JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 13, Number 5, 2006

Pp. 1028–1040

A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences

ALEKSANDR MORGULIS,* E. MICHAEL GERTZ, ALEJANDRO A. SCHÄFFER, and RICHA AGARWALA

ABSTRACT

The DUST module has been used within BLAST for many years to mask low-complexity sequences. In this paper, we present a new implementation of the DUST module that uses the same function to assign a complexity score to a sequence, but uses a different rule by which high-scoring sequences are masked. The new rule masks every nucleotide masked by the old rule and occasionally masks more. The new masking rule corrects two related deficiencies with the old rule. First, the new rule is symmetric with respect to reversing the sequence. Second, the new rule is not context sensitive; the decision to mask a subsequence does not depend on what sequences flank it. The new implementation is at least four times faster than the old on the human genome. We show that both the percentage of additional bases masked and the effect on MegaBLAST outputs are very small.

Key words: DNA database searching, sequence masking.

1. INTRODUCTION

NATURALLY OCCURRING nucleotide sequences often exhibit intervals with highly biased distribution of nucleotides ("low-complexity" intervals). Such intervals make DNA database search engines, such as BLAST (Altschul *et al.*, 1997), produce large number of high-scoring but biologically insignificant results. Making an interval of either the query sequence or a matching database sequence unavailable for starting a BLAST match is called *(soft) masking* the interval.

The DUST module (R. Tatusov and D.J. Lipman, unpublished data) included in BLAST is used to mask low-complexity regions of nucleotide queries. DUST is a heuristic algorithm that employs a scoring function based on counting nucleotide triplet frequencies in 64-base windows, which is similar to the scoring function used in the SIMPLE utility (Hancock and Armstrong, 1994). In this paper, we present a new implementation of the DUST module that corrects some deficiencies in all previous versions. We will refer to the new implementation of the DUST module by SDUST and to the original implementation of the DUST module by DUST.

In DUST, there were two situations in which a characteristic of the input sequence gives rise to anomalous masking. First, DUST is not symmetric with respect to reversing the input sequence and hence is not

National Center for Biotechnology Information, National Institutes of Health, Department of Health and Human Services, Bethesda, Maryland.

*Under contract to MSD Inc., Fairfax, VA.

symmetric with respect to reverse complement. Thus, DUST's choice of which DNA intervals to mask can depend on which of two complementary DNA strands is presented as the digitized input. For example, consider the following sequence of 89 nucleotides, which is the subsequence in positions [198630,198718] of Genbank (Benson *et al.*, 2004) entry AC009229.5:

```
ACCTGCACATTGTGCACATGTACCCTAAAACTTAAAGTATAATAATAATAAAATT
AAAAAAAAATGCTACAGTATGACCCCACTCCTGG
```

and its reverse complement

```
CCAGGAGTGGGGTCATACTGTAGCATTTTTTTTTAATTTTATTATTATTATACTT
TAAGTTTTAGGGTACATGTGCACAATGTGCAGGT.
```

DUST masks positions 56–64 in the forward sequence and positions 26–64 in the reverse complement. SDUST masks positions 26–64 in both cases. The subsequence of length 89 was selected so that the portion masked by SDUST is centered with 25 unmasked nucleotides on either side.

The second anomaly we sought to correct is that DUST is context sensitive. Two sequences may contain an identical low-complexity subsequence, but that subsequence may be masked in one and not in the other. For example, consider the following sequences of length 85:

```
ACCTGCACATTGTGCACATGTACCCAAAAAAAAAGCGCGCGCGCGTTTTTTTACA
GTATGAAAAAAAAAAAAACCCCACTCCTGG
```

and

```
ACCTGCACATTGTGCACATGTACCCACAGTATCCTGCACATTGGCTTTTTTTACA
GTATGACAGTATGACAGTCCCCACTCCTGG.
```

In the first sequence, the run of Ts has two longer runs of As nearby on both sides, while in the second sequence the runs of As are changed to some high-complexity sequences. DUST masks both runs of As (intervals 26–34 and 61–73) but leaves the run of Ts in the first sequence unmasked. However, the run of Ts in the second sequence (interval 46–52) is masked by DUST. SDUST masks the run of Ts in both cases.

The DUST module is used in another recently developed NCBI tool, Windowmasker (Morgulis *et al.*, 2006), which is designed to search and mask repetitive and low-complexity sequences in DNA databases for the purpose of improving BLAST DNA search results. We noticed that the running time of WindowMasker suffered when DUST was used for filtering low-complexity sequences. This prompted us to look for ways to improve the computational efficiency of the DUST module.

We present SDUST, a modified version of DUST, that fixes the two disadvantages described above and performs several times faster with the commonly used set of parameters. SDUST uses the same scoring function as DUST to assign high scores to sequences of low complexity but differs from DUST on which high-scoring sequences are masked. The bases masked by SDUST form a superset of those masked by DUST. We show that the difference in number of nucleotides masked is less than 2% on all human chromosomes and that the effect on MegaBLAST outputs is minimal, as desired.

