, although they work well for repeats that are intact, their performance drops in the presence of nested transposons. Greedier on the other hand considers nested transposons. It aims to find repeats in complete sequences. It creates graphs for each of the repeat by alignments between a complete chromosome and every repeat. It then selects the best path from all the graphs, which corresponds to a complete repeat. Greedier iteratively selects a path, removes the corresponding components from the graph and aligns the modified chromosome with the Blast as long as there is a single promising path. The very nature of the algorithm enables Greedier to locate the fragmented and nested transposons. However, Greedier works only for complete sequences. It can not identify repeats when a set of short fragments are available.

One typical limitation of the library based method is that, they can not discover any new repeat or new class of repeat. They can locate only the repeats specified in the repeat library.

Signature based approaches: These methods search for a specific structure or motif characteristic to a particular class of repeat. They can be further categorized based on the specific structure of repeats that they can identify such as LTR (long terminal repeat) retrotransposons, non-LTR retrotransposons, MITE etc. The TSDfinder [27], refines the coordinates of the L1 insertions that are detected by RepeatMasker. The SINEDR method detects the SINEs (short interspersed nucleotide elements) [30]. The RTAnalyzer program is used to detect the origin of retrotransposons [17] .

Several methods are available to detect the LTR retrotransposons in the genome, such as RetroTector [26], LTR_par [8], find_LTR [21] and LTR_STRUC [18]. These methods depend on a number of structural features of the transposons such as the distance between two LTRs in a sequence, the size of the LTR sequence, the identity between two LTRs and presence of critical sites for replication.

Methods have been proposed to locate MITE, which is a particular group of transposable elements, that appear in high copy number. FINDMITE looks for the MITEs that satisfy several criteria, such as particular TSDs (target site duplication), a certain length of TIR (terminal inverted repeat) and a minimum distance between two TIRs [29]. TRANSPO detects TIRs from query sequences [24]. The MAK tool kit is a group of program used to automate the MITE analysis [31].

Algorithm for RepFrag INPUT: A set of repeats, a set of fragments

1. Align the set of fragments to the set of repeats with a sequence alignment tool, such as Blast.
2. Create a graph for each repeat from the alignments.
3. **while** there are more vertices in the graphs
   (a) Select the path from each graph with the highest fitness.
   (b) From all the paths in (a), select the one with the highest fitness and mask the appropriate regions.
   (c) Remove all the vertices and edges from the graphs corresponding to the fragments on the selected path.
   (d) Stitch both non-masked ends of the selected path and cerate a new fragment.
   (e) Align that fragment with the repeat library.
   (f) Create vertices from the alignments and insert them to appropriate graphs.

Figure 2: Pseudocode of our method.

These methods focus on specific classes of transposons. Although they can be relevant for those specific classes, their success depends on the knowledge we have about the structures and characteristics of those specific classes of transposons. Furthermore, they can only work for complete sequences and do not take the effects of nested transposons into consideration.

De novo methods: These methods aim to find repeats without using a repeat library. They can be classified into two main groups. The first group of methods compare a sequence to itself, while the others search for small repeated words, known as k-mers.

In the self comparison group, The Repeat Pattern Toolkit depends on a sequence similarity scoring system and uses Blast for the comparison [3]. RECON is a popular program that also uses Blast for self comparison [6]. It, then, runs a clustering method to create repeat families. The PILER program differentiates between the tandem repeats, transposable elements and the terminal repeats [7]. The BLASTER method also uses Blast [20]. It matches the repeats to the genome and clusters the sequences into families.

The methods in the k-mer group search for multiple occurrences of identical or similar sequences. REPuter uses suffix tree for this purpose [10]. FORRepeats first locates the exact repeat sequences, then identifies the approximate repeat sequences and compares between them [13]. Programs such as ReAS [15], Tallymer [9] and RepeatScout [19] create a library of high frequency, fixed length k-mers and use them as seeds to find the family of repeats.

The advantage of de novo method is that they can identify new classes of repeats, that library based methods are not able to detect in general. However, these methods produce a large amount of false positives [14].

Most of the existing methods are designed for assembled sequence. Hence they can not produce accurate results when the chromosome is fragmented, which is a more difficult situation as we discussed. Here, we design our method specifically for identification of repeats from fragments.

3. METHODS

In this section, we discusses our algorithm. Section 3.1 summarizes the complete algorithm. The subsequent sections elaborate different parts of the algorithm. Section 3.2 describes how we build graphs from the alignments of fragments and repeats. Sections 3.3 discusses how we select a set of fragments to mask from the graph. Finally, Section 3.4 discusses how we update each graph after the removal of the selected path.

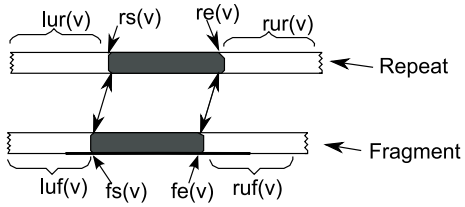


Figure 3: The figure describes different parts of a vertex v. Shaded intervals show the alignment between a repeat and a fragment. lur(v) and rur(v) denote the left and the right unaligned regions on the repeat respectively. luf(v) and ruf(v) stand for the left and the right unaligned regions on the fragment, respectively. rs(v) and re(v) are the start and end positions of the alignment on the repeat. Similarly, fs(v) and fe(v) are the start and the end positions of the alignment on the chromosome fragment.

3.1 Description of the complete algorithm

In this section, we provide a brief overview of the different steps of the complete algorithm.

Figure 2 presents an algorithmic outline of our method. In Step 1, we align the set of fragments $\mathcal{F}$ and the set of repeats $\mathcal{R}$ using a sequence alignment tool. In this paper, we use Blast, for this purpose. It is worth mentioning that one can use another sequence tool without changing the rest of the algorithm. Step 2 builds a graph for each repeats from these alignments. We create a vertex for each alignment. A vertex represents an alignment between a repeat and a fragment. Thus, there will be multiple vertices for a repeat-fragment pair if they have alternative high scoring based alignments. We store relevant alignment information at each vertex. Figure 3 illustrates an alignment between a repeat and a fragment and the information stored in the corresponding vertex. We, then, build a graph for each repeat in $\mathcal{R}$. We assign a vertex to a graph, if that vertex consists of the repeat for that graph. Then, based on a set of constraints, we create directed edges between the vertices in every graph. We elaborate on this step in Section 3.2.

