ABSTRACT

RAIDER: RAPID AB INITIO DETECTION OF ELEMENTARY REPEATS

by Nathaniel Figueroa

De novo repeat discovery is increasingly important due to the growth rate of new genomic data. Library-based programs such as RepeatMasker [1] effectively expand known families of repeats, but discovering new families is difficult due to their inexact nature. Tools relying on self-alignment (e.g RECON [2]), become prohibitively time-consuming with large sequences, while text-indexing methods, such as the Suffix Array or FM-Index, are poorly suited for the wildcard searches needed to account for single base mismatches. We present a tool, RAIDER, that uses spaced seeds in the spirit of PatternHunter [3] to identify inexact repeats with wildcard matching. RAIDER's speed allows extensive parameter tuning, processing Human Chromosome 22 in approximately 1 minute (as compared to 39 minutes for RepeatScout or longer for RECON). RAIDER shows great promise in terms of both sensitivity and specificity, with results comparable to RepeatScout, and much potential for improvement with unexplored spaced-seed patterns.

RAIDER: RAPID AB INITIO DETECTION OF ELEMENTARY REPEATS

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Department of Computer Science and Software Engineering

by

Nathaniel David Figueroa

Miami University

Oxford, Ohio

2014

Advisor_____
(John E. Karro)

Reader _____
(James Kiper)

Reader _____
(Chun Liang)

Table of Contents

1 Introduction 1

2 Background 2

2.1 Biological Background 2

2.2 RepeatMasker 3

2.3 Suffix tree techniques 3

2.4 Pattern Hunter 4

3 Definitions and Established Relationships 5

3.1 Repeat definitions 5

3.2 Lmer relationship to elementary repeats 7

3.3 Inexact matches 11

4 First Approach: SCANER 12

4.1 Seed-Pair 12

4.2 Pair-Chain 14

4.3 Discussion 15

5 RAIDER: A Refined Approach 15

5.1 Overview 15

5.2 Pseudocode 17

5.3 Inexact Matching 18

6 RAIDER Complexity Analysis 20

6.1 Space 20

6.2 Runtime 21

6.3 Space optimization via Prescan 22

6.4 Multi-Prescan 23

7 Results 24

7.1 Testing: Tool Parameters and Benchmark Sets 25

7.2 Metrics 25

7.3 Runtime 26

7.4 False Negatives 26

7.5 False Positives 26

7.6 Redundancy 26

7.7 Comparison Summary 27

7.8 Chromosome 1 and Whole-Genome Tests 28
8 Linear-time RAIDER 28
8.1 Overview 28
8.2 Step One: Count Lmers 29
8.3 Step Two: Mark Families 30
8.4 Pseudocode 31
8.5 Proof of Correctness 32
8.6 Inexact Matching 33
8.7 Performance 33
9 Conclusion 34
10 Appendices 34
10.1 Appendix A: Composite Repeats 34
10.2 Appendix C: The Pigeon-holed Repeats Problem 36
10.3 Appendix D: Simple RAIDER Python 37
11 Works Cited 40

List of Tables

Table 1: List of sequences analyzed and respective families preserved for identification benchmarking. 25

Table 2: Spaced seeds used for RAIDER benchmark. 25

Table 3: Real data test results. Notice that RAIDER is an order of magnitude faster while still covering nearly as much of the ancestor as does RepeatScout. False positive error is also generally lower. ErrMissed is high for RAIDER, but this is to be expected given that RAIDER only returns locations of whole elementary repeat instances, whereas RepeatScout uses RepeatMasker, which will identify all fragments of a given seed. Note also that this error generally decreases as spacing is added to the search seed 27

Table 4: Memory footprint and runtime for large sequences 28

Table 5: Results of SCANER + Composite-finding was still fast, but not as fast as RAIDER alone and did not perform any better. 35

List of Figures

Figure 1: Huo, et. al. method of tabulating Lmer frequencies. Frequencies occurring in tandem reveal possible repeats to be confirmed by the suffix tree. E.g. The consecutive frequency 2 for ABX and BXB reveals the repeat ABXB. 4

Figure 2: Three elementary repeat families. Note that although C only occurs after an instance of A or B, neither A or B can exclusively claim C as a subrepeat, thus C becomes its own element family. However, AC and BC both form composite repeat families. 6

Figure 3: When *L = 5*, a given base belongs to 5 Lmers 7

Figure 4: A family and its Lmer decomposition 9

Figure 5: Given a mask of 101, we still consider these two strings a match since all bases in each are covered by hits occurring in both strings. 12

Figure 6: Seed-Pair maximizes consecutive pairs found over the same distance. 13

Figure 7: Pair-Chain splits repeats into elements by merging on their intersecting regions. 14

Figure 8: A family is spliced into two families when an incomplete instance of the original family is found. 17

Figure 9: Spaced seeds allow a degree of error by treating spaced regions as wildcards. 20

Figure 10: Worst-case Lmer sequence requiring the original repeat be spliced in half *logk* times. 22

Figure 11 22

Figure 12: By only scanning for Lmers found in one region at a time, we eliminate the need to simultaneously store all Lmers in the hash table. 24

Figure 13: Step one of RAIDER: counting Lmers to create an ordered array of all significant Lmers. Assume *L*=3 and *f*=8. 29

Figure 14: Step two of RAIDER: Marking the boundaries of each family's Lmer series in LmerTable. Assume *L*=3 and *f*=8. 30

Figure 15: When both steps of RAIDER are complete, LmerTable will contain all elementary families, demarcated by start flags. 30

1 Introduction

Repeat regions of the genome are important for many reasons. They are a substantial portion of mammalian genomes, upwards of 45% in the human genome [4]. For example, ALU elements are a family of repeats where each instance is only about 300 bases long, but occurs so frequently in the human genome that they make up roughly 10% of the entire sequence [5]. Repeats are also suspected to play important roles in gene development. It is known that Transposable Elements (TEs), a special class of repeats, are capable of self-replication: they carry instructions which allow them to copy, or simply transpose (move), themselves to another location in the genome. On occasion portions of neighboring genes are moved or copied along with TEs, allowing for large mutations with unpredictable consequences [6].

The genome is traditionally modeled as a string over the alphabet {A,C,G,T}; repeats are (initially) identical substrings distributed throughout the larger genomic string. However, genome sequences tend to be large –running into billions of base-pairs (characters). Finding repeats by comparing a sequence to itself requires quadratic time with respect to length when using standard string-matching algorithms. Repetitive structures themselves can be difficult to identify [6], given the inexact nature of the repeats. Over time repeat sequences accumulate minor mutations (changes to the individual strings). These mutations may not affect the function of the sequence, but the sequence will be miscategorized by exact string-matching algorithms. Many repeats are inactive structures that have been heavily obfuscated by their mutations over millennia: they have not retained any function that would encourage conservation of the sequence pattern, yet their membership in a specific family remains scientifically significant. TEs introduce a third complication in that they can copy neighboring data: the challenge then becomes to discern where the repeat ends and the hitchhiking data begins [6].

Repeat discovery is currently best accomplished through tools that rely on a database of already annotated repeats, which is the approach taken by popular tools such as RepeatMasker [1]. This approach allows the efficient identification of new family members, but when analyzing new sets of genomic data only known families of repeats will be identified; the need for an effective *de novo* search method remains [7].

Several proposed methods rely on the use of suffix trees [8, 7, 9], which are capable of grouping repetitive sequences into branches of a tree in $O(n)$ time (where n is the length of the sequence) [10, 11]. Suffix trees, however, rely heavily on exact matching; allowing wildcard mismatches multiplies time and space requirements. Another potential drawback

is the shear sophistication of suffix tree algorithms, making them difficult to adapt to new approaches. Lastly, it is important that a *de novo* search be fast enough to allow for parameter tuning, as the size, count, and exactness of repeats being sought can vary widely between use cases.

2 Background

In this section we provide a cursory explanation of the molecular biology behind DNA sequences and repeat formation. We briefly describe other techniques for finding repeats, introduce the PatternHunter technique that will be important to our own method, and define terms necessary to understand its importance to our technique.

2.1 Biological Background

A deoxyribonucleic acid (DNA) sequence is a chain of *nucleotides* (or *bases*), each of which is one of four molecules: adenine, cytosine, guanine, and thymine. In sequence analysis, where we are looking purely at the sequence pattern and are unconcerned with the chemical properties of these molecules, we traditionally model DNA as a string representing the order of the molecules, representing each nucleotide type as a single character derived from the first letter of its name. Thus we represent a DNA sequence as a string over the alphabet {A,C,G,T}. A genome, the DNA sequence carried in each cell of a given organism, is then a very large string (of size $\approx 3.3 \times 10^9$ for humans). Components of the genome (e.g. chromosomes, genes, repeats) are substrings of that genomic string that have specific biological associations.

Two DNA sequences are *homologous* if they descend from a common ancestor. Note that a DNA sequence is not static, but is subject to substitutions – changes to the sequence, and thus to the representative string. The rate at which these changes occur depends on many factors, but given enough time almost any sequence will pick up some modifications. Thus, if we take one sequence, make two copies of it, and allow those copies to evolve independently, they will accrue differences over time. But the fact remains that they started in the same place (that is, both descend from the same original sequence), and hence remain homologous. Much study has been given to the development of techniques that will identify homologous sequences that have incurred extensive modifications [3, 11, 12].

A **repeat** is a sequence that is similar to another sequence within the same genome. There is no strict definition of similarity to qualify a repeat; RepeatMasker uses an alignment-based metric [1], and we subscribe to the elementary-repeat definition defined

in Section 3. Similar repeats are logically grouped into **families**; the aim of de novo repeat identification is to find candidate undiscovered families of repeats.

Transposable elements (TEs) are a special class of repeats that are capable of self-replication: a TE can copy itself to another part of the genome, or simply move (transpose) its sequence from one part of the genome to another. Some retroviruses that have become part of the human genome also possess this ability to self-replicate and are considered TEs as well. Unsurprisingly, TEs are known to be responsible for a number of diseases [13]. Most TEs are, or eventually become, evolutionarily neutral, having no impact on the biology of its host organism.

As TEs copy themselves, the host genome ends up with a family of homologous TE copies that begin to diverge over generations. Eventually mutations occur that disable these TEs' abilities to self-replicate, but their sequences remain in the genome and those that are neutral continue to change over generations, without conservation pressure (since they do not affect the organism's fitness).

2.2 RepeatMasker

Techniques for finding known repeats in a given sequence are well developed. RepeatMasker [1] is the de facto standard for *library-based* repeat searches: given a list of consensus sequences representing known families, RepeatMasker searches for new members of each family based on similarity to the consensus sequences. Because RepeatMasker matches are based on alignment similarity, RepeatMasker is able to account for the sequence variation between family members introduced by the effects of molecular evolution.

Of course, RepeatMasker is only useful given the existence of an initial library of known repeats. Without such a pre-compiled library, we must look at *de novo* search tools: tools attempting to find significant repeats based solely on the initial sequence composition. This result is then a library of repeats that can be used by tools such as RepeatMasker to find all further instances and fragments in the sequence.

2.3 Suffix tree techniques

Suffix trees and **suffix arrays** are data structures that can be built from a text S of length n in $O(n)$ time and then used to search S for any substring s of S in $O(|s|)$ time. Once the tree has been built, the search time is linearly bound in the length of the query [11] [14].

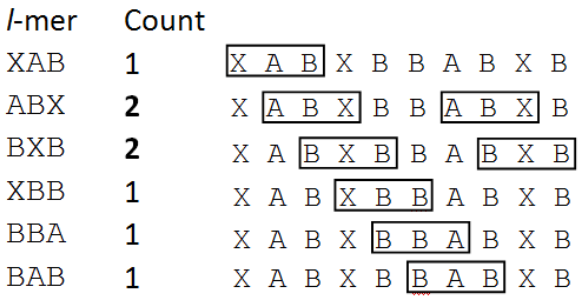


Figure 1: Huo, et. al. method of tabulating Lmer frequencies. Frequencies occurring in tandem reveal possible repeats to be confirmed by the suffix tree. E.g. The consecutive frequency 2 for ABX and BXB reveals the repeat ABXB.

In *Huo et al.* the authors introduce a repeat finding technique that employs suffix trees, but takes a mixed approach that leaves room for expansion [8]. Each Lmer (a string of size L) contained within the sequence is identified and put into a table, allowing the tabulation of the number of times each Lmer occurs in the entire sequence [8] using an $O(n)$ hash-table based approach. After creating the table and counting the frequencies of all Lmers, those which occur in tandem and with equal frequency are grouped together, and a suffix tree is used to see if the frequency of the combined Lmers equals the frequency of the individual Lmers. If so, and if that frequency is sufficiently high, a significant repeat has been found. Later the repeat is checked to see if it is elementary, a concept discussed in Section 3.