2. DEFINITIONS AND NOTATION

Let a be a sequence of n letters from the 4-letter alphabet $\mathcal{A} = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$. A *triplet* is a sequence of length 3. Any sequence of length $n > 2$ contains as subsequences exactly $n - 2$ triplets.

We assign a *score* $S(a)$ to any sequence a of length $n > 2$ in the following way. Let $\mathcal{R}$ denote the set of all 64 possible triplets over the alphabet $\mathcal{A}$. For $t \in \mathcal{R}$, let $c_t(a)$ be the number of times a triplet with

value t appears in a. We define $S(a)$ to be 0 if it has length 3. For sequence a of length $n > 3$, we define

$$S(a) = \frac{\sum_{t \in \mathcal{R}} c_t(a)(c_t(a) - 1)/2}{(\ell - 1)}, \tag{1}$$

where $\ell = n - 2$ is the number of triplets in a. Each triplet value t contributes $0 + 1 + 2 + \cdots + (c_t(a) - 1)$ to the numerator of the score function.

Both DUST and SDUST require two parameters: a positive integral *window size* W and a positive real-valued *score threshold* T. In practice, $W = 64$, and $T = 2$* for both DUST and SDUST. Both DUST and SDUST mask a subset of the intervals of q with length less than W and score greater than T. The critical concept in SDUST is the following definition that succinctly characterizes the intervals that are masked.

Definition 2.1. We define the set $\mathcal{P}(q, W, T)$ of *perfect intervals* to be the set of all subsequences a of q of length at most W and score greater than T that satisfy the property that the score of any subsequence of a is at most as high as the score of a.

SDUST masks a subsequence if and only if it is a perfect interval in q. DUST masks a subset of the nucleotides masked by SDUST but does not have a natural algebraic or combinatorial characterization of which intervals get masked. However, using algorithmic terminology and allowing the asymmetric use of left and right, the set of intervals masked by DUST can be characterized as described in the next section.

3. DUST ALGORITHM

DUST looks at all subsequences of fixed length W of the input nucleotide sequence; subsequences of length less than W are considered at the beginning and at the end of the input sequence. For each such subsequence a, it finds the high-scoring prefix a' with the largest score where ties are resolved in favor of the leftmost such prefix. If the score of the selected prefix is greater than a given threshold value T then the algorithm finds a subsequence a'' of a' with the maximum score (again the ties are resolved in favor of the leftmost subsequence). The algorithm then masks the union of all such a'' over all the windows in the input sequence. Pseudocode is included in Supplemental Information.

Because ties are broken in a directional fashion, the algorithm is not symmetric with respect to reversing the input sequence. The context-sensitivity problem, which was illustrated by an example in the Introduction, arises in the following general situation. Suppose there are three subsequences a_1, a_2, a_3, occurring in that order, such that the scores of a_1 and a_3 exceed the score of a_2. If the subsequences are sufficiently close together that every window containing a_2 contains either a_1 or a_3, then there may be no window containing a_2 in which a_2 is the highest scoring subsequence. Therefore, a_1 and a_3 form part of a flanking context that prevents a_2 from being masked.

4. OVERVIEW OF SDUST ALGORITHM

SDUST masks exactly those bases contained in $\mathcal{P}(q, W, T)$ for specified values of W and T. Because the score function S is invariant with respect to the operations of reverse and complement, it follows that $\mathcal{P}(q, W, T)$ is invariant with respect to taking reverse complements. Moreover, the property of being *perfect* does not depend on the position of the interval within a larger sequence. Thus, the set of masked intervals is not context sensitive.

Below is a high-level description of SDUST that illustrates how $\mathcal{P}(q, W, T)$ can be found. The algorithm considers a series $\{w_k\}$ of subsequences, known as *windows*, of the sequence q. Usually, w_k has length

*Both T and $S(a)$ are multiplied by 10 in the implementation. Therefore, the default for T in the dustmasker executable is 20 instead of 2.

exactly W and starts and ends exactly one base to the right of w_{k-1}, but shorter windows at the beginning of q are treated specially. We defer a precise discussion of how windows are defined at the beginning of q to the pseudocode in Supplemental Information.

For each window w_k, we compute a collection $\mathcal{P}_k$ of perfect intervals within w_k. The full set of perfect intervals is then the union of the perfect intervals in each window. In symbols,

$$\mathcal{P}(q, W, T) = \bigcup_{k=1}^{m} \mathcal{P}_k$$

where m is the number of windows considered. In practice, we need the collection of perfect intervals for only a single window in memory at any one time. As the iteration progresses from window w_{k-1} to w_k, we mask any bases contained in any of the perfect intervals in $\mathcal{P}_{k-1} \setminus \mathcal{P}_k$.

The algorithm maintains the following data structures that are modified in place as the iteration progresses.