Once the graphs are built, we iteratively process and update the graphs. In each iteration, we traverse all the graphs to select the path that has the highest fitness (Steps 3(a) and 3(b)). We discuss fitness computation in Section 3.3. A path is a sequence of vertices, that represents an alignment with a complete repeat, had the genome been non-fragmented. We remove all the vertices and edges related to the fragments in this path from all graphs. We mask the aligned fragments in the vertices of this path as a repeat. After this, we stitch the non-masked portion of the path and create a new fragment. The new fragment represents a potential portion of the original sequence before a transposon splits it. We, then, align the new fragment with the repeat library. We create a vertex for each alignment if there is any. Then, we insert the new vertices into the appropriate graphs as described in Step 3(f). We create edges to connect the new vertices with the existing set of graphs. We repeat steps 3(a) to 3(f) till number of vertices drops below a cutoff.

3.2 Creation of graph from alignments

This section describes how we create a graph from the alignments of the fragments in $\mathcal{F}$ with the repeats in $\mathcal{R}$.

Blast generates a set of alignments between pairs of repeats and fragments. These alignments provide us the seeds which will help us locate the repeats. We have observed that a lot of false posi-

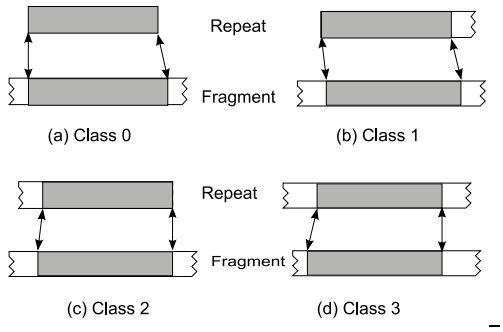


Figure 4: An illustration of the four classes of vertices. The straight boundary on the lateral side indicates the start or the end of a repeat. The zigzag side indicates that the boundary is non-terminal. Class 0 represents a match between a fragment and a complete repeat. Class 1 shows that the fragment matches with some prefix of a repeat. In class 2 the fragment matches with some suffix of a repeat. Class 3 consists of the vertices where a fragment matches with the middle portion of a repeat.

tives arise when the alignment length is small. So, we only consider the alignments that contain 100 or more bases. We create a vertex from every alignment. Thus, we can create multiple vertices between a fragment and a repeat. Consider a vertex v_i. Let us denote its fragment and repeat by $f(v_i)$ and $r(v_i)$ respectively. Let $fs(v_i)$ and $fe(v_i)$ denote the start and end of alignment on the fragment. We represent the start and end of alignment on the repeat with $rs(v_i)$ and $re(v_i)$. Let $a(v_i)$ and $b(v_i)$ denote the number of identical bases and the alignment length respectively. Let $o(v_i) \in \{+, -\}$ denote the orientation of alignment, direct or complemented, respectively. In complemented alignment, the fragment is aligned with the reverse complemented version of the repeat. Also, let us denote the length of the complete repeat $r(v_i)$ by len$(r(v_i))$. Thus, we define a vertex as an ordered tuple $v_i =< f(v_i), r(v_i), fs(v_i), fe(v_i), rs(v_i), re(v_i), a(v_i), b(v_i), o(v_i) >$. The subscript i to all the components of the vertex indicate that they belong to the ith vertex. Figure 3 illustrates different parts of a vertex.

Based on the start and end position of the alignment on the repeat sequence, we classify the set of vertices into four different classes. Figure 4 illustrates them.

1. Class 0: The alignment consists of the complete repeat (i.e. $rs(v_i) = 1$ and $re(v_i) =$ len$(r(v_i))$).
2. Class 1: The alignment contains only a prefix of the repeat (i.e. $rs(v_i) = 1$ and $re(v_i) <$ len$(r(v_i))$).
3. Class 2: The alignment contains only a suffix of the repeat (i.e. $1 < rs(v_i)$ and $re(v_i) =$ len$(r(v_i))$).
4. Class 3: The alignment does not contain the start and the end of the repeat (i.e. $1 < rs(v_i)$ and $re(v_i) <$ len$(r(v_i))$).

For every repeat $r_k \in \mathcal{R}$, we create a graph denoted by $\mathcal{G}_k$. We assign a vertex v_i to $\mathcal{G}_k$, if $r_k = r(v_i)$; i.e. the repeats for both the vertex and the graph are same. We create directed edges between the vertices that belong to a particular graph. Intuitively, an edge corresponds to two close consecutive portions of a repeat on the genome before fragmentation. The participating vertices can belong to the same or different fragments. We denote an edge as *internal* if the corresponding vertices share the same fragment, otherwise we call it an *external* edge. If we create an edge from vertex v_i to v_j, we denote v_i as *incoming* vertex to v_j and v_j as *outgoing* vertex to v_i. We can connect two vertices by an edge if they satisfy both *class constraints* and *position constraints*. We discuss these constraints next.

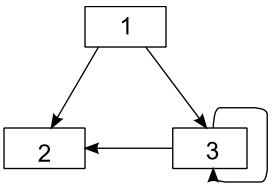


Figure 5: The figure graphically describes the class constraints, i.e. which classes of vertices can be connected to which classes. We can create edge from vertices of class 1 to that of class 2, vertices of class 1 to that of class 3 and vertices of class 3 to that of class 2. We can also create an edge from a vertex of class 3 to another vertex of the same class.

Class constraints: We can create a directed edge from vertex v_i to v_j if one of the following holds.

1. v_i belongs to class 1 and v_j belongs to class 2.
2. v_i belongs to class 1 and v_j belongs to class 3.
3. v_i belongs to class 3 and v_j belongs to class 3.
4. v_i belongs to class 3 and v_j belongs to class 2.

Figure 5 summarizes these constraints.

Class constraints are necessary but not sufficient to create an edge. If two vertices satisfy the class constraints, we can potentially create an edge between them, however that is not guaranteed. We also need to enforce that the end position of repeat on the incoming vertex matches closely to the start position of repeat on the outgoing vertex. We refine the edge creating process through the following position constraints.

