Reducing storage requirements for biological sequence comparison *

Michael Roberts, Wayne Hayes, Brian Hunt, Steve Mount, James Yorke †
Institute for Physical Science and Technology
University of Maryland
College Park, MD 20742-2431

DRAFT of November 20, 2003

PNAS Classification: Biophysics

Corresponding Author:
whayes@ipst.umd.edu — (e-mail **MUCH** preferred)

Wayne Hayes
IPST Building 224
University of Maryland
College Park, MD 20742.
Phone: 240-893-1612
FAX: 301-314-9363

8 pages, 6 figures, 0 tables, 160 words in abstract, 37000 characters in paper.

*We thank NIH and NSF for support.
†For questions or comments, please write to bhunt@ipst.umd.edu and yorke@ipst.umd.edu.

Abstract

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 find 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. 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 small fraction of the matches found using all seeds.

1 Introduction

Sequence comparison is a fundamental tool of computational biology, used in applications such as overlap determination in genome sequence assembly [3, 4, 5], as well as gene finding and comparison, and protein sequence comparison. The dominant method of sequence comparison, used for example by BLAST [1] and the Celera assembler [3] is the "seed and extend" approach [1, 2, 6, 9, 12, 7], although some methods use seeds without an explicit extend step [8]. Our particular application was inspired by sequence assembly, although we believe the ideas may be more widely applicable. Assume we wish to compare two strings T_1 and T_2. In this approach, we first choose a set of short sub-sequences 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 overlap "extends" beyond the seeds. More generally, we may be interested in many-to-many comparisons, finding sequences common to many strings $\{T_i\}_{i=1}^{N}$.

Given a set of N strings $\{T_i\}_{i=1}^{N}$, the first step in the seed-and-extend approach is to choose the set of seeds $\mathbf{S}_i = \{s_{i1}, s_{i2}, ...\}$ that are to represent each string T_i. We use seeds that are contiguous k-letter sub-strings called **k-mers**. A common approach to finding common k-mers between strings 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 3-mers `231`, `310`, `103`, `034`, and `343`. Given a database of all k-mer triples 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. This property—conveniently placing matching k-mers side-by-side for easy application of the extension algorithm—we call the *collection criterion*. Since *any* seed-and-extend algorithm must eventually collect matching k-mers together, it is obviously important for this criterion to be easily and efficiently satisfied.

The number of k-mer entries and the space required to store the entire list of k-mers can be staggering, however. 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, plus the pair of integers (i, p) identifying the string T_i and the position p within T_i at which the k-mer appears. We call (T_i, i, p) a **k-mer triple**. If the total number of letters in all the sequences in the database is L (which is already a large number), 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 gigabytes for the entire k-mer database. Although some currently available methods use algorithms that require only half of this storage [7], our method uses 1/10th the above storage, and can be easily tuned to use an arbitrarily small amount of memory (with an accompanying loss of sensitivity).

2 Minimizers

2.1 Interior 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. But this does not work well in general because it is possible for two sequences to share a sub-sequence much longer than k

letters, but not share a k-mer in the database. Matches with a target T_i could still be found in the database by searching for each of k adjacent k-mers from T_i, since if T_i and T_j have an exact match of $2k-1$ letters then *one* of the matching k-mers from T_j is in the database. However, this could require as many as k searches of the database *for each target*, which is expensive. Furthermore, this method does not place matches side-by-side, thus failing the collection criterion. In contrast, our method allows *all* matches of length $2k-1$ to be found by sorting the database *once*[1], simultaneously satisfying the collection criterion.

Our method uniquely chooses a *representative* k-mer from a group of adjacent k-mers in such a way that different T_i's choose the same representative if they share a long enough sub-sequence. 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:

(P1) If two strings have significant exact overlap, then at least one of the minimizers chosen from one will also be chosen from the other.

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 4-mer. We defer discussion of orderings to section 2.4. For now, our examples will use strings of digits with digit-wise lexicographic 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.

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

Figure 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 3-mers ($w = 5, k = 3, l = 7$). (b) Choosing the (6,7)-minimizer from 6 adjacent 7-mers ($w = 6, k = 7, l = 12$).

Referring to Figure 1, a set of w consecutive k-mers covers a string of exactly $w+k-1$ letters, where "consecutive" means 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. We call w the **window size**.

We define the set of (w, k)-**minimizers** for a string T_i to be those k-mers in T_i that are minimizers across all w-sized windows in T_i. For simplicity we often refer to a (w, k)-minimizer simply as a minimizer. Refer to the example in Figure 2, where we have $w = 4$ and $k = 3$. There are five (4,3)-minimizers for the string `231032101233101`, namely `103`, `032`, `012`, `123`, and `101`.