We wished to expand on this approach by incorporating the PatternHunter technique (described in the next section) to allow more flexibility in identifying inexact matches. In the process we found that we could use hash table to store more than just counts, ultimately eliminating the need for a suffix tree altogether.

2.4 Pattern Hunter

Techniques for efficiently querying large sequences of data for string matches have become critical to bioinformatics due to the dramatic growth of available data. One particularly successful technique, PatternHunter [3], has shown remarkable sensitivity despite the simplicity of the approach: as a substitute for searching for exact substring matches, we search for portions of a longer string. We call this a **spaced-seed** pattern: essentially a search string with portions of it replaced with wild card characters (the "spaces"), leaving multiple spaced segments to be searched for (the "seeds"). The spacing allows for mismatches and thereby increases sensitivity, but has been shown not to significantly increase false positive matches [3].

Following the convention of *Ma et al.* [3], we represent a spaced with a binary string: a 1 characters require a match and the 0 characters represents a one-character wild card. For example, the spaced seed pattern 10101 would require matches in the first, third and fifth character of any substring being evaluated, as illustrated in Figure 5. For example, given a spaced seed s of length L and two genomic strings q and t, also of length L, we say that q **hits** t with respect to s if $q_i = t_i$ for all i such that $s_i = 1$. For example, for $s = 10011$, $q = AACAA$ hits $t = AAAAA$ (with respect to s) because they match at all positions where s is 1; $q = AACAA$ does not hit t = TACAA.

3 Definitions and Established Relationships

In this section we lay out several concepts that are crucial to understanding the RAIDER algorithm. In Section 3.1 we give formal definitions for elementary repeat and elementary repeat families, as the goal of RAIDER is to identify elementary repeats and group them into families. In 3.2 we address the significance of Lmers in searching for elementary repeats, and pose several lemmas to establish relationships that are significant to RAIDER. Finally in Section 3.3 we give a formal criteria for accepting inexact repeats.

3.1 Repeat definitions

We will be formulating the problem computationally in the framework of string matching, and we will follow the notation and definitions proposed by Zheng and Lonardi for working with repeats [9].

A family of transposable elements is a set of copies of a substring occurring in multiple locations across the genome. To make the problem amenable to computation we would like a clean mathematical definition of a repeat, but the nature of biological systems makes this difficult [2]. For example, we would like to define a family as a set of similar substrings that are maximal in length. However, because transposable elements can occasionally (accidently) carry neighboring sequences in their transition [6], the emphasis on maximality can diminish the quality of repeat identification when a few repeats that happen to carry neighboring data lead to the exclusion of shorter but equally significant repeats of the same family. Zheng and Lonardi [9] attempt to deal with this by considering both maximality and **frequency**: the number repeat instances in a sequence. Instead of trying to initially deal with entire repeats, Zheng and Lonardi propose to define **elementary repeats** as the basis for defining repeat families. Elementary repeats are the basic building blocks of full repeats, meeting minimum criteria for length and frequency,

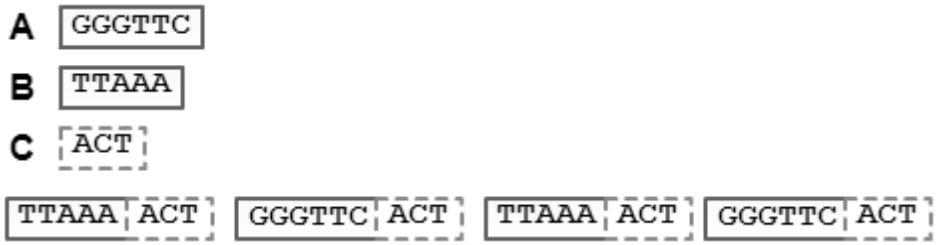


Figure 2: Three elementary repeat families. Note that although C only occurs after an instance of A or B, neither A or B can exclusively claim C as a subrepeat, thus C becomes its own element family. However, AC and BC both form composite repeat families.

and must be elementary in the sense that they themselves cannot be composed of other elementary repeats [9].

Definition 1 Elementary Repeat

A repeat family is considered *elementary* if there are a significant number of repeats, the repeats are maximally identical and significant in length, and no significant substring of the repeat occurs more times than the family has repeats. That is, *P* is an elementary repeat if:

1. *P* has a length of at least some fixed threshold value *L*.
2. *P* occurs at least *f* times, where *f* is a specified minimum frequency for elementary repeats.
3. Every substring of *P* that meets condition (1) must have the same number of occurrences as does *P* (i.e. does not occur apart from *P*).
4. There is no string *P'* that properly contains all occurrences of *P* in the query sequence.

In other words, *P* must be long enough, must appear with sufficient frequency, cannot have significantly long substrings appearing independently of *P*, and cannot be a part of a sequence that could itself be regarded as an elementary repeat. We say an algorithm finds an elementary repeat instance if it properly identifies the start and end coordinates of the repeat, and an algorithm groups a repeat if it assigns all instances to one family of repeats.

Defining elementary repeats is useful in biological scenarios because it is both precise and flexible. The advantages are illustrated in Figure 2: focusing on maximality would have absorbed family C into families A and B. C's reuse in multiple families might make it an important component worth identifying.

Alternatively, a biologist less interested in these basic elements could focus instead on composite repeats. We define and discuss our work with composite repeats in Appendix A.

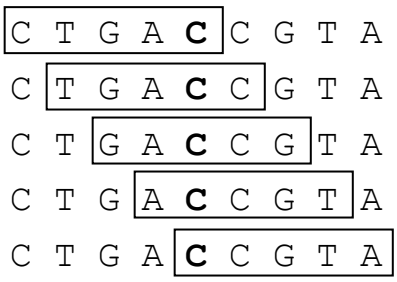


Figure 3: When *L = 5*, a given base belongs to 5 Lmers

3.2 Lmer relationship to elementary repeats

The approach we will be taking towards elementary repeat identification involves finding repeated Lmers and expending them to find covering repeat elements. To do this, we first must establish a few relationships between Lmers and elementary repeats, starting with two basic observations:

Observation 1: Any Lmer occurring *f* or more times must be either an elementary repeat or part of a longer elementary repeat.

When an Lmer occurs at least *f* times, conditions (1) and (3) of the Definition 1 clearly hold, and (4) must hold trivially (since the Lmer has no substrings of length *L*). Thus the only way such an Lmer can fail to be an elementary repeat is if it is not maximal – if it is properly contained with an elementary repeat.

Observation 2 Any given base will belong to *L* different Lmers.

Notice a specific Lmer occurring at a specific index *i* will overlap another Lmer occurring at index *i*+1. They will overlap by *L*-1 bases, and thus a specific (*L*-1)mer can belong to 2 Lmer. Likewise a specific (*L*-2)-mer can be overlapped by 3 Lmers, and so forth until we have a 1-mer, which can be overlapped by *L* Lmers. This is illustrated in Figure 3, and tells us that distinct elementary repeats will frequently have overlapping ends (of length no greater than *L*-1).

Now: given that an Lmer occurring at least *f* times is either an elementary repeat or contained in one, the question is how to find the boundaries of the containing elementary

repeat. In order to do this we will exploit a few properties of Lmers and elementary repeats, proved to be true in the following.

Lemma 1 Every Lmer belongs to at most one elementary repeat family.
Proof: Assume an Lmer x belongs to two families, F_1 and F_2. This makes x a substring of F_1 that is significant in length but must occur more times in the query sequence than does F_1 (as x also occurs in F_2) – thus violating condition (4) of the Definition 1. □

Lemma 1 reduces the complexity of searching for the boundaries of a containing repeat, as it ensures us that there is a unique set of boundaries. It is perhaps the most important point when trying to understand our final algorithm (RAIDER). In the course of the algorithm, we will say an Lmer x is assigned to a family F – meaning that *all* instances of x belong to F. If we re-assign x to distinct family G, it means all instances of x now belong to family G, and F must no longer contain x (as the lemma tells us x can only be in one family).

To continue discussion it will now help to have an operator used to denote the merging of strings based on the largest string that is both a suffix of one and a prefix of the other. For example, given the strings AA<u>CC</u> and <u>CC</u>GG, we would like to indicate that we will merge then over the common CC substring, resulting in AACCGG.

Definition 2 Given strings x and y, we define **x∘y** as the string abc, where: $x=ab$, $y=bc$, and b is the longest substring that is both a suffix of x and a prefix of y.

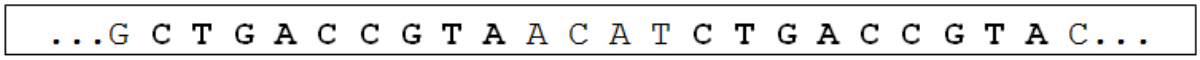
Note the distinction between ∘ and ·, the standard symbol used for string concatenation in the literature.

Having defined this, we can now define the concept of an Lmer series:

Definition 3 An **Lmer series** is sequence of Lmers $x_0, x_1, \ldots, x_{k-1}$ in the query sequence such that the start coordinate of x_i is one greater than the start coordinate of x_{i-1}. We say a sequence F is **composed** from an Lmer series if $F = x_0 \circ x_1 \circ \ldots \circ x_{k-1}$.

Note that for any Lmer series $x_0, x_1, \ldots, x_{k-1}$:

- We refer to the length of the series S, $||S||$, as the number of Lmers in the series.
- We denote the i^{th} Lmer in series S as $S[i]$.



A **family** refers to a set of repeat instances in the query sequence. The sequence above contains a family with two instances of CTGACCGTA.

Each family has a unique **Lmer decomposition**, the **Lmer series** which defines the family. If L=5, the family in the sequence shown above has the decomposition $x_0 \circ x_1 \circ x_2 \circ x_3 \circ x_4$ (illustrated on right).

Figure 4: A family and its Lmer decomposition

- For $0 \le i < k$-1: $|x_i \circ x_{i+1}| = L+1$ (or: $x_i = a \circ x$ and $x_{i+1} = x \circ b$, where $|a| = |b| = 1$ and $|x| = L$-1).
- If a sequence $F = x_0 \circ x_1 \circ \ldots \circ x_{k-1}$, then $|F| = L+k$-1.

Definition 4 Given a subsequence F of the query sequence, the **Lmer decomposition** of F is the Lmer series $x_0, x_1, \ldots, x_k$ such that F is composed of the series.

It should be obvious to the reader that any subsequence F has a unique Lmer decomposition of length $|F|+1-L$. We can think of an elementary repeat either as a sequence repeated many times, or an Lmer series (the sequence decomposition) repeated many times. For convenience, we will start to use the two interchangeably (referring to an Lmer as either a substring of the repeat or an element of the repeat – meaning an Lmer of its decomposition) as context demands.

As we are discussing RAIDER, we will consider a potential elementary sequence in terms of its Lmer decomposition and working bottom up – identifying Lmers that are substrings of elementary repeats and merging them into Lmer series.

Finally, we wish to present an alternative definition of an elementary repeat. While Definition 1 is more intuitive, the following will be more useful in our future discussions.

Definition 5 Let F be a subsequence of the query sequence with an Lmer decomposition of $x_1, x_2, \ldots, x_k$ (where $k = |F| + L - 1$). F is an elementary repeat if:

1. $k \ge 0$
2. $freq(F) \ge f$
3. $freq(x_i) = freq(F)$ for all Lmers x_i in the decomposition.

4. k is maximal: there is no Lmer y such that the sequence $y \circ F$, or $F \circ y$, meets conditions 1-3. (That is, the length of the defining sequence of F is maximal with respect to conditions 1-3.)

Here $freq(s)$ denotes the number of occurrences of substring s in the query sequence.

Theorem: Definitions 1 and 5 are equivalent.

Proof: For brevity, we will refer to condition (x) of Definition 1 as condition 1.x, and condition 5.y will refer to condition (y) of Definition 5. Then we have:

- Conditions 5.1 and 1.1 are equivalent. F has at least k elements in its decomposition if and only if $|F| \geq L + 0$, which is condition 1.1.
- Conditions 5.2 and 5.2 are equivalent. Proof is trivial.
- Conditions 5.3 and 1.3 are equivalent:
    - (⇐) If a sequence F conforms to condition 1.3, then every substring of length at least L has a frequency equal to that F. Hence all Lmers have a frequency equal to that of F, satisfying 5.3
    - (⇒) Let $F_{u,v}$ ($u \leq v$) be the subsequence F composed from the Lmer series $x_u, x_{u+1}, \ldots, x_v$. Note that since all x_i have the same frequency, it follows that $freq(F_{u,v}) = freq(x_i) = freq(F)$ for any u,v. Thus all substrings of F have the same frequency of F, hence conforms to condition 1.3.