Position constraints: Let us denote the average identity between two random sequences per nucleotide by ρ. We assume the value of ρ to be 0.25, as for every nucleotide there can be four different options in the other sequence. Assume that there is an edge from vertex v_i to v_j. The position constraints for external and internal edges are:

- Direct orientation, external vertices:
  1. $rs(v_j) - re(v_i) \leq \epsilon$, if $rs(v_j) > re(v_i)$.
  2. $re(v_i) - rs(v_j) \leq \epsilon$, if $rs(v_i) \leq rs(v_j) \leq re(v_i)$.
  3. $max\{ruf(v_i), luf(v_j)\} - \rho \times min\{ruf(v_i), luf(v_j)\} \leq \epsilon$.
- Direct orientation, internal vertices:
  1. $rs(v_j) - re(v_i) \leq \epsilon$, if $rs(v_j) > re(v_i)$.
  2. $re(v_i) - rs(v_j) \leq \epsilon$, if $rs(v_i) \leq rs(v_j) \leq re(v_i)$.
  3. $fs(v_j) - fe(v_i) \leq \epsilon$, if $fs(v_j) > fe(v_i)$.
  4. $fe(v_i) - fs(v_j) \leq \epsilon$, if $fs(v_i) \leq fs(v_j) \leq fe(v_i)$.

The first three constraints are applicable for the external edges where the two vertices belong to different fragments. Constraint 1 indicates that on the repeat side, the end position of alignment

on the first vertex, i.e. $re(v_i)$ is at most the starting position of alignment on the second vertex, i.e. $rs(v_j)$. Constraint 2 is analogous to the earlier one, but applies for overlapped repeat alignments on the two vertices. It enforces an upper bound to the length of the overlapping region. This condition indicates that there is an unaligned region on the repeat from position $re(v_i)$ to $rs(v_j)$. Thus, this constraint sets an upper bound to the length of this region with the parameter ϵ. On the fragment side, it is not possible to know if the two fragments overlap or not, as we do not have information about the position of the fragments on the chromosome. $\min\{ruf(v_i), luf(v_j)\}$ is the smallest possible overlap on the fragment, for any external edges. The expected number of identities on the overlapping region in that case is $\rho \times \min\{ruf(v_i), luf(v_j)\}$. Subtracting this value from the length of the largest possible overlaps, we obtain an upper bound to the expected number of mismatching bases on the fragment side. The third constraint enforces an upper bound to this value as ϵ.

Constraints 1 and 2 for the internal edges are exactly similar to the first two constraints of external edges. So we do not repeat the discussion for them. However, the constraints 3 and 4 (on the fragment side) are different, as we know the relative distance between the fragments in the two vertices. Constraint 3 indicates that on the fragment side, the end position of alignment on the first vertex, i.e. $fe(v_i)$ is at most the starting position of alignment on the second vertex, i.e. $fs(v_j)$. Constraint 4 is analogous to the earlier one, but applies for overlapped fragment alignments on the two vertices. It enforces an upper bound to the length of the overlapping region. This condition indicates that there is an unaligned region on the fragment from position $fe(v_i)$ to $fs(v_j)$. Thus, the constraint sets an upper bound to the length of this region as the parameter ϵ.

Based on the position and class constraints, edges can be classified into five categories. Figure 6 illustrates four of them.

- *External edge.* There is no overlap on the repeat side.
- *External edge.* There is an overlap on the repeat side.
- *Internal edge.* There is no overlap on either repeat or fragment side.
- *Internal edge.* There is an overlap on the fragment side, but not on the repeat side.
- *Internal edge.* There can be a fifth kind of edge, analogous to the fourth kind, where the overlap is on the repeat side, not on the fragment side.

We use a similar set of constrains when the two vertices have complemented alignment. However, due to space limitation and their similarity with the described ones, we do not discuss them here.

3.3 Graph traversal and path selection

In this section, we describe how we select the best path from the set of all graphs (Steps 3(a) and 3(b)). A path is a sequence of vertices that are connected by edges. All vertices on a path belong to the same repeat as there are no edges between the vertices of different repeats. Each path denotes a potential repeat that is spread across the fragments corresponding to the vertices on that path. In order to describe how we select a promising path (i.e., a path that leads to a complete repeat) we first define the concept of *fitness* for vertices and edges. In our definition, a large fitness value of a vertex (edge) indicates that the corresponding vertex (edge) will lead to a complete repeat with high chance.

Fitness value of a vertex: We define the *penalty zone* of a vertex v_i as $\kappa(v_i) = \min\{lur(v_i), luf(v_i)\} + \min\{rur(v_i), ruf(v_i)\}$.

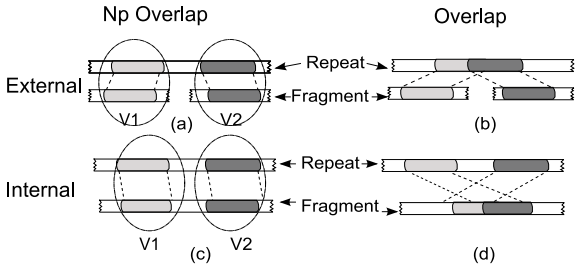


Figure 6: Four different kinds of edges: (a) This corresponds to external edges and no overlap on the repeat regions. (b) This is for external edges and overlap between the aligned repeat regions on the vertices. (c) Internal edges, however no overlap on either fragment or repeat side. (d) Internal edges, overlap on the fragment side. There can be a fifth kind of edge, analogous to (d), where the overlap is on the repeat side, not on the fragment side.

To create the penalty zone, we obtain the minimum of left unaligned regions on the repeat and the fragment of the vertex. Similarly, we obtain the minimum of the right unaligned regions on repeat and fragment on the vertex. The summation of these left and right regions defines the penalty zone. Intuitively, this penalty zone identifies the *non-compliant* region of a vertex, which is so divergent from the repeats in the library $\mathcal{R}$, that Blast could not align this region. A smaller penalty zone leads to a higher fitness value for that vertex and vice versa. We denote the fitness value of the vertex as,

$$f(v_i) = \frac{a(v_i) + \rho \times \kappa(v_i)}{b(v_i) + \kappa(v_i)}$$

The numerator of fitness function corresponds to the sum of the observed and expected number of identities. The denominator of this function is the sum of the expected and the observed alignment length. Note, that we define the observed number of identities based on the alignment between the repeat and the fragment, while we define the expected identities and alignment based on the penalty zone.