[1]Using $w = k$ as defined later.

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**

Figure 2: Example of choosing the set of all (4,3)-minimizers in a string, *ie.,* choosing the smallest 3-mer from every 4 adjacent 3-mers. 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.

Since there are a total of thirteen 3-mers in the string, having only five minimizers as seeds gives substantial space savings over using all thirteen 3-mers. 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 $2/(w+1)$.

Not all bases in Figure 2 are contained in minimizers: positions 1–3, 7, and 12 are not covered. Although it may not be required for every base 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, 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 bases are covered except at most $w - 1$ at each end of the string (see Figure 3). More generally, $w \leq k$ guarantees no gaps, whereas if $w \gg k$ then minimizers are sparse in the string.

Formally, we make the following assertion, which is a formalization of Property P1:

> Claim 1: If two strings share an error-free string of length $(w + k - 1)$ (that is, if they share a $(w + k - 1)$-mer), then they also share a (w, k)-minimizer.

For example, suppose we are using 20-mers and the window size is 20. Then two strings sharing a 39-mer will have a $(20, 20)$-minimizer in common. If instead 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 error-free overlap of 400 bases 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

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**

Figure 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.

up a related question: suppose two strings match each other on the ends such that they can be concatenated together to form a longer 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 (see Figure 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.

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

> Claim 2: if the ends of two strings have an error-free overlap of k letters or more, then they share at least one k-end-minimizer.

However, since end-minimizers become sparse towards the interior of a string, a single letter error in an end-minimizer can potentially destroy a short match.

2.3 A mixed strategy

If we combine (w, k)-minimizers using $w \leq k$ with (u, k)-end-minimizers for $u = 1, \ldots, w-1$ at both ends of the string, then every base in a string can be covered with some minimizer, and furthermore the ends of strings will have high coverage with both minimizers and end-minimizers, increasing the likelyhood of catching matches on the ends of even error-prone strings. The end-minimizers for $u = 1, \ldots, w - 1$ for the same string as Figure 2 are shown in Figure 5. Hence if $w \leq k$ and we include end-minimizers, Property P1 holds, and furthermore two strings with error-free overlap of many more than $w + k - 1$ bases will have several common minimizers.

2.4 Orderings

We now briefly discuss our experiences in ordering the list of all possible k-mers in genomic data. For ordering non-genetic k-mers, a similar amount of care may be necessary to achieve good results.

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**

Figure 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**

Figure 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.

Although for illustrative purposes we use lexicographic ordering in our examples, this is in fact a bad ordering for genomic data because using it results in minimizers that tend to start with strings of `A`'s, but strings of identical letters tend to have high error rates because they are difficult to sequence correctly. We would, of course, like to avoid choosing seeds from such low-quality data. Furthermore, strings of identical letters sometimes occur more frequently in real genomes than randomness would suggest, leading (even in the absence of errors) to many strings having common seeds that are not a part of longer matches.

Genomic tandem repeats have similar problems to strings of identical letters (frequent occurrence with high sequencing error rates), so it is also desirable to choose an ordering that avoids seeds containing tandem repeats. Finally, the letters `C` and `G` tend to occur less frequently than `A` and `T` in the genomes we have studied. This means that they are less likely to occur in a random k-mer and thus are good choices to include in minimizers. To achieve all these desired properties, 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 avoids strings of single letters, avoids odd-length tandem repeats, and tends to start minimizers with the valuable letters `C` and `G`. There are many other possibilities. For example, we could first order k-mers by the *number* of `C`s and `G`s 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 4 bases (preferably all four bases different), which would tend to avoid mono-, di-, and tri-nucleotides, which also appear frequently and lead to spurious matches. 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.

The reverse complement is handled simply by choosing the minimizer of each window W to be the smaller of the two minimizers from W and its reverse complement.

3 Results and discussion

Assuming $k \gg 1$ and $w \gg 1$, then as the window 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 Figure 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 Figure 3). Now, consider two adjacent windows, covering $w+1$ adjacent k-mers, and assume every k-mer has an equal probability of being a minimizer.[2] Then the k-mers at position 1 and $w+1$ each have equal probability of $1/(w+1)$ of being minimizers. Thus, on average, about a fraction $2/(w+1)$ of all k-mers are (w,k)-minimizers, independent of k. This is also the fraction of storage required to use minimizers as seeds rather than all k-mers. For example, in the case of the genome assembly of *Rattus norvegicus* [10], we used $k = w = 20$, giving a space savings of a factor of 10 while suffering no detectable loss in sensitivity or quality of overlap detection.

