# Finding and Characterizing Repeats in Plant Genomes

Jacques Nicolas, Pierre Peterlongo, Sébastien Tempel

# Finding and characterizing repeats in plant genomes

**Jacques Nicolas**[1 1] **, Pierre Peterlongo**[1] **and Sébastien Tempel**[2]

[1] Irisa/Inria Centre de Rennes Bretagne Atlantique,

Campus de Beaulieu, 35510 Rennes cedex, France

[2]LCB, CNRS UMR 7283

31 Chemin Joseph Aiguier - 13402 Marseille cedex 20, France

**Abstract:** Plant genomes contain a particularly high proportion of repeated structures of various types. This chapter proposes a guided tour of available software that can help biologists to look for these repeats and check some hypothetical models intended to characterize their structures. Since transposable elements are a major source of repeats in plants, many methods have been used or developed for this large class of sequences. They are representative of the range of tools available for other classes of repeats and we have provided a whole section on this topic as well as a selection of the main existing software. In order to better understand how they work and how repeats may be efficiently found in genomes, it is necessary to look at the technical issues involved in

the large--scale search of these structures. Indeed, it may be hard to keep up with the profusion of proposals in this dynamic field and the rest of the chapter is devoted to the foundations of the search for repeats and more complex patterns. The second section introduces the key concepts that are useful for understanding the current state of the art in playing with words, applied to genomic sequences. This can be seen as the first stage of a very general approach called linguistic analysis that is interested in the analysis of natural or artificial texts. Words, the lexical level, correspond to simple repeated entities in texts or strings. In fact, biologists need to represent more complex entities where a repeat family is built on more abstract structures, including direct or inverted small repeats, motifs, composition constraints as well as ordering and distance constraints between these elementary blocks. In terms of linguistics, this corresponds to the syntactic level of a language. The last section introduces concepts and practical tools that can be used to reach this syntactic level in biological sequence analysis.

**Keywords:** Repeats, Transposon, Indexing, Algorithmics on words, Pattern matching.

## 1. Introduction

A salient feature of eukaryotic genomes is the number of repeated sequences that they contain. Many processes contribute to this accumulation of genomic material. Even if the polyploidy speciation mechanism were not taken into account, plant genomes are particularly rich in copy events that explain the remarkable range of their size variation and may considerably increase this size. Indeed, the genome length world record,

$1.5x10^{11}$ DNA base pairs, is currently held by a plant, *Paris japonica*, an octoploid native to sub-alpine regions of Japan. The aim of this chapter is to propose several methods that help identify the various repeats that populate DNA sequences. The objective is more to collect and explain key concepts rather than to present an exhaustive and somewhat tedious study of the search for the various known repeat types, which may quickly have become obsolete.

In fact, due to its importance, we will describe in detail only the exploration of the major source of repeats, the transposable element (TE) super-class. Transposable elements often account for up to 40% of plant genomes and, for example, account for about 60% of *Solanum lycopersicum* and *Sorgum bicolor*, and 80% of the *Triticum aestivum* or the *Zea mays* genome sizes. A whole section is devoted to practical methods that have been developed to extract TEs. This includes both generic tools and tools tailored for a particular class of RNA or DNA transposons. Another important class of repeats in some plants, such as *Olea europaea* (1), the tandem (or satellite) repeats, are not treated specifically, although some pointers to *ab initio* methods for transposable elements, such as RepeatExplorer (2), may be useful for the search of tandem repeats on reduced sets of genomic reads. We refer the interested reader to a fine review on this topic in (3).

We then propose a more advanced section on the algorithmic basis of the detection of copies. This will help the reader to gain a better understanding of the technical terms often used in the description of the previous tools. All these software solutions derive much of their power from a crucial step, which finds all exact multiple occurrences of

words in a sequence. We introduce the state -of -the -art data structures that are used for this task and describe how approximated repeats can be searched for efficiently from these. In all cases, we provide pointers to free available software. Next Generation Sequencing (NGS) data are tending to predominate the current technology and need to be addressed specifically. Even if NGS avoids some issues with repeat cloning in BACs, sequencers have a certain bias that can affect the search for repeats. For instance, Illumina GA sequencers have some difficulties with GGC repeats and long inverted repeats (4) and Roche 454 sequencers have a higher error rate on long A or T homopolymers (5). Moreover, assembly is error-prone with respect to repeats, and we discuss to what extent it is possible to work directly at the read level without requiring an assembly step. The study of repeat variations is certainly the most important challenge that is currently addressed in advanced research on repeats by using NGS data, and the section ends with a small discussion on this subject.

We end the chapter with a more prospective section, which emphasizes a global and long-term approach that may be very useful in the systematic study or the discovery of particular repeat classes. It is motivated by the fact that as knowledge on repeat families grows, software solutions have to be adapted to take into account such knowledge, at an increasing cost. The linguistic approach is based on the design of some specific language for the description and search of complex chaining structures, which may occur in nucleic acid sequences as well as in protein sequences. In this paradigm, the description of the state-of-the-art knowledge is left to the biologist acting as modeler.

Each model written in the representation language can then be searched in a systematic way in the input data, using generic software, a parser, which is able to recognize the occurrences of any pattern written in the language. Of course, the more expressive languages are more desirable but this has a computational cost, which may even be beyond the reach of any computing system if the language and models are not designed carefully. A first subsection recalls the basic aspects of the theory of languages that clarify these computational limitations. A second subsection introduces some practical tools that may be experimented with for this modeling approach.

## 2. Detecting transposable elements in plant genomes

Many tools have been developed to search for transposable elements, forming a representative set illustrating the variety of techniques that can be applied for the search of genomic repeats in general.

The most common approach used to detect TEs is by homology to already-known TE sequences. This means that the search starts from an existing database of identified TE families and new elements are classified in the family containing the most similar sequence. In practice, this is often done on protein coding sequences (e.g. reverse transcriptase) since reaching a significant level of similarity requires a sufficient conservation level throughout evolution. In all cases, it assumes that a database of known TEs is available. A well-known generic database for eukaryotic consensus repeated sequence models is Repbase Update (6), maintained by the Genetic Information Research Institute (GIRI, Mountain View, CA). The current version, at the date of this publication (Jul. 2014), contained 8,300 loci related to the Viridiplantae clade. Some more specific databases exist. In Table 1, we list a few plant transposable element databases that cover several plant genera. Two of them cover all kinds of repeats that can appear in genomes, the most complete to date being mips-REdat.

A second approach for detecting TEs, called *structure-based* (7), tries to be less dependent on known sequences by taking advantage of a priori knowledge of

characteristic features of transposon families. For instance, LTR sequences have been analyzed for more than twenty years and a number of common elements have been discovered in their architecture: a short direct repeat string marking the insertion and flanking the 5' and 3' extremities of the LTR, a similar TG..CA box at each extremity, a polypurine tract about 12 bp in length, and various protein domains. All these features can serve as specific constraints enabling the detection of new elements in genomes. They usually form the basis of specific programs or scripts. We will see in the last section that this constrain solving problem may also be seen as an instance of a general pattern-matching issue, which could be treated by generic programs (parsers).

The most complex approach for detecting TEs is *ab initio*, without any assumption of the type of transposon being looked for. Typically, this is used at large scale to annotate new genomes. For this reason, *ab initio* methods make use of the most advanced data structures and string algorithms.

Of course, it is sometimes difficult to clearly state the status of a particular method, which may incorporate several steps belonging to different approaches, for instance, *ab initio* and by homology. We have created a fourth category, pipeline, to account for these more complex frameworks which implement a workflow combining existing software components with their own specific glue code and possibly new contributions.

Most detection methods are listed on the Bergman website[1]. Previous reviews classified the software according to the detection method (7) rather than detected transposable element superfamilies. This is a valuable approach for Bioinformatics and for whole-genome annotation and we start with a comprehensive subsection on this issue of systematic TE search in a newly-sequenced genome. Since biologists are often primarily interested in certain TE superfamilies, looking at methods as a second step, we propose, in the remaining subsections, different algorithms targeting specific transposable element (TE) families and then subdividing their presentation by the method itself (*ab initio*, *homology-based*, *structured* and *pipeline* methods). Our chapter focuses on software that is currently available via code downloading, a website or by email. For each TE class and each type of method, a table summarizes the corresponding list of software and provides for each a reference, a download site and installation requirements.

[1] http://bergmanlab.smith.man.ac.uk/?page_id=295

### 2.1. Large-scale search of transposable elements

First, we present tools that detect all TE superfamilies. These methods correspond to the majority of the TE detection tools. They generally use sophisticated algorithms to ensure an efficient search of elements and the interested reader is referred to Section 3 for having more details on the way it works on computers and having the possibility to compare related techniques.

**All TEs; *ab initio* methods**

Available *ab initio* methods are listed in Table 2. They can be roughly split into two categories: methods that detect *exact repeats* of fixed size and assemble or extend them, and methods that directly detect *non-exact repeats*.

Except for Tallymer (8) and RepARK which detect a range of exact repeats, all methods in the first group detect exact repeats of length *k* called *k*-mers. These repeats are stored in various data structures offering a tradeoff between the required amount of memory and the amount of computation. These repeats are stored in various data structures offering a tradeoff between the needed amount of memory and the amount of computation. Reputer (9) and RepeatFinder (10) use the suffix tree data structure for storing and retrieving *k*-mers, while Tallymer (8) uses the more compact suffix array

data structure. A number of methods such as WindowMasker (11) use a structure that does use the fact that elements are strings, a hash index (not to be confused with h-index… !). RepARK counts *k*-mers up to *k*=31, making use of the Jellyfish software (12), which is based on a multithreaded, lock-free hash index. PClouds (13) uses a bit array and a hash index. A software such as RepSeek (14) uses a lower and an upper bound for *k* and keeps only the maximal repeats, those that cannot be extended without losing an occurrence.

Available software solutions offer various alternatives regarding the choice of *k*, some more practical than others depending on the data set: it is most often a parameter whose value has to be user-provided (ReAS, Repseek, REPuter and RepeatScout); it equals $log_4(n)+1$ for PClouds (where *n* is the sequence size); it is the smallest integer that satisfies the equation $n/4^k < 5$ for WindowMasker. In the case of Tallymer, it is an optimal value calculated from a user-provided range fixing the minimal and maximal values. To be more precise, Tallymer calculates this value from the *uniqueness* ratio, which is the ratio of *k*-mers occurring exactly once relative to the total number of *k*-mers in the genomic sequence. Then the selected *k* corresponds to the least value where the ratio does not increase significantly with the increase of *k* (inflection point in the uniqueness ratio graph). All repeats greater than *k* are written out by Tallymer. RepARK assumes a Poisson distribution for the *k*-mers unique in the whole genome

and thus determines a frequency threshold to filter significant *k*-mers that occur at least twice in the genome.

Unlike other methods that detect repeats on genomic sequences, ReAS and RBR work on Whole Genome Shotgun (WGS) and Expressed Sequence Tags (EST) data respectively. ReAS selects WGS that contains *k*-mers with a high copy number and divides them into 100-bp segments centered on that *k*-mer. ReAS clusters the WGS containing the same *k*-mers and creates the consensus sequence of the 100-bp segments. In the RBR method, *k*-mers are considered as repeats if their frequency is higher than a calculated threshold based on a binomial distribution model.

The next important feature for comparing these tools is the way they build approximated repeats from exact ones. Dynamic programming is a widely used method for this purpose, although it is not the only one and the way it is applied may vary.

Tools such as REPuter, RepSeek and RepeatScout use *k*-mers as seeds and try to *extend them on both sides*. REPuter allows a fixed number of mismatches and uses a dynamic programming algorithm. For each repeat found, an *E*-value score is computed using the Kurtz and Myers procedure (15) which corresponds to the significance of mismatch repeats. REPuter also possesses a graphical interface to display the position of copy pairs along the genome, something that may help provide a global view of

duplicated material between chromosomes. RepSeek also extends the *k*-mer seeds by a dynamic programming approach but accepts more errors (substitutions or indels). Finally, RepSeek calculates for each extended repeat a score based on its length and nucleotide composition. The probability for a given repeat score is computed by estimating the probability P(Sbest_repeat ≥ S) that the score of the best local alignment observed between two random sequences of fixed size is larger than a given score S and is expressed as a function of the sequence length and the GC ratio. A minimum threshold score is derived from this probability above which no repeats are expected to be found in random sequences and only repeats with a score higher than this threshold are kept. RepeatScout uses a greedy algorithm to extend the exact repeat to the right and to the left and build a consensus. For each extended nucleotide, the algorithm calculates a score that is computed by summing up local similarities of each sequence with the consensus and uses an incomplete-fit penalty for sequences that partially match the consensus. The extension is stopped when the algorithm finds a locally optimal value value that does not increase after a fixed number of steps (100 by default). RepARK uses a de Bruijn graph assembler for NGS data, Velvet (16), to get the repeat library.

The previous strategy can be enhanced by specific preprocessing treatments. PClouds first excludes tandem repeats. It then clusters similar *k-mers* (called '*P-cloud core*') and extends them according to empirical user thresholds (13).

Another approach considers that every relevant approximate repeat is made of a *mosaic of exact repeats*. RepeatFinder merges exact repeat pairs if the gap distance between the repeats is smaller than the maximal allowed gap size (a user parameter to be defined in the command line), or if one repeat overlaps another by more than 75%. RepeatFinder then clusters the merged repeats that have a same maximal repeat or a common hit with an E-value lower than a user-specified parameter, using WU-BLAST (17).

ReAS starts from the consensus sequence of 100-bp segments built on WGS clusters containing the same *k*-mers. The ReAS algorithm recursively extends the consensus sequence to the left or/and to the right if another 100-bp consensus sequence overlaps it.

Finally, WindowMasker (11) aims solely to *mask low complexity regions* and repeats in genomic sequences and it does not attempt to annotate or classify them. WindowMasker screens the sequence twice for processing repeats: the first screen computes the frequency of each *k*-mer (for the direct and reverse strands) and defines several thresholds corresponding to various percentiles of the empirical cumulative distribution of repeat occurrences; the second screen uses a frequency-based score and masks *k*-mers whose score exceeds the 99.5% percentile or is between two masked k-mers and has a score exceeding the 99% percentile.

In the second group of *ab initio* methods, no index is built and repeats are directly detected through the self-alignment of genomic sequences. These methods are thus heavily reliant on dynamic programming and their sensitivity depends on the way in which significant aligned pairs are filtered. PILER (18) and RECON (19) proceed by first aligning each sequence against itself and then, in a second step, selecting repeats that present a high copy number or a score higher than a fixed threshold.

In more detail, RECON 1) creates a graph *G(V,E)* such that vertices *V* correspond to subsequences involved in WU-BLAST pairwise alignments and edges *E* represent overlapping vertices in a global alignment; 2) removes from this graph sequences that do not overlap a significant number of times with sequences belonging to the same cluster of sequences; 3) groups elements that belong to the same repeat family. RECON then creates a new graph *H(V',E')* for each family such that a vertex *V'* corresponds to a RECON element and each edge *E'* corresponds to the overlap between two elements in a global alignment. A RECON repeat family is attributed to each repeat position in genomic sequences.