- A structure that represents the right and left endpoints and the contents of the current window w_k; and
- A data structure that represents the collection $\mathcal{P}_k$ of all perfect intervals found so far within the current window.

For the first iteration, we add all perfect intervals in w_1 to $\mathcal{P}_1$. For subsequent iterations, we perform the following procedure to compute the set $\mathcal{P}_k$ and mask bases in q.

1. *Keep intervals from the previous window:* Initialize $\mathcal{P}_k$ to be all intervals in $\mathcal{P}_{k-1}$ that are subsequences of w_k. Mask the bases covered by any interval in $\mathcal{P}_{k-1}$ that is not a subsequence of w_k.
2. *Check suffixes of the current window:* For each suffix of w_k with score higher than T, add the suffix to $\mathcal{P}_k$ if the suffix does not contain either a higher-scoring interval already in $\mathcal{P}_k$ or a shorter, higher-scoring suffix of w_k.
3. *Check termination condition:* If w_k is the final window in Q, mask any bases covered by any interval in $\mathcal{P}_k$.

5. ALGORITHMIC OPTIMIZATION

Step 2 of SDUST is the most computationally expensive. In this section, we describe an efficient implementation of Step 2.

The following two propositions present conditions that allow us to remove some or all window suffixes from consideration in Step 2. For the first proposition, we seek a threshold value M, independent of the length of the sequence a, so that if $c_t(a) \leq M$ for all triplets t, then $S(a) \leq T$. Proposition 1 shows that $M = 2T$.

Proposition 1. *For any sequence a of length at least 3, if*

$$\max_{t \in \mathcal{R}} \{c_t(a)\} \leq 2T, \tag{2}$$

then $S(a) \leq T$.

Proof. Let $\ell + 2$ be the length of a. We seek to eliminate the variable ℓ from the inequality

$$S(a) = \frac{\sum_{t \in \mathcal{R}} c_t(a)(c_t(a) - 1)/2}{(\ell - 1)} \leq T, \tag{3}$$

to obtain a bound that is independent of the length of the interval a. We do so by substituting the identity $\ell = \sum_{t\in\mathcal{R}} c_t(a)$ into (3) and simplifying to obtain the inequality $\sum_{t\in\mathcal{R}} f(c_t(a)) \geq T$, where

$$f(c) = \frac{(2T+1)c - c^2}{2}.$$

The factorization $(2T+1)c - c^2 - 2T = (c-1)(2T-c)$ implies that $f(c) \geq T$ whenever $1 \leq c \leq 2T$.

Suppose that $c_t(a) \leq 2T$ for all triplets t. Then for any triplet t, either $c_t(a) = 0$ and $f(c_t(a)) = 0$, or $c_t(a) > 0$ and $f(c_t(a)) \geq T$. Since $c_t(a) > 0$ for the leftmost triplet in a, it follows that $\sum_{t\in\mathcal{R}} f(c_t(a)) \geq T$. Therefore, $S(a) \leq T$ whenever $\max_{t\in\mathcal{R}}\{c_t(a)\} \leq 2T$. ■

The bound in Proposition 1 is tight. If a consists of $\lfloor 2T+1 \rfloor$ identical triplets, then $S(a) = \lfloor 2T+1 \rfloor / 2 > T$.

If u is a subsequence of a, then $c_t(u) \leq c_t(a)$ for every t, and so u satisfies (2) whenever a does. In other words, if a satisfies (2), then no subsequence of a has score that exceeds T. Therefore, for every window w_k, we compute the value $L(w_k)$, the number of triplets in the largest suffix of w_k that satisfies (2). We then only need consider suffixes of w_k longer than $L(w_k)+2$ when performing Step 2.

Proposition 1 allows us to ignore those suffixes of the current window that do not have enough occurrences of any triplet. In contrast, the following proposition shows that if $L(w_k)$ is sufficiently long and $S(w_k)$ sufficiently small, then we may skip Step 2 entirely. In practice, many windows satisfy the next inequality.

Proposition 2. *Let w be a window with ℓ triplets so that w has length $\ell+2$. If*

$$S(w) \leq \frac{L(w)T}{\ell - 1}, \tag{4}$$

then no suffix of w has a score that exceeds T.

Proof. For $1 \leq j \leq \ell$, let u_j denote the suffix of w of length $j+2$. From the definition of u_j, it follows that $c_t(u_j) \leq c_t(w)$ for all triplet values t. Therefore, for any $1 \leq j \leq \ell$,

$$\sum_{t\in\mathcal{R}} \frac{c_t(u_j)(c_t(u_j)-1)}{2} \leq \sum_{t\in\mathcal{R}} \frac{c_t(w)(c_t(w)-1)}{2} = S(w)(\ell-1).$$

Suppose then that (4) holds. For $L(w) < j \leq \ell$, it follows that

$$\sum_{t\in\mathcal{R}} \frac{c_t(u_j)(c_t(u_j)-1)}{2} \leq TL(w) \leq T(j-1),$$

and therefore that $S(u_j) \leq T$. If, on the other hand, $1 \leq j \leq L(w)$, then the suffix u_j satisfies (2), and so the bound $S(u_j) \leq T$ also holds. ■

To take advantage of the optimization suggested by Propositions 1 and 2, we use a method for computing the values of $L(w_k)$ and $S(w_k)$ that is more efficient than scanning all of w_k. Moreover, we compute the score of suffixes of length greater than $L(w_k)+2$ without scanning the entire suffix.

