Bioinformatics, 2026, **42**(3), btag112
https://doi.org/10.1093/bioinformatics/btag112
Published: 13 March 2026
Original Paper | Sequence analysis

OXFORD

Finding low-complexity DNA sequences with longdust

Heng Li[1,2,3,*] and Brian Li[4]

[1]Department of Biomedical Informatics, Harvard Medical School, Boston, MA 02215, United States
[2]Department of Data Science, Dana-Farber Cancer Institute, Boston, MA 02215, United States
[3]Broad Insitute of MIT and Harvard, Cambridge, MA 02142, United States
[4]Commonwealth School, Boston, MA 02116, United States
*Corresponding author. Department of Biomedical Informatics, Harvard Medical School, Boston, MA 02215, United States; Department of Data Science, Dana-Farber Cancer Institute, Boston, MA 02215, United States; Broad Insitute of MIT and Harvard, Cambridge, MA 02142, United States.
E-mail: hli@ds.dfci.harvard.edu
Associate Editor: Dr. Can Alkan

Abstract

Motivation: **Low-complexity (LC) DNA sequences are compositionally repetitive sequences that are often associated with spurious homologous matches and variant calling artifacts. While algorithms for identifying LC sequences exist, they either lack concise mathematical definition of complexity or are inefficient with long or variable context windows.**

Results: **Longdust is a new algorithm that efficiently identifies long LC sequences including centromeric satellite and tandem repeats with moderately long motifs. It defines string complexity by statistically modeling the *k*-mer count distribution with the parameters: the *k*-mer length, the context window size and a threshold on complexity. Longdust exhibits high performance on real data and high consistency with existing methods.**

Availability and implementation: **https://github.com/lh3/longdust**

1 Introduction

A string is of low complexity (LC) if it is repetitive in composition. LC strings may lead to spurious homologous matches (Morgulis *et al*. 2006) and cause variant calling artifacts (Li 2014; Qin and Li 2025; Li 2025). In computer science, string complexity can be measured with Kolmogorov complexity (Kolmogorov 1998; Silva *et al*. 2022), Lempel-Ziv complexity (Lempel and Ziv 1976; Orlov and Potapov 2004) or Shannon entropy (Shannon 1948). In genomics, topological entropy (Crochemore and Vérin 1999), linguistic complexity (Troyanskaya *et al*. 2002) and Markov additive process (Spouge 2007) have also been explored. A complication in finding LC regions is that we want to identify boundaries of local LC regions, instead of classifying an entire string as LC or not. Measuring complexity with fixed-sized sliding windows is not adequate, either, as we could not pinpoint the boundaries with such methods.

SDUST (Morgulis *et al*. 2006) is a popular algorithm to identify local LC regions. It defines string complexity by running 3-mer counts and finds exact solutions under this definition. However, with the $O(w^3L)$ time complexity, where w is the window size and L is the genome length, SDUST is inefficient given a large w. In addition, the complexity scoring function used by SDUST grows quadratically with the string length. It tends to classify long strings as LC over short ones. A large window size w would make this worse. As a result, SDUST is not suitable for finding satellite or tandem repeats with long motifs.

In genomes, LC strings are usually tandemly repetitive due to DNA polymerase slippage. Nonetheless, due to point mutations in evolution and recurrent slippage to the same region but over different bases, long tandem repeats are often "impure" in that units in a tandem repeat differ from each other. Such impure tandem repeats can be identified with tandem repeat finding algorithms such as TRF (Benson 1999), ULTRA (Olson and Wheeler 2024) and pytrf (Du *et al*. 2025). They report repeat motifs and copy numbers, which are useful in downstream analyses. TANTAN (Frith 2011) models tandem patterns but can often identify general LC regions in practice. Unlike SDUST, these algorithms do not define string complexity with concise mathematical equations.

Inspired by SDUST, we sought an alternative way to define the *k*-mer complexity of a string and to identify local LC regions in a genome. Our complexity scoring function is based on a statistical model of *k*-mer count distribution and our algorithm is practically close to $O(wL)$ in time complexity, enabling the efficient identification of LC strings in long context windows.

2 Methods

Similar to SDUST (Morgulis *et al*. 2006), we define the complexity of a DNA string with a function of *k*-mer counts. In this section,

Received: 8 September 2025. **Revised:** 28 January 2026. **Accepted:** 3 March 2026

we will model the k-mer count distribution of random strings with each k-mer occurring at equal frequency and describe the complexity scoring function and the condition on bounding LC substrings in a long string. We will also discuss alternative scoring functions such as the SDUST scoring and Shannon entropy.