When not using minimizers, if the RAM of one computer is too small to store the entire k-mer database, it can be distributed across a network as was done at Celera during the overlap determination phase for the Human genome. However, this incurs a quadratic amount of work

[2] This assumption is not quite right. Take for example the minimizer `001`, and assume it occurs at the left end of the window, so the next window *must* choose a new minimizer. Regardless of which letter `X` occurs after the `001`, the resulting 3-mer `01X` has a good chance of being the minimizer of the next window. This dependency can be decreased by choosing an ordering for k-mers that puts lexicographically adjacent k-mers far from each other in the ordering.

w	T_{ratio}	F/T	Run time
1	99.993%	2.61	35:58
3	99.992%	2.41	27:48
8	99.987%	2.06	18:45
20	99.973%	1.66	12:26

Figure 6: Testing the effectiveness of minimizers using a dataset for which all true matches are known. All tests use k-mers with $k = 20$. The first column lists the window size w. Note that $w = 1$ means "use every k-mer". The second column, T_{ratio}, is the percentage of true matches that were found for the given value of m. The third column, F/T, is the "false to true ratio", ie., the ratio of false positive matches to true positive matches. The large number of false positives is due to repeat regions in *C. elegans*, giving false matches that are locally indistinguishable from true matches. The final column lists the run time to compute all matches in hours and minutes.

to bring the appropriate seeds together, so the distributed database fails the collection criterion. This resulted in a factor of approximately 100 slow-down compared to using minimizers. 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 biological string-matching problems on a single desktop computer, when they could previously only be run on a cluster or super-computer. For example, the 200-gigabyte k-mer database of the *Rattus norvegicus* genome mentioned in the introduction is now a 20 gigabyte minimizer database. Although there currently exist computers with 20 gigabytes of RAM, they are expensive. A more 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 *Rattus norvegicus* on a single desktop computer in about 3 days, a process which previously took hundreds of CPU days on a cluster [3]. The speed of this procedure also permits us to do high-quality, multi-read-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 match database [10, 11].

To test the reliability and speed advantage of minimizers, we created a data set of *faux reads* sampled computationally from the finished genome of *C. elegans*. Qualities for each base were taken from reads of the Human genome, and into each faux read we introduced errors (substitutions, insertions, and deletions) with probabilities based on true reads from the Human genome. We then computed overlaps between reads using minimizers with various window sizes as seeds. The results are in Figure 6. The first column shows that as the window size is increased from 1 to 20, using 20-mers as minimizers, the degradation in sensitivity of overlap detection is virtually nil. Similar results were found when using this technique in an assembly of *D. melanogaster* in collaboration with Celera [11], and an assembly of *Rattus norvegicus* in collaboration with the Baylor College of Medicine [10]. Finally, using less memory results in significant reduction in CPU requirements: Celera's computation of the overlaps between Drosophila reads took 10,000 CPU hours, which included using all k-mers as seeds, and significant $O(n^2)$ overhead to combine data that was gathered in parallel; ours took a total 100 hours on a single CPU. More details can be found in [11].

In summary, using minimizers as seeds for large-scale seed-and-extend string matching problems requires far less memory than storing every k-mer, and requires far less CPU time than searches using the store-every-kth-k-mer method.

References

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

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

[3] E. W. Myers et al. A whole-genome assembly of *drosophila*. *Science*, 287:2196–2204, 2000.

[4] S. Batzoglou et al. ARACHNE: A whole genome shotgun assembler. *Genome Research*, 12:177–189, 2002.

[5] B. Ewing and P. Green. Phrap. Unpublished. `http://www.genome.washington.edu`, 1994.

[6] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[7] B. Ma., J. Tromp, and M. Li. Patternhunter: Faster and more sensitive homology search. *Bioinformatics*, 13(3):440–445, 2002.

[8] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A fast search method for large dna databases. *Genome Research*, 11:1725–1729, 2001.

[9] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:2444–2448, 1988.

[10] Michael Roberts, Wayne Hayes, Cevat Ustun, Brian Hunt, and James Yorke. Read extension and correction using multi-comparison based overlaps. In preparation, 2003.

[11] Michael Roberts, Brian Hunt, James Yorke, Randall Bolanos, and Art Delcher. A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, 000:1–2, 2003. Accepted.

[12] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *J. Comp. Biol.*, 7:203–214, 2000.