For the current window, the optimized SDUST algorithm maintains the following information:

- The number of triplets $L(w_k)$ in the longest window suffix v_k satisfying condition (2) of Proposition 1;
- The triplet counts $c_t(w_k)$ and $c_t(v_k)$ for every triplet t; and
- The *running counts* $r(w_k)$ and $r(v_k)$, where

$$r(a) = \sum_{t\in\mathcal{R}} \frac{c_t(a)(c_t(a)-1)}{2}.$$

Each of these quantities may be efficiently calculated using the corresponding values from the window w_{k-1}. The loops required to do so are presented in detail in the pseudocode provided in Supplemental Information. However, all updating algorithms are based on the principle that if the sequence a is derived from the sequence b by adding or removing a single base from either end of b, then $c_t(a)$ differs from $c_t(b)$ at precisely one value of t. Similarly $r(a)$ can be computed from $r(b)$ by adding single value of $c_t(a)$ or by subtracting a single value of $c_t(b)$ from $r(b)$.

The operations involved in shifting from w_{k-1} to w_k and in searching for the longest window suffix v_k that satisfies (2) are all written in terms of operations that add or remove a single base from a subsequence. The triplet counts $c_t(w_k)$ and $c_t(v_k)$ are maintained as arrays and are computed by updating in place the arrays that represent $c_t(w_{k-1})$ and $c_t(v_{k-1})$.

Step 2 may thus be implemented as follows:

- Compute the values of $c_t(w_k)$, $r(w_k)$, $L(w_k)$, v_k, $c_t(v_k)$, and $r(v_k)$.
- Use $L(w_k)$ and $r(w_k) = (\ell-1)S(w_k)$ to check condition (4). If the condition is satisfied, stop processing the current window w_k.
- If condition (4) is not satisfied, use $r(v_k)$ to efficiently compute the scores of suffixes of length greater than $L(w_k)+2$. Consider each suffix of w_k of length greater than $L(w_k)$ in turn, working from shortest to longest. If a suffix has score greater than T and does not cover a higher-scoring interval already in $\mathcal{P}_k$, add the suffix to $\mathcal{P}_k$.

The loop to test suffixes of length greater than $L(w_k)+2$ for inclusion in $\mathcal{P}_k$ has been implemented so that the entire loop traverses $\mathcal{P}_k$ only once. Detailed pseudocode derived from the C++ implementation is given in Supplemental Information.

6. PERFORMANCE EVALUATION

Several tests were performed to compare SDUST with DUST. The human (*Homo sapiens*) genome build 34 and the fruitfly (*Drosophila melanogaster*) genome were used to perform the tests. The tests address three questions:

1. How many more nucleotides are masked by SDUST than are masked by DUST?
2. How much faster is the SDUST implementation?
3. What fraction of high-quality MegaBLAST (Zhang *et al.*, 1998) matches disappear due to the extra masking?

The results of the tests are summarized in Table 1. The second column of the table shows the increase in the percentage of bases masked by SDUST compared to the number of bases masked by DUST. In all tests, the increase was less than 2%.

Columns 3–5 of Table 1 show the running times of the DUST, SDUST, and percentage of decrease in the running time when SDUST was used. The runs were performed on a Dual Pentium-4 Xeon 3.2-Ghz CPU Linux computer with 512 Kb of L2 cache per CPU and 4 Gb of RAM. Each test was performed three times, and the number in the table is the average over three runs. In all our tests, the SDUST is at least four times faster than DUST.

The effect of SDUST on nucleotide BLAST searches was tested by running MegaBLAST (Zhang *et al.*, 1998) on a set of DNA queries from the human and fruitfly genomes against the DNA genomes of the corresponding organism. The query set for each genome contains 100 sequences that are each 200–300 kbases long. One set of MegaBLAST runs was performed with the queries masked using DUST; another set was performed with the queries masked using SDUST. Masked bases were represented by lower case letters in the query FASTA files.