Finally we show the definitions are equivalent:

(⇒) Assume a sequence F conforms to Definition 5 (and thus, we now know, to conditions 1.1, 1.2, and 1.3. If F did not conform to 1.4, then there is a substring F' containing all occurrences of F. We can write $F' = x \circ F \circ y$ (where $\cdot$ is the string concatenation operator) and $|x|+|y| \geq 1$. Assume without loss of generality that $|y| \geq 1$, and notice that there would now be Lmer that could be added to the Lmer series for all occurrences of F – contradicting our assumption that 5.4 holds.

(⇐) Assume a sequence F conforms to Definition 1 (and thus, we now know, to conditions 5.1, 5.2, and 5.3). If F did not conform to 5.4, then there is an Lmer y that either immediately precedes F_0 or immediately follows F_k at all instances of F. Assume the latter without loss of generality. Then set $F' = F \circ x$, and we have the F' contradiction 3.4. □

From now on we will use Definition 5 by default, explicitly invoking Definition 1 when needed.

Lemma 2 If a substring s with length $k \geq L$ appears in two locations such that the two occurrences are maximally identical, s cannot be a proper substring of any elementary repeat.

Proof: Suppose the contrary: there is an elementary repeat r such that s is a proper substring. Let $x_0, x_1, \ldots, x_{k-1}$ be the series decomposition of r, and notice that the series decomposition of s must be $x_a, \ldots, x_b$ where either $0 < a$ or $b < k$-1 (since s is shorter than r). Let $s' = x_{a-1}$ if $a > 0$, or $s' = x_{b+1}$ otherwise. s' is then a substring of r, but renders s no longer maximal. Thus if r exists, s cannot be a maximal string. □

In short: given two maximally identically substrings, we have an immediate bound on the length of any elementary repeat. It may be that this substring is the consensus sequence for an elementary repeat, or it may be that this substring contains the consensus sequence for an elementary repeat – but Lemma 2 tells us that the substring cannot be contained *within* an elementary repeat, a result that will be crucial to our RAIDER algorithm in Section 5.

3.3 Inexact matches

To this point, our definitions of elementary repeats have required exact matches. That is, an Lmer series F is an elementary repeat if, by condition 1 of Definition 5, it occurs at least f or more times *exactly*. Each of these instance by be the same Lmer series, and thus represents a set of identical sequences. Our goal now is to generalize this definition to incorporate the earlier discussed spaced seed matching from PatternHunter [3].

Definition 6. Let s be a seed of width L and q be a sequence of length $n \geq L$. We say that sequence t matches q with respect to s if: (1) $|t| = n$; (2) for each i, $0 \leq i < n$, there exists a value j, $0 \leq j < n$-l such that $j \leq i < j+L$ and the substring $q[j : j+L]$ hits $t[j : j+L]$ with respect to seed s.

In other words, two repeats are a match (hence in the same family) if every base in each string is covered by a substring that will hit a substring in the opposing sequence with respect to a seed (as visualized in Figure 5). We can further expand this definition to allow for the use of multiple seeds, though we leave that to a future study.

Given this definition, it is a short step to relax the substring quality condition in our algorithm to that of a seed-based match. In relaxing the assumption of sequence identity we lose our algorithmic guarantee of finding all repeats: as with BLAST HSP searches [12], or general sequence alignment, badly placed mutations will defeat the seed and result in false negatives. But use of these seeds does allow for many errors, and PatternHunter has demonstrated that the appropriate seed combination can achieve very high sensitivity.

4 First Approach: SCANER

Our first elementary repeat identification algorithm is SCANER, which identifies elementary repeats via a novel chaining algorithm, and is accordingly names the ***S**equential **C**haining and **A**nalysis of **N**ew **E**lementary **R**epeats*. Our work developing SCANER served as inspiration for our more sophisticated RAIDER algorithm, and so we shall briefly describe the SCANER technique.

SCANER is comprised of two algorithms. The first algorithm, **Seed-Pair,** expands on the Huo method of using a tabulation of Lmers (see Section 2.3) to find pairs of maximal repeats in O(*n*) time. **Pair-Chain,** the second algorithm, takes the output of Seed-Pair and chains similar pairs into repeat families, splicing apart repeats that are non-elemental. Pair-Chain runs in O(*k log k*) time, where *k* is the number of elementary repeats.

4.1 Seed-Pair

Seed-Pair finds all maximal pairs of repeats longer than *L*. The algorithm scans the sequence from start to finish, pairing all successive instances of any reoccurring substring of length greater than *L*. We track necessary information through a hash-table recording

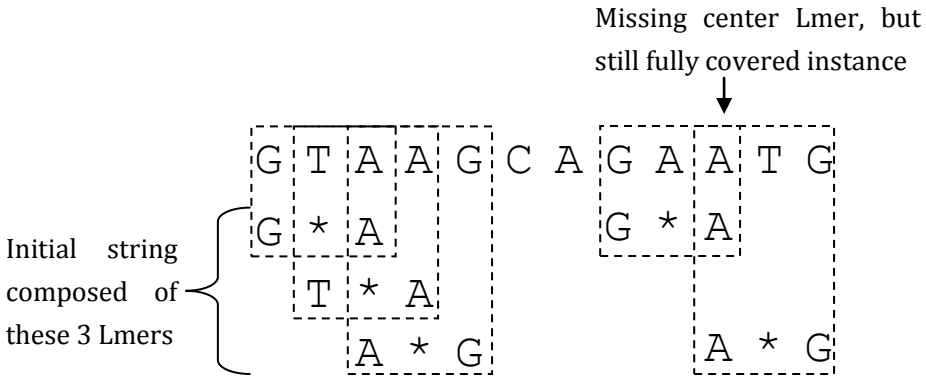


Figure 5: Given a mask of 101, we still consider these two strings a match since all bases in each are covered by hits occurring in both strings.

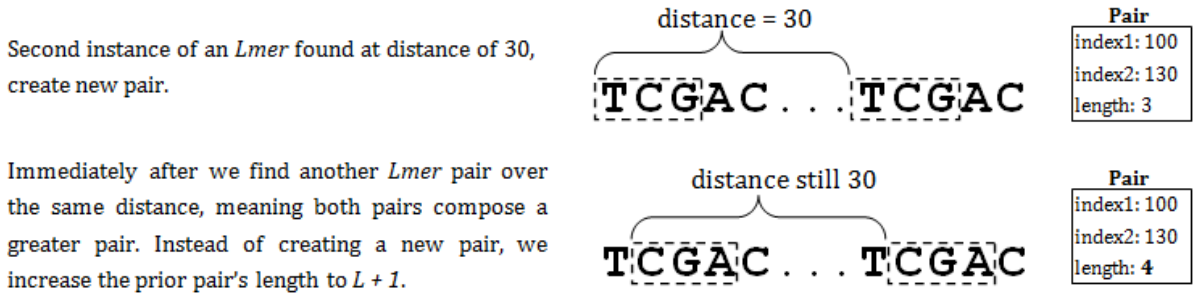


Figure 6: Seed-Pair maximizes consecutive pairs found over the same distance.

the last occurrence of each Lmer as we scan the query sequence. When an Lmer is encountered more than once, we look up its previous instance and create a new ordered pair. We then maximize both instances using another hash-table, as described below. Note that an instance of a repeat can, and is expected, to appear in two pairs: three instances of a repeat occurring in the order *r1*, *r2*, and *r3*, should produce the ordered pairs *(r1, r2)* and *(r2, r3).*

Algorithm 1 – Seed-Pair

```
Input: Text T of length n, minimum length L
Output: List of pairs of maximal repeats
1.      M ← Hash table    # maps Lmer (string) to position index (integer)
2.      D ← Hash table    #maps distance (integer) to a pair object {index1, index2, repeat length} stack
3.      for i ← 0 to n – L - 1 do
4.              p ← T[i:(i + L)]                # Get the Lmer at position i of T
5.              if M[p] is not null:            # Has the Lmer occurred before?
6.                      previous ← M[p]
7.                      distance ← i – previous
9.                      pair ← D[distance].top   # Get top of stack; the last pair found at this distance
8.                      if pair.index2 + pair.length is i + L – 1 then
9.                              pair.length ← pair.length + 1
10.                     else
11.                             push new pair (previous, i, L) onto D[distance]
12.             M[p] ← i
```

Take note of how pairs are maximized on lines 8 and 9: if a pair spanning the exact same distance occurred immediately before the current pair, we are dealing with a continuous pair, and so we increment the length of the prior pair to cover both. Figure 6 further illustrates this point.

The "seed" in "Seed-Pair" comes from the fact that instead of using Lmers, we can use the PatternHunter spaced-seed patterns discussed in Section 2.4. That is, given an Lmer, we apply a spaced-seed to allow for wildcard values. For example, given a spaced-seed pattern 10101, the Lmer TCGAC would become T*G*C, where * represents a wildcard.

The user specified seed allows for varying degrees of inexact matching when finding maximal pairs.

4.2 Pair-Chain

Taking the output from Seed-Pair, Pair-Chain finds pairs with overlapping members and merges them into chains. If three instances of a repeat appear in the order r_1, r_2, and r_3, resulting in the creation of ordered pairs (r_1, r_2) and (r_2, r_3) by Seed-Pair, Pair-Chain would merge these pairs at their intersecting member, resulting in the chain (r_1, r_2, r_3). Chains will merge together in this fashion to form longer chains, ultimately placing all similar repeats in the same family chains.

Recall that Seed-Pair simply found maximal pairs of repeats: it had no mechanism for knowing when to break a repeat into two or more elementary repeats. Instead, Pair-Chain provides an efficient method for doing this as chains are merged together.

Algorithm 2 – Pair-Chain

Input: List of pairs P, minimum repeat length L
Output: Set of linked-lists of elementary repeats

```
1.  C ← Set of linked lists made from P.  Each list has a repeat length, each node an index
2.  H ← Min-Heap of nodes. Insert all nodes from C, sorting by index
3.  while H is not empty do
4.      first ← H.pop
5.      second←H.peek
6.      if overlap(first, second) > L then
7.          mergeAndSpliceChains(first, second, H)
```

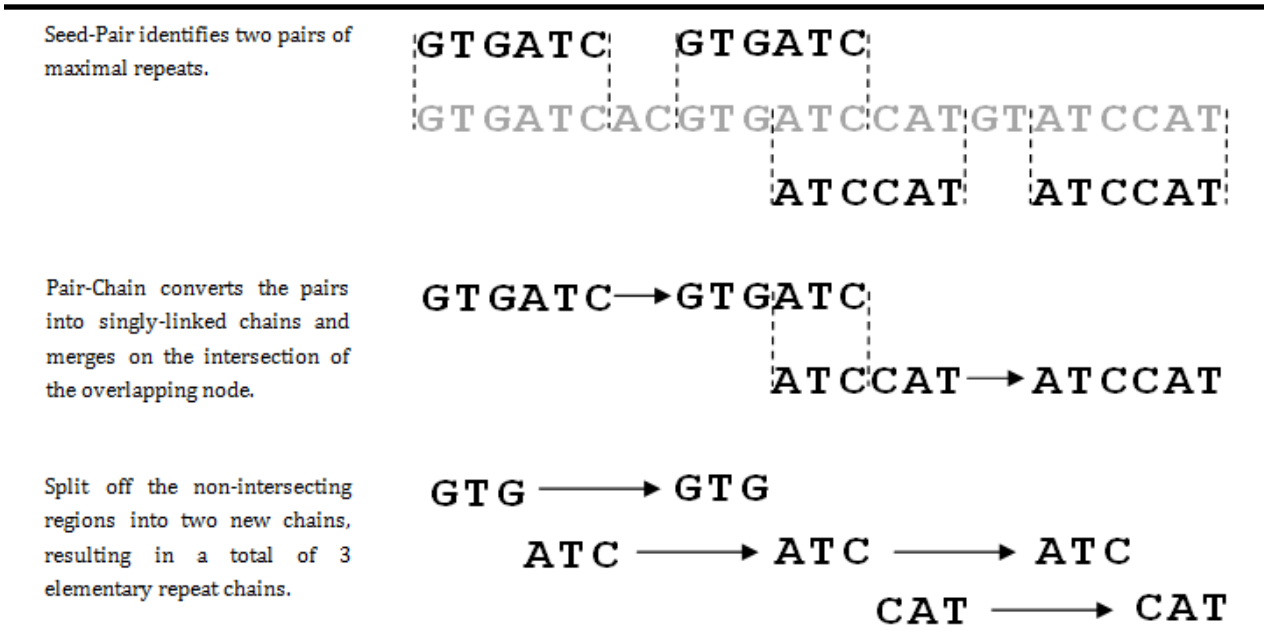


Figure 7: Pair-Chain splits repeats into elements by merging on their intersecting regions.

