Supporting Online Material for "Adaptive seeds tame genomic sequence comparison"

Szymon M. Kiełbasa[1], Raymond Wan[2], Kengo Sato[3],
Paul Horton[2], Martin C. Frith[2]

[1]Department of Computational Biology, Max Planck Institute
for Molecular Genetics, Ihnestr 63-73, 14195 Berlin, Germany

[2]Computational Biology Research Center, AIST,
2-4-7, Aomi, Koto-ku, Tokyo, 135-0064, Japan

[3]Graduate School of Frontier Sciences, University of Tokyo,
5-1-5 Kashiwanoha, Kashiwa, Chiba, 277-8561, Japan

December 15, 2010

Abstract

This document provides additional information to accompany the paper "Adaptive seeds tame genomic sequence comparison". We first describe our algorithm for finding adaptive seeds and how this is implemented in our software, LAST (Section 1 – Methods). This is followed by additional results with datasets that complement and expand on the main paper (Section 2 – Additional Results). Next, we describe the systems used for our experiments, more information about "pooling", and our dataset sources as well as any default settings chosen for local alignment (Section 3 – Materials). Finally, information pertaining to our analogy in Box 1 (Section 4 – Analogy with Text) and some related work not mentioned in the main paper is given (Section 5 – Related Work).

1 Methods

1.1 Methods for finding alignment seeds

Here we describe efficient methods for finding seeds (i.e. initial matches) between two sequences. We will refer to one sequence as the "target", and the other as the "query". The target contains T bases and the query contains Q bases. A is the number of distinct bases (4 for DNA, 20 for proteins). The sequences are first translated into a numeric encoding, for example: a $\rightarrow$ 0, c $\rightarrow$ 1, g $\rightarrow$ 2, t $\rightarrow$ 3. All of these methods consist of two phases: a preprocessing phase and a scanning phase. In the preprocessing phase, we analyze the target sequence to make some data structures (the "index"). In the scanning phase, we scan across the query sequence, looking up matches in the index.

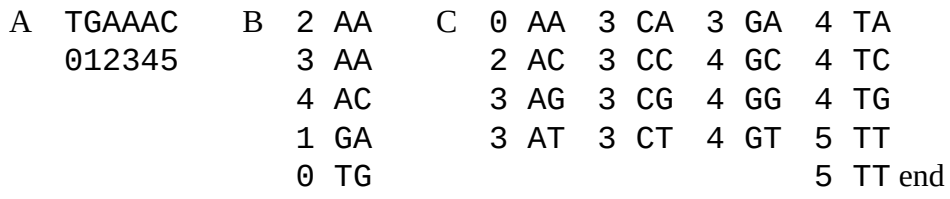

Figure S1: An index for finding fixed-length seeds of length 2. (A) A short DNA sequence, with positions written below it. (B) The position table, with corresponding 2-mers to the right. (C) The K-mer table, with 17 entries. Each possible 2-mer is written next to its start offset. The end offset of each 2-mer is the same as the start offset of the next 2-mer.

1.1.1 Fixed-length seeds

For sake of comparison, we first describe a straightforward method to find fixed-length seeds. In other words, we wish to find all exact matches of length K between two sequences.

In this case, the index consists of two tables: the position table and the K-mer table (Figure S1). The position table has one entry per position in the target sequence. It stores all the positions in the target where `aaa...aa` occurs, then all the positions where `aaa..ac` occurs, then all the positions where `aaa..ag` occurs, etc. The K-mer table requires $O(A^K)$ memory. It stores, for each possible K-mer, the start and end offsets in the position table for that K-mer.

In the scanning phase, we scan the query sequence from left to right, obtaining the K-mer starting at each position. This can be done in $O(Q)$ time, independent of K, by removing one base from the start of the K-mer and adding one base to the end at each step. The K-mers are represented by arithmetically computable indices, for example: $I(\texttt{gtgt}) \rightarrow 2 \times 4^3 + 3 \times 4^2 + 2 \times 4^1 + 3 \times 4^0$. Note that the indices of overlapping K-mer's can be computed in constant time, e.g. $I(\texttt{tgtc}) = 4 \times (I(\texttt{gtgt}) - 2 \times 4^3) + 1$. For each K-mer, in constant time, we can use the K-mer table to find the start and end offsets in the position table. Thus, we can count all the K-mer matches in $O(Q)$ time. The number of matches, however, might be $O(QT)$ in the worst case.

The preprocessing phase can be done in $O(T + A^K)$ time. One way is to scan the target twice. In the first scan, count all the K-mers. Use these counts to construct the K-mer table. In the second scan, populate the position table.

This method is practical only if A^K is not too large, e.g. less than 2^{32}. `BLAST`-like methods often have such a restriction on A^K.

1.1.2 Fixed-length spaced seeds

The method described above can be adapted straightforwardly to spaced seeds, where K is the number of must-match positions. In this case, however, we cannot compute successive K-mer indices by removing one base from the start and adding one base to the end. So the scanning phase now requires $O(QK)$ time, and the preprocessing phase requires $O(TK + A^K)$ time.

1.1.3 Adaptive seeds: definition

The simplest definition of adaptive seeds is: all exact matches between query and target sequences, such that each matching sequence occurs at most f times in the target. Our algorithm, in fact, finds something slightly different: all right-minimal exact matches that occur at most f times in

the target. "Right-minimal" means that the matches cannot be shortened on the right-hand side and still satisfy the criterion. Our algorithm records only the left-hand edge of each seed match, and subsequently extends an alignment starting from this edge. Thus, non-right-minimal matches would be redundant. Figure 1C-D in the main paper shows counts of right-minimal matches.