Fitness of an edge: Similar to the fitness of a vertex, the fitness of an edge is the ratio of the sum of the expected and the observed number of identities over the sum of the expected and observed length of alignments after merging the two fragments of that vertex. We define the observed and expected number of identities of edge e_{ij} by $\alpha_o(e_{ij})$ and $\alpha_e(e_{ij})$. We define the observed and expected alignment length of e_{ij} by $\beta_o(e_{ij})$ and $\beta_e(e_{ij})$ respectively. Thus, the fitness value of e_{ij} is given by,

$$g(e_{ij}) = \frac{\alpha_o(e_{ij}) + \alpha_e(e_{ij})}{\beta_o(e_{ij}) + \beta_e(e_{ij})}$$

We compute $\alpha_o(e_{ij}) = a(v_i) + a(v_j)$, which is the sum of observed identities of the two vertices. $\beta_o(e_{ij})$ is the sum of the observed alignment length of the two vertices, given by $b(v_i) + b(v_j)$.

Recall from Section 3.2 that we classified edges into five categories. Computation of the functions α_e and β_e for an edge depends on the category that edge belongs to. Next, we discuss the computation of α_e and β_e functions for each of the five categories. We discuss only the vertices with direct orientation as the complementary orientation is similar to it.

- *External edge, non-overlapping repeats:* The term $\alpha_e(e_{ij})$ is given by $\rho \times \min\{ruf(v_i), luf(v_j), (rs(v_j) - re(v_i))\} +$

$\rho \times \{lur(v_i)\} + \rho \times \{rur(v_j)\}$. The first term is the expected number of identities between the aligned regions of v_i and v_j. The second term is the expected number of identities for the prefix of the repeat prior to its aligned region in v_i. The last term is the expected number of identities for the suffix of the repeat after its aligned region in v_j. We formulate $\beta_e(e_{ij}) = \max\{ruf(v_i), luf(v_j), (rs(v_j) - re(v_i))\} + lur(v_i) + rur(v_j)$. Similar to $\alpha_e(e_{ij})$, this formulation considers the unaligned regions between, before and after the two aligned regions.

- *External edge, overlapping repeats:* In this case, we have to adjust the number of identities and the alignment length due to overlapping of aligned repeat regions between the two vertices. The term $\alpha_e(e_{ij})$ is given by $\rho \times \{lur(v_i)\} + \rho \times \{rur(v_j)\} - \zeta \times \{re(v_i) - rs(v_j)\}$. The first term is the expected number of identities for the prefix of the repeat prior to its aligned region in v_i. The second term is the expected number of identities for the suffix of the repeat after its aligned region in v_j. $\{re(v_i) - rs(v_j)\}$ is the overlapped region on the repeat. ζ is the identity frequency of the overlapped region, calculated as $\frac{a(v_i)+a(v_j)}{b(v_i)+b(v_j)}$. So, the last term denotes the number of identities in the overlapped region, which we subtract from the expected number of identities to amend the effect of overlap. The length of the expected alignment becomes $\beta_e(e_{ij}) = \max\{ruf(v_i), luf(v_j)\} + lur(v_i) + rur(v_j) - \{re(v_i) - rs(v_j)\}$. Similar to $\alpha_e(e_{ij})$, this formulation considers the unaligned region between, before and after the two aligned regions. We subtract the last term, as, we assume that there is also an overlap on side of the the fragments.

- *Internal edge, non-overlapping repeats and fragments:* When the edge belongs to internal category, the formulation is different, as we have the relative distance between the two fragments available. $\alpha_e(e_{ij})$ is given by $\rho \times \min\{(rs(v_j) - re(v_i), fs(v_j) - fe(v_i)\} + \rho \times \{lur(v_i)\} + \rho \times \{rur(v_j)\}$. The first term denotes the expected number of identities on the non-aligned region between the two vertices. The second term is the expected number of identities for the prefix of the repeat prior to its aligned region in v_i. The last term is the expected number of identities for the suffix of the repeat after its aligned region in v_j. The expected length of alignment is given by $\beta_e(e_{ij}) = \max\{(rs(v_j) - re(v_i), fs(v_j) - fe(v_i)\} + lur(v_i) + rur(v_j)$. The first, second and last term correspond to the expected alignment lengths between, before and after the aligned regions.

- *Internal edge, non-overlapping repeats, overlapping fragments:* In this condition, the fragments overlap. We subtract the overlapped identities on the fragment side from the expected number of identities, to amend to effect of the overlap. Thus, $\alpha_e(e_{ij})$ is $\rho \times \{lur(v_i)\} + \rho \times \{rur(v_j)\} - \zeta \times \{fe(v_i) - fs(v_j)\}$. The first term is the expected number of identities for the prefix of the repeat prior to its aligned region in v_i. The second term is the expected number of identities for the suffix of the repeat after its aligned region in v_j. $\{fe(v_i) - fs(v_j)\}$ is the overlapped region on the fragment. ζ is the identity frequency of the overlapped region, calculated as $\frac{a(v_i)+a(v_j)}{b(v_i)+b(v_j)}$. So, the last term denotes the number of identities in the overlapped region on the fragment which we subtract from the first two terms. The expected length of alignment is given by $\beta_e(e_{ij}) = \{rs(v_j) - re(v_i)\} + lur(v_i) + rur(v_j)$. The first term represents the expected alignment length on the repeat, as there is an overlap on the fragment side. The second and last terms correspond to the expected alignment length before and after the aligned regions, respectively.