MegaBLAST was run with the following command line options: `"-UT -Fm -D3 -m8 -e 0.1"`. The `-UT` option signals the desire to retain nucleotides present as lower-case letters in the query as lower case rather than turning them into upper case. The `-Fm` option disables all internal masking (including masking

TABLE 1. SEVERAL TESTS TO COMPARE SDUST AND DUST

	Bases masked	*Running time (sec)*			*Megablast results*
Database	*SDUST (% increase)*	*DUST*	*SDUST*	% decrease	*(D-S)/D*
D. melanogaster	5368015 (+1.67%)	113.79	25.05	77.99%	1/26393
H. sapiens chr1	9962990 (+1.36%)	231.86	52.67	77.28%	3/48559
H. sapiens chr2	10534797 (+1.32%)	245.16	54.87	77.62%	3/52687
H. sapiens chr3	8291266 (+1.41%)	195.81	44.05	77.50%	1/44143
H. sapiens chr4	8612129 (+1.29%)	210.38	43.29	79.42%	1/46831
H. sapiens chr5	7711431 (+1.37%)	197.54	40.63	79.43%	2/43717
H. sapiens chr6	7429406 (+1.37%)	184.34	38.84	78.93%	1/37868
H. sapiens chr7	7231658 (+1.34%)	162.33	36.76	77.35%	1/33972
H. sapiens chr8	6292534 (+1.33%)	156.36	33.03	78.88%	0/30517
H. sapiens chr9	5251123 (+1.33%)	130.81	27.15	79.24%	0/26385
H. sapiens chr10	6086515 (+1.29%)	144.12	30.98	78.50%	0/25012
H. sapiens chr11	5539418 (+1.30%)	133.29	30.04	77.46%	1/29650
H. sapiens chr12	5895050 (+1.35%)	135.87	30.43	77.60%	1/28727
H. sapiens chr13	4377601 (+1.33%)	101.65	22.34	78.02%	0/22167
H. sapiens chr14	3828803 (+1.38%)	92.81	20.22	78.21%	0/19249
H. sapiens chr15	3587821 (+1.35%)	85.92	18.99	77.90%	1/15829
H. sapiens chr16	3946925 (+1.21%)	104.46	19.60	81.24%	0/14161
H. sapiens chr17	3817441 (+1.34%)	102.5	19.93	80.56%	0/13099
H. sapiens chr18	3303535 (+1.34%)	76.56	17.07	77.70%	0/16223
H. sapiens chr19	3368114 (+1.22%)	68.59	14.90	78.28%	1/9078
H. sapiens chr20	2654471 (+1.30%)	62.44	13.88	77.77%	0/11018
H. sapiens chr21	1651742 (+1.24%)	37.12	8.14	78.07%	0/6681
H. sapiens chr22	1651307 (+1.27%)	38.21	8.36	78.12%	0/4261
H. sapiens chrX	7044967 (+1.28%)	167.67	36.96	77.96%	0/47100
H. sapiens chrY	1547697 (+0.87%)	32.91	6.61	79.91%	1/6584

The second column of the table gives the number of bases masked by SDUST and the percentage increase over DUST. In the last column, D [S] is the number of MegaBLAST matches, summed over 100 queries, reported when queries are masked with DUST [SDUST]; the numerator is the number of matches missed due to additional nucleotides masked by SDUST.

by the DUST module) and enables "soft masking" of nucleotides in the query that are in lower case. Soft masking means that masking is only used for initial seed selection but not for the extension of the alignments. The `-e 0.1` evalue threshold was chosen to limit the number of results. The `-D3 -m8` options affect the format of the output but not the alignments found. In addition, we considered only *high quality alignments*, defined as being at least 92 bp long and having an identity percentage of at least 97%. The same definition of high-quality alignments is used in several production applications at NCBI.

The last column of Table 1 presents counts of the number of high-quality alignments found. The denominator shows the total number summed over 100 queries of matches reported when alignments were done with the queries masked with DUST. The numerator shows the number of high-quality MegaBLAST matches reported when alignments were done with the queries masked with DUST but not reported when alignments were done with the queries masked with SDUST. The fraction of missed matches is generally <0.0001, which is negligible in practice.

We give some details about the few matches that disappeared. Most of the query part of the only missing match in the case of *Drosophila* is masked by both DUST and SDUST, but SDUST masks an additional run of seven "T"s, thus blocking the only possible alignment seed from being considered. Below is the corresponding subinterval of the query sequence with GenBank accession AC009206.20 starting at position 85386 and ending at 85544. The extra bases masked by SDUST are in brackets.

```
AAATTAAGGG[ttttttt]GCTTAATTAAACGCAAttttttttataaaatataat
taaaaaatatttatttttacttataaatcaaaaaacaaattaaaaatattaaatat
acaagaaaataaacaacaaatTCCAAGTTTACACACTTTTGAGACTGTCAA
```

In the case of the human genome, all the missing MegaBLAST matches are due to different masking by DUST and SDUST of the following sequence fragment.

```
ACCTGCACATTGTGCACATGTACCC[taaaacttaaagtataataataataaaa
tt]aaaaaaaaa
```

This sequence is the prefix of the sequence used in the Introduction to demonstrate the lack of symmetry of DUST. When its reverse complement is used as a query, DUST masks the same bases as SDUST and the corresponding matches are not reported by MegaBLAST.