From the alignment, PILER defines a *pile* as a list of pairwise hits covering a contiguous region in sequences. PILER saves only hits that cluster with more than *p* instances (*p* is a user-defined parameter). An interesting original feature of PILER is that it distinguishes different categories of repeats that correspond to different pairwise hit definitions (or repeat builds) and implements a specific method for each category. Thus,

PILER-DF looks for *intact transposable elements* and aligns at least three similar piles (not hits) to create a repeat, with this pile alignment avoiding generating fragmented sequences. PILER-TR searches *TEs with terminal repeats* (LTR retrotransposon) and aligns *banded* hits (i.e. hits separated by a maximum distance) to create the repeats.

*AAARF (*Assisted Automated Assembler of Repeat Families (20)) is a simple Perl script (downloadable at aaarf.sourceforge.net/) that is representative of methods oriented towards the use of incomplete genomic information collected through large sets of reads. It has been tested with success on the maize genome, either with a sample of the TIGR Sanger reads (780 bp on average) or with simulated 454 reads (100 bp on average). It is most useful for trying to detect repeats from the short 454 reads, a widespread situation for biological labs. It is possible to tune a number of parameters using BLAST (21) and ClustalW. Of course, it has some limitations since it is a purely *de novo* method working on highly fragmented data and it should be used with care particularly when discriminating families of degenerated repeats, such as a series of tandem repeats or a mix of non-autonomous and autonomous transposable elements.

**All TEs; Homology-based methods**

Available *homology-based* methods are listed in Table 3. Most of them make direct use of the BLAST software (Altschul et al., 1990) and a BLAST-parser. Currently, these

methods used three BLAST versions: AB-BLAST (22), NCBI-BLAST (21) and PSI-BLAST (23). The BLAST method is a famous and widely-used seed-based approach that looks for the presence of a query from its *k*-mers and estimates a E-value for the significance of matches. Its principles are recapped in Section 3. For all homology-based methods, the query sequence is a library of consensus sequences from TE copies.

*AB-BLAST* is the new and commercial version of WU-BLAST (no longer supported) and is only free in a limited version available to academic users. The input and output files and the binary names are identical to WU-BLAST. Because there are so far no papers published and no source code, it is not possible to describe the algorithm of this new version in further detail. *NCBI-BLAST* is a widely-used BLAST algorithm variation. By default, *k* is set to 11 for the DNA (or RNA) and to 3 for the proteins. PSI-BLAST is another version from the NCBI laboratory which is much more sensitive in picking up distant evolutionary relationships than a standard protein-protein BLAST (23).

A *BLAST parser* is a method that reads the BLAST hits and tries to assemble them to obtain summarized results. Since large scale analysis may produce an amount of results that can reach the amount of input sequences, such a component has a decisive role in practice. It determines the type of annotations that can be expected as a result. The language used to program the parser is an indication of its type of usage: Censor (24), RepeatMasker (25; 26), TESeeker (27), TransposonPSI (28) and RelocaTE (29) are written in Perl and are routines that can be included in more complex workflows; TARGeT (30)

is written in PHP as is intended for use only via a website. Note that, independently of languages, and perhaps because it may be a tedious task for biologists, some authors give only a very brief account or do not write out their algorithm in the corresponding paper, something that is not favorable for the interpretation of results and rational improvement of software by the community. For example, the authors of RepeatMasker did not write a paper about it, although it is possibly the most widely-used software for TE detection. *TransposonPSI* includes a library of ORFs coding for transposon proteins and uses PSI-BLAST to detect intact coding TEs and TBLAST for slightly degenerated coding TEs.

To be more precise, *TARGeT* (Tree Analysis of Related Genes and Transposons) is a webserver that does not only detect TEs in genomic sequences but also establishes a phylogeny of these elements. It first uses NCBI-BLAST to find the TE copies, aligns them with MUSCLE (31) and determines consensus with the TE family. Finally, it launches FastTree (32) to create the phylogenetic tree of the TE copies. *Censor* is a BLAST-parser that can use AB-BLAST or NCBI-BLAST. Censor first launches DUST and NSEQ to remove the micro- and mini-satellites from the genomic sequences. From the BLAST hit positions, it tries to assemble them based on the TE consensus families available in the RepBase database. Censor finally calculates for each match the new similarity score and a hit score. The software *TESeeker* detects preferentially autonomous and complete transposable elements, in three steps. In the first step,

TESeeker uses NCBI-BLAST to find the partial hits of the transposable element copies, assembles them using CAP3 (33) and creates new consensus using ClustalW (34). Hits with *E*-values higher than $1x10^{-20}$ are eliminated. In the second and third steps, TESeeker iterates the same method with NCBI-BLAST, CAP3 and ClustalW, but starting from the TE consensus library created during the previous step. *RelocaTE* is a list of Perl scripts aiming at the identification of given reference TE in NGS short reads (paired or unpaired). It produces the locations of TE insertions that are either polymorphic or shared between the reference and short reads. The identification is based on three tools, BLAT, Bowtie and SAMtools. RelocaTE has been used to characterize the amplification of mPing in *Oryza sativa* (29).

Like Censor, *RepeatMasker* first detects and removes the micro- and mini-satellites, by applying Tandem Repeat Finder (TRF) (35). RepeatMasker is very flexible and can launch many BLAST-like software solutions: cross-match (36), NCBI-BLAST (21), AB-BLAST (22), or Decypher (37). The NCBI laboratory has made available a special BLAST version for RepeatMasker called RM-BLAST (38). RepeatMasker identifies the TE superfamily of the fragmented BLAST-like hits and assembles them using methods tailored to each superfamily.

RepeatMasker is a base component used in many applications. There are (at least) four other methods that read the RepeatMasker hit outputs and re-assemble them: Process_hits (39), REannotate (40), RepeatRunner (41) and One code to find them all

(42). *Process_hits*, a Perl script that can also read a BLAST output, processes a refined output using various formats depending on the user parameters. *REannotate* defragments RepeatMasker outputs following certain rules: the two fragmented hits have the same orientation; the distance between the first and the last fragment must not exceed a user-specified parameter (default 40kb), and the overlap between two fragmented hits must exceed 10% of the reference length sequence. *RepeatRunner*, written in Perl, mainly detects the autonomous transposable elements (those that contain an ORF). RepeatRunner first launches BLASTX (22) and RepeatMasker, merges the results of both in a XML-based output and eventually masks the repeats in the input genomic sequence (like RepeatMasker). *One code to find them all* assemble RepeatMasker hits into complete copies, retrieve corresponding TE sequences and flanking sequences, and compute summary statistics for each TE family.

We end with three other methods that do not use BLAST-like output to detect transposable elements but can nevertheless be related to the homology-based methods, RetroSeq, T-lex and HMMER.

*RetroSeq* (43) and *T-lex* (44) detect TEs in next-generation sequencing (NGS) reads and are described for this reason in Subsection 3.3. *HMMER* (45) is a widely-used pattern-matching and discovery software package which creates and searches profile Hidden Markov Models (profile HMMs) against a sequence. It was not specifically designed for

TE detection. Three steps are involved in looking for a transposable element belonging to a fixed family using HMMER. One must first align the known copies of the TE family (HMMalign) and then create the HMM profile of this family (HMMbuild and HMMcalibrate). The final step consists in scanning the genomic sequence with the HMM profile as a model (HMMsearch).

**All TEs; Structured methods**

To our knowledge, three *structured* methods have been applied to detect transposable elements, SMaRTFinder and SMOTIF for the Copia retrotransposons in *A. thaliana* (also with SMaRTFinder) and STAN used for Helitrons in *A. thaliana*. This is the reason for citing them here although these methods are not dedicated to the recognition of transposable elements. The corresponding tools are listed in Table 4. They are generic and can be used to detect any biological pattern. This is the subject of Section 4.2, where more complete parsers such as Vmatch are described.

Apart from SMOTIF (46) , SMaRTFinder (47) and Stan (48) use the suffix-tree data structure. SMaRTFinder and SMOTIF first detect all positions of each element of the motif and join the positions that satisfy the distance between the elements of the motif. STAN, on the other hand, creates a list of the possible sequences from the user motif and looks for each of them. Only STAN can detect motifs with substitution errors and

non-fixed gaps. STAN is no longer maintained and a more expressive tool, Logol, can be used instead. It is also described in Section 4.2.

**All TEs; Pipeline methods**

We end this section with the most elaborate tools, which chain several methods to enhance the repeat annotations and are listed in Table 5. Apart from DAWGPAWS (49), which uses the three kinds of method, RepeatModeler (25), REPET (50), TriAnnot (51) and RISCI (52) use *de novo* and *homology-based* methods to detect TEs. All these programs are meta-tools: they launch other software and assemble and rewrite their results. Among TE detection software, only RepeatMasker (26) is used by all these pipelines.

In more detail, *DAWGPAWS,* written in PERL, launches LTR_STRUCT (53), LTR_seq (54), LTR_FINDER (55), FINDMITE (56), Find_LTR (57), TRF (35), Repseek (14), RepeatMasker, HMMER, TE_Nest (58), and BLAST. DAWGPAWS assembles and rewrites the output of these tools. After removing the tandem repeats with TRF (35), *RepeatModeler*, also written in PERL, uses first RECON (19) and RepeatScout (59). Finally, RepeatModeler launches after RepeatMasker with a library of consensus sequences to identify and assembles the previously detected hits. *REPET* is a sophisticated method composed of two main pipelines. The first pipeline (*de novo*) compares the genome against itself using BLASTER (a BLAST-like method written by

the authors (60)). Then, hits are clustered with RECON, PILER (18) and GROUPER (also written by these authors (60)). A consensus sequence is created by multiple alignment [MAFFT (61) and MAP (62)] and then classified. The second pipeline (*homology-based*) looks for repeats using a library of sequences, for example the created consensus, with RepeatMasker, CENSOR (24) or BLASTER (60). This pipeline also uses TRF and mreps (63) to detect micro- and mini-satellites. *TriAnnot* (51) is an annotation pipeline which first detects transposable elements and then predicts the other genetic elements. It launches BLASTX (21), RepeatMasker, TEAnnot (50) (*homology-based*), and Tallymer (8) (*ab initio* method). Finally, *RISCI* (Repeat Induced Sequence Changes Identifier) is a set of Perl scripts specialized in the comparative genomics of transposons. Starting from a reference genome and comparative genomes, it is able to infer intra-species and inter-species structural variations introduced by transposons (e.g. Target Site duplication, inversion and truncation of repeat sequence or post insertion modifications like disruption) . This pipeline uses RepeatMasker, Blast and the EMBOSS module and the Genbank annotation file if made available.

### 2.2. LTR retrotransposons

Concerning the search and analysis of specific transposable elements, LTR retrotransposons (LTRR) form a large family and offer a rich structure that justifies the

existence of several dedicated tools. An overview of the most interesting ones is provided in Table 6.

#### Homology-based methods

*LTR_MINER* (64), written in Perl, is the only homology-based method specialized in LTR retrotransposon detection. It launches first RepeatMasker and WU-BLAST, and then assembles the multiple hits of the Long Terminal Repeat (LTR) (e.g. the extremities of the retrotransposon) under some constraints: a maximal distance between hits (550 bp), a common orientation, a same LTRR family, and a combined length no larger than a complete LTRR. LTR_MINER also gives information about the probable age of the LTR retrotransposon insertion.

#### Structure-based methods

LTR_FINDER (55) and LTRharvest (65) detect only full length LTR retrotransposons while LTR_STRUC (53) and MGEScan-LTR (57) detect all LTRs. Aside from LTR_STRUC, LTR_FINDER, LTRharvest and MGEScan-LTR use the suffix-array data structure to find the exact maximal repeats in the genome, and extend/merge them into non-exact repeats. The three methods look also for LTRR signals such as Target Site Duplication (TSD) and PBS and PPT retrotransposon motifs.

In more detail, *LTR_FINDER* (55) is a web server that merges exact repeat pairs with a similarity score higher than a minimum extension threshold. LTR_FINDER combines dynamic programming (Smith-Waterman algorithm) and the search of structured motifs like the TG..CA box and TSD, in order to adjust LTRR candidate extremities. Next, LTR_FINDER looks for LTRR motifs such as the PBS and PPT signals (by aligning the LTR retrotransposon candidate with the 3' tail of tRNAs) or LTRR protein motifs (using ScanProsite on protein domains such as IN and RH). *LTRharvest* first builds the suffix array of the genomic sequence using GenomeTools (66) and stores all maximal repeats that are longer than a user-defined threshold. Optionally, LTRharvest looks for motifs such as the TSDs, which can be derived from the maximal repeat occurrences, and the palindromic LTR motif, which corresponds to the dinucleotide palindrome in the LTR extremities (often TG with CA). Finally, LTRharvest checks candidates that have the TSD site, together with some constraints on the size and similarity of the two LTRs and the distance between them. *LTR_STRUC*, written in C++, first identifies similar pairs subject to a set of constraints: common matching size (larger than 40 bp), similarity (higher than 70%), and distance (lower than a user-defined parameter value). In a second step, LTR_STRUC tries to extend from the initial pair to a second pair in the 3' direction, then the 5’ direction. The extension proceeds by looking in neighboring regions of fixed size (100 bp) for a largest match pair that can be aligned at similar distances from the previous ones and greedily produces the alignment of the whole region by filling the gap by largest matches. This extension process continues until the

similarity falls consistently below the 70% threshold. A last step determines progressively the exact termini of the LTR retrotransposon by calculating the number of matches in a sliding window. Finally, *MGEScan-LTR* (57), written in C++ and Perl, uses an algorithm similar to LTR_seq (54) (not available) and LTR_STRUC. In a first step, MGEScan-LTR finds all maximal repeat pairs longer than 40 bp and within a range of distances (between 1000 and 20000 bp). Two exact pairs are merged if they are close (less than 20 bp) and share a similarity greater than 80%. In a second step, the method scans the ORFs inside the LTR candidates using HMMER (45), and removes the candidates that match with DNA transposons. In the third step, MGEScan-LTR looks for solo LTRs. It clusters the LTR discovered previously and aligns them to create a profile HMM of each cluster. HMMsearch, a procedure of HMMER, is used to discover these solo LTR retrotransposons.

**Pipeline methods**

MASiVE (67), written in Perl, is typical of the methods based on carefully designed specific models: it only detects full autonomous LTR of the plant-specific Sireviruses and takes advantage of the highly conserved motifs they share for a sensitive and accurate search. It uses Vmatch (68) - see Section 4.2 - to detect clusters of Sirevirus-specific PPT motifs, and LTRharvest (65) - see *Structured methods* - to detect complete LTR retrotransposons. Only candidates that possess both hits, the right PBS site, an

admissible distance between the different elements and include LTRS and an internal sequence longer than 500 bp are retained. Wise2 (69) – see *Homology-based methods* - is used to detect the reverse transcriptase (RT) present in almost intact LTR retrotransposon.

### 2.3. Non-LTR retrotransposons

The two methods that detect non-LTR retrotransposons first launch homology-based tools, via Perl language scripts, and also look for non-LTR motifs to confirm the TE candidates. They are listed in Table 7.

*MGEScan-nonLTR* (70) uses dedicated HMM profiles and the pHMM module from the HMMER package (45) to detect autonomous non-LTR retrotransposons. These profiles correspond to the apurinic/apyrimidinic endonuclease APE, the linker and the Reverse Transcriptase protein domains. MGEScan-nonLTR then classifies the candidates into one of the 12 known non-LTR clades.