After grouping repeats into chains we sort all chain nodes by index and check successive nodes for overlap. mergeAndSpliceChains() represents the logic used for deciding where to splice chains together. Overlapping repeat chains are merged into longer chains, and chains' nodes are spliced into two chains when overlap is incomplete. For example, if there exist chains $c_1 = (r_1, r_2)$ and $c_2 = (r_3, r_4)$ such that a strict suffix of r_2 overlaps a strict prefix of r_3, we splice the chains on the intersection of their overlap and merge the intersecting regions together. We end up with three chains of resulting fragments: $c_1 = (r_{1,prefix}, r_{2,prefix})$, $c_2 = (r_{1,ntersection}, r_{2,intersection,} r_{3,intersection})$ and $c_3 = (r_{2,suffix}, r_{3,suffix})$ (Figure 7). Note that c_1 could have ended up covering the suffixes instead of the prefixes if instead the suffix of r_3 overlapped the prefix of r_2.

This is a simplification of the full algorithm, but illustrates the concept: after grouping repeats into family chains we combine their nodes into a min-heap, check for overlap between pairs, and merge overlapping repeat chains.

4.3 Discussion

We compared SCANER to RepeatScout [15], a leading de novo search tool [16], using the same benchmarking techniques for RAIDER described in Section 7. Those results can be obtained in appendix A. SCANER is competitive and very fast, but it served primarily as inspiration for the faster and simpler RAIDER. Despite its fast runtime, SCANER is essentially an O($n \log n$) search tool, as the instances of repeats have to be sorted (via the heap) to find overlap in the Pair-Chain algorithm. Since the number of repeats may be proportional to n, SCANER takes O($n \log n$) operations; though in practice, the ratio of repeats to n is so low that the $n \log n$ runtime remains remarkably fast.

5 RAIDER: A Refined Approach

5.1 Overview

In Section 4 we looked at SCANER, which found elementary repeats by finding maximal pairs, splitting on their intersections, and chaining like regions together. While efficient, it suffered two drawbacks: it had to maintain an object for every repeat instance, and it was a difficult algorithm to implement and describe. Here we improve upon SCANER and present the simpler and faster RAIDER, which requires only a single pass over the query sequence, and can reduce space usage by a constant factor if multiple passes are allowed (described in Section 6.4).

As with SCANER, we scan the sequence and look at the Lmers in the order they occur, and initialize maximal pairs as potential elementary repeats. We use a hash table to track the positions of Lmers, initialize a new series of Lmers whenever we find the second occurrence of an Lmer *x*, and add subsequent Lmers until we encounter an Lmer *y* such that *y* has not occurred exactly once previously. That is, if we find the second occurrence of Lmer *u* occurring as position *j*, with the first having occurred at *i* ($i < j$), we add it to a new table as the first of its Lmer series, and check to see if the Lmer at position *j+1* is the second occurrence of the Lmer that occurred at *i+1*; if so that Lmer is added to the series as well. We continue adding until we find a position *j+k* such that the Lmer is not the second occurrence of the Lmer starting at *j+k* and terminate the series. By Lemma 2 we know now that either this series is either an elementary repeat or contains one. Note that we track all pairs regardless of our threshold minimum *f*, which will be invoked at the end of the algorithm to remove identified elements with insufficient frequency. (In essence: we set *f=2* until the end of the algorithm, then invoke the filter.)

Continuing with the algorithm: let *F* be repeat element defined by a series of ||*F*|| Lmers (recalling that ||F|| indicates the number of Lmers in thee Lmer decomposition of *F*), which presently has its first two instances in *S* (counting from the left end). Let *F*[*i*] denote the i^{th} Lmer in the *F*'s Lmer decomposition. Note that *F* is elementary (for *f*=2) with respect to the portion of the sequence scanned so far, since no Lmer in *F* has yet been seen outside of *F*. As we continue scanning *S*, we may find another instance of *F*[0] at some position *k*, which marks the beginning of a third instance of *F*. We check to see that the Lmer at *k*+*i* is identical to *F*[*i*], for $i < ||F||$ (that is, verify that the sequence of Lmers making up the new sequence match those making up *F*). If we find that some $I < ||F||$ violating this condition, we need to split the series into *F* (the Lmer series consisting of Lmers 0 through *i-1*), and *G*, (the Lmer series consisting of Lmers *i* to then end of what was *F*). Thus we have effectively spliced the Lmers of *F* into two families, so that the new repeat, starting at the i^{th} Lmer of *F*, forms a complete repeat in its newly defined family.

A series *F* will be split if and only if an incomplete copy of the repeat is found in the sequence after *F* was defined through the identification of an initial repeat pair. Given that the algorithm scans from beginning to end, this manifests in two scenarios: discovery of an independent prefix, or discovery of an independent suffix. Discovery of a prefix takes place as described above (an illustrated in Figure 8), with the encounter of an independent repeat instance not containing the needed tail Lmers. A suffix of a repeat instance is discovered when an Lmer *F*[*i*], $i > 0$, is encountered apart from its position in an instance of *F*. The

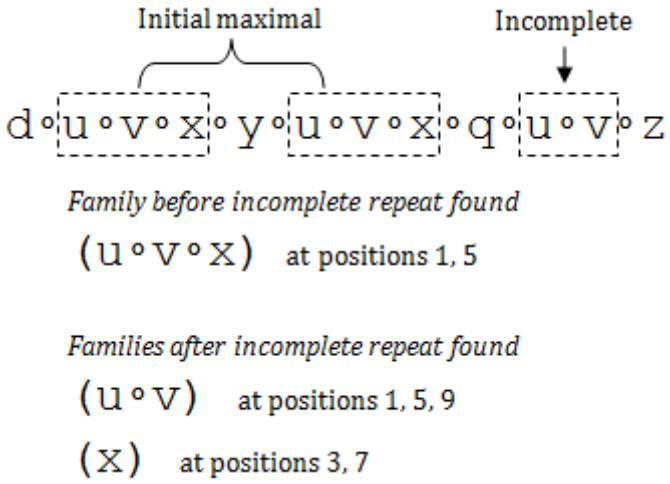


Figure 8: A family is spliced into two families when an incomplete instance of the original family is found.

series is immediately split, with Lmers $F[0]$ through $F[i\text{-}1]$ kept in the original series and the remaining Lmers assigned to a new series starting with $F[i]$.

Note that it is also possible to encounter an infix, but this will actually appear to the algorithm as a suffix followed by a prefix. For example, let F be a family with the Lmer decomposition $u \circ v \circ x$. If the Lmer v is encountered alone, apart from its position in F, the family's series will immediately be spliced, F now comprising only Lmer u, and a new family G with the decomposition $v \circ x$. (Noting that elementary repeats can overlap by L-1 or less bases without violation of the conditions). Next, x is expected to follow v, but since v is encountered alone, x is missing and v is now a prefix of its family G. Thus G is now spliced into two families, composed of Lmers v and x respectively.

5.2 Pseudocode

It must be emphasized that Lmer refers to a string, not to a specific location. If an Lmer p is assigned from family F to family G, all instances of p now belong to family G (Lemma 1). To say two Lmers belong to the same family is to say all instances of those two strings belong to the same family.

LmerSeries *– An ordered list of Lmers*
count(x) *– returns the number of instances of* Lmer *x found so far*
series(x) *– returns the LmerSeries to which* Lmer *x belongs,*
previous(x) *– returns last index of* Lmer *x prior to i*
SpliceSeries (LmerSeries *T*, Lmer *x*)
Input: LmerSeries *T*, Lmer *x* belonging to *T*

Output: Modified LmerSeries T, new LmerSeries T_{new}
1. T_{new} ← LmerSeries created from x and all subsequent Lmers in T
2. T ← truncate T to only contain Lmers preceding x
3. return T_{new}

RAIDER (sequence S, integer L, integer f):
Input: Genomic sequence S, minimum length L, minimum family size f
Output: Elementary repeat families with frequency f or greater as a set of Lmer series
1. *expectedSeries* ← NULL, *expectedRank* ← 0
2. for i ← 0 to $|S|$-L: # Scan the Lmers of S in sequence
3. x ← S[i:i+L] # x is the i^{th} Lmer of S
4. if *expectedSeries* ≠ NULL: # If we are in a repeat of a known family
5. *expected* ← *expectedSeries*[*expectedRank*] # Get expected Lmer
6. if x does not hit *expected*: # Repeat incomplete; series is non-elementary
7. SpliceSeries(*expectedSeries*, *expected*)
8. *expectedSeries* ← NULL
9. else: *expectedRank* ← *expectedRank* + 1
10. if count(x) = 2: # New family is being discovered
11. y ← S[i-1:i+L-1] # Preceding Lmer
12. if count(y) = 2 and previous(y) = previous(x)-1:
13. assign x to series(y)
14. else: create new LmerSeries, initialized with Lmer x
15. else if count(x) > 2: # x must already belong to a series
16. if series(x)[0]=x: # x is first Lmer in series(x)
17. *expectedSeries* ← series(x)
18. *expectedRank* ← 1
19. else if *expectedSeries* = NULL: # then series(x) is non-elementary
20. *expectedSeries* ←SpliceSeries(series(x), x)
21. *expectedRank* ← 1
22. if *expectedSeries* ≠ NULL and x is last Lmer in *expectedSeries*:
23. *expectedSeries* ← NULL
24. return all created Lmer series containing at least f elements

In our implementation we use a hashmap to map Lmers to Lmer objects which list Lmer occurrences. An LmerSeries is simply an ordered list of pointers to the Lmer objects of the Lmers it contains. When a family is spliced, the pointers are simply split between the two resulting LmerSeries. Thus, in a single operation, all instances of an Lmer are moved from one family to another by moving a pointer from one LmerSeries object to another.

5.3 Inexact Matching

What we have introduced so far still suffers from a problem common to repeat finding algorithms: the inability to cope with inexact repeats. Two homologous repeats with minor differences will be placed in separate categories by an exact string-matching

algorithm. Many repeats are inactive structures that have been heavily obfuscated over millennia and have not retained any function that would encourage conservation of the sequence pattern, yet their identification remains scientifically significant.

This is perhaps the most important reason why we have forsaken string indexing algorithms (e.g. Suffix Tree, FM Index), as they cannot account for inexact matches without multiplying storage or processing needs. By tailoring our algorithm to hunt for families of Lmers, we gain flexibility to define what constitutes an Lmer and what conditions are necessary to maintain a family.

As with SCANER, we utilize the PatternHunter method (Section 3.3). Instead of defining an Lmer to be an exact string of length L, we define a spaced seed pattern of length L.

One further algorithm enhancement is necessary to accommodate inexact matching: we have to allow for missing Lmers in repeat instances. A single base at position i in the sequence will belong to up to L Lmers (see Observation 2 and Figure 3 in Section 3.2), thus a family F with a defining sequence p may have an instance with a substituted base, call it p'. Even though p' may have only one substitution, this can cause up to L Lmers in p to no longer hit their counterparts in p'. A spaced seed will increase the number of hits since some Lmers will ignore the substitution when it is masked by a space, but there will still be some missed hits which we must allow for if we do not wish to splice the family.

To allow for missing Lmers, we relax the definition of elementary repeat from Section 3, such that constraint 2 now becomes: for an Lmer x_i in a family F, $Freq(x_i) \leq Freq(F)$. That is, Lmers in a family may occur less frequently than instances of the family itself, which means there are instances with missing Lmers.

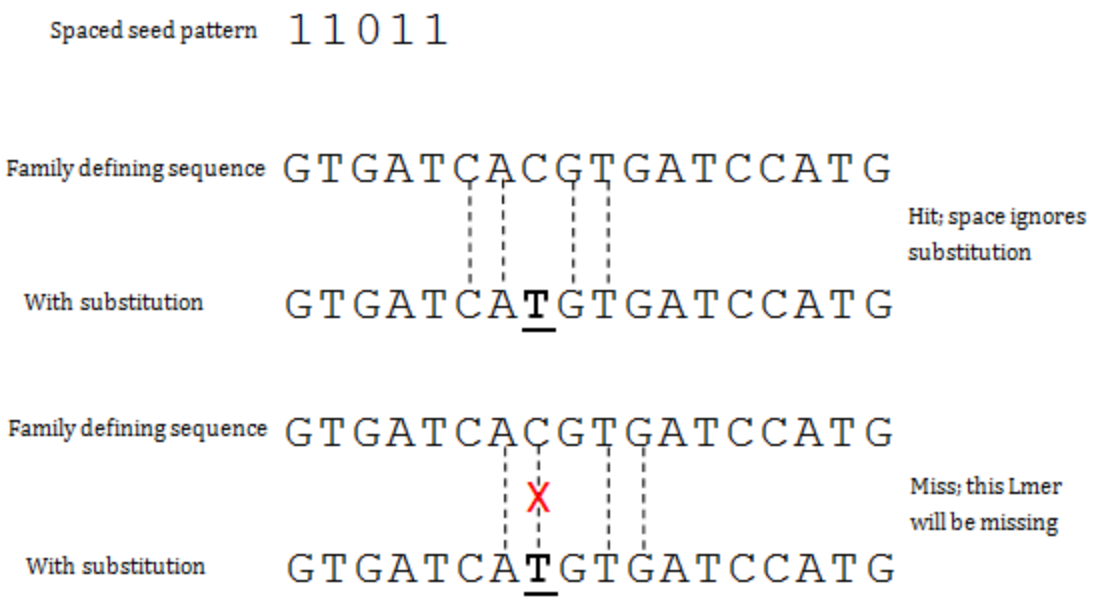


Figure 9: Spaced seeds allow a degree of error by treating spaced regions as wildcards.

