BIOINFORMATICS

Vol. 20 no. 18 2004, pages 3363–3369
doi:10.1093/bioinformatics/bth408

Reducing storage requirements for biological sequence comparison

*Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount and James A. Yorke**

Institute for Physical Science and Technology, University of Maryland, College Park, MD 20742-2431, USA

Received on April 16, 2004; revised on July 5, 2004; accepted on July 7, 2004
Advance Access publication July 15, 2004

ABSTRACT

Motivation: Comparison of nucleic acid and protein sequences is a fundamental tool of modern bioinformatics. A dominant method of such string matching is the 'seed-and-extend' approach, in which occurrences of short subsequences called 'seeds' are used to search for potentially longer matches in a large database of sequences. Each such potential match is then checked to see if it extends beyond the seed. To be effective, the seed-and-extend approach needs to catalogue seeds from virtually every substring in the database of search strings. Projects such as mammalian genome assemblies and large-scale protein matching, however, have such large sequence databases that the resulting list of seeds cannot be stored in RAM on a single computer. This significantly slows the matching process.

Results: We present a simple and elegant method in which only a small fraction of seeds, called 'minimizers', needs to be stored. Using minimizers can speed up string-matching computations by a large factor while missing only a small fraction of the matches found using all seeds.

Contact : {yorke,bhunt}@ipst.umd.edu

1 INTRODUCTION

Sequence comparison is a fundamental tool of computational biology, used in applications such as overlap determination in genome sequence assembly (Myers *et al.*, 2000; Batzoglou *et al.*, 2002; Ewing and Green, 1994, http://www.genome.washington.edu or http://www.phrap.org), as well as gene finding and comparison, and protein sequence comparison. The dominant method of sequence comparison, used for example by BLAST (Altschul *et al.*, 1990) is the 'seed and extend' approach (Altschul *et al.*, 1990, 1997; Lipman and Pearson, 1985; Pearson and Lipman, 1988; Zhang *et al.*, 2000), although some methods use seeds without an explicit extend step (Ning *et al.*, 2001). Assume we wish to find similar subsequences of two strings T_1 and T_2. In this approach, we first choose a set of short subsequences called 'seeds' from each of T_1 and T_2; then, for each seed common to both, we align T_1 and T_2 so that the seeds align, and check to see if the match 'extends' beyond the seeds.

Given a set of N strings $\{T_i\}_{i=1}^{N}$ to compare with each other pairwise, the first step in the seed-and-extend approach is to choose the set of seeds $\mathbf{S}_i = \{s_{i1}, s_{i2}, \ldots\}$ that are to represent each string T_i. We use seeds that are contiguous k-letter substrings called k-mers. A common approach to finding k-mers that are contained in more than one string is to store every k-mer that appears in each string T_i for all i. For example, the string `2310343` depicted in Figure 1 contains, in order of position, the 3mers `231`, `310`, `103`, `034` and `343`. Given a database of all k-mers contained in a set of strings, we can sort the list by k-mer. This conveniently puts identical k-mers side-by-side, giving us all the required k-mer seeds at which to apply the extend algorithm in an effort to find longer matches. We call this ability to recognize matches as soon as the database is sorted the *collection criterion*.

However, the number of k-mer entries and the space required to store the entire list of k-mers can be staggering. If $|T_i|$ represents the length of the string T_i, then the number of k-mers in T_i is $|T_i| - k + 1$, or roughly $|T_i|$ assuming $k \ll |T_i|$. Furthermore, each k-mer entry in the database requires k letters of storage for the k-mer string s, plus the pair of integers (i, p) identifying the string T_i and the position p within T_i at which s appears. We call (s, i, p) a *k-mer triple*. If the total number of letters in all the sequences in the database is L, then the database size scales roughly as kL. As an example of the size of such a database, the genome sequence assembly of *Rattus norvegicus* uses about 33×10^6 sequences called reads, with an average of about 600 letters each, giving a total of 2×10^{10} k-mer entries in the database, each of size k. A typical k in this application is 20, giving a total database size of 4×10^{11} letters! Even utilizing compressed storage (we can store 4 letters per byte since genomic sequences require only 2 bits per letter), we require 5 bytes to store each k-mer, and 5 bytes to store (i, p). This gives a total of 200 GB for the entire k-mer database.