7. DISCUSSION

The DUST module for filtering low-complexity sequences has been used in conjunction with BLAST for many years. Blocking low-complexity, biologically uninteresting matches from appearing in BLAST output is essential to making the output of DNA BLAST runs useful. One would like masking of low-complexity sequences to be as fast and as predictable as possible. We presented a modified implementation for the DUST module, called SDUST in this paper, that eliminates two sources of unpredictability in the previous versions and runs several times faster.

The DUST module is not the only available method to identify low-complexity DNA sequences. We mention five other methods that can be used to identify at least some low-complexity DNA sequences.

Two methods are based on local alignment. Claverie and States (1993) proposed aligning the sequence against itself and looking for high-quality alignments in the off-diagonal regions of the dynamic programming graph. Their method is implemented in the program XNU. So far as we know, XNU is widely used for protein sequences but is not widely used for DNA sequences due to their lower information content. One can also use RepeatMasker (Smit *et al.*, 1996) and match the input sequence against a library of low-complexity sequences (Jurka, 2000).

A very different method, proposed by Crochemore and Verin (1999), uses the notion of topological entropy. The method works by counting all different subwords in 512-bases-long overlapping windows. A sequence is considered low-complexity if the fraction of possible subwords it contains is below a predefined threshold. So far as we know, this method has not been used in any large-scale DNA analysis of biological interest.

Finally, one can use methods that search for tandem repeats to identify some classes of low-complexity sequences. Two software packages that search effectively for tandem repeats are Tandem Repeats Finder (Benson, 1999) and STAR (Delgrange and Rivals, 2004). In either case, some parameter tuning would be needed to apply these programs to the masking problem. In this regard, we left the parameters of DUST (window size and score threshold) at the same setting as they were previously.

SDUST is currently used at NCBI in the version of BLAST (Altschul *et al.*, 1997) called the "new BLAST engine," which is currently used for nucleotide searches on NCBI's Web pages. It is also used in Windowmasker, our new tool for masking repetitive and low-complexity sequences (Morgulis *et al.*, 2006). SDUST is available in the NCBI C++ toolkit as the source code for a stand-alone program called "dustmasker." The source code for the entire toolkit is available at: <ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/CURRENT/>.

Once the entire toolkit in installed, the code for dustmasker can be found in subdirectories include/algo/dustmask, src/algo/dustmask, and src/app/dustmasker. File README.build in subdirectory src/app/dustmask contains instructions for building dustmasker in a UNIX environment.

SUPPLEMENTAL INFORMATION

We present pseudocode for the DUST and SDUST algorithms. In the first subsection, we present the SDUST algorithm; the DUST algorithm will be presented in the second subsection. In the pseudocode, the arguments to all functions and procedures are passed by reference, so that assigning a value in the function or procedure changes the value in the caller. Indices into arrays and sequences start at zero. The function

NEW_ARRAY(length, initial_value) creates and returns a new array containing the given number of elements each initialized to the given value. The procedure APPEND(A, e) appends element e to the end of array A; the array will expand, if necessary, to hold the new element.

SDUST implementation

SDUST maintains arrays of length 64 that represent the counts of the number of triplets in a specific subsequence and integer variables that represent the running counts within the same subsequence. The array element $c_w[t]$ represents the value denoted $c_t(w_k)$ in the main text, although the array c_w may not hold the correct counts for the window w_k at intermediate stages in the computation. Similarly, $c_v[t]$, r_v and r_w represent the values denoted $c_t(v)$, $r(v)$ and $r(w)$ in the main text, respectively. The following two procedures ADD_TRIPLET_INFO and REM_TRIPLET_INFO, update the counts and running counts of a sequence when its length is changed by adding or removing one nucleotide from either end.

Algorithm S1. Update triplet counts when a sequence changes.

```
procedure ADD_TRIPLET_INFO(r, c, t)
    r ← r + c[t]
    c[t] ← c[t] + 1
end procedure
procedure REM_TRIPLET_INFO(r, c, t)
    c[t] ← c[t] − 1
    r ← r − c[t]
end procedure
```

The SDUST algorithm counts triplets in windows within a sequence q. Triplets are represented internally as integers in the range 0–63. The function TRIPLET(q, i) returns an integer representing the triplet $q_i q_{i+1} q_{i+2}$. A deque (double-ended queue), represented by the variable w, is used to hold the integer values for all triplets in the current window. The function NEW_DEQUE() creates a new, empty deque. The procedure APPEND_RIGHT(w, t) appends a value to the right end of the deque, and the function POP_LEFT(w) removes and returns an element from the left end. The elements of a deque may be accessed in random order using the same syntax used to access elements of an array.

When the current window is changed, the following procedure updates the deque w, the triplet counts c_w and c_v, and the running counts r_w and r_v. Moreover, it calculates the number of triplets L in the longest suffix of w that satisfies condition (2), the quantity denoted $L(w_k)$ in Section 5.