2.1 Notations

Let $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ be the DNA alphabet, $x \in \Sigma^*$ is a DNA string and $|x|$ is its length. $t \in \Sigma^k$ is a k-mer. For $|x| \geq k$, $c_x(t)$ is the count of k-mer t in x; $\ell(x) = \sum_t c_x(t) = |x| - k + 1$ is the total number of k-mers in x. $\overrightarrow{c}_x$, of size 4^k, denotes the count array over all k-mers.

In this article, we assume there is one long genome string of length L. We use closed interval $[i,j]$ to denote the substring starting at i and ending at j, including the end points i and j. We may use "interval" and "substring" interchangeably.

2.2 Modeling k-mer counts

For a k-mer $t \in \Sigma^k$ in a random string x, q_t is its frequency with $\sum_t q_t = 1$. $\overrightarrow{c}_x$ follows a multinomial distribution under a bag-of-words model:

$$P(\overrightarrow{c}_x; \overrightarrow{q}) = \ell(x)! \prod_{t\in\Sigma^k} \frac{q_t^{c_x(t)}}{c_x(t)!}$$

For convenience, introduce $r_t \triangleq 4^k q_t$. We have

$$\log P(\overrightarrow{c}_x; \overrightarrow{q}) = g(\ell(\S)) - \sum_t \log c_x(t)! + \sum_t c_x(t) \log r_t \quad (1)$$

Where

$$g(\ell) = \log \ell! - k\ell \log 4$$

Is a function of string length but does not depend on k-mer counts.

To get an intuition about the effect of low-complexity strings, suppose $q_t = 1/4^k$ and $\ell(x) \ll 4^k$. In this case, the last term in Eq. (1) is 0 and $c_x(t)$ is mostly 0 or 1 for a random string. $\log P(\overrightarrow{c}_x)$ will be close to $g(\ell(x))$. Given an LC string of the same length, we will see more $c_x(t)$ of 2 or higher, which will reduce $\log P(\overrightarrow{c}_x)$. Thus the probability of an LC string is lower under this model.

Although $\log P(\overrightarrow{c}_x)$ can be used to compare the complexity of strings of the same length, it does not work well for strings of different lengths because $\log P(\overrightarrow{c}_x)$ decreases with $\ell(x)$. We would like to scale it to $Q(\overrightarrow{c}_x)$ such that Q approaches 0 given a random string. We note that the mean of $\log P(\overrightarrow{c}_x)$

$$H(\ell) = \sum_{\overrightarrow{c}_x} P(\overrightarrow{c}_x) \log P(\overrightarrow{c}_x)$$

Is the negative entropy of $P(\overrightarrow{c}_x)$, which is

$$H(\ell) = \log \ell! + \ell \sum_t q_t \log q_t - f(\ell) = g(\ell) + \ell \sum_t q_t \log r_t - f(\ell)$$

Where

$$f(\ell; \overrightarrow{q}) \triangleq \sum_t \sum_{n=0}^{\ell} \binom{\ell}{n} q_t^n (1-q_t)^{\ell-n} \log n!$$

Under the Poisson approximation,

$$f(\ell; \overrightarrow{q}) \approx \sum_t e^{-\ell q_t} \sum_{n=0}^{\infty} \log n! \cdot \frac{(\ell q_t)^n}{n!} \quad (2)$$

Now introduce

$$\begin{aligned}\tilde{Q}(\overrightarrow{c}_x) &\triangleq H(\ell) - \log P(\overrightarrow{c}_x) \\ &= \textstyle\sum_t \log c_x(t)! - f(\ell(x)) + \sum_t [\ell(x) q_t - c_x(t)] \log r_t\end{aligned}$$

$\tilde{Q}(\overrightarrow{c}_x)$ approaches 0 given a random string x regardless of its length. We can thus compare strings of different lengths. However, $\tilde{Q}$ measures both k-mer repetitiveness in the first term and k-mer usage in the last term. It is possible for $\tilde{Q}$ to reach a large value if string x is composed of rare unique k-mers. To cancel the effect of rare k-mers, we ignore the last term and use

$$Q(\overrightarrow{c}_x; \overrightarrow{q}) \triangleq \sum_t \log c_x(t)! - f(\ell(x); \overrightarrow{q})$$

To measure the complexity of string x. $Q(\overrightarrow{c}_x)$ also approaches 0 given a random string x because $\tilde{Q}(\overrightarrow{c}_x)$ approaches 0 and $c_x(t)$ approaches $\ell(x)q_t$. $Q(\overrightarrow{c}_x)$ is higher for LC strings.

In practice, k-mer frequency $\overrightarrow{q}$ is computed from genome-wide GC content. Low or high GC content leads to elevated $f(\ell)$ (Fig. 1), which is expected as $c_x(t)$ grows faster with more biased GC content.