Our modification leaves us with an important question: how many missing can be tolerated before the family is spliced? Let b be the length of F, i.e. the total number of bases in an instance of F, and let s be a string of length b and a potential instance of F. Our strategy is defined by Definition 6 in Section 3.3: a string s an instance of F so long as every base in s is hit by at least one Lmer in F. Remember that any given base is hit by up to L Lmers in F (Observation 2), thus we could be missing up to L-1 Lmers from an instance of F and still have every base hit by at least one Lmer. The very first character of x_0 and the very last character of x_k are the only characters hit by only one Lmer in F, thus a substitution of either must result in the removal of the respective Lmer, making F one base shorter.

6 RAIDER Complexity Analysis

The following analyses are conducted with respect to the optimized RAIDER of Section 5. Let n be the number of nucleotides in the given input sequence, L be the minimum length for a significant repeat, and f be the minimum frequency needed for a family to be significant.

6.1 Space

The space components for RAIDER are as follows,

1. One **Map** - a map data structure, maps Lmer strings to **Locations**.
2. A **Locations** array for each Lmer – records integer locations of a given Lmer.
3. An **Lmers** array for each family – records which Lmer belongs to which family

Map will require, at a minimum, space for every unique Lmer and a pointer to the associated **Locations**. The size of a key is $w \leq L$, where w is the weight of the seed being used (the number of ones contained in the PatternHunter pattern). In our implementation we only consider the 4 nucleotides, regardless of case, and do not map any Lmers containing ambiguity codes. Thus there are at most 4^L possible Lmers to be mapped, a theoretical upper-bound to space usage for our map if $4^L < n$. This is unlikely to serve as a useful bound, however, since 4^L is much greater than the size of the human genome when $L \geq 16$, and our best seeds so far have tended to be in the range $24 \leq L \leq 160$. Thus in the theoretical worst case, we may have $O(n)$ unique Lmers. In practice we will never encounter n unique Lmers in a genome, as genomic data is highly repetitive; nonetheless our map will take $O(n)$ space to map every possible Lmer to a list of occurrences.

Unlike **Map**, the **Locations** lists will require $\Theta(n)$ space, as each position in the genome must initially be recorded by a list. (Technically $\Theta(n\text{-}(L\text{-}1))$ space since the tail of the sequence will be too short to contain an Lmer, but we assume L will always be negligibly small; for example, $L = 2^{12}$ would be ineffectively huge and still negligible compared to n). No amount of redundancy can reduce the amount of space needed to record every position in the genome, unless we allow runtime to be increased by a constant factor: every Lmer points to at most one family, and not every Lmer has to belong to a family; only those with f or more occurrences need be retained. Using the space optimization described in Section 6.3 below, the total number of families created can be reduced to $\Theta(n/f)$.

Since each Lmer belongs to at most one family, the sum of the sizes of all **Lmers** arrays must be less than or equal to the total number of unique Lmers. Thus all **Lmers** arrays combined use $O(n)$ space.

In summary: $O(n)$ **Map** + $\Theta(n)$ **Locations** arrays + $O(n)$ **Lmers** arrays = $O(n)$ space usage.

6.2 Runtime

The algorithm's main loop (line 2) will run $O(n)$ iterations ($n\text{-}L$ iterations, to be exact). Re-assigning all occurrences of an Lmer from family A to family B is simply a matter of moving a pointer to a list of occurrences from family A to family B.

Splicing a family involves splitting the list of contained Lmers at some pivot Lmer sequence, moving the pivot and all subsequent Lmers to the new family. Thus if family F is being spliced, and family F contains an array of k Lmers, then it will take at most $k\text{-}1$ operations to splice that family. But we can reduce this to at most $k/2$ operations if we allow the algorithm to choose which Lmers are moved. That is, if the pivot Lmer is i, then

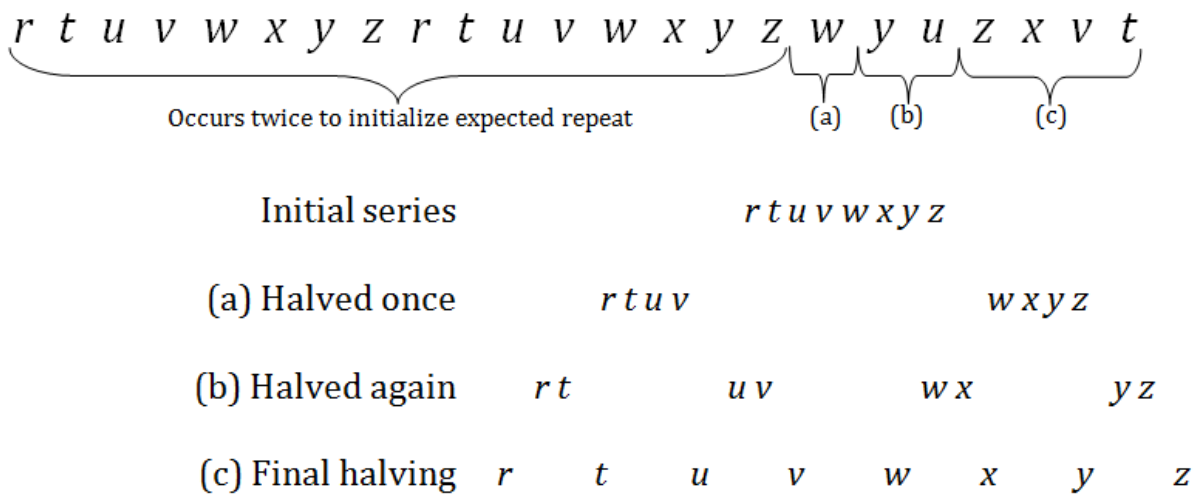


Figure 10: Worst-case Lmer sequence requiring the original repeat be spliced in half *logk* times.

we have two options: move all Lmers at *i* and greater to a new array, or move all Lmers at less than *i* to a new array: then assign the arrays to the appropriate families. We simply evaluate whether $k-i < i$, and move the corresponding number of Lmers.

Though RAIDER excels in terms of runtime (see Table 1), there is a worst-case scenario which prevents us from achieving linear runtime. Let T be an Lmer series where $||T||=k$, and T has been found twice – thus is a potential elementary repeat. A scenario could arise where we encounter the center Lmer of T, $T[k/2]$, which would trigger a splice of the T requiring $k/2$ operations to move half the Lmers of T into a new LmerSeries. This could be followed by the Lmers $T[k/4]$ and $T[k/4+k/2]$, each requiring $k/4$ operations to splice their respective halves of T into quarters of T, for another $k/2$ total operations. And so forth, splicing T until only single-Lmer families remain. T can be halved $log_2 k$ times, and each halving requires a total of $k/2$ operations to move Lmers to new LmerSeries. Thus $O(k \log k)$ operations are required, and since k can be a factor of n, RAIDER runs in $O(n \log n)$ time.

The final step of filtering out the families with fewer than f members (line 23) would require at most $O(n)$ operations should there somehow be $O(n)$ families. Thus RAIDER remains $O(n \log n)$ time.

6.3 Space optimization via Prescan

Any families with fewer than f instances will ultimately be filtered out (line 23), hence are not worth tracking. We can eliminate the space overhead by pre-scanning the genome before running RAIDER: we count the number of times each Lmer occurs, and then initialize our Lmer map to only consider Lmers with counts ≥ f. This prevents the unnecessary initialization of families that will end up having too few instances, prevents

MultiplyFamily() being called to account for Lmers with too few occurrences, and saves the associated space. Prescanning requires one pass over the genome, preserving the $\Theta(n)$ runtime, while reducing the amount of space used by families and Lmers to Θ(n/f).

However, counting Lmer occurrences during prescanning will still require a map holding each Lmer as a key, keeping the space requirement at $\Theta(n)$. But even if there has been no asymptotic space gain, our experiments have shown the gains to be significant, reducing the space required for human chromosome 1 from 24 GB to 10 GB while increasing runtime from 10 minutes to 18 minutes (Table 4).

6.4 Multi-Prescan

We can reduce space by a factor p if are willing to increase runtime by a factor of (p+1)/2. We simply map only $(1/p)^{th}$ of the sequence at a time, scanning the rest of the sequence for Lmers occurring in that $1/p$ region. After considering the first length n/p partition of the sequence, we need not scan it again (since we already scanned the rest of the genome for Lmers contained in that segment), and repeat the process with the next segment of size n/p. Thus we scan the entire sequence once, scan ((p-1)/p)n of the sequence the second time, then ((p-2)/p)n, and so forth. Thus the number of operations required is:

$$n \cdot \sum_{i=1}^{p} \frac{i}{p} = n \cdot \frac{p(p+1)}{2} \cdot \frac{1}{p} = n \cdot \frac{p+1}{2}$$

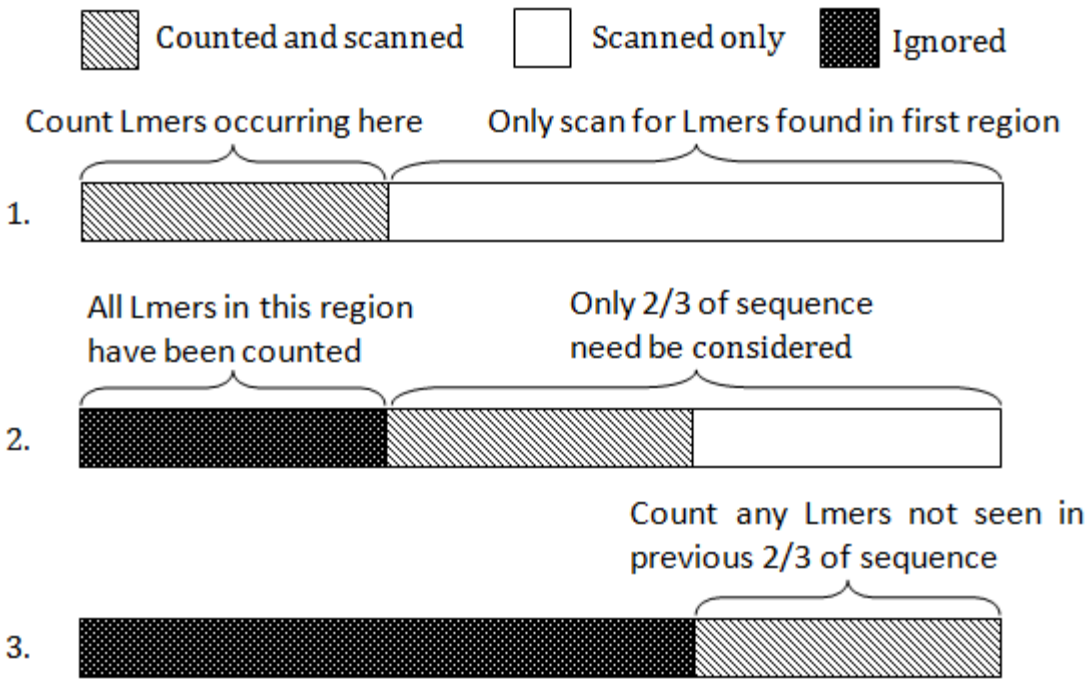


Figure 12: By only scanning for Lmers found in one region at a time, we eliminate the need to simultaneously store all Lmers in the hash table.

As an example, let $p = 3$. We begin by scanning the first hird of the sequence, counting all occurrences of the Lmers found there. We proceed to scan the last o thirds of the sequence, but do not count any Lmers we did not find in the first third of the genome. Thus we only need $O(n/3)$ space to map the Lmers found in the first third of the sequence, and simply increment the counters associated with those Lmers as we scan the remainder of the sequence. This requires $O(n)$ operations. We mark any Lmers which occur f or more times and delete the rest. We repeat the process, this time only counting Lmers occurring in the middle third of the sequence, again only needing $O(n/3)$ space, and this time not needing to consider the first third of the sequence at all, having already counted all Lmers occurring there. This requires at most 2n/3 operations. This process is repeated for the last thrid of the sequence, requiring only $n/3$ operations, for a total of $2n$ operations.