*To whom correspondence should be addressed.

Position	1	2	3	4	5	6	7		1	2	3	4	5	6	7	8	9	10	11	12
Sequence	2	3	1	0	3	4	3		4	2	6	4	7	2	8	1	4	7	5	1
k-mers	2	3	1						4	2	6	4	7	2	8					
with		3	1	0						**2**	**6**	**4**	**7**	**2**	**8**	**1**				
minimizer			1	0	3						6	4	7	2	8	1	4			
in				**0**	**3**	**4**						4	7	2	8	1	4	7		
bold					3	4	3						7	2	8	1	4	7	5	
	(a)								(b)					2	8	1	4	7	5	1

Fig. 1. Illustration of all k-mers in two windows of sequence as well as their minimizers. The sequence in the window and the position within the window are listed in the first two rows. The adjacent k-mers are listed in the rows below. The minimizer is highlighted in bold. Note that w adjacent k-mers correspond to a window of $l = w + k - 1$ letters. (**a**) Choosing the (5,3)-minimizer from 5 adjacent 3mers ($w = 5, k = 3, l = 7$). (**b**) Choosing the (6,7)-minimizer from 6 adjacent 7mers ($w = 6, k = 7, l = 12$).

2 MINIMIZERS

To reduce the storage space simply requires storing fewer k-mers, but which ones? One could store, for example, every k-th k-mer, so that each letter is covered exactly once. However, in that case two strings T_i and T_j with long identical subsequences that start at positions p_i and p_j need not have a stored k-mer in common unless $p_i - p_j$ is a multiple of k. Thus, the database would not satisfy the collection criterion, in the sense that sorting it by k-mer would yield seeds for only a small fraction of matching pairs (T_i, T_j). To find most of the matches, one would have to make a second pass through the strings and compare every k-mer to the database. In a BLAST-like scenario, a second pass is not necessary, because the goal is to find matches of other strings T to the strings that formed the database. However, the procedure would still be more efficient if we could compare only a fraction of the k-mers in T to the database.

Our method uniquely chooses a representative k-mer from a group of adjacent k-mers in such a way that different strings T_i and T_j choose the same representative if they share a long enough subsequence. The method allows us to select from each T_i a set of special k-mers (to be used as seeds) that we call minimizers. We choose them so that only a small fraction of the possible k-mers in a given T_i are *minimizers*, and so that they have the following property:

Property 1. If two strings have a significant exact match, then at least one of the minimizers chosen from one will also be chosen from the other.

2.1 Interior minimizers

As a first step in choosing minimizers, we select an ordering for the set of all k-mers. For strings of letters, one convenient ordering is simply lexicographic, so that, for example, AAAA is the 'smallest' possible 4mer. We defer discussion of orderings to Section 2.4. For now, our examples will use strings of digits with numerical ordering for illustrative purposes. Note that, despite all the examples in this paper, the mapping need not map letters to digits; it simply needs to apply an ordering to all the possible k-mers, and this ordering must be the same for all sequences being processed for minimizers.

Referring to Figure 1, a set of w consecutive k-mers covers a string of exactly $w+k-1$ letters, where 'consecutive' means that each k-mer is shifted by one letter from the previous one. To find a minimizer, we examine w consecutive k-mers and select the smallest, in the sense of our chosen ordering. In the case of a tie, each of the smallest k-mers is a minimizer. We call w the window size.

We say that a k-mer triple (s, i, p) is a (w, k)-minimizer for the string T_i if it is a minimizer for some window of w consecutive k-mers containing it. For simplicity we often refer to a (w, k)-minimizer simply as a minimizer. Refer to the example in Fig. 2, where we have $w = 4$ and $k = 3$. There are five (4,3)-minimizers for the string `231032101233101`, namely `032`, `012`, `123` and `101`. Since there are a total of 13 3mers in the string, having only 4 minimizers as seeds gives substantial space savings over using all 13 3mers. In practice we have found space savings of a factor of 10 using $w = k = 20$, and in general the space savings is about a factor of $2/(w + 1)$; see Section 3 for a heuristic explanation of this factor.