*RTAnalyzer* (71) launches BLAST to detect all non-LTR matches (autonomous and non-autonomous). From the hits, RTAnalyzer launches Matcher (72), which determines precisely the 5' extremity of the non-LTR retrotransposon. From the corresponding 5' Target Site Duplication (TSD), the algorithm extracts the 3' TSD. Finally, RTAnalyzer

determines the polyA tail, calculates a score from all these motifs and saves the non-LTR candidates that have a score higher than a threshold.

## 2.4. DNA transposons

DNA transposons are made of a transposase gene flanked by Terminal Inverted Repeats (TIRs) that make structure-based methods the best adapted for an efficient search. The main difference between methods is the way in which TIRs are selected. All methods are listed in Table 8.

### Homology-based methods

To our knowledge, there exists only one method specialized in homology-based detection of DNA transposons : *TRANSPO* (73). It implements a fast bit-vector dynamic programming algorithm (74) that finds the position of all matches similar to a given sequence in a library. The set of matches are then clustered using the SPAT program (75).

### Structure-based methods

Five methods use the palindromic structure of their 5' and 3' extremities to detect DNA transposons: Inverted Repeats Finder (IRF) (76), MITE Uncovering SysTem

(MUST) (77), Repetitive Sequence with Precise Boundaries (RSPB) (78), MITE-HUNTER (79) and MITE Digger (80). They all look first for the Terminal Inverted Repeat (TIR), and all but IRF also look for the TSDs that flank the TIR, created during the insertion of the element by a staggered cut in the target DNA.

In more detail, IRF searches TIR candidates that contain exact short inverted repeats (4 to 7 nt) that do not overlap, then extends them, calculates an alignment score for a larger palindromic arrangement and saves the candidates that obtain values higher than the thresholds for similarity (70%) and length (25 bp). *MUST* detects the TIR associated with the TSD (the minimal and the maximal length of TIR and TSD are user-defined parameters) from a sliding window (up to 500 bp). Candidates with a score higher than a similarity threshold are conserved. MUST then clusters them using the MCL algorithm (81) and writes out the MITE families that contain at least 3 occurrences. *RPBS* is a series of Perl scripts using a Blastn-based approach. It seems to have been essentially applied to non-autonomous transposons of the Miniature Inverted Transposable Elements family (MITE), but the core of its approach could be applied to a broader set of families. The principle is to build clusters of repeats (at least 5 elements, less than 1500 bp) sharing high similarity (*E*value < $10^{-15}$) and having precise boundaries (maximum 5 bp variation and dissimilar 100-bp flanking sequences <50% identity). This software is known to be resource-intensive (several days of computations for large genomes). MITE-HUNTER and MITE Digger have been designed exclusively

to detect MITEs. In *MITE-HUNTER,* MITE candidates (with their flanking regions) are compared with each other by BLAST (21), and MITE candidates that have similar flanking regions are considered to be part of a larger repeat element and are removed. MITE-HUNTER clusters the remaining candidates and defines a representative element (exemplar) for each cluster (an element that has the greatest similarity with all other elements). BLAST is used to detect its homologs in the input sequences. Candidates that have many copies are then aligned. Homologs such that flanking regions of the MITE sequences are similar (>60%) for the majority of occurrences are assumed to be false positive and discarded. The TSD is predicted again for the remaining candidates for better accuracy. The program creates a consensus sequence from the clustered homologs, compares all consensuses (using BLAST) and clusters them again in order to reduce the number of clusters. *MITE Digger* is another BLAST-based program designed to scale large genomes by making use of the fact that a MITE family typically contains several hundred highly similar copies that are scattered all over the genome. Rather than repeating for each copy TIR and TSD signature identification, screening, multiple sequence alignment and clustering, the search is first focused on a small portion of the genome and a possible family is filtered out as soon as the number of found copies reaches a threshold. For instance, the probability is very low (1%) of missing 50 copies in 8% of a genome database. Low-complexity regions and hits with very similar flanking regions are discarded from sequences and the process is iterated.

### 2.5. Helitrons

There seems to be only one method to date specializing in helitron search: HelSearch (see table ) (82) (see Table 9). This is based on the search for a helitron signature in the 5' and 3' extremities. HelSearch looks for the *CT[AG][AG]T* motif (3' termini) of the helitrons and the small hairpin structure (a 3' sub-termini with a *GC*-rich sequence) calculated using UNAFOLD (83). It then determines the 5' end (which contains the dinucleotide *TC*) from the multiple alignment of the potential hits. Finally, HelSearch classifies helitrons based on their 3' end similarity and uses BLAST to detect the fragmented helitrons in the input sequences.

## 3. Efficient search of repeats in genomic sequences

Some common background appears in all the tools we have seen so far: in most cases, even if it seems a little remote from reality, programs are looking first for *exact* repeats, often referred to as anchor points or *seeds*. The real, approximated repeats are obtained by extending the search further from these seeds, by introducing elementary operations

enabling the detection of copies with distant contents. Looking for exact and approximated repeats is part of the rich domain of string algorithms. In the most general setting, detecting repeats in genomic sequences, or in any kind of textual sequence, is organized on the basis of the detection of particular words (the queries**)** in a text (the bank). This setting appears clearly when processing alignments in homology-based methods or looking for particular motifs in structure-based methods.

There are basically two ways of performing such detection.

In the first case, the text is not preprocessed and the *query is searched on the fly* in the text. For instance, this is the case when using the well-known "Ctrl-F" shortcut key or search function on any piece of software dealing with texts (text editors, web browsers, etc.). Beginning in the early 1970s, there are strong and interesting theories about fast detection on the fly of queries in texts. This is known as *string-matching*. The naive way to achieve string-matching is to compare at each position in the text the presence of each character of the query, starting from this position. Obviously, if $n$ is the size of the text and $m$ is the size of the query, this would require $n.m$ comparisons. Thinking of large texts (several gigabytes for a genome), and even for relatively small queries (say 1000 bp), this quickly becomes an issue. It is possible to decrease this complexity for a linear behavior, although it will always depend of the size of the bank. We will not address the topic in this chapter but the interested reader can refer to the excellent book from Charras & Lecroq (84), which explores string-matching algorithms exhaustively and in a

didactic way. A more general approach, pattern-matching, is discussed in the last section (linguistic analysis).

Given their huge volume, the treatment of genomic sequences is generally based on the second way of performing text processing: the *text is preprocessed by indexation techniques.* In the previous section, many technical terms have been used, such as suffix trees and suffix arrays. These are structures developed for fast indexation and searching in texts. Research in this area has been fostered by developments in genomics and in Internet content querying and we propose a quick but up-to-date review of the key results achieved to date. We focus on indexing techniques which open the way both to fast queries and to repeat detection.

### 3.1. The art of indexing

Imagine what would happen without indexation while looking, for example, for a word (query) on the Internet (bank). Your favorite search engine would open and then read all words of all pages of the whole web in order to detect the presence/absence of your query in each page. Estimated to a few tens or hundreds of billions of webpages, querying the web would simply be impossible, whatever the computation resources available. The same would be the case if the bank is a large set of genomic sequences, such as Genbank. For an efficient search, indexes are necessary. An index is a mechanism that can answer the question: "where does this word occur in the bank?"

within an amount of time that *does not depend on the size of the bank*. To be precise, indexes are built once for a given bank (and within a time proportional to the size of the bank) and then the time it takes to perform every query in the bank depends only on the query size. Happily, indexes also have "side effects" which are valuable in the biological context, such as the efficient detection of repeats inside or between genomes.

The main idea is to use *data-structures* to organize and structure the information initially present in the banks. A classic and well-known structure is the dictionary: searching for a word in a paper dictionary does not require reading the entire book, as words are organized in lexicographic order. Similarly, the data-structures used for indexing texts are computational objects having the same advantages as classic dictionaries: they enable a portion of a sequence to be searched quickly by avoiding all portions where this query could not occur. However, while genomes can easily be seen as particular texts, the notion of "word" is not natural in this context since no general delimiter exists. This is why indexes used in this context are designed to answer questions regarding *any word* in the bank. For example, consider a sequence *S*=ATGCGCAGTTTAT as a bank, and a query *P*=GCGC: "does *P* occurs somewhere in *S*, and, if yes, where?" In this example the answer would be "*P* occurs at position 3 in *S*". From the point of view of strings, positions are associated to *suffixes* of the text, that is, words starting at this position and ending at the end of the text. For instance, position 3 in *S* is associated with the word GCGCAGTTTAT (and *P* is a *prefix* of this word, that is, it is

placed at its beginning). There are a number of indexes primarily powered by this important notion of the suffix: automaton, hash tables, suffix trees or suffix arrays. All of these data-structures are designed to answer at least the fundamental question "where does *P* occur in *S*?", but each has some specific skills either in terms of additional possibilities or in terms of performance (time efficiency and memory footprint).

Here we propose focusing on the most well-known and most widely-used data-structures: *suffix trees*, an "historical" data-structure which is no longer extensively applied in practice but which, in addition to useful educational and theoretical properties, also affords the most and the most flexible functionalities; *suffix arrays*, an efficient and elegant way to index large datasets and answer additional useful questions such as those related to exact repeats in a bank; and, lastly, the *Burrows-Wheeler transform* on which the compressed *FM-index* is based and which is a development that makes it possible to build compressed suffix arrays. Note that in all cases, an efficient program code appears quite short but is in fact very tricky and definitely not within a standard programmer's reach: use well-written libraries!

## Suffix trees

Suffix trees are represented by a classification tree structure that clusters the suffixes of the indexed text. All suffixes having a common prefix are clustered in the same class

under a common internal node that has this prefix as a label. Any suffix can be read along exactly one path from the unique root of the tree to one of its leaves and conversely, any path from the root to a leaf corresponds to a unique suffix of the text or, equivalently, to a position in this text. Figure 1 shows an example of the suffix tree of *S=ATTTAATAAC.* The suffix at position 8 in the text, AAC, can be read from the root to the leftmost leaf labeled *8:C* by collecting all the words along the path, A, A, and C.

There are strong theories and beautiful algorithms for constructing suffix trees quickly and with the lowest memory requirements. The most famous include the Weiner (85) and later the Ukkonen (86) algorithms which both propose a way of constructing the suffix tree of a sequence composed of *n* characters with a number of operations and a memory requirement proportional *to n*. To be precise, storing the suffix tree for a genomic sequence whose alphabet is limited to letters *A, C, G* and *T* requires in the worst case 20 bytes per indexed character in optimized applications (87). Indexing the human genome therefore requires 61.47 GB of memory. Even if the actual average amount of memory required is rather around 13 bytes per nucleic acid, storing the suffix tree puts a strain on the main memory for large-scale applications. Applied, for instance, to indexing an Illumina run composed of 100 million reads of length 100 would require 130 GB of memory. In order to save space, it is necessary to shift the space/time tradeoff in the program. One option is to save the tree to hard disk rather

than the main memory but this becomes much slower to access. The most recent implementations of suffix trees have used compressed structures, to the cost of slightly slower access (88). The corresponding program is available in C++ on the website (89).

The suffix tree makes it possible to answer a query in an amount of time proportional to the length of the query. Indeed, any word *P* read in the tree of an indexed sequence *S* from its root to any one of its leaves corresponds to a word in *S*. For instance, searching for the word *ATT* in the suffix tree of *S* =*ATTTAATAAC* can be read in bold in **Erreur ! Source du renvoi introuvable.** , leading to a leaf labeled 1: *ATTTAATAAC* meaning that the word *ATT* exists at starting position 1 in *S*.

Fortunately, the suffix tree offers many possibilities other than answering such simple queries. For instance, it is of great interest for detecting exact repeats. Indeed each *internal node* of the tree (squares in **Erreur ! Source du renvoi introuvable.**) determines a word which has several occurrences. This is the case for the word *TT* in our example (path with grey rectangles in the Figure): the corresponding cluster contains leaves labeled 3 and 2, meaning that there are two occurrences of *TT* in *S*, starting at positions 2 and 3. Moreover, simple algorithms can exploit the suffix tree to find easily distinct kinds of repeats: those having the maximal number of occurrences, the longest, those that are *maximal* in their length (adding a new character to the repeat would discard some occurrences), etc. The suffix tree can be applied to more than one unique sequence, in this case it is called a *generalized* suffix tree. The generalized suffix

tree is useful in genomics as it affords, for example, the possibility of easily determining the words that are copied between different sequences or chromosomes.

## Suffix arrays

The suffix array appeared in the early 1990s with the Manber and Myers paper (90). This data-structure is much simpler than the suffix tree, requires less memory space and answers the same range of questions. It is constructed by first sorting all suffixes of the indexed text and by storing the obtained set in alphabetic order, remembering the obtained starting positions.

For instance, the set of suffixes of *S*=ATTTAATAAC is made of ATTTAATAAC (starting position 1), TTTAATAAC (pos. 2), TTAATAAC (pos. 3), TAATAAC (pos. 4), AATAAC (pos. 5), ATAAC (pos. 6), TAAC (pos. 7), AAC (pos. 8), AC (pos. 9), and C (pos. 10). Their alphabetic order is: AAC, *AATAAC*, AC, ATAAC, ATTTAATAAC, C, TAAC, TAATAAC, TTAATAAC, and TTTAATAAC. Let us represent this set of suffixes in a table:

| *Position* | *Suffix* |
|---|---|
| 8 | *AAC* |
| 5 | *AATAAC* |
| 9 | *AC* |
| 6 | *ATAAC* |
| 1 | *ATTTAATAAC* |
| 10 | *C* |

| | |
|---|---|
| 7 | *TAAC* |
| 4 | *TAATAAC* |
| 3 | *TTAATAAC* |
| 2 | *TTTAATAAC* |

The suffix array may be reduced to the *Position* column of the previous table (the suffixes are represented to promote better understanding of the structure but are never stored in the computer memory). This sole column may appear rather minimalist, but this simple piece of information and the original indexed sequence are sufficient for efficiently answering a query. In its simplest form, it can be found by applying a dichotomic procedure: i) look for the word at the middle position of the array. If the query occurs at this position the search stops, or **ii/** if the query is alphabetically smaller than the suffix starting at this position, recursively apply the procedure to the upper part of the array, or apply it to the lower array.

For instance, searching the query *Q=AC* in the previously indexed text *S=ATTTAATAAC* would entail the following steps. First check the middle of the array. In this example, it corresponds to the suffix at position 1, that is, *ATTTAATAAC*. As it is alphabetically larger than the query *Q=AC,* the process restarts with the array limited to:

| *Position* | *Suffix* |
|---|---|
| 8 | *AAC* |

| | |
|---|---|
| 5 | *AATAAC* |
| 9 | *AC* |
| 6 | *ATAAC* |

Consider the middle line of this array: it corresponds to the suffix starting position 5, *AATAAC,* which is alphabetically smaller than the query *Q=AC*. Thus the process restarts with the lower part of the array. In this case, position 9, where the suffix AC starts, corresponds to the searched query. With this simple procedure, at worse, the number of steps to be performed is log(size of the array) = log(size of the indexed sequence). The query time can be further improved by the use of a new column called the LCP, which provides the length of the "Longest Common Prefix" between two consecutive lines in the suffix array. For instance, if 0 is a LCP default value for the first suffix of the array, the two lines of the previous suffix array would become:

| *Position* | *LCP* | *Suffix* |
|---|---|---|
| 9 | *0* | ***AA****C* |
| 6 | *2* | ***AA****TAAC* |

The value 2 indicates that *AAC* and *AATAAC* start with 2 common characters. The LCP somehow enables retrieval of the suffix tree topology, and then enables answering questions related to repeats. In the example, LCP =2 indicates that a repeat of size 2 exists in the indexed text, at positions 9 and 6 given by *Position*.