7 Results

Here we assess the quality of RAIDER results as compared to the RepeatScout tool [17], using genome-derived RepeatMasker-annotated benchmarks for comprehensive testing and synthetic data to test special cases. We initially attempted to include the tool RECON [14] in our benchmarks, but its runtime became prohibitive even when applied to small chromosome sequences (e.g. human chromosome 22 or mouse chromosome 19).

Sequence	Base-Pairs	Families Preserved
C. Elegans chrX	17,718,866	HAT1_CE PALTTTAAA2 CELE14B LINE2C_CE
C. Elegans chrI	15,072,423	SINE1_CE CELE45 LINE2B_CE LINE2G_CE
Human chr22	49,691,432	AluSx MIRb L1MB5 L1MC1
Mouse chr19	61,431,566	B1F1 PB1 L1_Mm L1_Mus3

Table 1: List of sequences analyzed and respective families preserved for identification benchmarking.

7.1 Testing: Tool Parameters and Benchmark Sets

Testing was primarily on modified genomic sequences using human chromosome 22, mouse chromosome 19, and *c. elegans* chromosomes I and X, rating success by the tool's ability to find significant portions of known ancestral repeats. However, in order to prevent unannotated repeats from confounding results by manifesting as false positives, we selected a set of target families (listed in Table 1) from each genome and "removed" all other repeat elements by randomly block-shuffling the intervening genomic sequence, thus preserving repeat characteristics and genomic base distribution while creating a sequence easier to analyze. Purely artificial genomic sequence were also created to test boundary cases, generating background sequences using a uniform base distribution and creating artificial repeat sequences of length 50 to 1000 base pairs (bp). These sequences served well for sanity testing during development, but as a result make poor benchmark data since these sequences purposely contain scenarios RAIDER is designed to handle; therefore we focus on performance with respect to biological data.

All tests were conducted using the default parameters for both the RAIDER and RepeatScout tools [17]. RAIDER default parameters were chosen to optimize overall performance, but were not tailored to individual benchmarks. Understanding how the spaced seed affects performance is also a goal of these benchmarks, so we test RAIDER using four different seeds with progressively increased spacing, as listed in Table 2.

7.2 Metrics

Given a query sequence S containing the set of families $R = \{r_1, r_2 .. r_{|R|}\}$, let $P = \{p_1, p_2 ..$

Name	Seed
S1	111111111111111111111111
S2	111110111111100111111011111
S3	000001100011100001111000001111100000011111100000001111111
S4	00001111...0^8...11111...0^{17}...111111...0^{32}...1111111...0^{64}...11111111

Table 2: Spaced seeds used for RAIDER benchmark.

$p_{|P|}\}$ be a set of predicted repeat families (where p_i is the set of repeat elements embedded in the genome corresponding to the i^{th} predicted family). Let $||p||$ denote the sum of all base-pairs covered by a family p, and $||P||$ be the total number of base-pairs covered by all families in P.

7.3 Runtime

We use an emperical measure of runtime based on the Linux time utility to precisely measure real (user) time elapsed to complete execution of the process being measured. All tests were run on Redhat Linux using a dual quad-core 2.4 GHz Intel Xeon E5620 CPU and 24 GB RAM.

7.4 False Negatives

Here we borrow from Bao and Eddy's assessment of their de novo tool RECON [14,15], focusing on errors at the base-level. For every family of repeats $r_i \in R$ we find the best matching family of repeats $p_j \in P$, and normalize the difference in coverage by the repeat length:

$$(||r_i|| - ||p_j||) / ||r_i||$$

This error is averaged over all families assessed (notated as ErrMissed).

7.5 False Positives

Again borrowing from Bao and Eddy [15], we test for false positives by noting which families p_{best} in P are considered best matches for families in R. That is, each $r_i \in R$ is best covered by at most one $p_{best} \in P$. For every p_{best}, we subtract $||p_{best}||$ from the total count of base pairs covered by P. We divide this by the total number of base pairs to get a percentage of false positives. This is averaged for all families assessed and reported as ErrFalse.

$$(||P|| - \Sigma||p_{best}||) / ||P||$$

7.6 Redundancy

Another metric used for RECON, this is meant to assess how many predicted repeats cover the same segment in S. Let v be the set of all regions overlapped by both P and R, without double-counting the same base-pairs. Let w be the same, only re-counting each

base-pair for every repeat in P that covers it. Redundancy is then $(||w|| - ||v||) / ||w||$. That is, the fraction of base-pairs shared between R and P that P covers multiple times. The definition of elementary repeat anticipates a given base being covered by up to L Lmers and therefore could theoretically be covered by up to L elementary repeats. Nonetheless, redundancy remains an important metric that may testify to the usefulness of the elementary repeat definition. Redundancy is averaged for all families assessed and reported as ErrRedun.

7.7 Comparison Summary

As can be seen in Table 3, RAIDER significantly outperforms RepeatScout in runtime, anywhere from 13x to 57x faster, depending on dataset and seed choice; 32x faster on average.

ErrMissed tends to be high, but this is likely a result of bias from the use of RepeatMasker-derived benchmarks: RepeatScout uses RepeatMasker to find all instances based on a given seed, whereas RAIDER simply returns what instances it was able to find without the aid of RepeatMasker. RepeatMasker marks recognizable fragments of a family's ancestor, and most of these fragments will not repeat with enough instances or length to form a RAIDER recognizable pattern. Note also that ErrMissed tends to decrease as spacing is added to the search seed, indicating sensitivity could be increased with a better search seed – requiring a more formal study of optimal seed structure.

	C. Elegans chrI				Human chr22			
Method	***Time (s)***	***Anc%***	***ErrFalse***	***ErrMissed***	***Time (s)***	***Anc%***	***ErrFalse***	***ErrMissed***
S1	23.08	0.622	0.010	0.909	53.88	0.326	0.004	0.990
S2	25.50	0.811	0.005	0.911	61.81	0.315	0.003	0.990
S3	38.51	0.582	0.005	0.854	93.94	0.689	0.008	0.986
S4	81.65	0.794	0.003	0.697	190.35	0.84	0.003	0.987
RepeatScout	1329.18	0.944	0.010	0.758	2343.53	0.777	0.258	0.759
	Mouse chr19				**C. Elegans chrX**			
Method	***Time (s)***	***Anc%***	***ErrFalse***	***ErrMissed***	***Time (s)***	***Anc%***	***ErrFalse***	***ErrMissed***
S1	91.24	0.131	0.103	0.993	29.19	0.485	0.008	0.965
S2	102.13	0.218	0.088	0.992	35.09	0.393	0.005	0.963
S3	154.94	0.21	0.093	0.989	45.93	0.587	0.007	0.959
S4	281.37	0.307	0.113	0.985	87.35	0.558	0.002	0.958
RepeatScout	3877.21	0.539	0.442	0.846	1183.70	0.604	0.337	0.674

Table 3: Real data test results. Notice that RAIDER is an order of magnitude faster while still covering nearly as much of the ancestor as does RepeatScout. False positive error is also generally lower. ErrMissed is high for RAIDER, but this is to be expected given that RAIDER only returns locations of whole elementary repeat instances, whereas RepeatScout uses RepeatMasker, which will identify all fragments of a given seed. Note also that this error generally decreases as spacing is added to the search seed.

Sequence	BPs	Method	Memory (GB)	Time (h:mm:ss)
Human genome	$2.9x10^9$	RAIDER (prescan)	96.09	4:26:34
Human chr1	$2.5x10^8$	RAIDER	24.19	0:10:03
Human chr1	$2.5x10^8$	RAIDER (prescan)	10.21	0:17:48
Human chr1	$2.5x10^8$	RepeatScout	4.24	1:30:40

Table 4: Memory footprint and runtime for large sequences

False positive error is also generally lower (with the exception of the mouse data), indicating RAIDER has a tendency towards precision, which is again unsurprising given that RAIDER does not perform alignment as is done in RepeatScout via RepeatMasker.

Ancestor coverage tends to favor RepeatScout, though RAIDER remains competitive, especially with the longer, more spaced seeds. In some cases RAIDER approaches 80% coverage, which is promising performance when there remains so much room for parameter tuning and seed selection.

7.8 Chromosome 1 and Whole-Genome Tests

For the sake of understanding memory and time requirements, we ran RAIDER against two very large sequences: human chromosome 1 and human chromosomes 1 through 22 combined (entire human genome except for chromosomes y, x and m). We were unable to run RepeatScout against the human genome as the software crashed immediately, most likely due to internal size constraints. We tested RAIDER with and without the single-prescan for reducing memory usage. We were only able to perform a whole genome run using the prescan method due to memory constraints. See results in Table 4.

8 Linear-time RAIDER

8.1 Overview

Towards the end of our research we realized RAIDER could be made even more efficient by exploiting the order in which Lmers appear, and in doing so achieve a linear time algorithm with respect to query sequence length. The algorithm is divided into two steps, each requiring one pass over the entire sequence. In the first pass, all Lmers are placed in a single array, ordered such that Lmers belonging to the same elementary repeat family will appear in consecutive positions and in the order dictated by the family's Lmer series. Once this array is created, the second task is to mark the boundaries between Lmer series on the array.

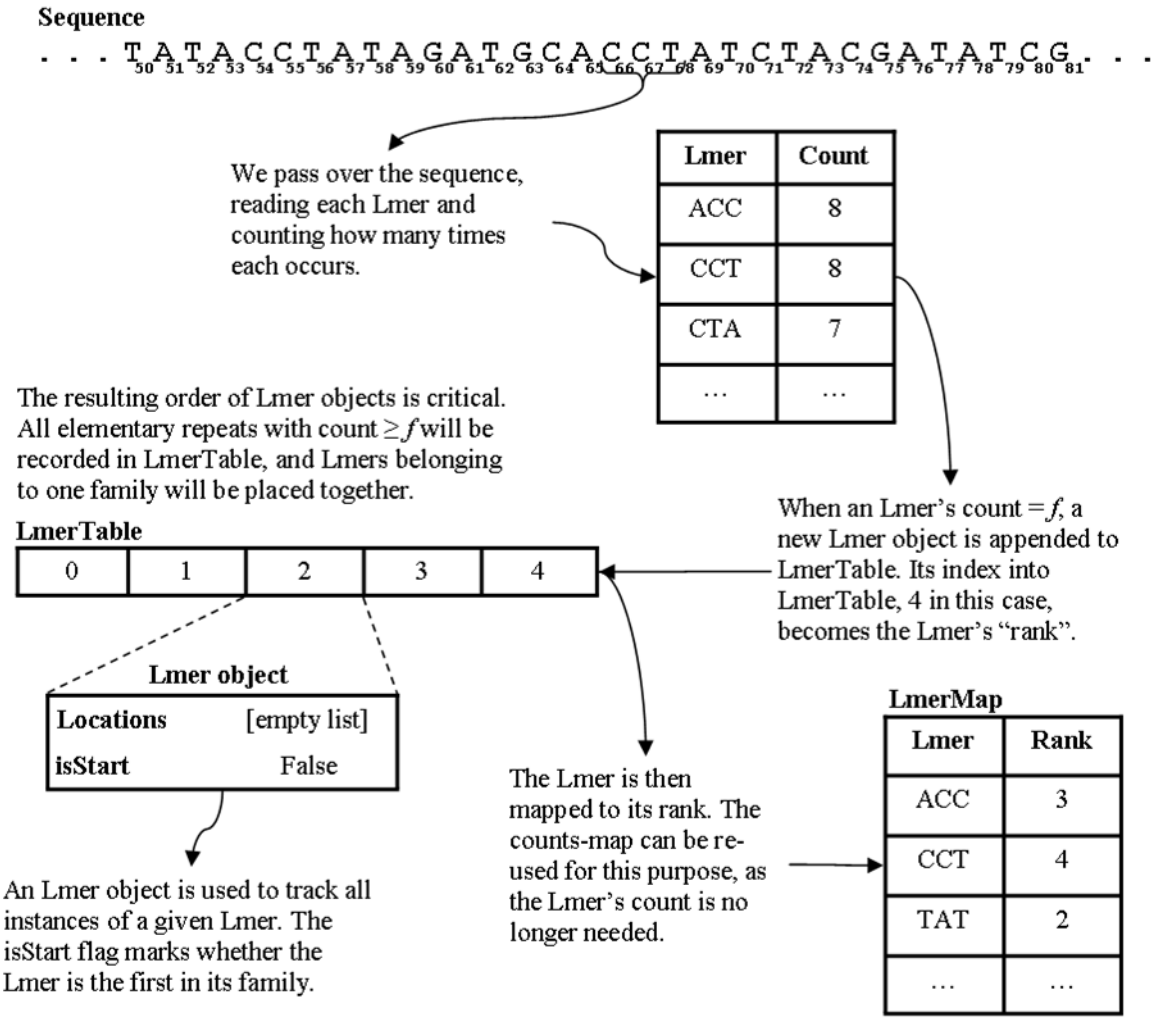


Figure 13: Step one of RAIDER: counting Lmers to create an ordered array of all significant Lmers. Assume *L*=3 and *f*=8.