2.3 Scoring low-complexity intervals

To put a threshold on the complexity, we finally use the following function to score string complexity:

$$S(\overrightarrow{c}_x) \triangleq Q(\overrightarrow{c}_x) - T \cdot \ell(x) = \sum_t \log c_x(t)! - T \cdot \ell(x) - f(\ell(x)) \quad (3)$$

Threshold T controls the level of complexity in the output. It defaults to 0.6, less than $\log 2$. At $k = 7$, $f(\ell)$ is close to 0 for small ℓ. 0.6ℓ is the larger term within the default window size of 5 kb (Fig. 1).

If $S(\overrightarrow{c}_x) > 0$, x is considered to contain an LC substring. However, we often do not want to classify the entire x as an LC substring in this case because the concatenation of a highly repetitive sequence and a random sequence could still lead to a positive score.

Recall that we may use close intervals to represent substrings. For convenience, we write $S(\overrightarrow{c}_{[i,j]})$ as $S(i,j)$. In implementation, we precalculate $f(\ell)$ and introduce

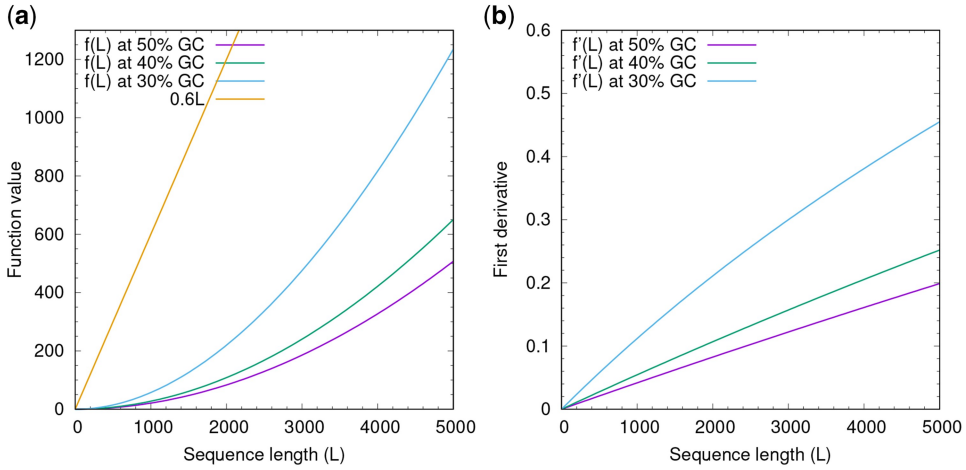

Figure 1 Scaling function. **(a)** Function value of $f(\ell)$ in Eq. (2) at $k=7$. **(b)** First derivative. Equivalent to per-base contribution.

$$\begin{aligned} U(i,j) &\triangleq \textstyle\sum_t \log c_{[i,j]}(t)! - T\cdot\ell([i,j]) \\ &= U(i,j-1) + \log c_{[i,j]}([j-k+1,j]) - T \end{aligned}$$

We can thus compute the complexity scores of all prefixes of $[i,j]$by scanning each base in the interval from left to right; we can similarly compute all suffix scores from right to left.

2.4 Finding low-complexity regions

We say x is a *perfect LC string* (or *perfect LC interval*) if $S(\vec{c}_x)>0$ and no substring of x is scored higher than $S(\vec{c}_x)$; say x is a *good LC string* (or *good LC interval*) if $S(\vec{c}_x)>0$ and no prefix or suffix of x is scored higher than $S(\vec{c}_x)$. We can use $U(i,j)$ above to test if $[i,j]$ is a good LC interval in linear time. If we apply this method to all intervals up to w in length (5000 bp by default), we can find LC regions of context length up to w in $O(w^2L)$ time. The union of all good LC intervals marks the LC regions in a genome. Algorithm 1 shows a faster way to find a good LC interval ending at j. Function Backward() scans backwardly from j to $j-w+1$ to collect candidate start positions (line 22). Variable v is the complexity score of suffix $[i-k+1,j]$ (line 20). By the definition of good LC interval, i can only be a candidate start if v is no less than all the suffixes visited before (line 21). We also ignore a candidate start i if $S(i,j)<S(i-1,j)$ because if $[i-1,j]$ is not a good LC interval, there must exist $i'>i$ such that $S(i',j)>S(i-1,j)<S(i,j)$, so $[i,j]$ would not be a good LC interval, either. In addition, if suffix $[i,j]$ is enriched with k-mers unique in the full window $[j-w+1,j]$, $[i,j]$ will not be a good LC interval (line 27) as there will exist $j'<j$ such that $S(i,j')>S(i,j)$. The time complexity of Backward() is $O(w)$.