The suffix array (together with the LCP array) requires (for classic genomic analyses) 5 bytes per indexed character. Thus, indexing a 3.3 billion-character human genome with a suffix array and its LCP would require 15.37GB. The indexation of a hundred million reads of length 100 would require 46.57GB of memory. A good C++ code is available in the appendix of the paper (91) (algorithm DC3) and two other C programs that are among the most advanced implementations to date are given in the appendix of the paper (92) (algorithms SA-IS and SA-DS). We mention the recent paper (93), a didactic presentation of the SA-IS algorithm for Bioinformatics specialists. Regarding access to an operational code in Java, please refer to (94). A recent development in using suffix arrays to look for repeats (finding the largest substring common to a set of sequences and finding maximal repeats exclusive of a sequence with respect to another set of sequences) is described in (95). The C code can be downloaded from (96). In fact, genomic repeats may be a source of inefficiency for certain implementations and care is required when choosing a program. A far more general use of the suffix array data-structure is described in the next section with the software Vmatch (68).

## FM-index and Burrows-Wheeler transform

The Burrows-Wheeler transform (97), inspired by the suffix array, was the starting point of the FM-index (98) (99), a new powerful indexing approach that is now widely used in many fields, in particular in computational biology. The Burrows-Wheeler transform (*BWT*) is a permutation of characters in a sequence that is easy to understand from a suffix array. Using our previous example, *ATTTAATAAC*, it is the sequence *TTAACAATTA*, which can be displayed next to the suffix array as follows:

| *Position* | *Suffix* | *BWT* |
|---|---|---|
| 8 | *AAC* | *T* |
| 5 | *AATAAC* | *T* |
| 9 | *AC* | *A* |
| 6 | *ATAAC* | *A* |
| 1 | *ATTTAATAAC* | *C* |
| 10 | *C* | *A* |
| 7 | *TAAC* | *A* |
| 4 | *TAATAAC* | *T* |
| 3 | *TTAATAAC* | *T* |
| 2 | *TTTAATAAC* | *A* |

For each position in the suffix array, the BWT corresponds to the letter located just before this position in the sequence (or the last letter for position 1). For instance, the letter indicated in the first line is 'T' as it is the letter at position 8-1=7 in the text. The *BWT* has astonishing properties that we review here briefly. Interested readers can refer to (97) for further algorithmic details.

A first important property of the BWT is that this permutation (*TTAACAATTA* in our example) is reversible: it is sufficient to retrieve the indexed sequence (*ATTTAATAAC*). This is what is called a *self-index,* i.e. it is not necessary to keep the indexed sequence in the memory, it is contained in the index.

A second appreciable property of the BWT is that it is highly compressible. Indeed this letter organization tends to create stretches of letters. For instance, consider the sequence "treat peat pea repeats" (spaces added to help reading). Its BWT sequence is "eeeepprprttetataasa". This is well-suited for compression algorithms (e.g. "eeee" could be rewritten as "4e").

In 2000, Ferragina and Manzini refined this compression approach to make it capable of answering queries in a text, leading to the FM-index (its full name, "Full-text index in Minute space" does not really help its understanding). To do this, they added a few pieces of information to the *BWT* while keeping the data structure extremely light. We

cannot enter into the details of this structure, but suffice to know that it can be considered as a kind of compressed suffix array requiring very little memory space. For instance, the human genome could be indexed with this approach using 1.23 GB, and one million reads of length 100 would require 3.73GB of memory. A good source for codes on the FM index and suffix array indexes is the Pizza&Chili reference site (100). The FM-index supports the following operations which generally use a tiny portion of the compressed file:

- *locate* finds the position in the text of an occurrence of the query, in $O(\log^c n)$ time, where c is a constant chosen at the time the FM-index is built.
- *count* computes the number of occurrences of the query, in time proportional to its length;
- *extract* returns the sequence of a given length starting at a given position in the text;
- *display* outputs for a given length $L$ the $L$ characters on each side of the occurrences of the query in the text.

## Hash tables versus Burrows-Wheeler transform

Another frequently employed indexing data structure is the hash table. The hash table concept stands on a very simple idea. For indexing a set of names for instance, each name is converted into an address (called a hash value) using a function that is

easy to compute. For instance, using a hash function that considers each substring as a number in base 11 (each letter being coded by an ASCII number), "AMY" is given the address $8801=65\times11^2+77\times11^1+89\times11^0$ and "BOB" is given the address $8921=66\times11^2+79\times11^1+66\times11^0$. In an initially empty array (called the hash table), "AMY" is stored at position 8801 and "BOB" at position 8921. As it is important to save as much space as possible, it happens that two different names get the same hash value. For instance, imagine now we add the name "AYM" that also has the hash value 8921. This causes a collision at position 8921 that already contains the name "BOB". There are several ways to manage collision, either by computing a new hash value for "AYM" (open addressing) or by storing at each position a list of name (“BOB” and “AYM”). In case of too many collisions, the hash table is overloaded and may be resized. This operation is expensive as it may reorganize all the already stored items.

Querying a hash table is fairly similar to what is done by the indexing algorithm. The hash value of the query is computed, and the content of the hash table at this position is visited in order to check the presence/absence of the queried object. Depending on the strategy used to manage collisions, either all the entries present in the list at this position have to be checked or new hash functions have to be computed for the query until it is found (match) or an empty position is reached (mismatch).

Most programming languages offer simple structures to build hash tables. It is easy to find efficient implementations of hash tables on the web (e.g. SpookyHash (101) or

SparseHash (102)). Some tools implementing hash tables are specifically designed for routine similarity search on NGS data such as RAPSearch2 (103), which allows similarity searches in proteins and is fully compatible with Blast. Implemented in C++, the source code is freely available for download at the RAPSearch2 website (104).

The main advantage of hash tables is their speed, in particular when collisions are absent or rare. Another important advantage stands in the fact that hash tables are dynamic and can host any additional piece of information for each of its items. Unlike the FM-Index, any new item can be added to a hash table, even after its construction. In a biological context, to each *k*-mer can be associated its list of occurrences in a genome for instance. In comparison, the FM-Index gives only access to the occurring position(s) of each query.

The theoretical indexing and querying times (respectively O(*n*) and O(*m*) in average with *n* being the size of the bank and *m* the size of the query) are the same for FM-Index and hash table approaches. However, the application range are a bit different. First the hash table contains *fixed items*. For instance if items are *k*-mers, a hash table does not allow to query *k+1*-mer or *k-1*-mer. Moreover, these items must be *explicitly stored*. For a human genome, storing 3 billion of 31-mers (coded on a binary alphabet) requires nearly 22 GB of memory. Indexing all 31-mers of a human genome would thus require approximately 23 GB of memory using a hash table (including the array itself). In comparison, the FM-Index is a self-indexed compressed data structure. This means that

the index itself contains the original sequence, and furthermore any *k*-mer of any size could be queried using this data structure. Indexing a human genome using the FM-index requires less that 2 GB of memory. The BWA methods described in (105) presents in details how the Burrows-Wheeler Transform (BWT) (often simply referred inaccurately as the FM-Index) can be used for performing the mapping of reads on a genome.

All the data structures presented above are designed for exact queries. However they are not made for answering questions such as "*where does Q=TAAT occur with at most one error in S=ATTTAATAAC*?". Some attempts to make them able to deal with such requests basically need to enumerate all possibilities. For instance, the previous request would be answered by searching all queries distant by at most one error with respect to *Q*, representing 4*(size of *Q*) queries: AAAT, CAAT, GAAT, TAAT, ACAT, etc. If more than one error should be tolerated, then the number of queries to perform explodes, making such solutions inapplicable for biological application. The search for approximated words requires additional techniques, as described in the next subsection.

### 3.2. Finding approximated words, a matter of seeds…

In terms of sequence approximation, two main distances are commonly used: the Hamming distance in which only substitutions are allowed and the Levenshtein distance in which insertions and deletions (called *indels*) are authorized in addition to substitutions. The Hamming distance between two words is particularly simple to compute: each couple of characters of the two words is read simultaneously and when the two characters differ, the distance is increased by one. Conversely, the Levenshtein distance (also called edit distance) is much more complex to measure. It involves a recursive organization of partial computations called *dynamic programming*. More precisely, computing the Levenshtein distance between two sequences requires a number of comparisons proportional to the product of their lengths, which becomes prohibitive when asking long queries in large banks. It is theoretically impossible to get rid of this complexity, however it is possible in practice to propose some techniques that look for an incomplete result. The goal becomes to find most of the solutions at the price of possibly missing some of them ("false negatives"), using techniques called *heuristics*. One of the most famous heuristics used in computation biology is BLAST (17), which is designed to find approximate occurrences (hits) of a query in databanks. The BLAST approach is representative of ***seed-based algorithms.*** In short, it uses the fact that two words similar enough exactly share at least some small sub-word, called the seed. For

instance, consider *TACACCCTAG* and *TCACCGCTTG*. These two words are similar and they share the seed *CACC*:

```
TACACC-CTAG
| |||| ||.|
T-CACCGCTTG
```

The seed-based algorithms use this simple idea in order to speed-up their computations. Given a query and a database, they first search for occurrences of sub-words of the query in the database. This first step is performed extremely fast, using indexing techniques presented above. The positions where the shared seeds occur in the query and in the database make it possible to limit the search space in which the query may have a hit. A dynamic programming computation is performed in a second step that searches this limited space. It is always useful to keep two warnings in mind when using heuristics: they may miss some solutions and default parameters are not always the best solution (for instance, the standard seed of size 11 is insufficient for finding weak homologies between ancient interspersed repeats and it is recommended to use 7 base seeds instead).

In the last decade, the notion of seeds has been vastly improved. Of course, the smaller the seed, the less likely one is to miss some similarities. But this has two drawbacks: the filtering effect is not very sensitive and the computation becomes slower since there may be many more spurious hits that occur by chance. Two ideas have been developed to increase the sensitivity of a seed with a fixed number of characters, allowing *spaced seeds* and using *multiple seeds*. Multiple seeds are sets of seeds that are

looked for simultaneously and used together to determine an E-value of the hits. The principle of spaced seeds is to choose noncontiguous characters to build them. For the previous example, *C-CT-G* ("-" means a don't care position in the text) has the same number of characters (the same *weight*) and thus the same selectivity than *CACC* but its sensitivity is likely to be better because it spans a longer region (6 instead of 4). Moreover, a software program like YASS (106) offers the possibility of introducing *subset* seeds, i.e. to define some positions where nucleic acids can only take a subset of possible values. For example, transition-constrained seeds (of weight ½) have to belong to a same class in the query and the text, either purine or pyrimidine. Noting # a match position, - a don't care position, and @ a transition position, the default seed of YASS, of weight 9, is #@# –– ## –– # – ##@#. Seeds of fixed weight can be optimized for a range of similarity between sequences and for a user-defined particular family of sequences in order to maximize the hitting probability. YASS can be used online or is available for download at (107). It can filter low complexity repeats and produce the same output format than BLAST. The paper (108) provides an example of YASS pairwise comparisons applied to a gene family encoding proteins with pentatricopeptide repeat (PPR) motifs in the radish genome.

It is possible to derive a "spaced" suffix array from a standard suffix array that takes into account don't care positions by applying a suitable transformation on the text (109) and the query. The seed-based algorithms enable the detection of repeats within a

sequence or between sequences. Indeed, the algorithmic "engine" based on seeds is also adapted for comparing two long sequences in order to search for similar sub-sequences [local alignment (110) and Mummer (111)] and for comparing a sequence against itself in order to search for repeats inside this sequence itself as, for instance, is the case in the Repseek software (14).

### 3.3. Using short sequence reads instead of contigs

The previously presented concepts are based on the use of queries which are smaller in length than those of the bank sequences. With the arrival of NGS (Next-Generation Sequencers) it is not uncommon to have to deal with unassembled data. In this context, queries and/or banks are composed of short sequences called reads of at most a few hundreds nucleotides. A set of such reads typically represents (all chromosomes of) an original genome. The read representation of a whole genome is neither adapted to human, nor to standard automatic analyses. For example, it becomes more difficult to compute the answer to the simple query: "Does this sequence occur in this set of reads?" and previous indexation methods must be adapted or, more radically, new sequences must be designed to be able to cope with such data representation.

When dealing with NGS reads, in particular while looking for repeats in a set of reads, there are several approaches that can be distinguished depending on the

presence of a third-party reference genome or not. Given a set of sequenced reads, and when a *reference genome* exists, this latter can be used as a bank and reads are used as queries. For each read, a BLAST-like search is performed in order to know where it occurs on the genome. This process is known as *read mapping*. As it is applied to millions of reads, the mapping must be answered in a short time. Numerous methods, specific to read mapping, have been developed in the past few years. They are adapted to the size of the requests and to their expected error profiles and high similarity to the reference. A great deal of information can be extracted from the mapping of reads, especially concerning polymorphism (SNPs but also repeated elements). A good read mapper in the case of highly-polymorphic genomes is the NextGenMap software (112). The reference genome is indexed in a hash table. There exists a GitHub site including a wiki page where the code is available for downloading on (113). When *no reference sequence* is available the user has two choices left: either reconstruct the sequence from reads (assembly process) and apply the reference-based approaches on this assembled sequence, or use *de novo* methods that seek elements of interest directly in non-assembled reads. The *de novo* approaches are thus useful when no reference sequence exists and when the assembly of reads is problematic or impossible. This can be the case for highly complex genomes such as plant polyploid genomes. The polyploid nature of the genomes of most of the major species of agronomic interest represents a strong barrier to analysis of the organization and variation of repeats, either for non-coding areas or for duplicated genes. The only reasonable way to conduct a repeat study in

these genomes seems to be to extract the repeat family of interest by careful primer design and PCR amplification. However, the emergence of successful tools for *de novo* detection of elements of interest in raw unassembled reads can be seen. For instance, simpler to detect than repeats, the SNP detection can be performed *de novo* with recent tools like Cortex (114) or discoSnp (115).

We have described in the list of homology-based methods for TE identification a software, RelocaTE, which is able to look for given reference TE in a set of next-generation sequencing (NGS) unassembled reads. Tools are now available to detect Transposable Elements directly from these reads. The general idea is that repeated elements are represented by a high number of reads and read frequency may be used together with sequence similarity to assemble and regroup them into repeat families. At least three methods have been designed for this purpose, RepARK (116), RetroSeq (43) and T-lex (44).

*RetroSeq* is a sophisticated method that can exploit mate pairs. Instead of the classic FASTA file, the input file of RetroSeq is a BAM format file. First, RetroSeq looks for discordant mate pairs: regions present in BAM sequences but not present in the reference sequence. These regions are identified as transposable elements by aligning them against the consensus library with the Exonerate software (117). *T-le*x first uses RepeatMasker (26) to remove the TE present in highly repetitive regions from the list of TE insertions and detects the insertion of TE copies by comparing the two Target Site

Duplications (TSD) and the termini of the TE from the NGS reads with the reference sequence. T-lex detects the deletion of TE copies by the deletion of termini regions in NGS reads.