We immediately have the following formalization of Property 1:

Property 1′. If two strings have a substring of length $w+k-1$ in common, then they have a (w, k)-minimizer in common.

The common substring implies a common window of w consecutive k-mers, which generate the same minimizer for each string.

Not all letters (digits) in Figure 2 are contained in minimizers: positions 1–3, 7 and 12 are not covered. Although it may not be required for every letter to be covered by a minimizer, it may be desirable for some applications. Gaps between minimizers are caused when the minimizers of two adjacent windows are more than k positions apart, as in the case of the windows starting in positions 4 and 5 in Figure 2: the minimizer of the window in position 4 covers positions 4–6,

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1
	2	3	1	**0**	**3**	**2**									
		3	1	**0**	**3**	**2**	1								
window			1	**0**	**3**	**2**	1	0							
with				**0**	**3**	**2**	1	0	1						
minimizer					3	2	1	**0**	**1**	**2**					
in						2	1	**0**	**1**	**2**	3				
BOLD							1	**0**	**1**	**2**	3	3			
								0	**1**	**2**	3	3	1		
									1	**2**	**3**	3	1	0	
										2	3	3	**1**	**0**	**1**

Fig. 2. Example of choosing the set of all (4,3)-minimizers in a string, i.e. choosing the smallest 3mer from every 4 adjacent 3mers. Note that in contrast to Figure 1, in this figure each row represents an entire window, with the window's minimizer highlighted in bold. Successive rows depict adjacent windows. As we see, adjacent windows often share the same minimizer. This is the fundamental reason why using minimizers (rather than all k-mers) as seeds reduces storage requirements.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1
	2	3	**1**	**0**	**3**										
		3	1	**0**	**3**	**2**									
			1	**0**	**3**	**2**	1								
				0	**3**	**2**	1	0							
					3	2	**1**	**0**	**1**						
						2	1	**0**	**1**	**2**					
							1	**0**	**1**	**2**	3				
								0	**1**	**2**	3	3			
									1	**2**	**3**	3	1		
										2	**3**	**3**	1	0	
											3	3	**1**	**0**	**1**

Fig. 3. (w,k)-minimizers with $w = k = 3$ for the same string as Figure 2; $w \leq k$ guarantees that every letter is covered by a minimizer except at most $w-1$ letters at the ends.

while the minimizer for the next window covers positions 8–10, leaving position 7 covered by no minimizer. However, note that the minimizers of two adjacent windows of size w can differ in their starting positions by at most w. Thus, gaps can be at most $w-k$ in size, so setting $w \leq k$ ensures no gaps occur between minimizers. Then, all letters are covered except at most $w-1$ at each end of the string (Fig. 3). On the other hand, if $w \gg k$ then minimizers are sparse in the string.

For example, suppose we are using 20mers and the window size is 20. If X is a string of length 400, then it has at least 19 minimizers: the first minimizer has position at most 20, the second at most 40, etc. Similarly, if two strings have an exact match of 400 letters, then they must have at least 19 minimizers in common.

2.2 End-minimizers

Having $w \leq k$ guarantees that no gaps appear between adjacent minimizers, but it still allows some (at most $w-1$) letters at each end of the string to be outside any minimizers. This brings up a related question. Suppose two strings match each other on their ends such that they can be aligned together to form a longer string; in this case we say that the strings overlap. (Strictly speaking only one of the strings needs to match on its end, if for example it is a substring with both ends matching the interior of a larger string.) If the match is less than $w+k-1$ letters, then it is possible for the strings to have no (w,k)-minimizer in common even if there are no gaps between minimizers. This problem is easily fixed (Fig. 4) by the introduction of end-minimizers. A (u,k)-end-minimizer is a (u,k)-minimizer chosen from a window of size u which is anchored to one end of the string, and the set of k-end-minimizers are comprised of all such (u,k)-end-minimizers for u from 1 up to some maximum window size v.