- *Internal edge, overlapping repeats, non-overlapping fragments:* In this condition, the repeats overlap and the equations are analogous to that of the earlier one. Thus, $\alpha_e(e_{ij})$ is $\rho \times \{lur(v_i)\} + \rho \times \{rur(v_j)\} - \zeta \times \{re(v_i) - rs(v_j)\}$. The first term is the expected number of identities for the prefix of the repeat prior to its aligned region in v_i. The second term is the expected number of identities for the suffix of the repeat after its aligned region in v_j. $\{re(v_i) - rs(v_j)\}$ is the overlapped repeat region. ζ is the identity frequency of the overlapped region, calculated as $\frac{a(v_i)+a(v_j)}{b(v_i)+b(v_j)}$. So, the last term denotes the number of identities in the overlapped region on the fragment which we subtract from the expected number of identities, to amend the effect of overlap. The expected length of alignment is given by $\beta_e(e_{ij}) = \{fs(v_j) - fe(v_i)\} + lur(v_i) + rur(v_j)$. The first term represents the expected alignment length on the fragment, as the overlap is on the side of the repeat. The second and last terms correspond to the expected alignment length before and after the aligned regions respectively.

Path selection: We select a path from each of the graphs using a greedy strategy as follows. In each graph, we start from the vertex with the highest fitness value.

Once we select the first vertex, we extend the path to incoming and outgoing directions. We discuss how we do the extension in the outgoing direction. We do not discuss extending the path in the other direction as it follows the same algorithm in the opposite direction. Let us denote the current vertex by v_i. Let us denote the set of outgoing vertices of v_i by the set $\mathcal{N}_R(v_i)$. Also, let $g(e_{ik})$ denote the fitness value of the edge by $e_{ik} = \{v_i, v_k\}, v_k \in \mathcal{N}_R(v_i)$. We select the neighbor v_j, such that $v_j = \arg\max_{v_k} g(e_{ik})$. We use this new vertex v_j to extend further, similarly. We extend the path till the last possible vertex. Once we complete extending the path in one direction, we repeat the same extension process to the other direction.

When the entire path is selected, we calculate the fitness value of the path. Briefly, the fitness value of a path is a generalization of the fitness value of an edge over multiple vertices (≥ 2). We calculate the expected and observed number of identities over the expected and observed alignment length on the path using equations similar to that of the fitness of an edge. We skip the detailed formula as it is trivial to derive it from the fitness of the edges.

We select a path from each graph using the algorithm described above. Among those paths, we then select the one with the highest fitness value. We mask the aligned portion on the fragments of this path as repeat.

3.4 Updating the graphs

Recall that a fragment can appear in multiple vertices as it can have high similarity with more than one repeats. It can even have alternative alignments with the same repeat. As a result of this, a fragment can be a part of many paths in multiple graphs. Once we pick the best path, we cannot use the fragments on that path to find another path as those fragments are already masked. In this section, we discuss how we update the graphs that we built to avoid such conflicts (Steps 3(c) to 3(g)). In addition to avoiding the conflicts, our update strategy enables us to find repeats that are split due to nested transposons.

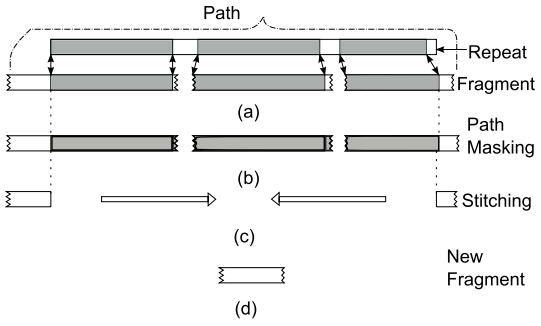


Figure 7: The figure illustrates how we mask the aligned region in a path and create a new fragment, if possible, from the unaligned regions of the terminal fragments. (a) A path consists of three vertices. (b) The shaded regions show the parts of the fragments we mask. Note, that there are some non-zero unaligned bases on the right side of first and second fragments. For continuity, we also mask them. Thus, we mask part of the first and last vertex and entire region of the intermediate fragments. (c) We select the left unaligned part of the first fragment and the right unaligned part of the last fragment to cerate a new fragment. The length of at least one of them has to be non-zero to create the new fragment. (d) The new fragment, that we now align with the repeat library.

Let $\omega(f(v_i))$ denote the set of vertices, created from $f(v_i)$. That is $\forall v_j \in \omega(f(v_i)), f(v_i) = f(v_j)$. Also, let $\varphi(f(v_i)) = \{e_{jk}\}$ denote the set of edges such that at least one vertex at the two ends of those edges contain $f(v_i)$. Mathematically, either $f(v_i) = f(v_j)$ or $f(v_i) = f(v_k)$. Let $\mathcal{P} = \{v_1, v_2, \cdots, v_N\}$ be the best path selected at the current iteration. $\forall f(v_k)$ such that $v_k \in \mathcal{P}$, we remove $\omega(f_k)$ and $\varphi(f_k)$ from all the graphs. In other words, we remove all the vertices and edges for all the fragments related to the selected path.

After we remove the vertices and edges as discussed, we stitch $luf(v_1)$ and $ruf(v_N)$ and create a new fragment v_{1N}. If there is more than two fragments in the path, we completely mask all the non-terminal fragments to potentially mask a continuous repeat region. However, we do not mask the unaligned left region on the first fragment and the unaligned right region on the last fragment as, according to our constraints, they are not part of the current repeat. We use them to create a new fragment. Figure 7 illustrates the stitching process for a hypothetical path with three external vertices. We align v_{1N} with the repeat library. We create new vertices, provided there are some valid alignments. We insert the new vertices into the existing set of vertices. If there are vertices with more that one repeat, we add them to appropriate graphs.

For a graph $\mathcal{G}_r$, we denote the set of existing and new vertices by $\mathcal{V}_{re}$ and $\mathcal{V}_{rn}$ respectively. We create possible edges $\{e_{ij}\}$ within $\mathcal{V}_{rn}$ itself, $v_i, v_j \in \mathcal{V}_{rn}$. We also, create the set of edges $\{e_{ij}\}$ between $\mathcal{V}_{re}$ and $\mathcal{V}_{rn}$, such that $v_i \in \mathcal{V}_{re}, v_j \in \mathcal{V}_{rn}$ or $v_i \in \mathcal{V}_{rn}, v_j \in \mathcal{V}_{re}$.

After we create the new edges, we iterate over the steps as we describe in Section 3.3 and Section 3.4 and find the next best path. We continue the iteration as long as there is a path with significant cutoff value.