Given a candidate start position i, function Forward() returns $j'=\text{argmax}_{i<j'\le j}S(i,j')$. $[i,j]$ will be a good LC interval if and only if $j'=j$ (Line 7). We call Forward() in the ascending order of candidate start positions (line 4). We may skip a start position if it is contained in an interval found from previous Forward() calls (line 5). This is an approximation as it is possible for a good LC interval to start in another good interval. An alternative heuristic is to only apply Forward() to the smallest candidate start in B. This leads to a guaranteed $O(w)$ with FindStart(). In practice, the two algorithms have almost identical runtime. We use Algorithm 1 in longdust as it is closer to the exact algorithm.

Function FindStart() finds the longest good LC interval ending at one position. We apply the function to every position in the genome to find all good LC intervals. We can skip j if $[j-k+1,j]$ is unique in $[j-w+1,j]$ because the forward pass would not reach j in this case. We also introduce a heuristic to extend a good LC interval $[j-w,j-1]$ to $[j-w+1,j]$ without calling FindStart().

Our definition may classify a short non-repetitive interval to LC if it is surrounded by highly repetitive LC intervals. We additionally use an X-drop heuristic (Altschul *et al.* 1997) to alleviate this issue. On the T2T-CHM13 genome, the total length of LC intervals is 0.5% shorter with X-drop. The heuristic has a minor effect.

The overall longdust algorithm is inexact and may result in slightly different LC regions on opposite strands. We run the algorithm on both the forward and the reverse strand of the input sequences and merge the resulting intervals. The default longdust output is strand symmetric.

2.5 The SDUST scoring

SDUST (Morgulis *et al.* 2006) uses the following complexity scoring function:

$$S_S(\vec{c}_x) = \frac{1}{\ell(x)}\sum_t \frac{c_x(t)(c_x(t)-1)}{2} - T$$

This function grows linearly with $\ell(x)$ for $\ell(x)\ge 4^k$, while our scoring function grows more slowly in the logarithm scale. The SDUST function is more likely to classify longer sequences as LC.

Furthermore, SDUST looks for perfect LC intervals rather than good LC intervals like longdust. It cannot test whether an interval is perfect in linear time. Instead, SDUST maintains the complete list of perfect intervals in window $[j-w+1,j]$ and tests a new candidate interval against the list. The FindStart() equivalent of SDUST is $O(w^3)$ in time, impractical for long windows.

Algorithm 1 Find LC interval ending at j

procedure FindStart(k, w, T, j, c')
  $B \leftarrow$ Backward(k, w, T, j, c')
  $j'_{\max} \leftarrow -1$
  for $(i, v') \in B$ in the ascending order of i **do**
    continue if $i < j'_{\max}$ ▷ this is an approximation
    $j' \leftarrow$ Forward(k, T, i, j, v')
    return i **if** $j' = j$ ▷ $[i, j]$ is a good LC interval
    $j'_{\max} \leftarrow \max(j'_{\max}, j')$
  end for
  return -1 ▷ No good LC interval ending at j
end procedure
procedure Backward(k, w, T, j, c')
  $u \leftarrow 0$; $v_0 \leftarrow -1$; $u' \leftarrow 0$
  $v_{\max} \leftarrow 0$; $i_{\max} \leftarrow -1$; $c \leftarrow [0]$
  $B \leftarrow \emptyset$
  for $i \leftarrow j$ **to** $\max(j - w + 1, k - 1)$ **do** ▷ i is descending
    $t \leftarrow [i - k + 1, i]$ ▷ the k-mer ending at i
    $c[t] \leftarrow c[t] + 1$
    $u \leftarrow u + \log(c[t]) - T$
    $v \leftarrow u - f(j - i + 1)$ ▷ $v = S(i - k + 1, j)$
    if $v < v_0$ **and** $v_0 = v_{\max}$ **then**
      $B \leftarrow B \cup \{(i + 1, v_{\max})\}$ ▷ a candidate start pos
    else if $v \geq v_{\max}$ **then**
      $v_{\max} \leftarrow v$; $i_{\max} \leftarrow i$
    else if $i_{\max} < 0$ **then**
      $u' \leftarrow u' + \log(c'[t]) - T$ ▷ $c'[t] \triangleq c_{[j-w+1,j]}(t)$
      break if $u' < 0$ ▷ Forward() wouldn't reach j
    end if
    $v_0 \leftarrow v$
  end for
  $B \leftarrow B \cup \{(i_{\max}, v_{\max})\}$ **if** $i_{\max} \geq 0$
  return B
end procedure
procedure Forward($k, T, i_0, j, v'_{\max}$)
  $u \leftarrow 0$; $v_{\max} \leftarrow 0$; $i_{\max} \leftarrow -1$; $c \leftarrow [0]$
  for $i \leftarrow i_0$ **to** j **do**
    $t \leftarrow [i - k + 1, i]$
    $c[t] \leftarrow c[t] + 1$
    $u \leftarrow u + \log(c[t]) - T$
    $v \leftarrow u - f(i - i_0 + 1)$
    if $v \geq v_{\max}$ **then**
      $v_{\max} \leftarrow v$; $i_{\max} \leftarrow i$
    end if
    break if $v > v'_{\max}$
  end for
  return $i_{\max}$
end procedure

SDUST hardcodes $k = 3$ and uses $w = 64$ by default for acceptable performance.