1.1.4 Adaptive seeds: suffix tree

For contiguous seeds, suffix trees (reviewed in [20]) could be used to implement adaptive seeds. The suffix tree Ψ of a target string $t = t_1...t_T\$$ is the trie whose root-to-leaf paths have a direct, one-to-one correspondence to the suffixes of t; while its root-to-internal-node paths have a one-to-one correspondence with the right-maximal repetitive substrings of t. Where a right-maximal repetitive substring of t is a substring which occurs followed by at least two distinct characters in t. We call it repetitive because it occurs at least twice, and right-maximal because it occurs more times than any string produced by adding one character to its right-hand side. The terminal character '$' is a special character whose purpose is to ensure that no suffix of t is a prefix of any other suffix of t.

Often suffix trees are considered to include one *suffix link* for each internal node. The suffix link attached to a node whose path from root spells out $t_1...t_i$, is a pointer to the node whose path from root spells out $t_2...t_i$. When $i = 1$ the suffix link points to the root, otherwise $t_2...t_i$ always corresponds to an internal node. This can be seen by observing that $t_1...t_i$ being right-maximal repetitive implies that $t_2...t_i$ is also right-maximal repetitive.

Suffix trees with their suffix links can be constructed in time and space linear in the size of the target text. Once Ψ is constructed, the frequency statistics of each substring in t can be computed with a depth first search traversal of the suffix tree. An example suffix tree with frequency statistics is shown in Figure S2.

For contiguous seeds, suffix links enable scanning a query $q = q_1...q_Q$ in $O(Q)$ time. The use of suffix links for this kind of computation is well known. Here we sketch the method and why it can be computed efficiently, but omit rigorous proofs.

Consider the process of finding all right-minimal matches (for some frequency f) between the target sequence t and a query sequence $q = q_1...q_Q$. This means finding matches for each suffix of the query, which is naturally done starting with $q_1...$, then $q_2...$, etc. Note that:

$$\text{frequency\_in\_target}(\ q_{i+1}...q_j\ ) \geq \text{frequency\_in\_target}(\ q_i...q_j\ )$$

Thus if $q_i...q_j$ is the right-minimal match for suffix $q_i...$, then the next right-minimal match will be either $q_{i+1}...q_j$ itself or an extension of it; and thus proper prefixes of $q_{i+1}...q_j$ never need to be examined. Conveniently, the last node encountered when looking for $q_i...q_j$ in Ψ has a suffix link pointing to exactly where the search for $q_{i+1}...$ should start. Using suffix links in this way, the work done moving *up* in sufT is constant for each suffix of Q. In contrast, the work done moving down in Ψ is not constant for each query suffix. Fortunately, each downward move in Ψ either terminates the search for that suffix or permanently moves the leftmost character to ever be examined again in Q to the right. Thus the amortized time of the entire scanning phase is $O(Q)$.

Despite the theoretical good fit that suffix trees have to adaptive seeds, we chose to use suffix arrays instead when implementing LAST. The reason is twofold. First, standard suffix trees do not work for spaced or subset seeds, although it seems likely that suffix trees can be generalized