Algorithm S2. Update data structures when the window is shifted.

```
procedure SHIFT_WINDOW(t, w, T, W, L, r_w, r_v, c_w, c_v)
    if LENGTH(w) ≥ W − 2 then
        s ← POP_LEFT(w)
        REM_TRIPLET_INFO(r_w, c_w, s)
        if L > LENGTH(w) then
            L ← L − 1
            REM_TRIPLET_INFO(r_v, c_v, s)
        end if
    end if
    APPEND_RIGHT(w, t)
    L ← L + 1
    ADD_TRIPLET_INFO(r_w, c_w, t)
    ADD_TRIPLET_INFO(r_v, c_v, t)
    if c_v[t] > 2T then
        repeat
            s ← w[LENGTH(w) − L]
            REM_TRIPLET_INFO(r_v, c_v, s)
```

```
            L ← L − 1
        until s = t
    end if
end procedure
```

In the SDUST algorithm, the list of perfect intervals for the current window is represented by the variable P. In the pseudocode, P is a singly linked list of structures each representing a perfect interval; in the C++ code a doubly-linked list is used. Each structure has four fields: start, finish, score, and next. The start and finish fields are indices into the sequence q, where both endpoints are included in the interval. The next field is used to chain intervals into a list. The list is maintained in sorted order, where it is sorted first by descending order of start and then by ascending order of finish. The value nil represents the empty list. The REVERSE(P) function returns a copy of P in reverse order. Where the REVERSE(P) function appears in pseudocode, the doubly-linked list is simply traversed in reverse order in the C++ code.

When the window w is shifted from one iteration of SDUST to the next, two actions must be taken. First, all perfect intervals that no longer lie in the current window must be removed from P, and the bases contained in those intervals must be masked. Second, the list P must be updated to include suffixes of the current window. The SAVE_MASKED_REGIONS procedure removes the appropriate intervals from P and records the bases to be masked. The array res (short for "results") contains pairs (s, f) of start and end points representing the maximal disjoint regions to be masked. The argument wstart is the position in the complete sequence q where the current window starts.

Algorithm S3. Save the endpoints of regions to be masked.

```
procedure SAVE_MASKED_REGIONS(res, P, wstart)
    P ← REVERSE(P)
    if P ≠ nil and P.start < wstart then
        ℓ ← LENGTH(res)
        if ℓ > 0 then
            (s, f) ← res[ℓ − 1]
            if P.start ≤ f + 1 then
                res[ℓ − 1] = (s, max{P.finish, f})
            else
                APPEND(res, (P.start, P.finish))
            end if
        else
            APPEND(res, (P.start, P.finish))
        end if
        while P ≠ nil and P.start < wstart do
            P ← P.next
        end while
        P ← REVERSE(P)
    end if
end procedure
```

The FIND_PERFECT procedure updates the current list of perfect intervals P to include suffixes of w. On entry, P contains exactly all perfect intervals in w that are not suffixes of w. On exit, P contains all perfect intervals in w.

Algorithm S4. Find perfect intervals that are suffixes of w.

```
procedure FIND_PERFECT(P, w, T, wstart, L, r_v, c_v)
    c ← COPY(c_v); r ← r_v
    perf ← P; previous_perf ← nil
    max_score ← 0
    for i ← LENGTH(w) − L − 1 down to 0 do
        t ← w[i]
```

```
        ADD_TRIPLET_INFO(r, c, t)
        new_score ← r/(LENGTH(w) − i − 1)
        if new_score > T then
            while perf ≠ nil and perf.start ≥ i + wstart do
                max_score ← max{max_score, perf.S}
                previous_perf ← perf
                perf ← perf.next
            end while
            if new_score ≥ max_score then
                max_score ← new_score
                new_perf =
                struct PERFECT {
                    start = i + wstart, finish = LENGTH(w) + 1 + wstart,
                    S = new_score, next = perf
                }
                if previous_perf = nil then
                    P ← new_perf
                else
                    previous_perf.next ← new_perf
                end if
                previous_perf ← new_perf
            end if
        end if
    end for
end procedure
```

Finally, the top-level SDUST algorithm may be defined in terms of the procedures and functions described above. The first **if** test applies the inequality of Proposition 2.