# 4. Towards a richer characterization of repeated structures

## 4.1. A gentle introduction to the theory of languages

As can be observed in this chapter, most of the current practice of pattern matching looks at efficient ways to index and compare sequences. This has proved very useful and remains extremely important for the efficiency of any search algorithm. However, it proves to be insufficient as the knowledge and understanding of some functional or structural aspects of the different repeat families increases. Analysts in molecular biology progressively shift from mere classification tasks to modeling tasks and develop complex scripts in order to fulfill their search needs. Programming scripts may become a tedious task because people need to express various hypothetical models of sequence architectures. It is widely acknowledged that they may even be hard to reproduce (118). A first line of progress has been proposed with the birth and development of Scientific

Workflow Management Systems [SWMS, (119)]. To search for complex patterns such as repeats, another approach is to clearly separate the descriptive part (the model of the studied sequence family) from the way this model is searched in genomes. This is precisely the goal of linguistic analysis and in the rest of this section we deal with the key concepts in this field. The description of patterns or languages on strings is the subject of the theory of formal languages. It is widely used for computer languages but can also be developed in the case of biological sequences. The search for pattern or models described in a language relies on the development of dedicated parsers that can accept any query model in the language.

The framework of *formal languages* introduces models of a possibly infinite set of sequences. The issue is to represent an infinite set in a finite way. A standard representation, called *grammar*, is a set of rewriting rules acting on a starting axiom. For instance, the following grammar (with axiom $S_1$) is able to recognize telomeric regions of eukaryotes like *A. thaliana*, known to be composed primarily of tandemly repeated blocks *5'-C(C|T)CTAAA-3'*:

$$\mathbf{S_1} \rightarrow \mathrm{C}\ \mathbf{S_2} \qquad \mathbf{S_2} \rightarrow \mathrm{C}\ \mathbf{S_3} \qquad \mathbf{S_2} \rightarrow \mathrm{T}\ \mathbf{S_3} \qquad \mathbf{S_3} \rightarrow \mathrm{C}\ \mathbf{S_4} \qquad \mathbf{S_4} \rightarrow \mathrm{T}\ \mathbf{S_5}$$

$$\mathbf{S_5} \rightarrow \mathrm{A}\ \mathbf{S_6} \qquad \mathbf{S_6} \rightarrow \mathrm{A}\ \mathbf{S_7} \qquad \mathbf{S_7} \rightarrow \mathrm{A}\ \mathbf{S_1} \qquad \mathbf{S_7} \rightarrow \mathrm{A}$$

In such a model, the left part of a rule (the head) rewrites ($\rightarrow$) into its right part (the body). Note that rules use two types of symbols, non-terminal symbols that have to be rewritten using any rule with a matching head ($S_i$, $i \in [1,7]$ in our example) and terminal

symbols that correspond to letters of the analyzed string (nucleic acids A, C or T in our example). Any genomic sequence that can be generated by a finite application of such rules, starting from the axiom rule, is accepted as a telomeric region by the model. Conversely, it is possible to check (i.e. to parse) a given sequence by applying the rules from right to left on the sequence, and possibly collect some information from the parsing structure (a tree). For instance, the number of tandem repeats in the region will be the number of times $S_1$ occurs in the parsing tree.

It appears that the general form of rules has a deep impact on the expressiveness of the associated languages and the complexity of standard operations on these languages such as the membership of a sequence to a language or the intersection of two languages. Furthermore, the categorization of rule types may be roughly achieved in very few classes. Thus, the rules in the example above are very specific: there exists a single, non-terminal symbol in the head and at most one non-terminal in the body, at the end of the body. It can be shown that this particular structure is characteristic of a well-known class of languages called *regular* languages. This class has been used in many pattern matching and bioinformatics tools [e.g. Unix grep for all types of text or ScanProsite (120) for proteins] and script languages (e.g. Perl) since it is possible to look for occurrences of any pattern in linear time (proportional to the length of the sequence). Often, regular expressions (e.g. *C[C|T]CTAAA* in our example) are used instead of grammars since they offer a more compact representation but this is strictly

equivalent and just a matter of notations. Despite their high utility, regular languages are limited for the recognition of repeats. They can only recognize (or serve to model) looping structures, e.g. fixed tandem repeats. Describing, for instance, the Terminal Inverted Repeats of DNA transposons is out of reach for this class of languages. Moreover, if the element that is repeated is unknown (looking for some unspecified tandem repeat or equivalently looking for all tandem repeats in a genome for instance), it is also impossible to represent the structure with a regular language.

For the case of *palindromic repeat structures*, of which the stem-loop structure in RNA sequences is one of the prime examples, it is necessary to accept grammar rules with a body containing any string of terminal and non-terminal symbols. The corresponding class of languages is called *context-free*. For instance, recognizing the TIRs of a DNA transposon could be described by the following grammar:

$$\mathbf{S_1} \rightarrow \text{A}\ \mathbf{S_2}\ \mathbf{U} \quad \mathbf{S_1} \rightarrow \text{C}\ \mathbf{S_2}\ \mathbf{G} \quad \mathbf{S_1} \rightarrow \text{G}\ \mathbf{S_2}\ \mathbf{C} \quad \mathbf{S_1} \rightarrow \text{U}\ \mathbf{S_2}\ \mathbf{A}$$

$$\mathbf{S_2} \rightarrow \text{A}\ \mathbf{S_2}\ \mathbf{U} \quad \mathbf{S_2} \rightarrow \text{U}\ \mathbf{S_2}\ \mathbf{A} \quad \mathbf{S_2} \rightarrow \text{C}\ \mathbf{S_2}\ \mathbf{G} \quad \mathbf{S_2} \rightarrow \text{G}\ \mathbf{S_2}\ \mathbf{C}$$

$$\mathbf{S_2} \rightarrow \text{A}\ \mathbf{S_2} \quad \mathbf{S_2} \rightarrow \text{C}\ \mathbf{S_2} \quad \mathbf{S_2} \rightarrow \text{G}\ \mathbf{S_2} \quad \mathbf{S_2} \rightarrow \text{U}\ \mathbf{S_2} \quad \mathbf{S_2} \rightarrow \varepsilon$$

Context-free rules in this grammar (e.g. $\mathbf{S_2} \rightarrow \text{A}\ \mathbf{S_2}\ \mathbf{U}$) serve to describe the Watson-Crick pairing of nucleic bases. Thus this logical structure on sequences may be clearly associated with a meaningful structure in space corresponding to chemical bonds. Other regular rules (e.g. $\mathbf{S_2} \rightarrow \text{A}\ \mathbf{S_2}$) describe the internal sequence between TIRs without further constraint. The last rule $\mathbf{S_2} \rightarrow \varepsilon$ is a termination rule, where ε denotes

the empty string. The programming languages are generally context-free languages (if you look at the html or xhtml code of a web page for instance, you will see the very same structure of pairing tags that are characteristic of context-free languages). This class of languages is more expressive but at some cost: recognizing if a model occurs in a sequence of size $n$ may require in the worst case in the order of $n^3$ operations.

The next question is to know if this class of language is sufficient for biological modeling. The answer is clearly no. Consider for instance the description of chloroplast microstatellites. These "simple sequence repeats" that are stretches of small words (size less than 7 generally) are complex from the point of view of structure: it is not possible to decide in advance the number of copies or the size of copies. It requires more advanced grammatical rules, called *context sensitive*, where the non-terminal symbol on the left and the body of the rule may be surrounded by as many symbols as necessary (there exists a context of rule application) providing that the same symbols are on both sides. Other examples of context sensitive models in biological sequences are the pseudo-knot structures in RNA or the disulfide bridge structure in proteins and the introduction of errors in repeat is another source of complexity. The cost of models in this category may become very high but fortunately it is not necessary to use the full expressive power of context-sensitive languages. In practice, the art of linguistic analysis entails getting the right tradeoff between the flexibility of the modeling language and the efficiency of model parsing. From the user's point of view, a number of models have to be tried, tuning them iteratively in order to get a reasonable number

of hits. Moreover, since parsers can provide not only the hits but also their internal structure, it may be necessary to filter in post-treatment structural alternatives that are not relevant for the biological analysis.

### 4.2. Linguistic analysis of genomic sequences

Once a language has been chosen for expressing models or patterns, any model can be searched in a bank of sequences using dedicated software. If the language is simple and specific to a sequence family, the query may generally be described by a string on a special alphabet and this task is referred to as pattern matching. If the language is more generic and allows the expression of more complex structures, it can take several rules to describe the language and the software is then called a *parser*.

**Dedicated Pattern Matching**

It is not possible here to provide an exhaustive review of the profusion of specific tools that have been made available to bioanalysts. Some are specific to a sequence family and others to a particular motif type.

We have already cited *ScanProsite* (120) as an example of a pattern matcher using motifs defined on a subset of regular languages. The current version accepts Prosite patterns, user-defined patterns in the Prosite syntax, a combination of patterns using logical

operators *and*, *or* and *not*, and can use contextual annotation templates (ProRules) to detect functional characteristics. They are searched either by a query in a precomputed database or with an algorithm called *ps-scan*. A website is available on the Expasy server but for a large-scale independent analysis, it is possible to download the ps-scan Perl script (121). A higher level parser has been developed for the *de novo* recognition of human polymeraseII promoter regions in (122). This study uses a two-level grammar. A regular grammar first allows recognition of promoter elements such as the TATA-box, the Initiator and the Downstream Promoter Element (DPE), etc. Then a context-free grammar is in charge of the recognition of a correct assembly of all these elements in a reasonable promoter. Unfortunately, although it is likely the authors use a generic context-free parser for this task, no tool is made available: we cite it here mostly for the purposes of illustration since it is characteristic of the linguistic approach.

A number of tools are dedicated to RNA sequences, in response to the increasing need for structure exploration in the complex RNA world, boosted by the recent importance of non-coding RNA studies. This is useful for checking structural features in retrotransposons. *RNAmotif* (123) is probably the most popular in this category as it combines a pattern description language and a language to tune the scoring. It has been designed for the description of patterns as a succession of content-constrained stems and loops, offering the possibility of choosing the standard Watson-Crick pairing (*A-U*, *G-C*) or any other user-defined pairing. The code is available for download at (124). The tool *Locomotif* (125) has almost the same expressiveness (slightly less) as the previous

one but proposes interesting additional features. The first is that it allows the user to graphically design his pattern in an editor by composing several stems and loops annotated with information on the sequence content and size. A dedicated parser is automatically derived from the graphical representation provided. A second feature filters a single matching result by optimizing a thermodynamical model. A more recent tool in this category, *Structator* (126), is representative of this new generation of tools that first use a lexical analysis to significantly improve the parsing time in a second step. It makes use of an index data structure that is suited to the analysis of palindromic structures and is derived from those we have presented, the *affix array*.

**General Purpose Pattern Matching**

Some tools have been designed for the analysis of several types of sequences (DNA, RNA, proteins) with a generic expressiveness, i.e. without targeting the recognition of a particular motif family. Among these general tools, two tendencies can be observed, efficiency-oriented and expressiveness-oriented software.

One of the most advanced software solutions from the point of view of *efficiency* is *Vmatch* (68), which offers a wide variety of search facilities in very large sequences. Vmatch is a package maintained since 2003 by S. Kurtz and resulting from long experience in the field of indexing and pattern matching for genomic sequences (the

initial version was called REPuter). Vmatch is free for academic research and can be obtained by downloading a license agreement form. It proposes a flexible command language with numerous constructions offering a very broad variety of possible queries. It is based on a careful implementation of enhanced suffix arrays (127) for the computation of a sequence index that provides fast access to every substring in that sequence. If the search for a motif contains some rare substrings, this technique is particularly efficient. As in the previous version, REPuter, Vmatch goes from exact to approximated strings with a fixed number of mismatches by using a dynamic programming algorithm and proposes a graphical interface for the bioanalyst.

The software Vmatch is the core search engine used in a number of more specialized tools working on specific sequence structures (e.g. tandem-repeats or LTR retrotransposons in MASiVE). It is used in some databases to generate genomic information or to propose extended search functionalities. For instance, in MIPSPlantsDB, the curation and clustering in the mips-REdat repeat database (128) has been achieved using Vmatch: repeats are put in a same cluster if they share 98% identity and the representative of each cluster is its longest sequence, a choice that makes it possible to remove incomplete sequences included in a cluster representative. PlantGDB proposes a server also based on Vmatch, *PatternSearch* (129), to look for short patterns with mismatches in *A. thaliana* or *O. sativa* genomes.

Another highly generic tool, although less expressive, is *Biogrep* (130), designed by MIT with the objective of quickly recognizing a large set of simple motifs (typically more

than 100) in biological sequence banks, using multi-processor optimizations. Biogrep allows queries in the POSIX language, a standard format of extended regular expressions, and can look for patterns in parallel on a set of processors.

The other approach for the analysis of biological sequences is more concerned with modeling the peculiarities of biological objects in the most relevant and *expressive* way. A major contribution in this respect is the work of D. Searls who laid the foundations for research in this domain. He was the first to supervise developments allowing users to design biological grammars and to apply them for the large-scale analysis of their genomic sequences (131; 132). One of D. Searls' key ideas is to try to find a balance between the well-founded framework of context-free languages that offer a good expressivity/efficiency trade-off, and the necessity of easily describing basic biological mechanisms such as copy that lie at the core of genome evolution. D. Searls introduced a very practical object in algebraic grammars, the *string variable*, which elegantly expresses this notion of copy (either direct or reverse). He has implemented the resulting logic formalism, called SVG - for StringVariable Grammars -, in the (no longer available) GenLang tool (133). From the point of view of expressivity on biological sequences, this makes it possible to take into account not only various forms of copy, distance, position and size constraints but also hierarchical aspects of genomic structures. For instance, in the case of LTR Retrotransposons, the top-level rule of the

grammar could be represented by the following expression – this is given for the purposes of illustration only and does not pretend to be fully realistic:

LTRR→ DR:[2..6], «tg», (U5,R,U3):[80..750], «ca »,
[1..100], pbs, [1..100], gag, [1 000..15 000], ppt, [1..100],
«tg», (U5:80%, R:90%, U3:80%), «ca», DR .

In this expression, DR, U5, R, and U3 are string variables. Its meaning is "the sequence is surrounded by two exact copies of a direct repeat (DR) of size between 2 and 6. The LTR start with nucleotides "tg", end with nucleotides "ca" and are made up of three parts named A, R and B with a total length between 80 and 750. The right LTR is an approximate copy of the left one. The central part (R) is the most preserved - because of the hybridization between both Rs during duplication - with a 90% minimum identity level whereas U3 and U5 only need to have 80% level identity. The central part of the sequence must contain at constrained distances a primer binding site (pbs), a group-specific antigen (gag), and polypurine tract (ppt), which are described by other grammatical rules.