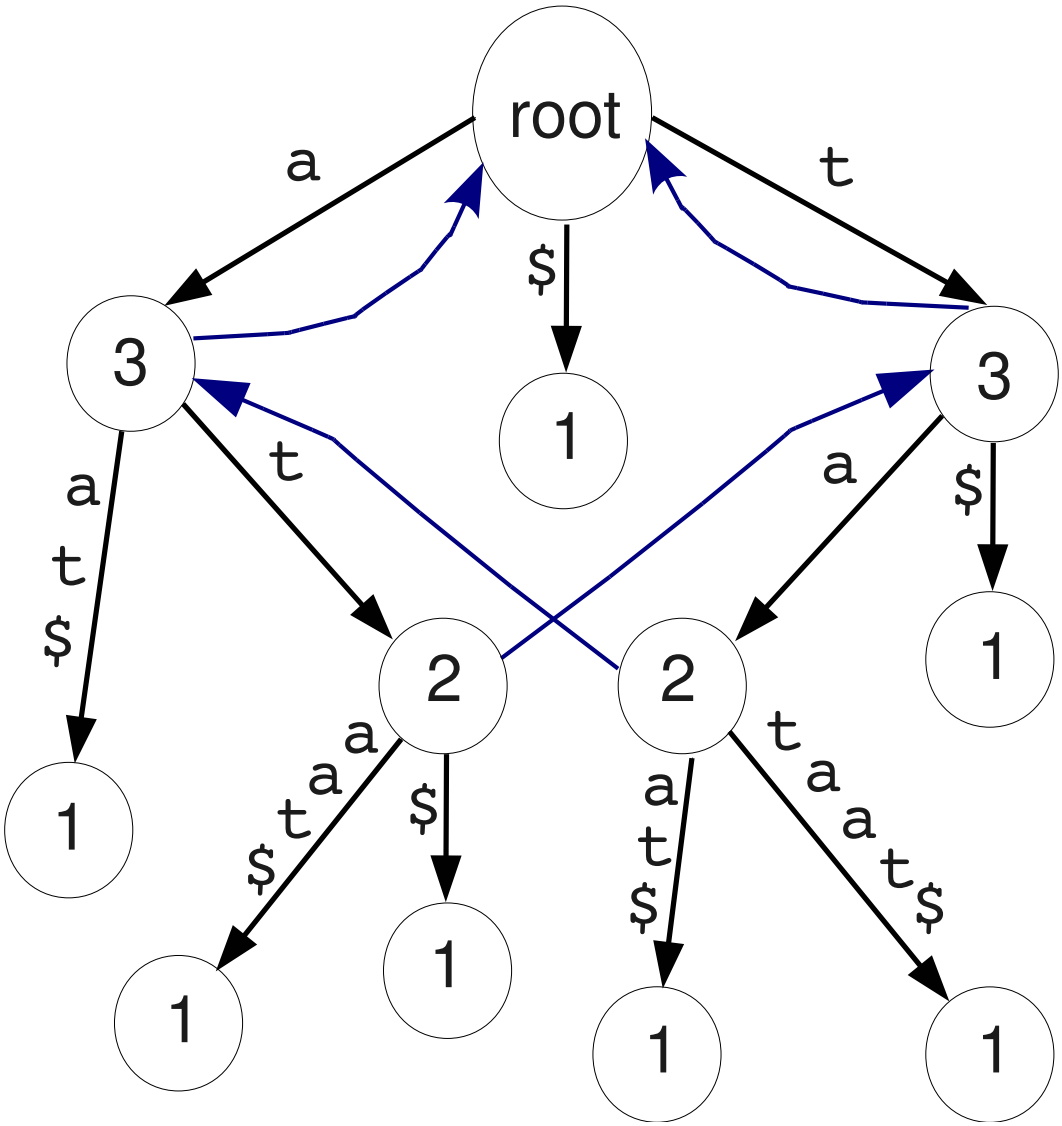

Figure S2: An example of a suffix tree with frequency statistics is shown. Colored arrows represent suffix links. The string is `tataat$`.

Table S1: The proportion of `lastal` run time used for finding adaptive seeds, for the protein sequences of Figure 3B in the main paper.

Adaptive seed frequency parameter (f)	1	2	5	10
Proportion of run time used for finding seeds	29%	15%	6%	4%

to support them. We have not confirmed this in detail, but we suspect that such as generalization would utilize the following inequality, where s is the span of the seed's repeating unit:

$$\text{frequency\_in\_target}(\ q_{i+s}...q_j\ ) \geq \text{frequency\_in\_target}(\ q_i...q_j\ )$$

Second, and more critically, suffix trees generally require more memory than suffix arrays. Efficient implementations of suffix trees require about 20 bytes per base [5]. So the human genome ($\sim$3 billion bases) would need $\sim$60 gigabytes, and the largest human chromosome ($\sim$250 million bases) would need $\sim$5 gigabytes.

1.1.5 Adaptive seeds: enhanced suffix array

Every algorithm that uses a suffix tree can be replaced with an algorithm that uses an enhanced suffix array and solves the same problem in the same time complexity [5]. The advantage is that enhanced suffix arrays require less memory: 8 bytes per base according to Table 6 of [5]. So the human genome would need $\sim$24 gigabytes, and the largest human chromosome would need $\sim$2 gigabytes.

1.1.6 Adaptive seeds: our implementation

Our implementation is simpler than a full-blown enhanced suffix array, and uses even less memory ($\leq$ 5 bytes per base). Its theoretical time requirement is inferior, but in practice it takes less time to find the seeds than to extend alignments from them, so the seed-finding is fast enough (Table S1).

Our index consists of two tables: a suffix array and a "bucket table". The suffix array has one entry per position in the target sequence. It stores the positions in the target sorted in alphabetical order of the sequences starting at those positions. Note how similar this is to the position table for fixed-length seeds (Figure S1B). The position table stores the positions sorted according to length K subsequences: it is like an incompletely-sorted suffix array. Conversely, the suffix array is like a position table valid for all values of K.

The bucket table is like the K-mer table for fixed-length seeds (Figure S1C). The difference is that we store entries for not just one value of K, but for all values of K up to some maximum: $K = 1, 2, 3, ..., B$. This requires $O(A^B)$ memory.

In the scanning phase, we scan the query sequence from left to right. At each position in the query, we find the shortest K-mer starting at that position, which occurs no more than f times in the target. To do this, we start with $K = 1$ and increase it one base at a time. When $K \leq B$, we look up the number of matches using the bucket table. When $K > B$, we find the K-mer's start and end offsets in the suffix array by binary search. The binary search is restricted to the offsets found in the previous ($K - 1$ $^{\text{th}}$) step.

```
A  ATATTC    B  2 ATtC
   012345       0 ATaTTc
                5 C
                1 TAtTC
                4 TC
                3 TTc
```

Figure S3: A spaced suffix array, using a spaced seed pattern of **110**. (A) A short DNA sequence, with positions written below it. (B) The spaced suffix array, with corresponding suffixes to the right. Lowercase indicates letters that are skipped when ordering suffixes.

For the preprocessing phase, we use an in-place radix sort to construct the suffix array [35, 26]. Again, this is not theoretically optimal: there are algorithms to construct suffix arrays in $O(T)$ time [40], and the radix sort does not achieve this guarantee. On the other hand, the radix sort is simple, fast in practice (Table S3), and can be implemented with only $O(\log T)$ memory overhead [35].