2.6 Scoring with Shannon entropy

Let $p_x(t) = c_x(t)/\ell(x)$. The Shannon entropy of string x is

$$H(\overrightarrow{c}_x) \triangleq -\sum_t p_x(t) \log p_x(t) = \log \ell(x) - \frac{1}{\ell(x)} \sum_t c_x(t) \log c_x(t)$$

When $\ell(x) \leq 4^k$, $H(\overrightarrow{c}_x)$ reaches the maximum value of $\log \ell(x)$ at $p_x(t) = 1/\ell(x)$. $H(\overrightarrow{c}_x)$ also grows with $\ell(x)$. For $\ell(x) \leq 4^k$, define

$$S_E(\overrightarrow{c}_x) \triangleq \log \ell(x) - H(\overrightarrow{c}_x) - T = \frac{1}{\ell(x)} \sum_t c_x(t) \log c_x(t) - T$$

We adapted longdust for S_E and found using S_E is more than twice as slow. We suspected some longdust heuristics did not work well with S_E.

3 Results

3.1 Low-complexity regions in T2T-CHM13

We applied longdust, SDUST v0.1 (Morgulis *et al.* 2006), pytrf v1.4.2 (Du *et al.* 2025), TRF v4.10 (Benson 1999), TANTAN v51 (Frith 2011), and ULTRA v1.20 (Olson and Wheeler 2024) to the T2T-CHM13 human genome (Nurk *et al.* 2022). The maximum period was set to 500 (Table 1). Notably, TRF would not finish in days with the default option `-l`. With `-l30`, TRF could run in 11 hours using 17.82GB memory at the peak.

Due to the lack of ground truth, we could not calculate the accuracy of longdust. We instead checked how often longdust reports regions unique to itself and how often it misses regions reported by other tools, especially by two or more of them.

Longdust finds 277.1 Mb of LC regions with 224.3 Mb overlapping with centromeric satellite annotated by the telomere-to-telomere (T2T) consortium (Fig. 2). Of the remaining 52.7 Mb, 34.1 Mb overlaps with TRF; 15.4 Mb of the remainder (18.6 Mb) is found by SDUST. Only 3.2 Mb is left, suggesting most longdust LC regions fall in centromeres or are found by TRF or SDUST. Longdust uses three parameters: k-mer size, window size w and score threshold T. Varying these parameters do not greatly alter the results. The GC adjustment also has minor effect even when we intentionally set the genome-wide GC below the actual value of 41%.

TRF, the most popular tandem repeat finder, finds 274.5 Mb of tandem repeats, 244.0 Mb of which have ≥ 4 copies of repeat units. 97.9% of the 244.0 Mb are identified by longdust. TRF additionally reports tandem repeats with < 4 repeat units. Only 14.8% of them overlap with longdust results. Longdust misses

Table 1 Command lines and resource usage for T2T-CHM13.

Tool	t_{CPU} (h)	Mem (G)	Command line
longdust	0.94	0.47	(default)
SDUST	0.07	0.23	−t30
pytrf	2.64	0.70	−M500
TRF	12.83	7.49	2 7 7 80 10 50 500 -l12
TANTAN	0.56	1.28	−w500
ULTRA	146.07	33.31	−p 500 -t 16

Performance measured on a Linux server equipped with Intel Xeon Gold 6130 CPU and 512GB memory.