GenLang is no longer available but *PatSearch* (134) is a restricted tool belonging to this family. It is based on the C program *scan_for_matches*, mainly written by R. Overbeek and which is downloadable from (135). It allows to describe approximated strings (including IUB codes for ambiguous nucleotides and mismatch/indel errors), gaps and

length constraints, stem/loops structures and alternative patterns. Insofar as regards repeats, they can be described by a statement

*nmax*>repeat(*patternident*=*pattern*) *dmin*..*dmax* >*nmin* ,

where *nmin* and *nmax* are integers fixing a range for the number of patterns, *dmin* and *dmax* fix a range for the edit distance between repeat units, and *patternident* and *pattern* are a string variable and a pattern constraining the content of this variable respectively. The keyword frepeat has to be used instead of repeat in case of exact repeat. In addition, PatSearch provides an assessment of the motif significance from a simulation experiment using Markov chains (estimating the number of instances that can be expected randomly).

Logol (136) is a highly descriptive language dedicated to the modeling of biological sequences and also derived from SVG. Starting from the sound basis of SVG grammars, the Logol language proposes several extensions - most notably by adopting a constraint approach - with the aim of allowing the expression of realistic biological motifs. Models use constrained string variables (supporting overlaps, substitution and distance errors) that can be subject to various transformations (e.g. inverse complement), gaps, and repetitions of a pattern along the sequence, negation and alternatives to define different possibilities. As in every formal grammar components can be grouped with a view to obtaining a high-level representation of a subset of components.

Repeats may be described either with string variables or with special repeat constructs. For example, the following model with *string variables I1* and *I2* can be used to look for 3 instances of the same string successively deriving from each other (e.g. *I1= aaaaa*, *I2=aaaca* and *I3=agaca*): *X1:{#[5,8],_I1},.*:{#[1,7]}, ?I1:{_I2}:{$[1,1]},.*:{#[1,7]}, ?I2:{$[1,1]}*

The second pattern, *?I1: {_I2}:{$[1,1]}*, reads as follows: the expected string must be similar to the previous *I1* string (*aaaaa* in our example), apart from 1 mismatch (*$[1,1]*). The matched string (*aaaca*) is saved in I2 ({_I2}) for further use in the last pattern (*{?I2}*). This individualization of instances means it is possible to adjust fine notions of sequence evolution.

The following example shows how palindromic repeats for the recognition of stem-loops whose stem length varies between 5 and 11 and loop size between 1 and 9 are represented. In this example, the Watson-Crick pairing is not required to be perfect: up to 2 substitutions and 1 indel are allowed.

*STEM1:{#[5,11],_IS1}, .*:{#[1,9]}, -"wc" ?IS1 :{$[0,2],$$[0,1]}*

The content of *STEM1* (first strand of the stem) is saved in IS1, (*_IS1*). The second stem strand is then defined as the exact reverse complement of the previous content (that is -"*wc*" *?IS1*), except for 2 mismatches and 1 indel.

The special constructor *repeat,* as in PatSearch, manages the characteristics of a series of occurrences. Its standard format is:

*repeat(<entity>,<distance>)+<occurrence number>*.

For instance, *repeat("acgt",[0,3])+[7,38]* states that substring *acgt* is repeated from 7 to 38 times, using a spacing of at most 3 characters between 2 repeats.

Logol is available as a web application and for download on (137). It includes a graphical editor. In the case of spacers in a model, Logol calls on an external program using indexing sequence techniques to directly look for positions of subsequent words. Two possibilities are offered by Logol to perform indexing, either Vmatch or Cassiopee, a Ruby tool specifically developed for Logol and which is generally not as efficient as Vmatch but enables installation independently of Vmatch.

# List of tables

**Table 1: List of plant transposable element databases that cover several plant genera**

| Database and reference | Content | Web site |
|---|---|---|
| **mips-REdat + mips-REcat** (128) | All repeats + repeat type index (82 genera) | http://mips.helmholtz-muenchen.de/plant/recat/ |
| **Plant Repeat Databases** (138) | | http://plantrepeats.plantbiology.msu.edu/ |
| **MASiVEdb** (139) | LTR (37 species) | http://databases.bat.infspire.org/masivedb/ |
| **P-MITE** (140) | MITE (41 species) | http://pmite.hzau.edu.cn/django/mite/ |

**Table 2: List of available *ab initio* methods detecting all types of transposable elements**

| Software Name | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|
| **PClouds** (13) | www.evolutionarygenomics.com/ProgramsData/PClouds/PClouds.html | Linux, MacOS X | C compiler | No help file |
| **PILER** (18) | www.drive5.com/piler/ | All | MUSCLE | User's guide (web site) |
| **RBR** (141) | www.ii.uib.no/~ketil/bioinformatics/tools.html | All, Linux preferred | Glasgow Haskell Compiler (GHC) | README Install file |

| | | | | |
|---|---|---|---|---|
| **ReAS** (142) | ://ftp.genomics.org.cn/pub/ReAS/software/ | All | Perl, C compiler, 64bits system | README Install file |
| **RECON** (19) | selab.janelia.org/recon.html | All | C compiler | README Install file. Makefile to be adapted |
| **RepeatFinder** (10) | cbcb.umd.edu/software/RepeatFinder/ | All | REPuter | README Install file |
| **RepeatScout** (59) | repeatscout.bioprojects.org/ | Linux, MacOS X | Perl, Tandem Repeat Finder, RepeatMasker | README Install file |
| **Repseek** (14) | wwwabi.snv.jussieu.fr/public/RepSeek/ | Linux, MacOS X | No | README Install file User's guide (Repseek_doc.pdf |
| **REPuter** (9) | bibiserv.techfak.uni-bielefeld.de/reputer/ | All | No | User's guide (web site) |
| **RepARK** (116) | https://github.com/PhKoch/RepARK. | All | Perl, Jellyfish, Velvet | README |
| **Tallymer** (8) | www.zbh.uni-hamburg.de/?id=211 | Linux, MacOS X | Perl, C compiler, Python, Ruby, Cairo & Pango lib. (optional), | User's guide (web site) |

| | | | | |
|---|---|---|---|---|
| | | | HMMER | |
| **Windowmasker** (11) | ftp.ncbi.nlm.nih.gov/pub/agarwala/windowmasker | Windows, Linux | No | User's guide (web site) |

**Table 3: List of available homology-based methods detecting all types of transposable elements**

| Software Name | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|
| **AB-BLAST** (wu-) (22) | www.advbiocomp.com/blast.html | | | User's guide (web site) |
| **Censor** (24) | www.girinst.org/censor/download.php | Linux, MacOS X | Perl | README Install file |
| **HMMER** (45) | hmmer.janelia.org/ | Linux, MacOS X | Perl | User's guide (web site) |
| **NCBI-BLAST** (21) | ftp.ncbi.nlm.nih.gov/blast/executables/blast+/ | All | No | README Install file |
| **One code to find them all** (42) | http://doua.prabi.fr/software/one-code-to-find-them-all | All | Perl | Tutorial (zip file on web site) |
| **Process_hits** (39) | processhits.sourceforge.net/ | All | Perl | README Install file |
| **REannotate** (40) | www.bioinformatics.org/reannotate/index.html | All | Perl | User's guide (web site) |
| **RelocaTE** (29) | https://github.com/srobb1/ | All | Blat, Bowtie 1, | README |

| | | | | |
|---|---|---|---|---|
| | RelocaTE | | BioPerl, SAMtools | User's guide (web site) |
| **RepeatMasker** (25) | www.repeatmasker.org/ | All | Perl, Blast or HMMER or RMBlast, and Tandem Repeat Finder | README Install file |
| **RepeatRunner** (41) | www.yandell-lab.org/software/repeatrunner.html | All | Perl, Blast and RepeatMasker | README Install file |
| **RetroSeq** (43) | github.com/tk2/RetroSeq | Linux | Perl, bedtools lib., Samtools and Exonerate | User's guide (web site) |
| **T-lex** (44) | petrov.stanford.edu/cgi-bin/Tlex_manual.html | Linux | Perl, RepeatMasker, Maq, SHRIMP2, BLAT, Phrap and FastaGrep | User's guide (web site) |
| **TARGeT** (30) | target.iplantcollaborative.org/ | All | Web browser | User's guide (web site) |
| **TESeeker** (27) | repository.library.nd.edu/view/27/teseeker | All | VirtualBox, BLAST, CAP3, ClustalW2, and BioPerl | User's guide (web site) |

| | | | | |
|---|---|---|---|---|
| **Transposon-PSI** (28) | transposonpsi.sourceforge.net/ | All | Perl, Psi Blast | README Install file |

**Table 4: List of available structure-based methods detecting all types of transposable elements**

| Software Name | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|
| **SMaRTFinder** (47) | services.appliedgenomics.org/software/smartfinder/ | All | C compiler | README Install file |
| **SMOTIF** (46) | www.cs.rpi.edu/~zaki/software/sMotif/ | All | C compiler | README Install file |
| **Logol** (136) | http://logol.genouest.org/ | Linux, MacOS X | Ruby | User's guide (web site) |

**Table 5: List of available pipeline methods detecting all types of transposable elements**

| Software Name | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|
| **DAWGPAWS** (49) | dawgpaws.sourceforge.net/ | Linux | Perl, emacs, Apollo, Blast, Cross_match,EuGène, find_ltr, FINDMITE, FGENESH, | User's guide (web site) |

| | | | | |
|---|---|---|---|---|
| | | | GeneID, GeneMarkHMM, Genscan, HMMER, LTR_FINDER, LTR_Seq, LTR_Struc, RepeatMasker | |
| **RepeatExplorer** (2) | galaxy.umbr.cas.cz:8080/ | Linux | Perl, R, Python, ImageMagick, Blast, RepeatMasker, Muscle, Fasty36 | User's guide (web site) |
| **RepeatModeler** (25) | www.repeatmasker.org/RepeatModeler.html | Linux, MacOS X | Perl, RepeatMasker, RECON, RepeatScout, Tandem Repeat Finder, and RMBlast or Ab-Blast | README Install file User's guide |
| **REPET** (50) | urgi.versailles.inra.fr/Tools/REPET | Linux, MacOS X | C compiler | User's guide (web |

| | | | | |
|---|---|---|---|---|
| | | | | site) |
| **TriAnnot** (51) | wheat-urgi.versailles.inra.fr/Tools/Triannot-Pipeline | All | Web browser | User's guide (web site) |
| **RISCI** (52) | http://www.ccmb.res.in/rakeshmishra/tools.html | Linux | Perl, EMBOSS, Blast, RepeatMasker | README |

**Table 6: List of available software for the search of Long Terminal Repeat Retrotransposons (LTRR)**

| Software Name | Method | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|---|
| **LTR_MINER** (64) | Homol. | genomebiology.com/content/supplementary/gb-2004-5-10-r79-s7.pl | All | Perl | No help file |
| **LTR_Finder** (55) | Struct. | tlife.fudan.edu.cn/ltr_finder/ | Linux (standal one) | No | User's guide (web site) |
| **LTR_STRUC** (53) | Struct. | www.mcdonaldlab.biology.gatech.edu/ltr_struc.htm | Windows | No | No help file |
| **LTRharvest** (65) | Struct. | www.zbh.uni-hamburg.de/forschung/genominformatik/software/ltrharvest.html | Linux, MacOS X | Perl, C compiler, Python, GenomeTools | README Install file User's guide (web |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | site) |
| **MGEScan-LTR** (57) | Struct. | darwin.informatics.indiana.edu /cgi-bin/ evolution/daphnia_ltr.pl | Linux, MacOS X | Perl, C compil. Tandem Repeat Finder, HMMER, EMBOSS | README Install file User's guide (web site) |
| **MASiVE** (67) | Pipel. | tools.bat.infspire.org/masive/ | Linux, MacOS X | LTRharvest, Vmatch, Wise2 and MAFFT | README Install file |

**Table 7: List of available homology-based methods detecting non LTR retrotransposons**

| Software Name | Web site, Download site (ftp, forge, galaxy or github), or email contact | OS | Requirements | Comments |
|---|---|---|---|---|
| **MGEScan-nonLTR** (70) | darwin.informatics.indiana.edu /cgi-bin/evolution/nonltr/nonltr.pl | Linux, MacOS X | Perl, C compiler, HMMER, EMBOSS package | README Install file |
| **RTAnalyzer** (71) | www.riboclub.org/cgi-bin/RTAnalyzer/index.pl | All | Perl, Internet connexion | No help file |

**Table 8: List of available software for the search of DNA transposons**

| Software Name | Method | Web site, Download site (ftp, forge, galaxy or | OS | Requirements | Comments |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| | | github), or email contact | | | |
| **TRANSPO** (73) | Homol. | alggen.lsi.upc.es/recerca/search/transpo/transpo.html | Windows (standalone) | cygwin1.dll (standalone) | No help file |
| **IRF** (76) | Struct. | tandem.bu.edu/irf/irf.download.html | All | No | No help file |
| **RSPB** (78) | Struct. | http://122.205.95.39/media/MITE/tools/RSPB_0.20.zip | Linux | Blast, Muscle, Mdust, Perl, RepeatMasker | Readme (http://122.205.95.39/media/MITE/tools/RSPB_Readme.txt) |
| **MITE-Hunter** (79) | Struct. | target.iplantcollaborative.org/mite_hunter.html | All | Perl, Blast, Muscle, mDust | Readme Install file User's guide (web site) |
| **MITE Digger** (80) | Struct. | http://labs.csb.utoronto.ca/yang/MITEDigger | Windows | Perl | Readme + Rice database |
| **MUST** (77) | Struct. | csbl1.bmb.uga.edu/ffzhou/MUST/ | Web server | No | No help file |

**Table 9: Software looking for Helitrons**

| Software Name | Web site, Download site (ftp, | OS | Requirements | Comments |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| | forge, galaxy or github), or email contact | | | |
| **HelSearch** (82) | helsearch.sourceforge.net/ | All | Perl, Blast, ClustalW | README Install file |

# List of figures

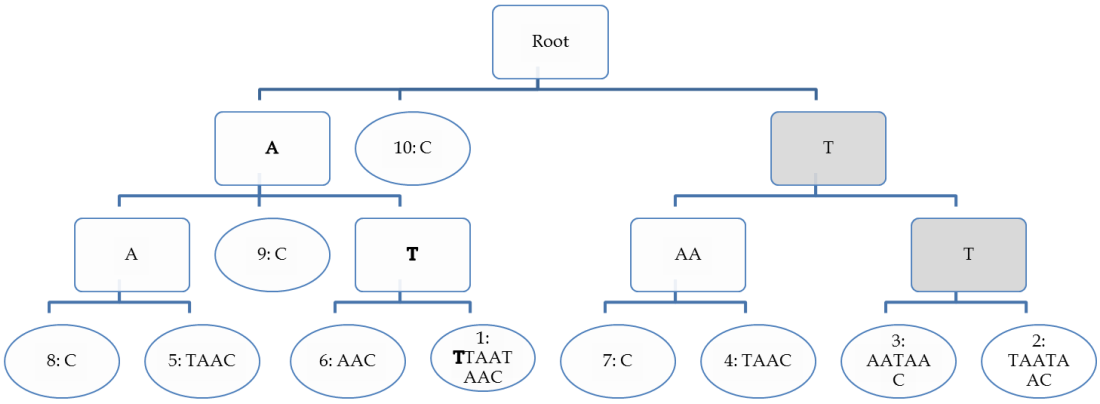


**Figure 1: Suffix- tree representation for the text ATTTAATAAC. The top square is the root of the tree. Circles are the leaves, and other squares are called "internal nodes".**

## References

1. *The peculiar landscape of repetitive sequences in the olive (Olea europaea L.) genome.* **Barghini, E., et al., et al.** 4, Apr 2014, Genome Biol Evol., Vol. 6, pp. 776-91. doi: 10.1093/gbe/evu058.