Our current suffix array implementation uses one 4-byte unsigned integer per entry. This means that we cannot handle chromosomes larger than 2^{32} bases (but we can handle genomes larger than 2^{32} bases, by voluming: see below). By default, B is set to the highest possible value such that the bucket table consumes at most one byte per entry in the suffix array.

1.1.7 Adaptive spaced seeds

To find adaptive spaced seeds, we use a "spaced suffix array" (Figure S3). Let us explain this with an example. Suppose we choose a spaced seed pattern of **110**. Then, the spaced suffix array will hold the positions in the target sorted in alphabetical order, except that every third position is skipped when ordering suffixes. So, for example, `acgc` comes before `acat`, and `ctgatgc` comes before `ctgatag`. The radix sort and scanning phase can be adapted straightforwardly to spaced suffix arrays. Moreover, it is possible to construct a spaced suffix array in $O(T)$ time [24].

1.1.8 Adaptive subset seeds

To find adaptive subset seeds, we use a "subset suffix array". Let us explain this with an example. Suppose we choose this subset seed pattern: `a, c, g, t` | `a, c, g, t` | `ag, ct` . This means that in the first two positions `a`, `c`, `g` and `t` are considered distinct and must match exactly, but in the third position `a` and `g` are considered equivalent, and so are `c` and `t`. As usual, we vary the length of this pattern by cyclically repeating it. The subset suffix array will hold the positions in the target sorted in alphabetical order, except that at every third position, `a` is considered equivalent to `g` and `c` is considered equivalent to `t` when ordering suffixes. So, for example, `acgc` comes before `acat`. The radix sort and scanning phase can be adapted straightforwardly to subset suffix arrays. Theoretically, it is possible to construct a subset suffix array in linear time using techniques similar to those for linear time spaced suffix array construction [24].

1.1.9 Multiple sequences and edges

The target may consist of multiple sequences (e.g. chromosomes). To handle this, we concatenate the target sequences into one big sequence, using special delimiter bases as separators. These

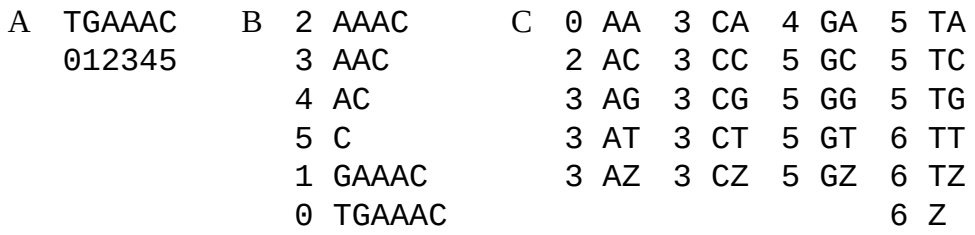

Figure S4: An index for finding adaptive seeds, using a bucket table with $B = 2$. (A) A short DNA sequence, with positions written below it. (B) The suffix array, with corresponding suffixes to the right. (C) The bucket table, with 21 entries.

delimiters are guaranteed not to match anything in any query sequence. This prevents artifactual seed matches that span two target sequences. Delimiters are also added to the very beginning and the very end of the query and target. These act as sentinels, so that the scanning phase does not need special logic to avoid falling off the end of the sequence.

When extending alignments from the seeds, we arrange for the delimiters to have a huge negative alignment score, so that they automatically terminate the gapless X-drop algorithm. For gapped extensions, unfortunately, special logic is needed to avoid crossing delimiters.

1.1.10 Reverse strands

In order to compare both strands of two DNA sequences, it suffices to compare one strand of one sequence to both strands of the other. This is easily done by comparing both strands of the query to one strand of the target, using two successive scanning phases.

1.1.11 Non-standard bases

DNA sequences in databases frequently contain bases other than `acgt`, and protein sequences often contain amino acids other than the standard twenty. Practical software needs to cope with such data. In our seed-finding method, non-standard bases are treated just like delimiters: they never match anything.

1.1.12 An implementation of the bucket table

The bucket table can be implemented in various ways, but it is desirable to optimize memory efficiency, so that we can maximize the value of B for a given amount of memory. Recall that the bucket table stores, for each possible K-mer where $K \leq B$, the start and end offsets in the suffix array for that K-mer. Our current implementation uses $(A^{B+1} - 1)/(A - 1)$ entries, where each entry is one 4-byte unsigned integer (Figure S4).

First, we need to store the start offset of every possible B-mer, which already requires A^B entries. This information includes the start offset of every K-mer with $K \leq B$: for example, the start offset of `ctg` is the same as the start offset of `ctgaaa`. This information also includes the end offsets of most K-mers: for example, the end offset of `tgca` is the same as the start offset of `tgcc`.

However, this information does not include the end offsets of K-mers that end with `t`. This is because of delimiters and non-`acgt` bases. The end offset of `tgact` might not be the same as

the start offset of `tgaga`, because if `tgacZ` occurs, it will sit between them in the suffix array. (`Z` represents a delimiter or non-`acgt` base, which sorts after the four normal bases.) Therefore, we additionally need to store the end offset of every K-mer that ends with `t`, taking a further A^{K-1} entries. So we need to store $A^B+A^{B-1}+...+A+1$ entries in total, which equals $(A^{B+1}-1)/(A-1)$ entries.