4. EXPERIMENTS

In this section, we evaluate the performance of RepFrag on real datasets.

Experiment setup.

We implemented RepFrag in Java. There are a significant number of repeat finding softwares available. Among them, RepeatMasker is the gold standard to find repeats because of its accuracy [14]. In this paper, we compared our method to RepeatMasker. We obtained RepeatMasker from `http://www.repeatmasker.org/`. We also compared our method to Crossmatch. However, due to poor accuracy of Crossmatch we do not report its results.

We ran all our experiments on a Quad-Core AMD Opteron. We allocated 5-10 GB of RAM for every experiment depending on the size of the fragment file.

We compare the accuracy and the running time of our method to those of RepeatMasker. We measure the accuracy as the number and percentage of true positives and false positives. We say that a nucleotide we identify as repeat is *true positive* is it is annotated as repeat in the literature. Otherwise, we call it *false positive*. We use the annotations of *A.Thaliana* from `http://www.arabidopsis.org/` as the gold standard for this purpose. We also calculated the ratio of true positive to false positive as it shows the trade off between the two metrics. Finally, we measured the running time for both the methods. We report the running time in minutes.

Description of datasets.

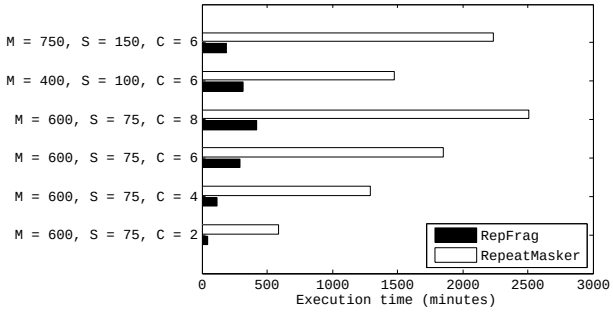
We downloaded 5 complete genome sequence files of *A.Thaliana* from `ftp://ftp.arabidopsis.org` for the locations of the repeats on these chromosomes are already annotated. In order to evaluate the performance of RepFrag for fragmented datasets with different characteristics, we created several datasets. To explain these datasets we first describe the variables that govern these datasets. Let us denote the length of chromosome g_i (g_i is the ith chromosome of *A.Thaliana*, $1 \leq i \leq 5$) with $l(g_i)$. The length of the fragments in each dataset is normally distributed with mean μ and standard deviation σ (i.e., $\mathcal{N}(\mu, \sigma)$). We describe the values of these two variables later in this section. Assume that the total coverage of the fragments in a dataset is C. We first draw a random number s from the uniform distribution $\mathcal{U}(1, l(g_i) - \mu)$. This number represents the starting location of a new fragment. To determine the length of that fragment, we pick a random number from the normal distribution $l(f) \sim \mathcal{N}(\mu, \sigma)$. We select the ending point of the fragment as $min(s + l(f), l(g_i))$. We continue this process of creating new fragments from the original chromosome until the total length of all the generated fragments equals or barely exceeds $C \times l(g_i)$. We say that this dataset has C-fold coverage.

We created datasets with six different configurations of parameters μ, σ and C from each of the five chromosomes of *A.Thaliana*. So totally we have 30 datasets. The value of these parameters for these datasets are as follows.

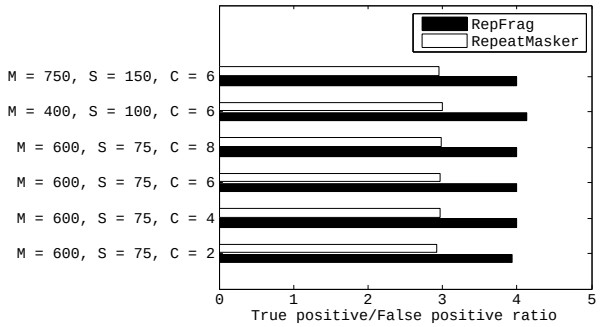
- $\mu = 600, \sigma = 75, C = 2$
- $\mu = 600, \sigma = 75, C = 4$
- $\mu = 600, \sigma = 75, C = 6$
- $\mu = 600, \sigma = 75, C = 8$
- $\mu = 400, \sigma = 100, C = 6$
- $\mu = 750, \sigma = 75, C = 2$

It is worth noting that the values of μ and σ in these configurations are taken from the actual sequencing machines [1, 22]. Also, the coverage values are compatible with real datasets.

We obtain the repeat library from `http://www.girinst.org`. RepeatMasker also uses the same repeat library.



(a) Comparison in execution time



(b) Comparison in TP/FP

Figure 8: The two figures highlight the comparison of our method and RepeatMasker. (a) In six different configurations our method outperforms RepeatMasker in terms of speed. On the average it is 7.3 times faster than the other method. (b) In the same six configurations our method consistently has higher true positive/false positive ratio than that of RepeatMasker. On the average it is 35% better that the other method.

Results.

In this section we present the running times and accuracies of RepFrag and RepeatMasker on all datasets. Tables 1- 6 present the results.

In terms of accuracy, we observe that on average RepFrag and RepeatMasker can identify 47.17% and 54.16% of the true positives over all the datasets. The first question that arises from this result is why both of these methods miss nearly half of the annotated repeats. In order to answer this question, we carried out another experiment as follows. We extracted the annotated repeats that are not found by these methods and compared them to the repeat library using Blast. There were no significant alignments between the two sets. This suggests that the repeat library is not complete and thus, the results can further improve if missing repeats are included in the library.