2. *RepeatExplorer: a Galaxy-based web server for genome-wide characterization of eukaryotic repetitive elements from next-generation sequence reads.* **Novák, P., et al., et al.** 6, 2013, Bioinformatics, Vol. 29, pp. 79279-3. doi: 10.1093/bioinformatics/btt054.

3. *Review of tandem repeat search tools: a systematic approach to evaluating algorithmic performance.* **Lim, K. G., et al., et al.** 1, 2013, Briefings in Bioinformatics, Vol. 14, pp. 67-81. doi :10.1093/bib/bbs023.

4. *Sequence-specific error profile of Illumina sequencers.* **Nakamura, K., et al., et al.** 13, may 2011, Nucl. Acids Res., Vol. 39, p. e90. doi:10.1093/nar/gkr344.

5. *Direct comparisons of Illumina vs. Roche 454 sequencing technologies on the same microbial community DNA sample.* **Luo, C., et al., et al.** 3, 2012, PLoS ONE, Vol. 7, p. e30087. doi: 10.1371/journal.pone.0030087.

6. *Repbase Update, a database of eukaryotic repetitive elements.* **Jurka, J., et al., et al.** 2005, Cytogenetic and Genome Research, Vol. 110, pp. 462-467.

7. *Discovering and detecting transposable elements in genome sequences.* **Bergman, C. M. et Quesneville, H.** 6, 2007, Briefings in Bioinformatics, Vol. 8, pp. 382-392.

8. *A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes.* **Kurtz, S., et al., et al.** 2008, BMC Genomics, Vol. 9, p. 517.

9. *REPuter: the manifold applications of repeat analysis on a genomic scale.* **Kurtz, S., et al., et al.** 22, 2001, Nucleic Acids Res., Vol. 29, pp. 4633-42.

10. *A clustering method for repeat analysis in DNA sequences.* **Volfovsky, N. et Haas, B.J. and Salzberg, S.L.** 8, 2001, Genome Biol., Vol. 2, p. RESEARCH0027.

11. *WindowMasker: window-based masker for sequenced genomes.* **Morgulis, A., et al., et al.** 2, 2006, Bioinformatics, Vol. 22, pp. 134-41.

12. *A fast, lock-free approach for efficient parallel counting of occurrences of k-mers.* **Marcais, G. et Kingsford, C.** 2011, Bioinformatics, Vol. 27, pp. 764-770.

13. *Identification of repeat structure in large genomes using repeat probability clouds.* **Gu, W., et al., et al.** 1, 2008, Anal Biochem, Vol. 380, pp. 77-83.

14. *Repseek, a tool to retrieve approximate repeats from large DNA sequences.* **Achaz, G., et al., et al.** 1, 2007, Bioinformatics, Vol. 23, pp. 119-21.

15. *Estimating the probability of approximate matches.* **Kurtz, S. et Myers, G.** s.l. : Springer Verlag, 1997, Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1264, pp. 52-64.

16. *Velvet: algorithms for de novo short read assembly using de Bruijn graphs.* **Zerbino, D.R. et Birney, E.** 2008, Genome Res., Vol. 18, pp. 821-829.

17. *Basic local alignment search tool.* **Altschul, S.F., et al., et al.** 3, 1990, J Mol Biol., Vol. 215, pp. 403-10.

18. *PILER: identification and classification of genomic repeats.* **Edgar, R.C. et Myers, E.W.** 2005, BMC Bioinformatics, Vol. 9, p. 18.

19. *Automated de novo identification of repeat sequence families in sequenced genomes.* **Bao, Z. et Eddy, SR.** 8, 2002, Genome Res., Vol. 12, pp. 1269-76.

20. *Discovery and assembly of repeat family pseudomolecules from sparse genomic sequence data using the Assisted Automated Assembler of Repeat Families (AAARF) algorithm.* **DeBarry, J., Liu, R. et Bennetzen, J.** 1, 2008, BMC Bioinformatics, Vol. 9, p. 235. 10.1186/1471-2105-9-235.

21. *NCBI BLAST: a better web interface.* **Johnson, M., et al., et al.** 2008, Nucleic Acids Res., Vol. 36, pp. W5-9.

22. **Advanced Biocomputing, LLC.** AB-BLAST. [En ligne] 2009. http://blast.advbiocomp.com/.

23. *Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements.* **Schäffer, AA., et al., et al.** 14, 2001, Nucl. Acids Res., Vol. 29, pp. 2994-3005. doi:10.1093/nar/29.14.2994.

24. *CENSOR - a program for identification and elimination of repetitive elements from DNA sequences.* **Jurka, J., et al., et al.** 1, 1996, Computers and Chemistry, Vol. 20, pp. 119-122.

25. **Smit, AFA.., Hubley, R. et Green, P.** RepeatMasker Open-3.0. [En ligne] 1996-2010. http://www.repeatmasker.org/.

26. *Using and understanding RepeatMasker.* **Tempel, S.** 2012, Methods Mol Biol., Vol. 859, pp. 29-51.

27. *An automated homology-based approach for identifying transposable elements.* **Kennedy, R.C., et al., et al.** 2011, BMC Bioinformatics, Vol. 12, p. 130.

28. **Haas, B.J.** TransposonPSI. [En ligne] 2010. http://transposonpsi.sf.net.

29. *The Use of RelocaTE and Unassembled Short Reads to Produce High-Resolution Snapshots of Transposable Element Generated Diversity in Rice.* **Robb, S. C., et al., et al.** 6, 2013, G3: Genes|Genomes|Genetics, Vol. 3, pp. 949-957. 10.1534/g3.112.005348.

30. *TARGeT: a web-based pipeline for retrieving and characterizing gene and transposable element families from genomic sequences.* **Han, Y., Burnette, J.M. et Wessler, S.R.** 11, 2009, Nucl. Acids Res., Vol. 37, p. e78.

31. *MUSCLE: multiple sequence alignment with high accuracy and high throughput.* **Edgar, R.C.** 5, 2004, Nucl. Acids Res., Vol. 32, pp. 1792-1797.

32. *FastTree: Computing Large Minimum-Evolution Trees with Profiles instead of a Distance Matrix.* **Price, M.N., Dehal, P.S. et Arkin, A.P.** 2009, Molecular Biology and Evolution, Vol. 26, pp. 1641-1650.

33. *CAP3: A DNA sequence assembly program.* **Huang, X. et Madan, A.** Sept. 1999, Genome Res., Vol. 9, pp. 868-77.

34. *Clustal W and Clustal X version 2.0.* **Larkin, M.A., et al., et al.** 21, 2007, Bioinformatics, Vol. 23, pp. 2947-2948.

35. *Tandem repeats finder: a program to analyze DNA sequences.* **Benson, G.** 18, 1999, Nucleic Acids Research, Vol. 20, pp. 573-580.

36. **Green, P.** phrap/cross_match/swat documentation. [En ligne] 1993-1996. http://www.phrap.org/phredphrap/general.html.

37. **TimeLogic.** Decypher. [En ligne] 2014. http://www.timelogic.com/.

38. **Smit, A.** RMBlast. [En ligne] 2013. http://www.repeatmasker.org/RMBlast.html.

39. **Smith, J.D.** Process_hits. [En ligne] 2010. http://sourceforge.net/projects/processhits/files/README.txt/download.

40. *Automated paleontology of repetitive DNA with REANNOTATE.* **Pereira, V.** 2008, BMC Genomics, Vol. 9, p. 614. doi: 10.1186/1471-2164-9-614.

41. *Improved repeat identification. ; masking in Dipterans.* **Smith, C.D., et al., et al.** 1, 2007, Gene, Vol. 389, pp. 1-9.

42. *"One code to find them all": a perl tool to conveniently parse RepeatMasker output files.* **Bailly-Bechet, M., Haudry, A. et Lerat, E.** 13, s.l. : Biomed Central, 2014, Mobile DNA,, Vol. 5. doi:10.1186/1759-8753-5-13.

43. *RetroSeq: transposable element discovery from next-generation sequencing data.* **Keane, T.M., Wong, K. et Adams, D.J.** 3, 2012, Bioinformatics, Vol. 29, pp. 389-390.

44. *T-lex: a program for fast and accurate assessment of transposable element presence using next-generation sequencing data.* **Fiston-Lavier, A.S., et al., et al.** 6, 2011, Nucl. Acids Res., Vol. 39, p. e36.

45. *HMMER web server: interactive sequence similarity searching.* **Finn, R.D., Clements, J. et Eddy, S.R.** 2011, Nucleic Acids Research, Vol. 39, pp. W29-W37.

46. *SMOTIF: efficient structured pattern and profile motif search.* **Zhang, Y. et Zaki, M.J.** 2006, Algorithms Mol Biol., Vol. 1, p. 22.

47. *Structured Motifs Search.* **Morgante, M., et al., et al.** 8, oct. 2005, Journal of Computational Biology, Vol. 12, pp. 1065-1082. doi:10.1089/cmb.2005.12.1065.

48. *Suffix-tree analyser (STAN): looking for nucleotidic and peptidic patterns in chromosomes.* **Nicolas, J., et al., et al.** 24, 2005, Bioinformatics, Vol. 21, pp. 4408-10.

49. *The DAWGPAWS pipeline for the annotation of genes and transposable elements in plant genomes.* **Estill, J.C. et Bennetzen, J.L.** 1, 2009, Plant Methods, Vol. 5, p. 8.

50. *Considering transposable element diversification in de novo annotation approaches.* **Flutre, T., et al., et al.** 1, 2011, PLoS One, Vol. 6, p. e16526.

51. *TriAnnot: A Versatile. ; High Performance Pipeline for the Automated Annotation of Plant Genomes.* **Leroy, P., et al., et al.** 2012, Front Plant Sci., Vol. 3, p. 5.

52. *RISCI - Repeat Induced Sequence Changes Identifier: a comprehensive, comparative genomics-based, in silico subtractive hybridization pipeline to identify repeat induced sequence changes in closely related genomes.* **Singh, V. & Mishra, R.** 609, 2010, BMC Bioinformatics, Vol. 11. 10.1186/1471-2105-11-609.

53. *LTR_STRUC: A novel search and identification program for LTR retrotransposons.* **McCarthy, E.M. et McDonald, J.F.** 2003, Bioinformatics, Vol. 19, pp. 362-367.

54. *Efficient algorithms and software for detection of full-length LTR retrotransposons.* **Kalyanaraman, A. et Aluru, S.** 2, 2006, J Bioinform Comput Biol., Vol. 4, pp. 197-216.

55. *LTR_FINDER: an efficient tool for the prediction of full-length LTR retrotransposons.* **Xu, Z. et Wang, H.** 2007, Nucleic Acids Res., Vol. 35, pp. W265-W268.

56. *Eight novel families of miniature inverted repeat transposable elements in the African malaria mosquito, Anopheles gambiae.* **Tu, Z.** 2001, PNAS, Vol. 98, pp. 1699-1704.

57. *De novo identification of LTR retrotransposons in eukaryotic genomes.* **Rho, M., et al., et al.** 2007, BMC Genomics, Vol. 8, p. 90.

58. *TEnest: Automated chronological annotation and visualization of nested plant transposable elements.* **Kronmiller, B.A. et Wise, R.P.** 2008, Plant Physiol., Vol. 146, pp. 45-59.

59. *De novo identification of repeat families in large genomes.* **Price, A. L., Jones, N. C. et Pevzner, P.A.** 1, 2005, Bioinformatics, Vol. 21, pp. 351-8.

60. *Detection of new transposable element families in Drosophila melanogaster. ; Anopheles gambiae genomes.* **Quesneville, H., Nouaud, D. et Anxolabéhère, D.** 1, 2003, J Mol Evol., Vol. 57, pp. S50-9.

61. *On global sequence alignment.* **Huang, X.** 1994, Computer Applications in the Biosciences, Vol. 10, pp. 227–235.

62. *Recent developments in the MAFFT multiple sequence alignment program.* **Katoh, K. et Toh, H.** 2008, Briefings in Bioinformatics, Vol. 9, pp. 286–298.

63. *mreps: efficient and flexible detection of tandem repeats in DNA.* **Kolpakov, R., Bana, G. et Kucherov, G.** 2003, Nucl. Acids Res., Vol. 31, pp. 3672–3678.

64. *Insertion bias and purifying selection of retrotransposons in the Arabidopsis thaliana genome.* **Pereira, V.** 10, 2004, Genome Biol., Vol. 5, p. R79.

65. *LTRharvest, an efficient and flexible software for de novo detection of LTR retrotransposons.* **Ellinghaus, D., Kurtz, S. et Willhoeft, U.** 2008, BMC Bioinformatics, Vol. 9, p. 18.

66. *GenomeTools: a comprehensive software library for efficient processing of structured genome annotations.* **Gremme, G., Steinbiss, S. et Kurtz, S.** 3, 2013, IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 10, pp. 645–656.

67. *MASiVE: Mapping and Analysis of SireVirus Elements in plant genome sequences.* **Darzentas, N., et al., et al.** 19, 2010, Bioinformatics, Vol. 26, pp. 2452–2454.

68. **Kurtz, S.** Vmatch: large scale sequence analysis software. *http://www.vmatch.de/vmweb.pdf.* [En ligne] December 2011.

69. *Genewise and genomewise.* **Birney, E., Clamp, M. et Durbin, R.** 2004, Genome Res., Vol. 14, pp. 988–995.

70. *MGEScan-non-LTR: computational identification and classification of autonomous non-LTR retrotransposons in eukaryotic genomes.* **Rho, M. et Tang, H.** 21, 2009, Nucl. Acids Res., Vol. 37, p. e143.

71. *RTAnalyzer: a web application for finding new retrotransposons and detecting L1 retrotransposition signatures.* **Lucier, J.F., et al., et al.** 2007, Nucleic Acids Res., Vol. 35, pp. W269-W274.

72. *EMBOSS: the European Molecular Biology Open Software Suite.* **Rice, P., Longden, I. et Bleasby, A.** 2000, Trends Genet., Vol. 16, pp. 276–277.

73. *Genome-wide analysis of the Emigrant family of MITEs of Arabidopsis thaliana.* **Santiago, N., et al., et al.** 12, 2002, Mol Biol Evol., Vol. 19, pp. 2285-93.

74. **Gordon, A.D.** *Classification*. New York : Chapman & Hall/CRC, 1999.

75. *A fast bit-vector algorithm for approximate string matching based on dynamic progamming*. **Myers, G.** s.l. : Springer-Verlag LNCS Series, 1998. Ninth Combinatorial Pattern Matching Conference. Vol. 1448, pp. 1–13.

76. *Inverted repeat structure of the human genome: the X-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes.* **Warburton, P.E., et al., et al.** 10A, 2004, Genome Res., Vol. 14, pp. 1861-9.

77. *MUST: a system for identification of miniature inverted-repeat transposable elements and applications to Anabaena variabilis and Haloquadratum walsbyi.* **Chen, Y., Zhou, F. et Li, G .and Xu, Y.** 1-2, 2009, Gene, Vol. 436, pp. 1-7.