1.2 Reducing memory usage

Our seed-finding implementation requires a somewhat large amount of memory: 4-5 bytes per base for the index, plus 1 byte per base to store the sequence itself. We can reduce this a bit by not indexing non-`acgt` bases, of which there are many in current reference genome sequences. In practice, the largest human chromosome needs $\sim$1.5 gigabytes, and the human genome needs $\sim$16 gigabytes. There are several ways to reduce this memory usage, though none is without drawbacks. As computer memories continue to become larger and cheaper, it is not clear whether these approaches will be needed.

1.2.1 Volumes

We can trivially reduce memory usage by splitting a genome into sets of chromosomes, where each set is sufficiently small, and making a separate index for each set. `LAST` does this automatically, so that the human genome can be processed using 2 gigabytes. In the scanning phase, the software compares the query sequence(s) to each volume in turn.

The drawback is that the scanning phase becomes slower, because the query has to be re-scanned for each volume. Moreover, adaptive seeds for each volume are generally different from adaptive seeds for the whole genome: so this approach does not always produce identical results.

1.2.2 Sparse indexing

Another simple approach is to index only every N^{th} base in the genome. This method has been used with success by previous software such as `BLAT` [27]. This approach carries over straightforwardly to adaptive seeds and suffix arrays [28], and is an option in `LAST`.

The drawback is that many seeds can no longer be found, so some sequence similarities may be missed. On the other hand, when using adaptive seeds, sparse indexing will cause many new seeds to be found. This is because some matches that occur more than (say) ten times with a full index will occur less than ten times with a sparse index. So the effect on sensitivity is unclear.

1.2.3 Prefix splitting

We can also reduce memory usage by making separate indexes for genome positions beginning with different prefixes. For example, we might make 16 indexes: one for positions starting with `aa`, another for positions starting with `ac`, and so forth. In the scanning phase, we would compare the query sequence(s) to each index in turn.

Prefix splitting is similar to voluming, but has some advantages. Firstly, prefix splitting should cause little slowdown, because we can rapidly skip over query positions that do not start with the right prefix. Secondly, adaptive seeds found with this approach will be the same as adaptive seeds

Table S2: Comparison of time and space performance between the original LAST and CSA version of LAST. All timings are measured on Mac OS X 10.6 with Core i7 2.8GHz. The 2nd column: the elapsed time for building index with `lastdb -p ce6 ce6-sangerPep.fa`. The 3rd column: the elapsed time for counting matches with `lastal -j 0 ce6 dm3-flyBasePep.fa`. The 4th column: the elapsed time for extending matched seeds to gapped alignments with `lastal ce6 dm3-flyBasePep.fa`. The 5th column: the memory usage required for suffix indices.

	build	count	extend	memory
Original	6.8s	9.6s	73.2s	42MB
CSA version	9.1s	27.2s	511.8s	9MB

found without prefix splitting. This is not quite guaranteed: adaptive seeds shorter than the prefix length may not be the same, but such short adaptive seeds do not typically occur.

Prefix splitting would complicate tracking gapless alignments on each diagonal (described below). This would cause a (perhaps slight) increase in run time and/or memory usage (if we store the same gapless alignment more than once).

In summary, prefix splitting seems promising. We have not yet implemented it, because it would add complexity and the need for it is unclear.

1.2.4 Compressed suffix array / FM index

Memory usage can be reduced by using the compressed suffix array (CSA) or Burrows-Wheeler Transform (BWT) techniques, which have been actively studied in the field of data compression and have been applied for indexing large texts including genomes [33, 34, 32]. CSA and BWT do *not* store suffix indices themselves, but more compressible indices, from which the suffix indices can be recovered.

We implemented an experimental version of LAST with compressed indices based on the FM index using the wavelet tree, a state-of-the-art data structure for CSA and BWT [19]. In order to employ subset seeds, we applied the lexical naming principle proposed in [24]. We conducted a benchmark on the *C. elegans* and *D. melanogaster* peptide sequences used in Figure 3B to compare time and space performance between suffix arrays and compressed suffix arrays.

Table S2 shows that the CSA dramatically reduced the memory usage for suffix indices. For counting matches, the CSA version is relatively fast (but 3 times as slow as the original) because of the efficient backward search algorithm on the FM index. Whereas, for extending matched seeds, the CSA version is 7 times as slow as the original, because it requires much time to recover the suffix indices from compressed data structures to obtain the matched positions.

In summary, CSA seems promising for small PCs to use LAST with large genomes. By managing the maximum hit and minimum length parameters for adaptive seeds, the CSA version of LAST might be sufficiently practical. In general, the benefit of CSA / BWT / FM index is not clear-cut, because these techniques achieve lower memory usage at the expense of higher time usage.

1.3 Cache (in)efficiency

Cache efficiency is a technique to speed up execution on modern computer hardware. Unfortunately, our suffix array algorithm does not have great cache efficiency. On the one hand, by using

a bucket table, we perform binary searches only in small chunks of the suffix array that are likely to all be in cache at once. On the other hand, every lookup in the suffix array is accompanied by a random-access lookup in the target sequence.

A cache-efficient alignment algorithm using fixed-length seeds was recently published [21]. It requires indexing the query sequence as well as the target. It is possible that a similar approach could be used for adaptive seeds, but we have not pursued this idea.