End-minimizers are ideal for matching the ends of strings, and satisfy the following property:

Property 2. If the ends of two strings have an exact overlap of at least k letters and at most $k+v-1$ letters, then they share at least one k-end-minimizer.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1	1
	2	**3**	**1**													
	2	**3**	**1**	0												
	2	3	**1**	**0**	**3**											
	2	3	1	**0**	**3**	**2**										
	2	3	1	**0**	**3**	**2**	1									
	2	3	1	**0**	**3**	**2**	1	0								
	2	3	1	**0**	**3**	**2**	1	0	1							
	2	3	1	0	3	2	1	**0**	**1**	**2**						
	2	3	1	0	3	2	1	**0**	**1**	**2**	3					
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3				
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1			
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1	0		
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1	0	1	
	2	3	1	0	3	2	1	0	1	2	3	3	1	**0**	**1**	**1**

Fig. 4. k-end-minimizers (for $k = 3$) for the left end of a string. We choose the (u, k)-minimizer for every window of length u that is anchored to the left end of the string, for $u = 1, 2, \ldots, l - k + 1$, where l is at most the length of the string.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1
	2	**3**	**1**												
	2	**3**	**1**	0											
												3	**1**	**0**	**1**
													1	**0**	**1**

Fig. 5. End-minimizers for the same string as in Figure 3. Including both these minimizers and the ones from Figure 3, we are guaranteed to cover every base with at least one minimizer.

However, since end-minimizers become sparse towards the interior of a string, two strings with a long but not quite exact overlap may not have an end-minimizer in common.

2.3 A mixed strategy

If we combine (w, k)-minimizers of a string with (u, k)-end-minimizers for $u = 1, \ldots, w - 1$ at both ends of the string, then if $w \leq k$, every base in a string will be covered with some minimizer, and furthermore the ends of strings will be well covered by minimizers, increasing the likelihood of finding low-fidelity matches on the ends of strings. Also, Properties $1'$ and 2 imply that two strings with an exact overlap of at least $2k$ bases have a minimizer in common. The end-minimizers for $u = 1, \ldots, w-1$ for the same string as Figure 2 are shown in Figure 5.

2.4 Orderings

We now briefly discuss the effect of different orderings in determining the minimizers for DNA sequence data. Similar considerations may apply to other types of strings.

Although for illustrative purposes we use lexicographic ordering in our examples, this has the following undesirable effect. If a string contains many consecutive zeros (or As in the case of genomic data), then several consecutive k-mers may be minimizers. While this is not a major problem, it counteracts our goal of sampling a fraction of the k-mers. One can mitigate this effect by choosing an ordering in which the letters that occur least frequently are deemed minimal, and/or by changing the ordering from one letter to the next. In DNA sequences, the letters C and G often occur less frequently than A and T. We assign the values 0, 1, 2, 3 to C, A, T, G, respectively, for the odd numbered bases of k-mers, and reverse the ordering for even numbered bases. This tends to start minimizers with the valuable (in the sense of the significance of a match) letters C and G, and makes the minimum k-mer CGCGCG . . . . There are many other possibilities. For example, we could first order k-mers by the number of Cs and Gs and then choose a minimizer from a restricted set containing more of these letters; or, we could demand that a minimizer has as many distinct bases as possible in its first four bases (preferably all four bases different). In general, we want to devise our ordering to increase the chance of rare k-mers being minimizers, thus increasing the statistical significance of matching minimizers.

These considerations are perhaps most important in the case that $w \gg k$; when minimizers do not cover all the letters

in a string, it is especially important that they cover 'valuable' substrings. We remark that by avoiding strings such as AAAA . . . , we avoid regions in DNA that are particularly prone to sequencing errors.

Finally, for DNA sequences we are also interested in matches between one string and the reverse complement of another string. Thus in choosing seeds, we identify each k-mer with its reverse complement. Then we choose the minimizer of each window W to be the smaller of the two minimizers from W and its reverse complement.