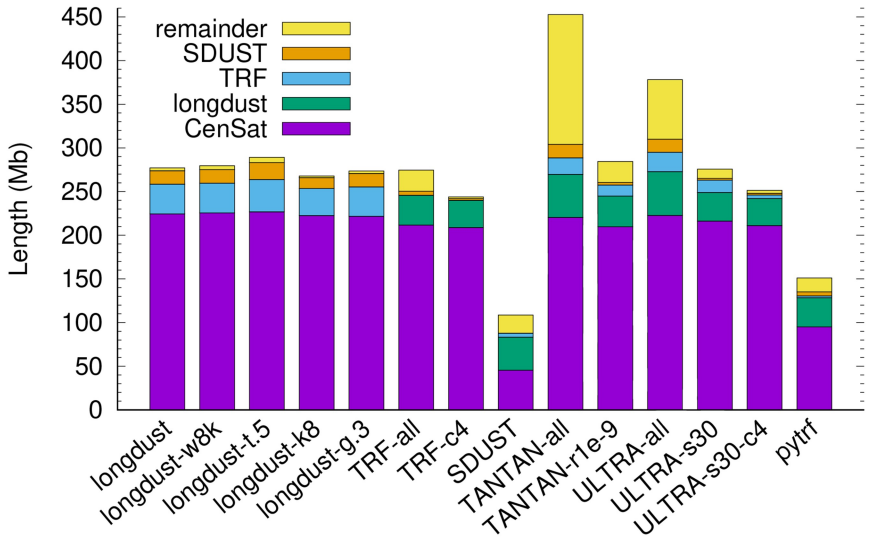

Figure 2 Lengths of low-complexity regions in T2T-CHM13. Low-complexity (LC) regions identified by each tool are first intersected with centromeric satellite annotation. The remainder is then intersected with longdust, TRF and SDUST in order. There are no overlaps between stacks. The total height is the length of LC regions found by each tool. Alternative settings – "longdust-w8k": window size $w = 8000$; "longdust-t.5": score threshold $T = 0.5$; "longdust-k8": using 8-mers; "longdust-g.3": genome-wide GC set to 30%; "TRF-c4": requiring ≥ 4 copies of the repeat unit; "TANTAN-r1e-9": run with "-r 1e-9"; "ULTRA-s30": requiring score ≥ 30 in the output.

repeats with low copy numbers. In fact, under the default threshold $T = 0.6$, the minimum numbers of exact copies longdust can find is approximately:

$$3 + \frac{k-1}{r} + \frac{3T - \log 2 - \log 3}{\log 4 - T} \approx 3.01 + \frac{k-1}{r}$$

This assumes the repeat unit length $r > k$, $f(\cdot) = 0$ and all k-mers are unique within the repeat unit.

ULTRA outputs 68.2 Mb of regions not reported by longdust, TRF or SDUST. Nevertheless, if we raise its score threshold to 30, the total region length is similar to that of TRF with most of repeats of ≥ 4 copies captured by longdust. Under the default setting, TANTAN finds the largest LC regions missed by longdust, TRF or SDUST. Decreasing the repeat starting threshold from 0.005 to 10^{-9} greatly reduces its unique regions from 148.5 Mb to 24.5 Mb. We suspect the remaining regions unique to TANTAN may have low copy number but we cannot control its output. TANTAN can optionally generate repeat units but this mode is six times as slow and reports only 27% of regions in the default mode.

As to other tools, pytrf cannot effectively identify alpha satellite with ~170bp repeat units, even though it was set to find tandem repeats with unit up to 500 bp. With a 64 bp window size, SDUST naturally misses tandem repeats with long units, including all alpha satellite.

3.2 Behaviors in long satellite repeats

Although longdust, TRF, TANTAN and ULTRA report similar amount of human satellites, the LC block sizes vary greatly. We will take the centromeric satellite of chromosome 11 for an example. Around this centromere, longdust reports a continuous block of 3.5 Mb from 50,999,020 to 54,488,059 on T2T-CHM13. TRF finds a nearly identical region, with the ending position shifted by only 3 bp. In contrast, although ULTRA finds 99.86% of this region, it breaks them into many smaller blocks. The longest block is 120 kb in size. TANTAN under the default setting misses 2.3% of this long alpha array and its output is more fragmented. The longest block is only 1587 bp. With "-r 1e-9," TANTAN misses 7.6% of this satellite. Across all satellite regions annotated on T2T-CHM13, the N50 block size is 2.9 Mb, 2.0 Mb, 10.2 kb and 340 bp for longdust, TRF, ULTRA and TANTAN, respectively. Longdust reports the longest blocks probably because it merges satellites of different types.

One application of LC masking is to speed up cross-species whole-genome alignment. 90.8% of satellites found by longdust lie in blocks longer than 10 kb and conversely, 99.2% of blocks longer than 10 kb fall in annotated satellites. By setting a block length threshold, we can mask the bulk of satellite without affecting the rest of the genome. A similar strategy works for TRF that also reports long blocks. We would need to stitch shorter blocks reported ULTRA and TANTAN to use this strategy, which would add another layer of complexity.

3.3 Checking strand symmetry

SDUST is strand symmetric in that running SDUST on either strand of the input sequences gives identical results. While longdust defines string complexity with strand symmetry, it uses heuristics in implementation and may report different LC intervals if it is run on the reverse complement of the input sequences. To achieve strand symmetry, longdust applies the same algorithm to both strands and output the union intervals. Without the union, 0.17% of LC regions are specific to one strand (Fig. 3).