8.2 Step One: Count Lmers

The first step is to create the array of Lmers, which we call **LmerTable**. To populate LmerTable we must identify all significant Lmers, i.e. Lmers occurring *f* or more times. Such Lmers are guaranteed by Observation 1 to appear as members of the Lmer decomposition of some elementary repeat. The method for identifying these is simple: simply scan over the sequence from start to finish, recording the number of occurrences of each Lmer in a hash table. When we see the f^{th} occurrence of an Lmer, append that Lmer to LmerTable. Appending Lmers in this fashion guarantees they are ordered as they appear in their respective series (a claim proved in the next section).

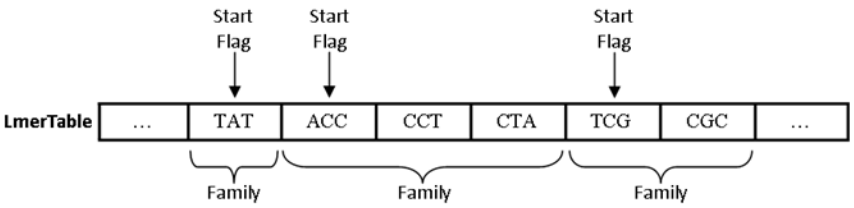


Figure 15: When both steps of RAIDER are complete, LmerTable will contain all elementary families, demarcated by start flags.

We will want to look up Lmers' positions in LmerTable, so we set up **LmerMap** be a mapping of Lmers to their positions in LmerTable, populating it as we append Lmers to LmerTable. With the first step finished, LmerTable and LmerMap will be fully populated. Let the position of an Lmer in LmerTable be known as its **rank**. Thus LmerMap can be described as mapping each Lmer to its corresponding rank.

8.3 Step Two: Mark Families

The second step is to mark boundaries in LmerTable to define which Lmers belong

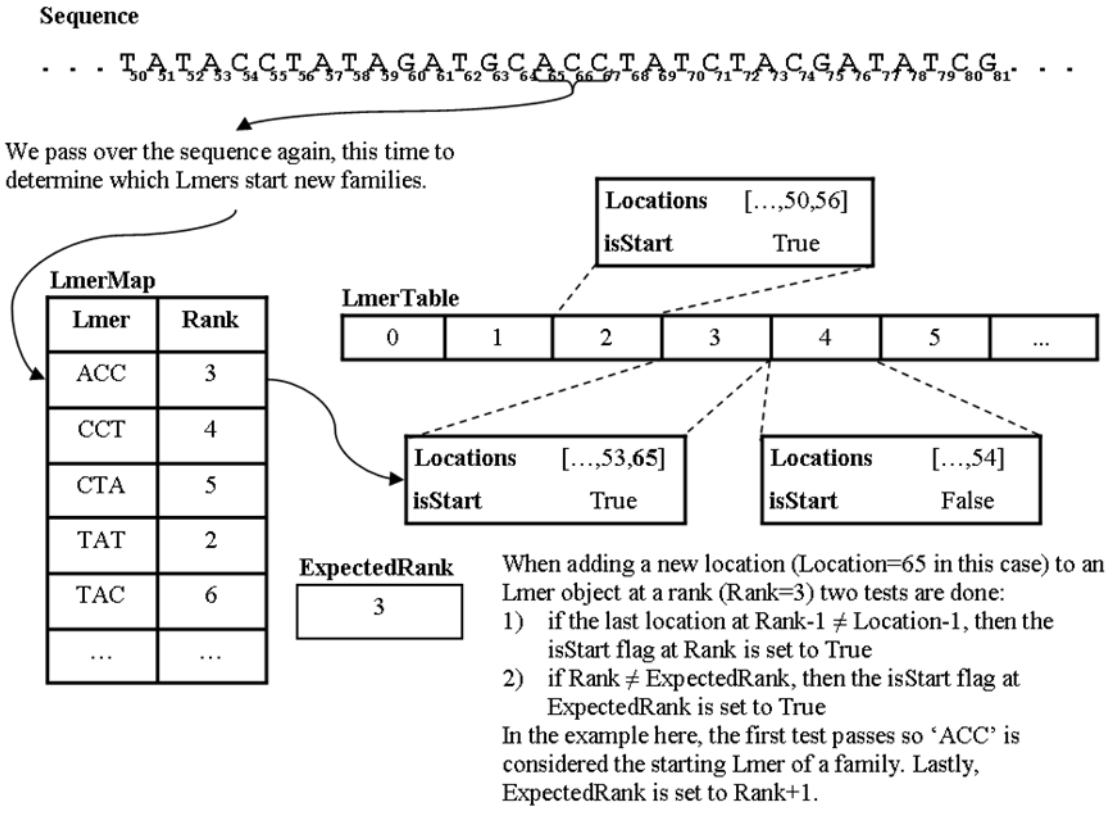


Figure 14: Step two of RAIDER: Marking the boundaries of each family's Lmer series in LmerTable. Assume *L*=3 and *f*=8.

to the same series decomposition. If all the Lmers in LmerTable belonged to the same repeat decomposition, then the instances of these Lmers would all appear as one continuous group in the query sequence. We would find every instance of an Lmer with rank *Rank* occurring immediately after an instance of the Lmer at *Rank*-1 and immediately before an instance of the Lmer at *Rank*+1. If an instance were found where this was not the case, that Lmer would be considered out of rank. Our goal is to find every place in the sequence where Lmers appear out of rank, as these mark the boundaries of family series.

8.4 Pseudocode

Lmer object *{ Locations : vector of integers, isStart : Boolean flag }*
LmerTable *– vector of Lmer objects*
LmerMap *– Map of Lmer keys to integer rank (index of Lmer in LmerTable)*
CountLmers (Sequence *S*, integer *L*, integer *f*):
Input: Sequence *S*, minimum length *L*, minimum family size *f*
1. counts ← Map of Lmer keys to integer values
2. for *i*=0 to (|*S*| - *L*):
3. *x* ← *S*[*i*:*i*+*L*]
4. counts[*x*] += 1
5. if counts[*x*] = *f*:
6. LmerMap[*x*] ← |LmerTable|
7. LmerTable.append(Lmer object)

markStart(rank) *– Sets the isStart flag to true for the Lmer at rank*
getLastLocation(rank) *– Returns last value appended to LmerTable[rank].Locations*
MarkFamilies (sequence *S*, integer *L*, integer *f*):
Input: Sequence *S*, minimum length *L*, minimum family size *f*
1. *expectedRank* ← NULL
2. markStart(0)
3. for *i*=1 to (|*S*| - *L*):
4. *x* ← *S*[*i*:*i*+*L*]
5. if *x* in LmerMap:
6. *rank* ← LmerMap[*x*]
7. if getLastLocation(*rank*-1) ≠ *i*-1:
8. markStart(*rank*)
9. if *rank* ≠ *expectedRank*:
10. markStart(*expectedRank*)
11. LmerTable[*rank*].*Locations* ← append *i*
12. *expectedRank* ← *rank* + 1
13. else if *expectedRank*:
14. markStart(*expectedRank*)
15. *expectedRank* ← NULL

When CountLmers() followed by MarkFamilies() have completed, LmerTable will contain a set of Lmer series demarcated by start flags (seeFigure 15). The locations the instances of a series F can then be obtained by taking all locations of the first Lmer in F, which will be the Lmer with the start flag set to true. It is evident from a simple examination of the pseudocode that RAIDER is linear with respect to input size, finishing in $O(|S|)$ operations: two passes over S are necessary, but no inner-loop, recursion, or $\omega(1)$ function call is required.

8.5 Proof of Correctness

Lemma 3: When CountLmers(S, L, f) completes: every Lmer in LmerTable belongs to the decomposition of an elementary repeat, and every elementary repeat F in S will be contained in LmerTable as an ordered Lmer series.

Proof: Every Lmer in LmerTable has a frequency of at least f, hence by Observation 1 is either an elementary repeat or a member of an elementary repeat. Let F be an elementary repeat with decomposition $x_0 \circ x_1 \circ \ldots \circ x_{k-1}$, each of which only appear in instances of F (Lemma 1). Recall from Definition 5 that all Lmers in the decomposition must have the same frequency. Since instances of F always begin with x_0, the j^{th} instance of x_i must appear immediately prior to the j^{th} instance of x_{i+1}. Since an Lmer is added to the table when it is found to occur f times, when the f^{th} instance of x_0 is encountered, the next k-1 Lmers to be encountered will be the next k-1 elements to be added to LmerTable. □

Lemma 4: When MarkFamilies(S, L, f) completes: An Lmer in S is the first Lmer of an elementary repeat if and only if it appears in LmerTable marked with the isStart flag set to true.

Proof: Since MarkFamilies() is only run after CountLmers(), we know that an Lmer is in LmerTable if and only if it belongs to an elementary family. Our tasks are to ensure that 1) the first Lmer of every elementary repeat is marked by the isStart flag and 2) no other Lmer is marked by the isStart flag.

As MarkFamilies() progresses, every instance of every Lmer will be evaluated. The goal is to look at every Lmer in LmerTable and determine whether it is the start of a new family. Let p and q be two Lmers in LmerTable with ranks r and r+1 respectively; we must determine whether they belong to the same elementary repeat. If they do not, then q must be the start of a new elementary repeat since it is not a continuation of p's family. q belongs to p's family if and only if every instance of p is immediately followed by an instance of q, and every instance of q is immediately preceded by an instance of p. Thus proving that q is

the start of a new family is simply a matter of finding an instance of p that is not followed by an instance of q, or finding an instance of q that is not preceded by an instance of p.

Since p and q are of consecutive ranks, we expect them to be in the same family until proven otherwise. So when we encounter an instance of p at position i of S we expect the Lmer at position i+1 to be an instance of q. We test this by setting *expectedRank* to p+1 (line 12), and if the next Lmer is not q (line 9) or is not in LmerTable at all (line 13) we know p was not followed by q, and thus q must be the start of a different family.

Likewise, if we encounter an instance of q at position i of S we want to verify that p is the Lmer at position i-1. So we look up the last location of q and ensure it equals i-1 (line 7). If not, q must be the start of a new family.

It follows that all Lmers which begin a new family will be marked accordingly: if the Lmer at position i does not belong to the same elementary repeat series as the family of the next Lmer, then the next position will be marked as the start of a new family. The Lmer at rank 0 is necessarily the start of a new family, thus all subsequent Lmers in LmerTable are either a continuation of the previous family or the start of a new one. □

8.6 Inexact Matching

We now have two mechanisms for inexact matching: the first by replacing Lmers with PatternHunter style spaced seeds (Section 2.4), as was done with SCANER and RAIDER in Sections 4 and 5.

The second mechanism is unique to this version of RAIDER. We simply replace the start flags with start counters: every time an out-of-rank Lmer is discovered, its start counter is incremented. We then determine boundaries as a ratio of start-counter value to Lmer frequency: if an Lmer's start-count is low compared to the Lmer's frequency, then we forgive the few times it appeared out of rank and do not mark a boundary. The user specifies what percentage of an Lmer's instances may appear out of rank before that Lmer is marked as a family start.

8.7 Performance

At the time of writing this algorithm only had a Python implementation, which did not offer a fair comparison to the C++ implementations of SCANER and RAIDER algorithms. Thus we do not yet know what practical benefit the linear runtime might achieve, or what performance-increase may be gained from the new mechanism for inexact matching.

9 Conclusion

RAIDER holds much promise in terms of both speed and sensitivity, and warrants further development. In its experimental state, our assessments show RAIDER is already able to outperform popular tool RepeatScout in a number of scenarios.

In terms of speed, RAIDER performed extremely well given the size of input, processing human chromosome 22 in under a minute and the human genome in four and a half hours. RAIDER also performed considerably faster than RepeatScout, often by orders of magnitude. The discovery of a linear-time version of RAIDER has created potential for even faster whole-genome processing times, and is not far from implementation and testing.

Performance in terms of sensitivity was competent, but there are many avenues that may lead to further improvements. We suspect our sensitivity could be greatly improved by following RepeatScout's method for masking elements: currently RAIDER simply outputs a list of locations where elementary repeat instances were found, whereas RepeatScout simply outputs consensus seeds and then uses RepeatMasker to mask locations. Having RAIDER analyze all instances of a repeat family and output consensus seeds in the same manner as RepeatScout should yield increased sensitivity.

A rigorous investigation of optimal spaced seed patterns could yield dramatic improvements. Furthermore, since our method only finds elementary repeats, it would be useful to create an additional step which uses the building-blocks of elementary repeats to form and search for larger, composite repeats (we discuss initial attempts in appendix A).