3 RESULTS AND DISCUSSION

As the window from which a minimizer is chosen slides along a random string, a new minimizer occurs about once every half-window width. This can be seen heuristically as follows. There are only two cases in which a minimizer changes as we shift a window to the right: either the minimizer was at the left end of the old window and is 'lost' (as when shifting from position 4 to position 5 in Fig. 3), or the new k-mer that appears on the right is smaller than the existing minimizer (as when shifting from position 5 to position 6 in Fig. 3). Now, consider two adjacent windows, covering $w + 1$ adjacent k-mers, and assume that every k-mer has an equal probability of being a minimizer. Then the k-mers at position 1 and $w + 1$ each have equal probability of $1/(w+1)$ of being minimizers. Thus, the probability that the minimizers of the two adjacent windows differ is $2/(w + 1)$, and hence on average, about a fraction $2/(w + 1)$ of all k-mers are (w, k)-minimizers, independent of k. (Owing to correlations between adjacent k-mers, our assumption that a k-mer at the end of a window is just as likely as any other to be a minimizer is not quite right. In our tests on random sequences and DNA sequences, the actual proportion of k-mers that are minimizers can be a few percent above $2/(w+1)$. The effect decreases substantially as the size of the alphabet increases. For proteins, which have an alphabet of size 20, the proportion is $2/(w + 1)$ to at least 4 significant digits over long random strings.) This is also the fraction by which the seed database is reduced when minimizers are used as seeds rather than all k-mers, so that larger values of w give larger space savings. If it is desired that every base be covered by a minimizer, then choosing $w = k$ gives the best space savings that satisfy these constraints. For example, in the case of the genome assembly of *R.norvegicus*, we used $k = w = 20$, giving a space savings of about a factor of 10, while suffering no detectable loss in quality of overlap detection.

When not using minimizers, if the RAM of one computer is too small to store the entire k-mer database, the sequence data can be distributed in batches across a network, as was done at Celera during the overlap determination phase of assembling the human genome (Venter *et al*., 2001). However, bringing the appropriate seeds together has a cost that is quadratic in the number of batches, so the distributed database fails the collection criterion. Since overlap determination consumed the vast majority of CPU resources and real time during the draft assembly of the human genome, using minimizers would have sped up the process considerably.

The use of minimizers results in the ability to attack the largest existing genome sequence assembly problems on a single desktop computer, when they could previously only be run on a cluster or supercomputer. For example, the 200 GB k-mer database of the *R.norvegicus* genome mentioned in the introduction is now a 20 GB minimizer database. Although currently there exist computers with 20 GB of RAM, they are expensive. Another important property of minimizers is that the list of minimizer triples (s, i, p) can be sorted by minimizer on disk, with sufficiently good locality of reference that the sort process is not I/O bound. Furthermore, once sorted, identical minimizers are conveniently placed side-by-side, satisfying the collection criterion and facilitating easy running of the extend algorithm against all potential matches, again without the seed-and-extend process being I/O bound. This allows us to compute the read overlaps for *R.norvegicus* on a single desktop computer in about 3 days. In comparison, Celera took about 400 CPU days on a cluster (Venter *et al*., 2001). The speed of our procedure also permits us to do high-quality, multiread-comparison-based error correction (since the collection criterion holds) and then repeat the entire string matching procedure on error-corrected strings, significantly improving the quality of the overlap database and of the subsequent genome assembly (M. Roberts, W. Hayes, C. Ustum, B. Hunt, J. Yorke and A. Zamin, manuscript in preparation). We have also used minimizers successfully in assembling *Drosophila melanogaster* [M. Roberts, B. Hunt, J. Yorke, R. Bolanos and A. Delcher (submitted for publication)].

Although in this paper we focus on k-mers as seeds, the idea could easily be extended to seeds with gaps, significantly reducing the storage required by methods such as those used by MEGABLAST (Zhang *et al*., 2000).

To test the reliability and speed of using minimizers, we used a faux dataset created by computationally shattering 100 MB (i.e. 10^8 letters) of finished *Caenorhabditis elegans* genome sequence into faux reads. There were a total of 1 065 846 reads with lengths distributed approximately normally with a mean of 537 and a standard deviation of about 90, giving 5.7-fold coverage of the genome. Quality values for the bases were taken from quality values for actual reads of the human genome. Base errors were then artificially inserted according to probabilities dictated by the quality values. We then computed overlaps between reads using minimizers with various window sizes as seeds. The results are in Figure 6. We first describe the 'No Sym.' case, listed in the first five columns. We see that with $w = 1, k = 20$, over 99.5% of the true overlaps can be found, but the number of false positive (spurious) overlaps is also quite large. (The large number of false positives is due to repeat regions in *C.elegans*, giving false matches that are locally indistinguishable from true matches.)