We also checked the strand differences of other tools. For most of them, a few percent of LC regions are strand specific. However, a large fraction of LC regions found by TANTAN was

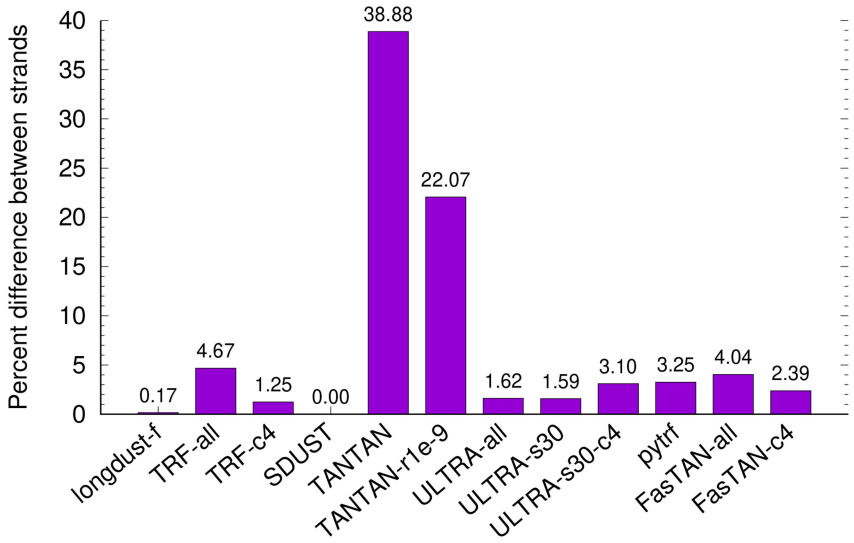

Figure 3 Strand asymmetry. Each tool is run on the original T2T-CHM13 and the reverse complement of the genome. The LC intervals outputted from the two runs are compared. The Y axis shows 1 − intersection/union for each tool. Longdust by default applies the algorithm to both strands and outputs the union; "longdust-f" refers to the mode that processes one strand only.

only reported on one strand. The difference is driven by regions outside centromeric satellites, where the percent difference reaches 60.0%. Although TANTAN intends to keep the leftmost repeat copy intact, the large difference is still unexpected given that human centromeres are mostly composed of several types of megabase-long tandem arrays.

3.4 Low-complexity regions in a gorilla genome

The near T2T gorilla genome (AC: GCF_029281585.2; Yoo *et al.* 2025) is 3546 Mb in size, 428 Mb larger than the human T2T-CHM13 genome. We ran longdust on the gorilla genome for 1.4 hours and found 656.8 Mb of LC regions. That is 379.7 Mb larger than the LC regions in T2T-CHM13. The genome size difference is primarily driven by LC regions.

To further confirm this observation, we extracted 298.8 Mb of regions in the gorilla genome that are $\geq$10kb in length without any 51-mer exact matches to 472 human genomes (Li 2024). 99.7% of them are marked as LC regions by longdust and none of them are alpha satellite. 95.8% of these gorilla-specific regions are distributed within 15 Mb from telomeres, broadly in line with Yoo *et al.* (2025).

TANTAN with "`-w500 -r1e-9`" reported 672.2 Mb of LC regions, 629.1 Mb of which overlapped with longdust. TANTAN still tends to report fragmented regions with an N50 block length of 3.7 kb. We also ran TRF on the gorilla genome. It did not finish in a week with option "`-l20`."

4 Discussions

The development of longdust was motivated by our recent work on small variant filtration (Li 2025) and structural variant stratification (Qin and Li 2025). In the first work, we initially used SDUST and noticed it missed long LC regions that harbor false variant calls. We improved the results with longdust. In the second work, we needed to scan hundreds of human assemblies to find LC regions not present on the reference genome. Longdust was chosen for its high performance. In both cases, we investigated the TRF output on the reference genome and found the conclusions remained similar.

Du *et al.* (2025) observed that each tandem repeat finding tool reported a large fraction of regions unique to itself. We think this is driven by weak tandem repeats that are composed of a few diverged repeat units. If we focus on highly repetitive LC regions, longdust, TRF, TANTAN and ULTRA agree reasonably well with each other. In lack of ground truth, it is difficult to tell which tool is more accurate in terms of genome-wide coverage. We do not intend to brand longdust as the best tool for LC finding; we emphasize more on a new probabilistic model for defining string complexity, which advances the SDUST algorithm and achieves a descent balance between performance, coverage, and strand symmetry.

From the theoretical point of view, longdust uses an approximate algorithm. It tests LC intervals ending at each position with Algorithm 1, but has not sufficiently exploited dependencies between positions. It will be interesting to see whether there is an exact $O(wL)$ algorithm under the longdust formulation or a meaningful alternative formulation that leads fast implementations.