78. *Miniature inverted-repeat transposable elements (MITEs) have been accumulated through amplification bursts and play important roles in gene expression and species diversity in Oryza sativa.* **Lu, C., et al., et al.** 3, 2012, Mol Biol Evol, Vol. 29, pp. 1005-1017. doi: 10.1093/molbev/msr282.

79. *MITE-Hunter: a program for discovering miniature inverted-repeat transposable elements from genomic sequences.* **Han, Y. et Wessler, S.R.** 22, 2010, Nucleic Acids Res., Vol. 38, p. e199.

80. *MITE Digger, an efficient and accurate algorithm for genome wide discovery of miniature inverted repeat transposable elements.* **Yang, G.** 186, june 2013, BMC Bioinformatics, Vol. 14. doi:10.1186/1471-2105-14-186.

81. *Graph clustering via a discrete uncoupling process.* **Dongen, S.V.** 2008, SIAM. Journal. on. Matrix. Analysis. and. Applications., Vol. 30, pp. 121–141.

82. *Structure-based discovery and description of plant and animal Helitrons.* **Yang, L. et Bennetzen, J.L.** 31, 2009, P.N.A.S., Vol. 106, pp. 12832-7.

83. *DINAMelt web server for nucleic acid melting prediction.* **Markham, N. et Zuker, M.** 2005, Nucleic Acids Res., Vol. 33, pp. 577–581.

84. **Charras, C. et Lecroq, T.** *Handbook of exact string matching algorithms.* s.l. : King's College publications, 2004. ISBN:0954300645.

85. *Linear pattern matching algorithms.* **Weiner, P.** s.l. : IEEE Computer Society Washington, DC, USA, 1973, SWAT '73 Proceedings of the 14th Annual Symposium on Switching and Automata Theory, pp. 1-11. doi:10.1109/SWAT.1973.13.

86. *On-line construction of suffix trees.* **Ukkonen, E.** 3, 1995, Algorithmica, Vol. 14, pp. 249-260. doi:10.1007/BF01206331.

87. **Aluru, S. et Ko, P.** *Handbook of Computational Molecular Biology.* [éd.] S. Aluru. s.l. : Chapman & Hall/CRC Computer and Information Science Series, 2006. p. Chapter 5 and 6.

88. *Compressed suffix tree--a basis for genome-scale sequence analysis.* **Välimäki, N., et al., et al.** 5, s.l. : Oxford University Press, 2007, Bioinformatics, Vol. 23, pp. 629-30. doi:10.1093/bioinformatics/btl681.

89. **Mäkinen, V.** Compressed Suffix Tree. [En ligne] 2013. http://www.cs.helsinki.fi/group/suds/cst/.

90. *Suffix arrays: a new method for on-line string searches.* **Manber, U.i et Myers, G.** 1993, SIAM Journal on Computing, Vol. 22, pp. 935–948. doi:10.1137/0222058.

91. *Linear work suffix array construction.* **Kärkkäinen, J., Sanders, P. et Burkhardt, S.** [éd.] ACM. 6, Nov. 2006, Journal of the ACM (JACM), Vol. 53, pp. 918-936. doi>10.1145/1217856.1217858.

92. *Two Efficient Algorithms for Linear Time Suffix Array Construction.* **Nong, G., Zhang, S. et Chan, W.H.** [éd.] IEEE. 10, oct. 2011, IEEE Transactions on Computers, Vol. 60, pp. 1471 - 1484. DOI:10.1109/TC.2010.188.

93. *A bioinformatician's guide to the forefront of suffix array construction algorithms.* **Shrestha, A. M. S., Frith, M. C. et Horton, P.** jan. 2014, Brief. Bioinform. doi:10.1093/bib/bbt081.

94. **Weiss, Dawid.** jsuffixarrays. [En ligne] 2011. https://github.com/carrotsearch/jsuffixarrays.

95. *Efficient repeat finding in sets of strings via suffix arrays.* **Barenbaum, P., et al., et al.** 2, 2013, Discrete Mathematics & Theoretical Computer Science, Vol. 15, pp. 59-70.

96. **Becher, V.** findrepset. [En ligne] 2013. http://www.dc.uba.ar/people/profesores/becher/software/findrepset.tar.bz2.

97. **Burrows, M. et Wheeler, D.J.** *A block sorting lossless data compression algorithm.* Digital Equipment Corporation. Palo Alto : s.n., 1994. Technical Report. 124.

98. *Opportunistic data structures with applications.* **Ferragina, P. et Manzini, G.** 2000, FOCS '00 Proceedings of the 41st Annual Symposium on Foundations of Computer Science, pp. 390-398. doi:10.1109/SFCS.2000.892127.

99. *An Experimental Study of an Opportunistic Index.* **Ferragina, P. et Manzini, G.** Washington, D.C., USA : Society for Industrial and Applied Mathematics, 2001. Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 269--278. isbn 0-89871-490-7.

100. **Ferragina, P. et Navarro, G.** Compressed Indexes and their Testbeds. [En ligne] sept. 2005. http://pizzachili.di.unipi.it/.

101. **Jenkin, B.** SpookyHash. [En ligne] 2012. http://burtleburtle.net/bob/hash/spooky.html.

102. **Google.** Sparsehash. [En ligne] 2012. http://code.google.com/p/sparsehash/.

103. *RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data.* **Zhao, Y., Tang, H. et Ye, Y.** 1, s.l. : Oxford pub., 2012, Bioinformatics, Vol. 28, pp. 125-126. 10.1093/bioinformatics/btr595.

104. **Zhao, Y. et Ye, Y.** RAPSearch2. [En ligne] June 2014. http://omics.informatics.indiana.edu/mg/RAPSearch2/.

105. *Fast and accurate short read alignment with Burrows–Wheeler transform.* **Li, H. et Durbin, R.** 14, s.l. : Oxford Pub., 2009, Bioinformatics, Vol. 25, pp. 1754-1760. 10.1093/bioinformatics/btp324.

106. *YASS: enhancing the sensitivity of DNA similarity search.* **Noe, L. et Kucherov, G.** 2, 2005, Nucleic Acids Research, Vol. 33, pp. W540-W543.

107. **Noe, L.** Yass. [En ligne] 2013. http://bioinfo.lifl.fr/yass/.

108. *Sequence analysis of two alleles reveals that intra- and intergenic recombination played a role in the evolution of the radish fertility restorer (Rfo).* **Mora, J.R.H., et al., et al.** 35, 2010, BMC Plant Biol, Vol. 10.

109. *DisLex: a transformation for discontiguous suffix array construction.* **Horton, P., Kiełbasa, S.M. et Frith, M.C.** 2008. Workshop on Knowledge, Language, and Learning in Bioinformatics, KLLBI. Pacific Rim International Conferences on Artificial Intelligence (PRICAI). pp. 1-11.

110. *Identification of common molecular subsequences.* **Smith, T.F. et Waterman, M.S.** [éd.] Elsevier. 1, March 1981, Journal of Molecular Biology, Vol. 147, pp. 195-197. doi: 10.1016/0022-2836(81)90087-5.

111. *Versatile and open software for comparing large genomes.* **Kurtz, S., et al., et al.** 2, 2004, Genome Biology, Vol. 5, p. R12. doi:10.1186/gb-2004-5-2-r12.

112. *NextGenMap: fast and accurate read mapping in highly polymorphic genomes.* **Sedlazeck, F.J., Rescheneder, P. et von Haeseler, A.** 21, 2013, Bioinformatics, Vol. 29, pp. 2790-1. doi: 10.1093/bioinformatics/btt468..

113. **Sedlazeck, F.J. et Rescheneder, P.** NextGenMap. [En ligne] 2014. http://cibiv.github.io/NextGenMap/.

114. *De novo assembly and genotyping of variants using colored de Bruijn graphs.* **Iqbal, Z., et al., et al.** 2, Jan. 2012, Nature Genetics, Vol. 44, pp. 226-232. doi: 10.1038/ng.1028.

115. **Peterlongo, P.** discoSnp. [En ligne] 2014. http://colibread.inria.fr/software/discosnp/.

116. *RepARK—de novo creation of repeat libraries from whole-genome NGS reads.* **Koch, P., Platzer, M. et Downie, B. R.** 9, 14 March 2014, Nucl. Acids Res., Vol. 42, p. e80. doi:10.1093/nar/gku210.

117. *Automated generation of heuristics for biological sequence comparison.* **Slater, G.S. et Birney, E.** 1, 2005, BMC Bioinformatics, Vol. 6. doi:10.1186/1471-2105-6-31.

118. *Replication of analysis of published microarray gene expression analyses.* **Ioannidis, J. P. A., et al., et al.** 2, Feb. 2009, Nature Genetics, Vol. 41, pp. 149–155. doi: 10.1038/ng.295.

119. *The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud.* **Wolstencroft, K., et al., et al.** W1, 2013, Nucl. Acids Res., Vol. 41, pp. W557-W561. doi:10.1093/nar/gkt328.

120. *ScanProsite: detection of PROSITE signature matches and ProRule-associated functional and structural residues in proteins.* **de Castro, E., et al., et al.** Jul. 2006, Nucl. Acids Res., Vol. 34(Web Server issue), pp. W362-W365. doi: 10.1093/nar/gkl124.

121. **de Castro, E.** ps_scan. [En ligne] 2002. ftp://ftp.expasy.org/databases/prosite/ps_scan/.

122. *A Composite Method Based on Formal Grammar and DNA Structural Features in Detecting Human Polymerase II.* **Datta, S. et Mukhopadhyay, S.** 2, 2013, PLoS ONE, Vol. 8, p. e54843. doi:10.1371/journal.pone.0054843.

123. *RNAMotif: A new RNA secondary structure definition and discovery algorithm.* **Macke, T., et al., et al.** 22, nov. 2001, Nucl Acids Res, Vol. 29, pp. 4724–4735. doi: 10.1093/nar/29.22.4724.

124. **Macke, T.** RNAMotif. [En ligne] april 2010. http://casegroup.rutgers.edu/casegr-sh-2.5.html.

125. *Locomotif: from graphical motif description to RNA motif search.* **Reeder, J., Reeder, J. et Giegerich, R.** 13, 2007, Bioinformatics, Vol. 23, pp. 392-400. doi: 10.1093/bioinformatics/btm179.

126. *Structator: fast index-based search for RNA sequence-structure patterns.* **Meyer, F., et al., et al.** 214, may 2011, BMC Bioinformatics., Vol. 12. doi:10.1186/1471-2105-12-214.

127. *Replacing suffix trees with enhanced suffix arrays.* **Abouelhoda, M.I., Kurtz, S. et Ohlebusch, E.** 1, march 2004, Journal of Discrete Algorithms, Vol. 2, pp. 53-86. doi:10.1016/S1570-8667(03)00065-0.

128. *MIPS PlantsDB: a database framework for comparative plant genome research.* **Nussbaumer, T., et al., et al.** 2013, Nucl. Acids Res., Vol. 41 (Database issue), pp. D1144-51.

129. **Brendel, V.** Pattern Search. [En ligne] 2007. http://www.plantgdb.org/cgi-bin/vmatch/patternsearch.pl.

130. **Jensen, K., Stephanopoulos, G. et Rigoutsos, I.** Biogrep: a multi–threaded pattern matcher for large pattern sets. *kljensen/biogrep · GitHub.* [En ligne] 2002. https://github.com/kljensen/biogrep.

131. *The language of genes.* **Searls, D.B.** 6912, 2002, Nature, Vol. 420, pp. 211-217.

132. *String Variable Grammar: a logic grammar formalism for DNA sequences.* **Searls, D. B.** 1,2, 1995, Journal of Logic Programming, Vol. 24, pp. 73-102.

133. *Gene structure prediction by linguistic methods.* **Dong, S. et Searls, D. B.** 1994, Genomics, Vol. 23, pp. 540-551.

134. *PatSearch: A program for the detection of patterns and structural motifs in nucleotide sequences.* **Grillo, G., et al., et al.** 13, 2003, Nucl. Acids Res., Vol. 31, pp. 3608-12. doi: 10.1093/nar/gkg548.

135. **Overbeek, R.** ScanForMatches. [En ligne] jul. 2010. http://blog.theseed.org/servers/2010/07/scan-for-matches.html.

136. *Expressive Pattern Matching with Logol. Application to the Modelling of -1 Ribosomal Frameshift events.* **Belleannée, C., Sallou, O. et Nicolas, J.** Rennes : s.n., 2012. JOBIM'2012. pp. 5-14. http://jobim2012.inria.fr/jobim_actes_2012_online.pdf.

137. **Sallou, O.** Logol. [En ligne] 2014. http://logol.genouest.org.

138. *The TIGR Plant Repeat Databases: a collective resource for the identification of repetitive sequences in plants.* **Ouyang, S. et Buell, C.R.** 2004, Nucl. Acids Res., Vol. 32 (Database issue), pp. D360-3.

139. *MASiVEdb: the Sirevirus Plant Retrotransposon Database.* **Bousios, A., et al., et al.** 158, 2012, BMC Genomics, Vol. 13.

140. *P-MITE: a database for plant miniature inverted-repeat transposable elements.* **Chen, J., et al., et al.** 2013, Nucl. Acids Res., Vol. 42 (Database issue), pp. D1176-81. doi: 10.1093/nar/gkt1000.

141. *RBR: library-less repeat detection for ESTs.* **Malde, K., et al., et al.** 18, 2006, Bioinformatics, Vol. 22, pp. 2232-6.

142. *ReAS: Recovery of ancestral sequences for transposable elements from the unassembled reads of a whole genome shotgun.* **Li, R., et al., et al.** 4, 2005, PLoS Comput Biol., Vol. 1, p. e43.

143. *The TIGR Plant Repeat Databases: a collective resource for the identification of repetitive sequences in plants.* **Ouyang, C.R. et Buell, S.** 2004, Nucleic Acids Res., Vol. 32 (Database issue), pp. D360-3. http://plantrepeats.plantbiology.msu.edu/.

144. *Rapid Identification of Repeated Patterns in Strings, Trees and Arrays.* **Karp, R.M., Miller, R.E. et Rosenberg, A.L.** New York, NY, USA : ACM, 1972. STOC '72, Fourth Annual ACM Symposium on Theory of Computing. pp. 125-136.

145. *RJPrimers: unique transposable element insertion junction discovery and PCR primer design for marker development*. **You, F.M., et al., et al.** sup. 2, 2010, Nucl. Acids Res., Vol. 38, pp. W313-20.

146. *Automated paleontology of repetitive DNA with REANNOTATE.* **Pereira, V.** 2008, BMC Genomics, Vol. 9, p. 614.

147. *MITE Digger, an efficient and accurate algorithm for genome wide discovery of miniature inverted repeat transposable elements.* **G., Yang.** 186, 2013, BMC Bioinformatics, Vol. 14. doi:10.1186/1471-2105-14-186.

148. *ScanProsite: detection of PROSITE signature matches and ProRule-associated functional and structural residues in proteins.* **de Castro E, Sigrist CJA, Gattiker, A., et al., et al.** 2, 2006, Nucl Acids Res., Vol. 34, pp. W362–W365. doi: 10.1093/nar/gkl124.

149. *Transposon Insertion Finder (TIF): a novel program for detection of de novo transpositions of transposable elements*. **Nakagome, M., et al., et al.** 71, s.l. : Biomed, 2014, BMC Bioinformatics, Vol. 15. doi:10.1186/1471-2105-15-71.