		No Sym.			w/ Sym.		
w	k	T_{ratio}	F/T	Run time	T_{ratio}	F/T	Run time
1	20	99.528%	1.79	25:26	99.993%	2.61	35:58
3	20	99.379%	1.64	18:26	99.992%	2.41	27:48
8	20	98.931%	1.37	11:47	99.987%	2.06	18:45
20	20	97.467%	1.07	7:09	99.973%	1.66	12:26
1	28	97.404%	1.24	21:43			
1	30	97.669%	1.30	22:49			

Fig. 6. Testing the speed and effectiveness of minimizers using a dataset for which all true matches are known. The first column lists the window size w. Note that $w = 1$ means 'use every k-mer'. The second column lists the size of the k-mers used to seed matches. Next are two groups of three columns, without and with 'Symmetrizer' (explained in the text), respectively. Within each group, the first column T_{ratio}, is the percentage of true matches with at least 40 letters of overlap that were found for the given value of m; the second column, F/T, is the 'false to true ratio', i.e. the ratio of false-positive matches to true-positive matches; the third column lists the run time to compute all matches in hours and minutes on a dual-processor Linux computer.

As w is increased to 20, the fraction of true overlaps found drops to about 97.5%, but the number of false positives drops by a much larger factor. Comparing to the last two rows, we see that using $w = 20$ is comparable to using all k-mers of size 28 or 30 in terms of T_{ratio}, while attaining a lower false-positive rate. In the last three columns we have improved the T_{ratio} to well beyond 99.9% by adding a 'Symmetrizer' step. Symmetrizer was introduced in M. Roberts, B. Hunt, J. Yorke, R. Bolanos and A. Delcher (submitted for publication), and finds the vast majority of missing overlaps. It simply notes that if read X plausibly overlaps reads Y and Z and the offsets of Y and Z relative to X suggest that Y and Z overlap, then Y and Z are checked for overlap. As can be seen, applying Symmetrizer after using minimizers with $w = k = 20$ finds virtually all missing overlaps, while having a total runtime about half that of using $w = 1, k = 30$ and using about 1/10th the memory. (With $w = 20$, the runtime is dominated by the extension part of the algorithm, which is why the speedup is significantly less than a factor of 10.)

As with any method of choosing seeds, the parameters k and w that determine our minimizers are subject to a trade-off between specificity (the proportion of seeds that are indicative of longer matches) and sensitivity (the proportion of the desired matches that are represented by a seed). Increasing k increases the specificity and decreases the sensitivity. The results above indicate that, not surprisingly, increasing w also decreases the sensitivity. They also indicate that (w, k)-minimizers achieve similar sensitivity to using all $(k + w/2)$-mers. We conjecture that this is true more generally, for the following reason. Recall that a (w, k)-minimizer occurs on average every $(w + 1)/2$ letters along a string. If minimizers occurred exactly every $(w + 1)/2$ letters, then every substring of length $k + (w - 1)/2$ would contain a minimizer. Thus, if two strings have a $(k + w/2)$-mer in common, then they are likely to have a (w, k)-minimizer in common. Whether the specificity of (w, k)-minimizers is comparable to that of $(k + w/2)$-mers depends on how large k is relative to the size of the string database, as we now discuss.

Consider a string database of length L. The expected frequency of a given base-b k-mer s is every b^k places ($b = 4$ for DNA and $b = 20$ for proteins), for a grand total of about L/b^k matches. That is, we expect a given base-b k-mer s to occur L/b^k times in a random string of length L. If k is chosen large enough that $L/b^k \ll 1$, then if s occurs twice in L, the match is unlikely to have occurred at random. Assuming further that non-random matches tend to be long compared to k, the specificity of k-mers is then close to 1, and increasing k further does not improve it significantly. For example, in computing overlaps for reads from a mammalian-sized genome, we have $L \approx 10^{10}$ (corresponding to multiple coverage of a gigabase-sized genome), and $b = 4$. We choose $k = 20$ in this case since $L/b^k \approx 0.01$. Furthermore, we store only those k-mers that actually appear in the read database (rather than an index with all b^k of them, most of which do not appear in the read database). Using (w, k)-minimizers instead of all k-mers reduces the number of k-mers we store by a factor of $2/(w + 1)$, as described above.