In comparison to tandem repeat finding tools such as TRF and ULTRA, longdust is unable to report the repeat units in case of tandem repeats. This will limit the use cases of longdust. Another practical limitation of longdust is the restricted window size. The genome of Woodhouse's scrub jays, for example, contains satellite with a 18 kb repeat unit (Edwards *et al.* 2025). This would be missed by longdust under the default setting. Increasing the window size would make longdust considerably slower. This is partly due to the $O(wL)$ time complexity and partly due to the speedup strategies in longdust that are more effective given $\ell(x) \ll 4^k$. It would be ideal to have an algorithm that remains efficient given large windows or, better, does not require specified window sizes.

Acknowledgments

We are grateful to Qian Qin for evaluating the effect of low-complexity regions in structural variant calling, and to Maximilian Haeussler for explaining the role of masking low-complexity regions in cross-species alignment. We thank the reviewers for suggesting cleaner derivation.

Author contributions

Heng Li(Conceptualization [Lead], Investigation [Lead], Methodology [Lead], Software [Lead], Supervision [Lead], Visualization [Lead], Writing—original draft [Lead], Writing—review & editing [Lead]), Brian Li(Software [Supporting]).

Conflict of interest

None declared.

Funding

This work is supported by US National Institute of Health grant R01HG010040, R01HG014175, U01HG013748, U41HG010972, and U24CA294203 (to H.L.).

Data availability

https://github.com/lh3/longdust

References

Altschul SF, Madden TL, Schäffer AA *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res* 1997;**25**:3389–402.

Benson G. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Res* 1999;**27**:573–80.

Crochemore M, Vérin R. Zones of low entropy in genomic sequences. *Comput Chem* 1999;**23**:275–82.

Du L, Sun D, Chen J *et al.* Pytrf: a python package for finding tandem repeats from genomic sequences. *BMC Bioinformatics* 2025;**26**:151.

Edwards SV, Fang B, Khost D *et al.* Multispecies pangenomes reveal a pervasive influence of population size on structural variation. *Science* 2025;**390**:eadw1931.

Frith, M. C. (2011). A new repeat-masking method enables specific detection of homologous sequences. *Nucleic Acids Res*, **39**: e23.

Kolmogorov AN. On tables of random numbers (reprinted from "Sankhya: The Indian Journal of Statistics", series a, vol. 25 part 4, 1963). *Theor. Comput. Sci* 1998;**207**:387–95.

Lempel A, Ziv J. On the complexity of finite sequences. *IEEE Trans Inform Theory* 1976;**22**:75–81.

Li H. Toward better understanding of artifacts in variant calling from high-coverage samples. *Bioinformatics* 2014;**30**:2843–51.

Li, H. (2024). BWT construction and search at the terabase scale. *Bioinformatics*, **40**:btae717.

Li, H. (2025). Finding easy regions for short-read variant calling from pangenome data. *Gigascience*, **14**:giaf103.

Morgulis A, Gertz EM, Schäffer AA *et al.* A fast and symmetric DUST implementation to mask low-complexity DNA sequences. *J Comput Biol* 2006;**13**:1028–40.

Nurk S, Koren S, Rhie A *et al.* The complete sequence of a human genome. *Science* 2022;**376**:44–53.

Olson, D. R. and Wheeler, T. J. (2024). ULTRA–effective labeling of tandem repeats in genomic sequence. *Bioinform Adv*, **4**:vbae149.

Orlov, Y. L. and Potapov, V. N. (2004). Complexity: an internet resource for analysis of DNA sequence complexity. *Nucleic Acids Res*, **32**: W628–33.

Qin, Q. and Li, H. (2025). Challenges in structural variant calling in low-complexity regions. *Gigascience*, **14**:giaf154.

Shannon CE. A mathematical theory of communication. *Bell Syst Tech J* 1948;**27**:379–423.

Silva, J. M., Qi, W., Pinho, A. J., and Pratas, D. (2022). Alcor: alignment-free simulation, mapping, and visualization of low-complexity regions in biological data. *Gigascience*, **12**:giad101.

Spouge JL. Markov additive processes and repeats in sequences. *J Appl Probab* 2007;**44**:514–27.

Troyanskaya OG, Arbell O, Koren Y *et al.* Sequence complexity profiles of prokaryotic genomic sequences: a fast algorithm for calculating linguistic complexity. *Bioinformatics* 2002;**18**:679–88.

Yoo D, Rhie A, Hebbar P *et al.* Complete sequencing of ape genomes. *Nature* 2025;**641**:401–18.

Bioinformatics, 2026, 42, 1–7
https://doi.org/10.1093/bioinformatics/btag112
Original Paper