The true positive percentage of RepeatMasker is around 7% higher than that of RepFrag. However, as Tables 1- 6 demonstrate, RepeatMasker achieves this at the expense of increased false positives. On average the false positive rates of RepFrag and RepeatMasker are 15.12% and 22.15% respectively. A standard way to evaluate the accuracy in this case is to consider the ratio of true positives to false positive. On the average, true positive to false positive ratio is 4.01 for our method, while it is 2.96 for RepeatMasker. *This indicates that our method can identify one more true positive than RepeatMasker (i.e., 4.01 - 2.96) for each false positive. We conclude that our method is (4.01/2.96) = 1.35 times, i.e. 35%, more accurate than RepeatMasker.*

Our results also demonstrate that that our method is significantly faster than RepeatMasker on all the datasets. For different database characteristics we tested, RepFrag is 4.6 to 12.5 times faster. The gap between the speeds of the two methods increases as the coverage decreases. This is probably because as the coverage drops, the density of the graphs in our method drops as well. As a result, the time required for searching a path with a high fitness value and updating the graph reduces greatly. On the other hand, the running time of RepeatMasker is linear in the size of the database of fragments. So its running time drops only linearly with coverage. We also observed that the gap between the running times of the two methods grows as the mean fragment length increases. This is because as the length of the fragments grow (keeping the coverage fixed), the number of fragments drop. As a result, the number of vertices in our graphs reduce. This reduces the time to search and update our graphs. The net result of reduced running time for RepFrag can be seen in Tables 5, 3 and 6. On the average over all of our 30 datasets, RepFrag is 7.3 times faster than RepeatMasker.

If we concentrate on the individual chromosome level across the setup, we observe some interesting patterns. For example, Chromosome 4 always have the highest percentage of true positive and highest TP/FP ratio for both the methods. We can conclude that, repeats in this chromosome are least divergent compared to that of the others. On the contrary, the $5th$ chromosome, has always the lowest true positive and TP/FP ratio. This implies, that the repeats and transposons in this chromosome have diverged from the library specified repeats to a high extent.

We highlight the comparison of the two methods in Figure 8. Figure 8(a) shows that our method is consistently faster than RepeatMasker. On the average RepFrag is 7.3 times faster than RepeatMasker. We also observe that when we increase C or decrease μ, both the methods requires more time. According to our estimation, the time requirement of our method varies linearly with the number of edges in all the graphs. Figure 8(b) compares the two methods for their true positive-false positive (TP/FP) ratio. We observe that our method attains a higher value of that ratio for all the configurations. On the average it is 35% better that the other method.

The consistency of results across all datasets testifies the stability of our method with different size of the fragments and the coverage.

5. CONCLUSION

In this paper, we considered the problem of identifying repeats from fragments generated from sequencing machines. We developed a graph based method, named *RepFrag*, which solves this problem efficiently even in the presence of nested transposons. Given a database of short fragments from a genome and a library of possible repeats, RepFrag uses Blast to compare the fragments to repeats. It creates a graph for each repeat based to these alignments. It searches for the path with best fitness value in the graph, which corresponds to a potential complete repeat sequence. It then masks the aligned regions of fragments on the path and creates a new fragment from the unaligned leftovers. It aligns the new fragment with the repeat library and updates the graph using the alignments. It continues this iterative process of path selection and modification of graph till there are no more promising paths.

We compared RepFrag to RepeatMasker, which became a gold standard among the existing methods due to its accuracy. We have compared the two methods on the fragment 30 datasets with different characteristics from five chromosomes of *A.Thaliana*. On average RepFrag had a 35% better true positive-false positive ratio. RepFrag was 7.3 times faster than RepeatMasker on the average. Thus, our method improved significantly over RepeatMasker in terms of speed and true positive- false positive ratio.

Table 1: Comparison of RepFrag with RepeatMasker. $\mu = 600$, $\sigma = 75$, $C = 2$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	10.5	43.72	15.6	2.80	59	51.34	24.35	2.10	752
Chr2	9.8	46.72	15.51	3.01	42	53	22.64	2.34	478
Chr3	10.1	42.98	17.12	2.51	42	48.79	24.47	1.99	577
Chr4	8.2	60.7	6.45	9.41	40	70.44	10.72	6.57	455
Chr5	10.3	40.85	20.58	1.98	53	47.48	28.89	1.64	666
Avg	9.8	47	15.05	3.94	47	54.21	22.21	2.92	585

Table 2: Comparison of RepFrag with RepeatMasker. $\mu = 600$, $\sigma = 75$, $C = 4$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	20.9	44.13	15.78	2.79	138	51.46	24.4	2.10	1593
Chr2	19.6	46.93	15.91	2.94	109	52.98	22.82	2.32	1006
Chr3	20.4	43.19	17.31	2.49	105	48.72	24.35	2.00	1366
Chr4	16.5	61.31	6.27	9.77	96	70.44	10.32	6.82	1067
Chr5	20.6	41.72	20.93	1.99	126	48	29	1.65	1414
Avg	19.6	47.45	15.24	3.99	115	54.32	22.17	2.97	1289

Table 3: Comparison of RepFrag with RepeatMasker. $\mu = 600$, $\sigma = 75$, $C = 6$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	31.3	44.15	15.71	2.81	312	51.32	24.29	2.11	2365
Chr2	29.4	47.06	16.04	2.93	247	52.86	22.86	2.31	1491
Chr3	30.7	43.33	17.41	2.48	351	48.66	24.38	1.99	1814
Chr4	24.8	61.43	6.25	9.82	162	70.43	10.29	6.84	1441
Chr5	30.9	41.91	21.33	1.96	389	47.93	29.23	1.63	2136
Avg	29.4	47.57	15.34	4	292	54.24	22.21	2.97	1849

Table 4: Comparison of RepFrag with RepeatMasker. $\mu = 600$, $\sigma = 75$, $C = 8$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	41.7	44.26	16.01	2.76	491	51.28	24.52	2.09	3258
Chr2	39.2	47.13	16.16	2.91	433	52.85	22.97	2.30	2025
Chr3	41.0	43.35	17.48	2.48	414	48.52	24.45	1.98	2466
Chr4	33.1	61.61	6.23	9.88	331	70.44	10.21	6.89	1923
Chr5	41.3	42.15	21.29	1.97	444	48.06	29.08	1.65	2866
Avg	39.2	47.70	15.43	4	422	54.23	22.24	2.98	2507

Table 5: Comparison of RepFrag with RepeatMasker. $\mu = 400$, $\sigma = 100$, $C = 6$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	31.2	42.43	14.96	2.83	367	50.51	23.93	2.11	1926
Chr2	29.4	45.61	15.29	2.98	314	52.03	22.53	2.30	1209
Chr3	30.8	41.78	16.56	2.52	316	47.6	23.84	1.99	1458
Chr4	24.8	59.58	5.76	10.34	253	69.42	9.95	6.97	1120
Chr5	30.9	40.58	20.24	2.00	341	47.38	28.59	1.65	1683
Avg	29.4	45.99	14.56	4.13	318	53.38	21.76	3	1479

Table 6: Comparison of RepFrag with RepeatMasker. $\mu = 750$, $\sigma = 150$, $C = 6$.