Algorithm S5. Symmetrically mask low-complexity regions in a sequence q, using score threshold T and window size W.

```
function SDUST(q, T, W)
    res ← NEW_ARRAY(length = 0); P ← nil
    r_v ← 0; r_w ← 0
    c_v = NEW_ARRAY(length = 64, initial_value = 0)
    c_w = NEW_ARRAY(length = 64, initial_value = 0)
    w = NEW_DEQUE()
    L ← 0
    for wfinish = 2 to LENGTH(q) − 1 do
        wstart ← max{wfinish − W + 1, 0}
        SAVE_MASKED_REGIONS(res, P, wstart)
        t ← TRIPLET(q, wfinish − 2 )
        SHIFT_WINDOW(t, w, T, W, L, r_w, r_v, c_w, c_v)
        if r_w > L × T then
            FIND_PERFECT(P, w, T, wstart, L, r_v, c_v)
        end if
    end for
    wstart ← max{0, LENGTH(q) − W + 1}
    while P ≠ nil do
        SAVE_MASKED_REGIONS(res, P, wstart)
        wstart ← wstart + 1
    end while
    return res
end function
```

The original DUST implementation

The original DUST algorithm is implemented differently from the SDUST algorithm described above, but many of the data structures are the same. We present pseudocode for the original DUST algorithm, using the same notation we used to describe SDUST. The result returned by the DUST algorithm, represented by the variable res, is subtly different from the return value of SDUST. SDUST returns a maximal disjoint set of regions to be masked, but DUST return an overlapping collection of intervals. Because SDUST returns a smaller data structure, we believe its behavior is preferable.

Algorithm S6. Find the high-scoring prefix of a subwindow whose first and last triplets are istart and ifinish, respectively.

```
function BEST_PREFIX(w, istart, ifinish)
    c_w ← NEW_ARRAY(length = 64, initial_value = 0)
    r_w ← 0
    finish ← istart − 1; max_score ← 0
    for i ← istart to ifinish do
        t ← w[i]
        r_w ← r_w + c_w[t]
        c_w[t] ← c_w[t] + 1
        if i > istart and r_w/(i − istart) > max_score then
            finish ← i
            max_score ← r_w/(i − istart)
        end if
    end for
    return (finish + 2, max_score)
end function
```

Algorithm S7. Mask low-complexity regions in a sequence.

```
function DUST(q, W, T)
    res ← NEW_ARRAY(length = 0)
    w ← NEW_DEQUE()
    for i ← 2 to LENGTH(q) + W − 4 do
        wstart ← max{i − W + 1, 0}
        wfinish ← min{i, LENGTH(q) − 1}
        if i < LENGTH(q) then
            t ← TRIPLET(q, i − 2)
            APPEND_RIGHT(w, t)
        end if
        if wstart > 0 then
            POP_LEFT(w)
        end if
        (limit, max_score) ← BEST_PREFIX(w, 0, LENGTH(w) − 1)
        start ← 0; finish ← limit
        if max_score > T then
            for s ← 1 to limit − 3 do
                (f, new_score) = BEST_PREFIX(w, s, limit − 2)
                if new_score > max_score then
                    max_score ← new_score
                    start ← s; finish ← f
                end if
                APPEND(res, (start + wstart, finish + wstart))
            end for
        end if
    end for
end function
```

ACKNOWLEDGMENTS

We thank David Lipman for encouragement. This research was supported by the Intramural Research Program of the NIH, National Library of Medicine.

REFERENCES

Altschul, S.F., Madden, T.L., Schäffer, A.A., *et al.* 1997. Gapped BLAST and PSI-BLAST—a new generation of protein database search programs. *Nucleic Acids Res.* 25, 3389–3402.

Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., *et al.* 2004. Genbank: Update. *Nucleic Acids Res.* 32, D23–D26.

Benson, G. 1999. Tandem Repeats Finder: A program to analyze DNA sequences. *Nucleic Acids Res.* 27, 573–580.

Claverie, J.-M., and States, D.J. 1993. Information enhancement methods for large scale sequence analysis. *Comput. Chem.* 17, 191–201.

Crochemore, M., and Verin, R. 1999. Zones of low entropy in genomic sequences. *Comput. Chem.* 23, 275–282.

Delgrange, O., and Rivals, E. 2004. STAR: An algorithm to Search for Tandem and Approximate Repeats. *Bioinformatics* 20, 2812–2820.

Hancock, J.M., and Armstrong, J.S. 1994. SIMPLE34: An improved and enhanced implementation for VAX and Sun computers of the SIMPLE algorithm for clustered repetitive motifs in nucleotide sequences. *Comput. Appl. Biosci.* 10, 67–70.

Jurka, J. 2000. Repbase update: A database and an electronic journal of repetitive elements. *Trends Genet.* 16, 418–420.

Morgulis A., Gertz, E.M., Schäffer, A.A., *et al.* 2006. WindowMasker: Window based masker for sequenced genomes. *Bioinformatics* 22, 134–141.

Smit, A.F.A., and Green, P. 1996. RepeatMasker. Available at <http://repeatmasker.genome.washington.edu> and <http://www.repeatmasker.org/webrepeatmaskerhelp.html>.

Zhang, Z., Schwartz, S., Wagner, L., *et al.* 2000. A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.* 7, 203–214.

Address correspondence to:
Richa Agarwala
NCBI/NIH
Bldg. 38A, Rm. 1003N
8600 Rockville Pike
Bethesda, MD 20894

E-mail: richa@helix.nih.gov