The development of metrics not requiring the of repeat databases potentially assembled by the tools we are competing with would be of value. Metrics that can recognize and award the correct grouping of biological components are also needed, as the potential advantage of elementary repeats is in their ability to separate hitch-hiking data from TEs.

With further refinement, RAIDER has the potential to be the fastest and the most accurate de novo search tool available, as well as the only tool that can perform whole-genome searching in a reasonable amount of time.

10 Appendices

10.1 Appendix A: Composite Repeats

Elementary repeats alone tend to be small for use by a tool such as RepeatMasker (see Section 2.3). Thus we also implemented an algorithm for finding composite repeats using

the output of SCANER. However, the definition of composite repeat allows for much redundancy, and further work was needed to refine our ancestral seed database.

Definition 7 – Composite repeat

A composite repeat C is a sequence of two or more elementary repeats which occurs with frequency c in a given text T, meeting the following conditions:

1) $c \geq F_c$, where F_c is the specified minimum frequency for a composite.
2) C is maximal in terms of elementary repeat components.

Note that this does not contain a condition analogue to (3) of the elementary repeat definition. This allows the possibility of composites comprised of smaller composites, so long as the smaller composites remain maximal. For example, let A, B and C represent three distinct elementary repeats. Suppose we find the following sequence in our text, with * representing unidentified sequence data: ABC*AB*ABC. BC is not a composite, because it

	C. Elegans ChrI			Human Chr22		
Method	***Time (s)***	***Ancestor Coverage***	***False Positives***	***Time (s)***	***Ancestor Coverage***	***False Positives***
SCANER	38.35	0.847	0.209	159.20	0.763	0.191
RepeatScout	1238.86	0.944	0.010	2244.93	0.777	0.258
	Mouse Chr19			**C. Elegans ChrX**		
Method	***Time (s)***	***Ancestor Coverage***	***False Positives***	***Time (s)***	***Ancestor Coverage***	***False Positives***
SCANER	174.81	0.337	0.822	50.16	0.503	0.055
RepeatScout	4208.35	0.539	0.442	1240.68	0.604	0.337

Table 5: Results of SCANER + Composite-finding was still fast, but not as fast as RAIDER alone and did not perform any better.

would not be maximal as it only occurs as a part of ABC. However, AB is a composite because its occurrence outside of ABC allows it to be considered maximal.

We created another step for SCANER which takes the output of Pair-Chain and finds composite repeats in the resulting chains, with a runtime bound by $O(m|C|^2)$, where m is the number of composites and $|C|$ is the length of the largest composite in terms of elements. It worked by simply looking all elementary repeats, and any elementary repeats that were adjacent or overlapping were considered continuous and a potential composite. If the same sequence of continuous elementary repeats was found f_C or more times, it was recorded as a composite.

The definition of composite allows for high redundancy, and finding composites in this manner also meant adding a third, rather inefficient step to SCANER. It still completes in

reasonable time (Table 5) but remains slower than RAIDER with similar performance. Thus for the duration of this paper we focused on the development of RAIDER.

Research is needed to find an optimal formula for assembling elementary repeats into useful composite repeat seeds. A formula that exploits elementary repeats ability to eliminate hitch-hiking data, and simultaneously avoids the issues with redundancy we encountered, would likely boost RAIDER's sensitivity considerably.

10.2 Appendix C: The Pigeon-holed Repeats Problem

Let L=5 be the minimum length needed for a sequence instance to be considered significant. Let's also assume f=2 is the minimum frequency required to qualify as an elementary repeat. So if ATATA occurs twice, it is an elementary repeat.

Now because of the pigeon-hole principle, an elementary repeat can only occur so many times before it must produce a second family that is actually a false positive. For example: let F_i be the i^{th} instance of the family F. Let p_i be the base immediately before F_i, and s_i be base immediately after F_i. For example, if

$p_1 + F_1 + s_1$=CATATAT and

$p_2 + F_2 + s_2$=GATATAG

then p_1=C, s_1=T, p_2=s_2=G.

Now since every p_i must come from the set {A,C,G,T}, we cannot have more than four instances of F without repeating some p_i.

AATATA
CATATA
GATATA
TATATA
*x*ATATA

If x is A, then we have a new elementary repeat: AATAT, since it now occurs twice. If it is C, we have the new elementary repeat CATAT, and so forth. Additionally, we have all the bases s_i which occur immediately after ATATA, so if $||F||$=5, we now must have three elementary repeat families.

It gets worse. If $||F|| > 16$, we now must have at least *five* families, because the two bases immediately prior and immediately after each instance are limited to 16 possible 2-base strings. By now the pattern has emerged: for every power of 4 in $|F|$, we add two more families. Thus if we have one true family F with $||F||$ instances, we end up with $2log_4(||F||)$

false positive families. So for families such as the ALUs with thousands of instances in the human genome, this becomes problematic.

Also note that the number of instances in each of those false-families grows at a linear pace: at least one in four instances of F will have the same p_i, which means that if $||F||=100$, we have a false-family with at least 25 instances. Increasing f may mitigate the issue, but could also eliminate potentially important data that just did not occur many times. This is exacerbated some by the fact that bases don't occur with even distribution, thus increasing the probability of certain bases will generate false positives.

Perhaps the best solution to this is to filter out families that overlap significantly but do not occur independently of F, or do not occur at least 50% of the time where F occurs (or some similar significant fraction). Exploring the math further should reveal at what point a repeat is statistically likely to be just noise. Until that avenue is thoroughly explored, experimentally increasing the minimum frequency f to a power of 4 should be an effective strategy. E.g. if f=16, then an even base-distribution will not produce any pigeon-holed false-positives for any family $||F|| \leq 64$.

10.3 Appendix D: Simple RAIDER Python

The linear version of RAIDER presented in Section 8 is attached here. A C++ implementation of inexact-matching RAIDER described in Section 5 is available for download at http://handouts.cec.miamioh.edu/karroje/RAIDER/

```
import os
import sys
import argparse

# Lists Lmers in order of first occurrence
LmerTable = []

# Maps strings to Lmer objects
LmerMap = {}

class Lmer():
    def __init__(self):
        self.locations = []
        self.isNewFamily = False

def isNewFamily(rank):
    return LmerTable[rank].isNewFamily

def setNewFamily(rank):
    LmerTable[rank].isNewFamily = True

def getLmer(S, i, L):
    S.seek(i)
    return S.read(L).upper()
```

```
def countLmers(S, S_len, C, L):
    counts = {}
    for i in range(S_len - L):
        text = getLmer(S, i, L)
        count = 1
        if text in counts:
            count = count + counts[text]
        counts[text] = count
        if count == C:
            LmerMap[text] = len(LmerTable)
            LmerTable.append(Lmer())

        if i % 1000 == 0:
            showProgress(i, S_len)

def markFamilies(S, S_len, C, L):
    expectedRank = 0
    setNewFamily(0)
    for i in range(S_len - L):
        text = getLmer(S, i, L)
        if text in LmerMap:
            rank = LmerMap[text]
            current = LmerTable[rank]
            prevLocations = LmerTable[rank - 1].locations
            if not prevLocations or prevLocations[-1] != i - 1:
                setNewFamily(rank)
            if rank != expectedRank and expectedRank < len(LmerTable):
                setNewFamily(expectedRank)
            current.locations.append(i)
            expectedRank = rank + 1
        elif expectedRank and expectedRank < len(LmerTable):
            setNewFamily(expectedRank)
            expectedRank = 0

        if i % 1000 == 0:
            showProgress(i, S_len)

def writeFamily(length, locations, S, O):
    count = len(locations)
    O.write(">" + str(length) + ":" + str(count) + "\n")
    S.seek(locations[0])
    O.write(S.read(length) + "\n")

def writeFamilies(C, L, S, O):
    length = L
    familyStartRank = 0
    locations = LmerTable[familyStartRank].locations

    countFams = 1
    countRepeats = len(locations)

    tableSize = len(LmerTable)
    for rank in range(1, tableSize):
        if isNewFamily(rank):
            countFams += 1
            countRepeats += len(locations)
```

```
                writeFamily(length, locations, S, O)
                locations = LmerTable[rank].locations
                length = L
            else:
                length += 1

            if rank % 100 == 0:
                showProgress(rank, tableSize)
        writeFamily(length, locations, S, O)
        print(str(countFams) + " families identified")
        print(str(countRepeats) + " repeats identified")

    def raider(S, S_len, C, L, O):
        print("Counting Lmers...")
        countLmers(S, S_len, C, L)
        print("\nMarking families...")
        markFamilies(S, S_len, C, L)
        LmerMap = None
        print("\nWriting results...")
        writeFamilies(C, L, S, O)

    def showProgress(progress, total):
        percent = int(float(progress) / (float(total) / 100.0))
        sys.stdout.write('\r' + "[")
        sys.stdout.write("".join(["|" for i in range(percent)]))
        sys.stdout.write("".join(["-" for i in range(100 - percent)]))
        sys.stdout.write("]" + str(percent) + "%")
        sys.stdout.flush()

    def bootstrap(input, output, C, L):
        S_len = os.path.getsize(input)
        S = open(input, 'r')
        O = open(output, 'w')
        raider(S, S_len, C, L, O)
        S.close()
        O.close()

    def parseArgs():
        parser = argparse.ArgumentParser("RAIDER")
        parser.add_argument('-c', '--count', dest = 'count', help = "Minimum
number of occurrences to be significant", default = 8)
        parser.add_argument('-l', '--length', dest = 'length', help = "Minimum
length required to be elementary", default = 16)
        parser.add_argument("input_file", help = "File containing reference
sequence. Must be simply the sequence, with no line breaks")
        parser.add_argument("output_file", help = "Output file name")
        return parser.parse_args()

    if __name__ == "__main__":
        args = parseArgs()
        bootstrap(args.input_file,      args.output_file,      int(args.count),
int(args.length))
```

11 Works Cited

[1] A. H. R. Smit, “RepeatMasker Open-3.0,” 2008-2010. [Online]. Available: http://www.repeatmasker.org.

[2] Z. E. S. R. Bao, “Automated de novo identification of repeat sequence families in sequenced genomes,” *Genome Research,* no. 12.8, pp. 1269-1276, 2002.

[3] J. T. M. L. Bin Ma, “PatternHunter: faster and more sensitive homology search,” *Bioinformatics,* 2001.

[4] "Initial Sequencing and Analysis of the Human Genome.," *Nature,* vol. 409, no. 6822, pp. 860-921, 2001.

[5] K. S. Julien Häsler, “Alu elements as regulators of gene expression,” *Nucleic Acids Research,* pp. 5491-5497, November 2006.

[6] H. Q. Casey M. Bergman, “Discovering and detecting transposable elements in genome sequences,” *Briefings in Bioinformatics,* vol. 8, no. 6, pp. 382-384, October 2007.

[7] D. He, “Using Suffix Tree to Discover Complex Repetitive Patterns in DNA Sequences,” in *28th IEEE*, New York City, USA, 2006.

[8] H. Huo, V. Stojkovic and X. Wang, “An adaptive suffix tree based algorithm for repeats recognition in a DNA sequence,” *Systems Biology and Intelligent Computing,* 2009.

[9] S. L. Jie Zheng, “Discovery of Repetitive Patterns in DNA with Accurate Boundaries,” in *5th IEEE Symposium on Bioinformatics and Bioengineering*, 2005.

[10] V. S. Hongwei Huo, “A Suffix Tree Construction Algorithm for DNA Sequences,” in *Bioinformatics and Bioengineering*, Boston, MA, 2007.

[11] D. Gusfield, Algorithms on Strings, Trees, and Sequences, New York, NY: Press Syndicate of the University of Cambridge, 1997.

[12] S. F. M. T. L. S. A. A. Z. J. Z. Z. M. W. &. L. D. J. Altschul, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.,” *Nucleic Acid Research,* vol. 25, no. 17, pp. 3389-3402, 1997.

[13] D. J. H. P. D. Victoria P. Belancio, “Mammalian non-LTR retrotransposons: For better or worse, in sickness and in health,” Cold Spring Harbor Labarotory Press, 2008, pp. 343-358.

[14] E. M. McCreight, “A Space-Economical Suffix Tree Construction Algorithm,” *Journal of the ACM,* vol. 23, no. 2, pp. 262-272, April 1976.

[15] A. L. J. N. C. &. P. P. A. Price, “ De novo identification of repeat families in large genomes.,” *Bioinformatics,* vol. 21(suppl 1), pp. 351-358, 2005.

[16] S. B. S. M. Z. P. D. Saha, “Empirical comparison of ab initio repeat finding programs,”

Nucleic acids research, vol. 36, no. 7, pp. 2284-2294, April 2008.

[17] R. Hariharan, "Optimal Parallel Suffix Tree Construction," *ACM,* p. 290, 1994.