Chr	True repeats $\times 10^6$	RepFrag				RepeatMasker			
		% of TP + FN		TP/FP	Running time (m)	% of TP + FN		TP/FP	Running time (m)
		TP	FP			TP	FP		
Chr1	31.3	43.92	15.53	2.82	216	51.62	24.42	2.11	2871
Chr2	29.5	46.87	15.74	2.97	194	53.35	22.94	2.32	1838
Chr3	30.7	43.13	17.2	2.50	176	49.01	24.64	1.98	2227
Chr4	24.7	60.93	6.26	9.73	143	70.76	10.43	6.78	1724
Chr5	31.0	41.7	21.01	1.98	196	48.32	29.41	1.64	2535
Avg	29.4	47.31	15.14	4	185	54.61	22.36	2.96	2239

6. REFERENCES

[1] http://www.npac.syr.edu/projects/cpsedu/summer98summary/examples/hpf/hpf.html.

[2] http://www.advbiocomp.com/blast.html.

[3] P. Agarwal and D. States. The Repeat Pattern Toolkit (RPT): analyzing the structure and evolution of the C. elegans genome. *Proc Int Conf Intell Syst Mol Biol*, 2:1–9, 1994.

[4] S. Altschul, W. Gish, and W. M. et al. Basic local alignment search tool. *J Mol Biol*, 215(3):403–10, 1990.

[5] Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant Arabidopsis thaliana. *Nature*, 408(6814):796–815, December 2000.

[6] Z. Bao, , Z. Bao, and S. R. E. et al. Automated de novo identification of repeat sequence families in sequenced genomes. *Genome Res*, 12:1269–1276, 2002.

[7] R. Edgar and E. Myers. PILER: identification and classification of genomic repeats. *Bioinformatics*, 21 Suppl 1:i152–8, 2005.

[8] A. Kalyanaraman and S. Aluru. Efficient algorithms and software for detection of full-length LTR retrotransposons. *J Bioinform Comput Biol*, 4(2):197–216, 2006.

[9] S. Kurtz, A. Narechania, and J. S. et al. A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9:517, 2008.

[10] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–7, 1999.

[11] E. Lander, L. Linton, and B. B. et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[12] E. S. Lander, L. M. Linton, and B. e. a. Birren. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, February 2001.

[13] A. Lefebvre, T. Lecroq, and H. D. et al. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–26, 2003.

[14] E. Lerat. Identifying repeats and transposable elements in sequenced genomes: how to find your way through the dense forest of programs. *Heredity*, 2009.

[15] R. Li, J. Ye, and S. L. et al. ReAS: Recovery of ancestral sequences for transposable elements from the unassembled reads of a whole genome shotgun. *PLoS Comput Biol*, 1(4):e43, 2005.

[16] X. Li, T. Kahveci, and A. Settles. A novel genome-scale repeat finder geared towards transposons. *Bioinformatics*, 24(4):468–76, 2008.

[17] J. Lucier, J. Perreault, and J. N. et al. RTAnalyzer: a web application for finding new retrotransposons and detecting L1 retrotransposition signatures. *Nucleic Acids Res*, 35(Web Server issue):W269–74, 2007.

[18] E. McCarthy and J. McDonald. LTR_STRUC: a novel search and identification program for LTR retrotransposons. *Bioinformatics*, 19(3):362–7, 2003.

[19] A. L. Price, N. C. Jones, and P. A. Pevzner. De novo identification of repeat families in large genomes. *Bioinformatics*, 21(suppl_1):i351–358, June 2005.

[20] H. Quesneville, C. Bergman, and O. A. et al. Combined evidence annotation of transposable elements in genome sequences. *PLoS Comput Biol*, 1(2):166–75, 2005.

[21] M. Rho, J.-H. Choi, and S. K. et al. De novo identification of LTR retrotransposons in eukaryotic genomes. *BMC Genomics*, 8(90), 2007.

[22] J. Rothberg and J. Leamon. The development and impact of 454 sequencing. *Nat Biotechnol*, 26(10):1117–24, 2008.

[23] P. SanMiguel, A. Tikhonov, and Y. J. et al. Nested retrotransposons in the intergenic regions of the maize genome. *Science*, 274(5288):765–8, 1996.

[24] N. Santiago, C. Herráiz, and J. G. et al. Genome-wide analysis of the Emigrant family of MITEs of Arabidopsis thaliana. *Mol Biol Evol*, 19(12):2285–93, 2002.

[25] H. R. . G. P. Smit, AFA. Repeatmasker open-3.0. Website, 1996-2004. http://www.repeatmasker.org.

[26] G. O. Sperber, T. Airola, and P. J. et al. Automated recognition of retroviral sequences in genomic dataŮRetroTector. *Nucleic Acids Res*, 35(15):4964Ű4976, 2007.

[27] S. Szak, O. Pickeral, and W. M. et al. Molecular archeology of L1 insertions in the human genome. *Genome Biol*, 3(10):research0052, 2002.

[28] G. Tóth, G. Deák, E. Barta, and G. Kiss. PLOTREP: a web tool for defragmentation and visual analysis of dispersed genomic repeats. *Nucleic Acids Res*, 34(Web Server issue):W708–13, 2006.

[29] Z. Tu. Eight novel families of miniature inverted repeat transposable elements in the african malaria mosquito, anopheles gambiae. *Proc Natl Acad Sci U S A*, 98(4):1699–704, 2001.

[30] Z. Tu, S. Li, and C. Mao. The changing tails of a novel short interspersed element in Aedes aegypti: genomic evidence for slippage retrotransposition and the relationship between 3' tandem repeats and the poly(dA) tail. *Genetics*, 168(4):2037–47, 2004.

[31] G. Yang and T. Hall. MAK, a computational tool kit for automated MITE analysis. *Nucleic Acids Res*, 31(13):3659–65, 2003.