1.4 Alignment extension in LAST

Having found seeds, LAST tries to extend alignments from them. The extension procedure is broadly the same as in BLAST, but we describe the details here.

1.4.1 Gapless extensions

As each seed is found, we perform gapless extensions in both directions from the left-hand edge of the seed match. The aligned bases are scored using an arbitrary scoring matrix, and the extension terminates when the score drops more than some value Y below the maximum seen for that extension [6]. If the total score of the gapless alignment is below a threshold value D, we discard it.

We also check whether each gapless alignment is "optimal". In other words, we check that all prefixes and suffixes of the alignment have positive score, and that the alignment has no segment with score $< -Y$. Alignments that fail these checks are discarded.

1.4.2 Gapped extensions

In this phase, LAST tries to extend gapped alignments, starting from the gapless alignments. It first sorts the gapless alignments in descending order of score (breaking ties in an arbitrary but deterministic way, to make the results reproducible). Then, taking each gapless alignment in turn, it finds the longest run of identical matches within the alignment, and performs gapped extensions from either end of the run. This procedure trims off possibly unreliable parts at either end of the gapless alignment, but it also avoids needless gapped extensions when comparing a sequence to itself.

The gapped extensions are done with an X-drop algorithm [7]. Our method proceeds anti-diagonal by antidiagonal (Figure S5), as described in Section 2 of [48], except that we do not use "half-nodes". For increased speed, we use the dynamic programming rearrangement described in [14].

If the gapped alignment has score less than some threshold E, or if it is not "optimal" (see above), it is discarded. Otherwise, we check whether it overlaps (i.e. shares paired bases with) any of the remaining gapless alignments. If it does, those gapless alignments are marked as redundant, and we will not waste time extending gapped alignments from them.

Our alignment extension procedure occasionally produces gapped alignments that overlap one-another. To reduce this redundancy, LAST has a final step where it discards any alignment that shares an endpoint (i.e. identical coordinates in both sequences) with a higher-scoring alignment.

		c	g	a	t	t	c	a
	0	-2	-4	-6				
g	-2	-1	-1					
g	-4	-3						
a	-6							
Z								

Match score: +1
Mismatch score: -1
Gap score: -2
Maximum score drop: 10

Figure S5: A gapped extension, with the first four antidiagonals completed. Each cell holds the optimal score for a gapped extension ending at that point. Z indicates a delimiter. For simplicity, this figure uses linear gap costs instead of affine gap costs.

1.4.3 Faster gapped extension near sequence ends

The gapped X-drop algorithm is inefficient when it starts near the end of a sequence, which often happens when aligning short sequences. A simple change makes it more efficient.

Consider the example in Figure S5, where the first four antidiagonals have been filled. When preparing to fill the fifth antidiagonal, the method checks for delimiters, and finds one (Z) in the vertical sequence. At this point, the method deduces that the maximum possible score increase is 3 (three matches with a maximum score of +1 each). Therefore, the maximum score drop can be reduced from 10 to 2, without changing the result. This reduces the number of cells that get filled.

1.5 Self-comparison of large sequences

The basic alignment extension procedure described above will go horribly wrong if we compare a large sequence to itself. However, it is not uncommon to compare two sequences containing largely identical segments. Therefore we employ several special techniques to cope with this.

1.5.1 Tracking gapless alignments on each diagonal

Each gapless alignment is on a fixed "diagonal", which is the coordinate in one sequence minus the coordinate in the other sequence. While doing gapless extensions, we update a "diagonal table" that records, for each diagonal, the right-most query-sequence coordinate covered by any non-discarded gapless alignment so far. By checking this table, we avoid performing gapless extensions from seeds that are already covered by a gapless alignment. Without doing this, comparison of a large sequence to itself would be catastrophically slow.

Since the number of diagonals can be huge, the diagonal table does not actually have a separate entry for each diagonal. Instead, it has 256 slots, each of which stores information for multiple diagonals. Information for diagonal d is stored in slot number $d \bmod 256$. The memory used by a slot grows when information for a new diagonal is added. To counteract this, we regularly discard entries with query-sequence coordinates to the left of the current scan position. Fortunately, the memory and time requirements of the diagonal table are negligible in practice.

We tried a slight variant of this method: tracking all "optimal" gapless alignments even if their score is less than the gapless extension threshold D. This variant was slower.

1.5.2 Redoing gapless extensions with gapped parameters

Apart from gaps, gapless and gapped extensions may differ in two ways. Firstly, they may use different X-drop values. Secondly, they may score lowercase letters differently (by using different score matrices): this allows a kind of soft repeat-masking, where lowercase letters are treated as mismatches for gapless extensions but not for gapped extensions.

These differences cause trouble when self-comparing a large sequence. The highest-scoring gapless alignment might only cover a small fraction of the sequence, e.g. because the sequence has many runs of unknown bases (`nnn...`), or because of lowercase masking. The gapped extension might then expand over the remaining, large fraction of the sequence. We call this "death by dynamic programming". For example, a gapped extension across the length of human chromosome 1 (250 million bases) would perform dynamic programming in an X-drop-defined band around the main diagonal, requiring excessive amounts of time and memory.