On the other hand, if k is chosen so that L/b^k is greater than 1, then a typical k-mer will occur multiple times in a random string of length L. Choosing such a small value of k will result in low specificity, but may be necessary to achieve acceptable sensitivity when one is looking for low-fidelity matches, such as those between the genomes of different species. In this case, the value of k is limited by the expected size of exact matches within longer matches of the desired minimum fidelity. For example, suppose that again $L \approx 10^{10}$ and $b = 4$, but that we want to use seeds of at most 15 letters. Based on the discussion above, we might expect similar sensitivity from using $(10, 10)$-minimizers as from using all 15-mers. However, the specificity of $(10, 10)$-minimizers will

be much worse. For $k = 15$, we have $L/b^k \approx 10$, while for $k = 10$, we have $L/b^k \approx 10,000$. Even though a given 10-mer may be a minimizer only a fraction of the times it appears in the string database, using $(10, 10)$-minimizers will yield vastly more spurious seeds than using 15-mers. In this case, since we expect most 15-mers to appear in the string database, it may be most efficient to create a lookup table of every possible 15-mer, along with the list of places that that 15-mer occurs. Thus, minimizers may not be useful in the case of searching for small matches in a large database, or when the fidelity of long matches is low. On the other hand, if one is interested only in finding longer, high-fidelity matches, then large k-mers and thus minimizers, can be used to great effect.

In summary, compared to using every k-mer, using minimizers as seeds for large-scale, high-fidelity seed-and-extend string matching problems can significantly reduce storage requirements. It can also significantly reduce CPU requirements by restricting the number of seeds that need to be considered, and by satisfying the collection criterion. Finally, can achieve these gains without significant loss in sensitivity.

ACKNOWLEDGEMENTS

This material is based on work supported by National Science Foundation Grants 0104087, 0312360, and 0114792, as well as The National Institutes of Health grant 1R01HG0294501.

REFERENCES

Altschul,S.F. Gish,W., Miller,W., Myers,E.W. and Lipman,D.J. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Altschul,S. Madden,T.L., Schaffer,A.A., Zhang,J., Zhang,Z., Miller,W., and Lipman,D.J. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Batzoglou,S., Jaffe,D.B., Stanley,K., Butler,J., Gnerre,S., Mauceli,E., Berger,B., Mesirov,J.P. and Lander,E.S. (2002) ARACHNE: A whole genome shotgun assembler. *Genome Res.*, **12**, 177–189.

Lipman,D. and Pearson,W. (1985) Rapid and sensitive protein similarity searches. *Science*, **227**, 1435–1441.

Myers,E., Sutton,G., Delecher,A.L., Dene,I.M., Fasulo,D.P., Flanigan,M.J., Kravitz,S.A., Mobarry,C.M., Reinert,K.H., Remington,K.A. *et al*. (2000) A whole-genome assembly of *Drosophila*. *Science*, **287**, 2196–2204.

Ning,Z., Cox,A.J. and Mullikin,J.C. (2001) SSAHA: a fast search method for large dna databases. *Genome Res.*, **11**, 1725–1729.

Pearson,W. and Lipman,D. (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci., USA*, **85**, 2444–2448.

Venter,J.C., Adams,M.D., Myers,E.W., Li,P.W., Mural,R.J., Sutton,G.G., Smith,H.O., Yandell,M., Evans,C.A., Holt,R.A. *et al*. (2001) The sequence of the human genome. *Science*, **291**, 1304–1351.

Zhang,Z., Schwartz,S., Wagner,L. and Miller,W. (2000) A greedy algorithm for aligning DNA sequences. *J. Comp. Biol.*, **7**, 203–214.