To address this problem, each gapless extension is repeated using the gapped score matrix and X-drop value, before proceeding to the gapped extension phase. This mostly solves the problem, but it is still possible to get huge gapped extensions if the GEP (gap extension penalty) is less than half the cost of aligning `n` to itself. (Minor details: The gapless extensions are re-done before sorting them in order of score, and we discard any that are not "optimal".)

1.5.3 Avoiding gapped extensions from tandem repeats

Biological sequences often contain tandem repeats. When self-comparing a sequence, tandem repeats produce gapless alignments that are close to, but not on, the main diagonal. When we perform gapped extension from such a gapless alignment, there is a danger that the alignment might wander onto the main diagonal and then extend over the entire length of the sequence.

Currently, we avoid this problem in an ad hoc way. When we check for overlaps between a gapped alignment and remaining gapless alignments, we make one other check too. We take the run of identical matches that was used to seed the gapped alignment, and look for remaining gapless alignments that represent tandem repeats within this run, with repeat period ≤ 1000. Any such gapless alignments are marked as redundant, and we do not expand gapped alignments from them. `LAST` has an option to turn off this procedure.

To avoid confusion, let us point out that tandem repeats cause two separate problems. One is the problem just described. The other is that short-period repeats give rise to strong alignments that are probably not homologous. The maximum period length that must be considered when addressing these two issues need not be the same.

2 Additional Results

The sensitivity of `LAST` and its running time has been evaluated for additional datasets and parameters which do not appear in the main paper. We report and discuss these results below, according to data type.

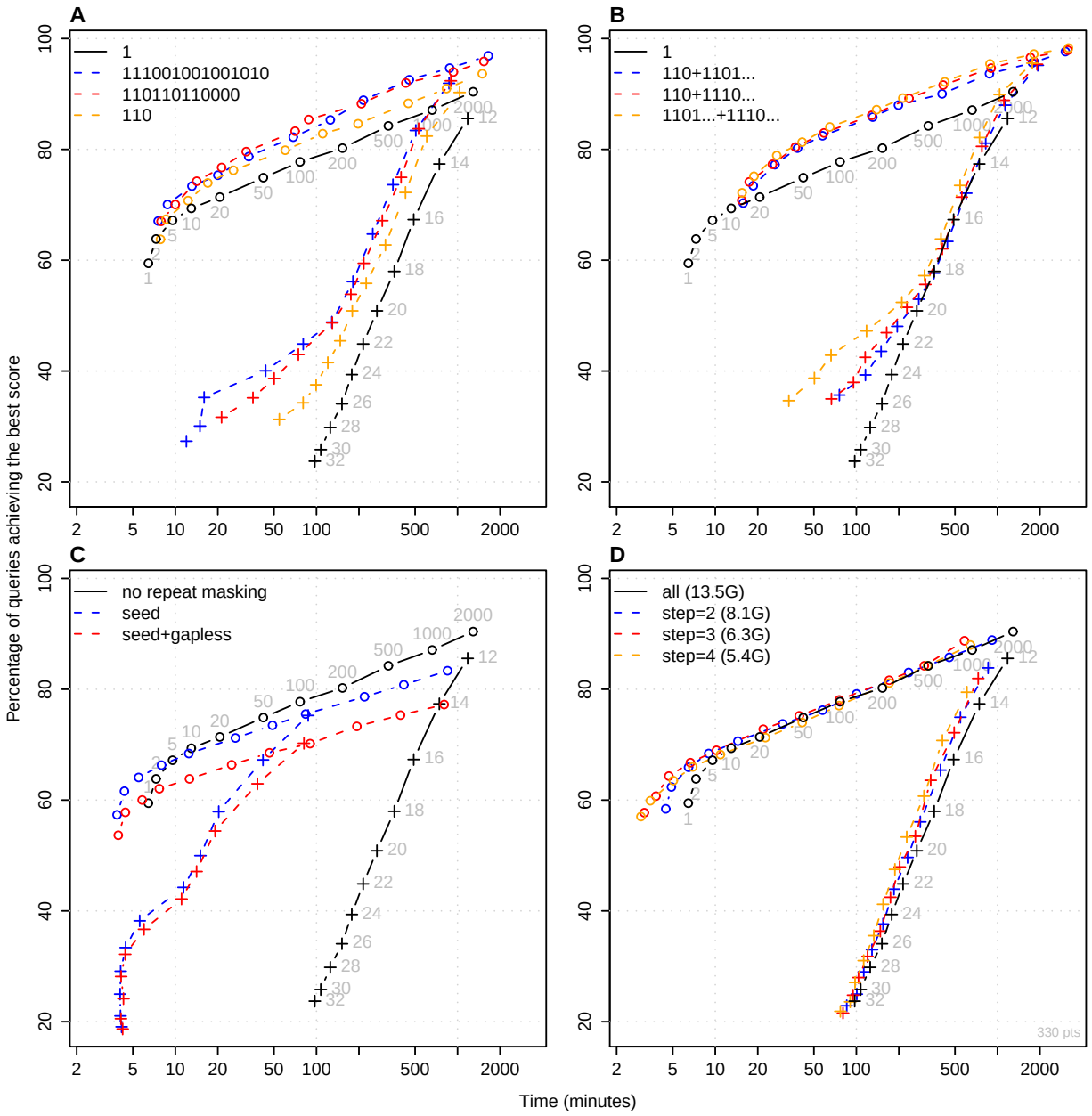

Figure S6: Performance of various methods for aligning *H. sapiens* promoters from the Eukaryotic Promoter Database (limited to max. 5000nt) to a database of the complete *M. musculus* genome. In each panel, circles indicate adaptive seeds and crosses indicate fixed-length seeds. The numbers provide adaptive seed maximum frequency or fixed seed length. (A) Contiguous vs. spaced seeds. (B) Merging best results for two seeds. (C) Effect of different query repeat masking strategies (using `WindowMasker`). (D) Sparse indexing of the target sequence for different step sizes.

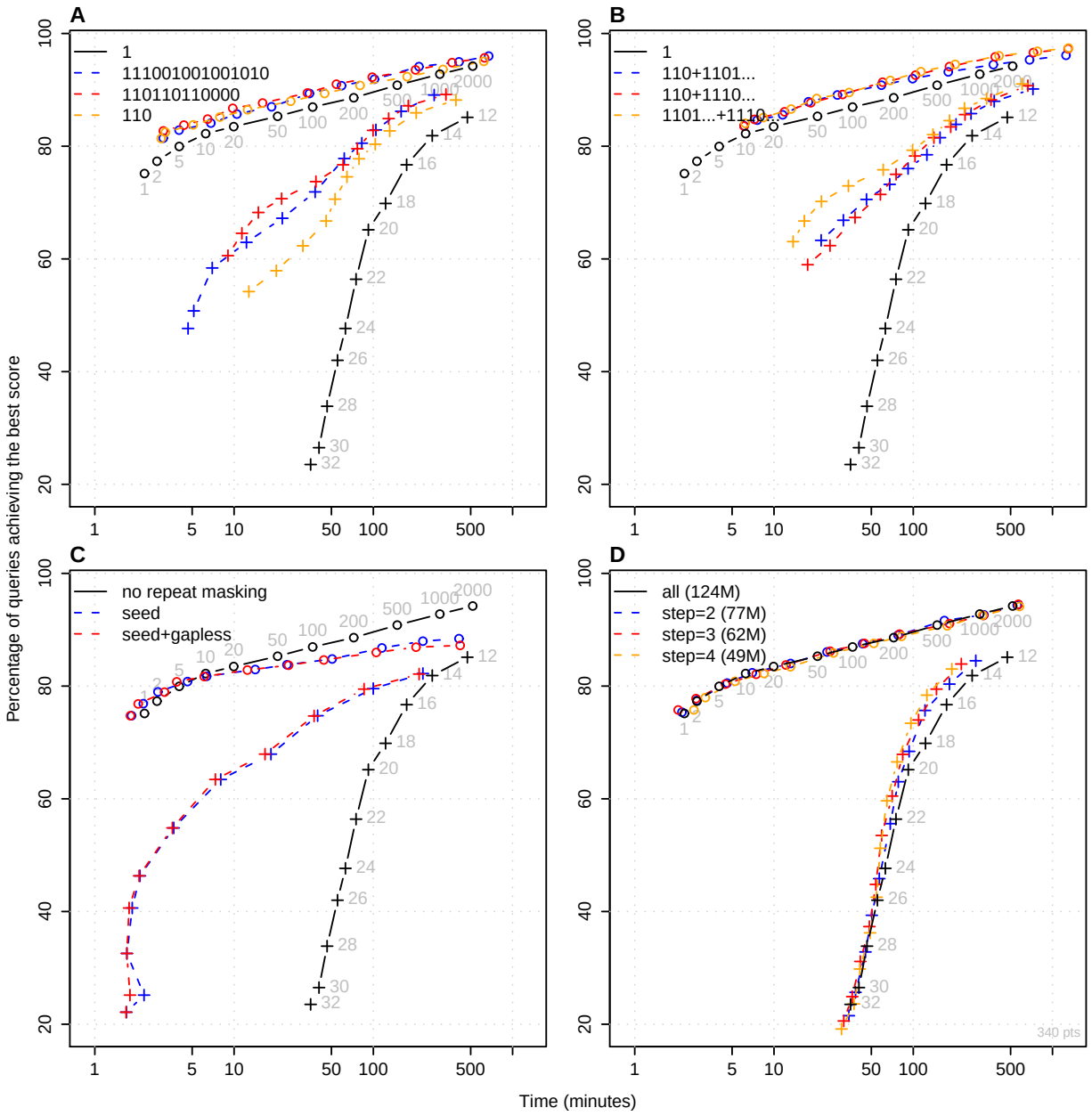

Figure S7: Performance of various methods for aligning *P. yoelii* query contigs to a database made of *P. falciparum* chromosomes. In each panel, circles indicate adaptive seeds and crosses indicate fixed-length seeds. The numbers provide adaptive seed maximum frequency or fixed seed length. (A) Contiguous vs. spaced seeds. (B) Merging best results for two seeds. (C) Effect of different query repeat masking strategies (using `Tandem Repeats Finder`). (D) Sparse indexing of the target sequence for different step sizes.

2.1 Genomic sequence comparisons

In Figure S6 and Figure S7, we present performance of LAST aligning human promoters against the complete mouse genome and *P. yoelii* contigs against the whole *P. falciparum* genome, respectively. Each panel presents evaluation of a different LAST parameter and is discussed below. All four panels of each figure are drawn to the same coordinate system, and the black points correspond to the same reference alignments obtained with contiguous seeds and unmasked sequences (Figure 3A,C in the main paper). The circles denote alignments for adaptive seeds with maximum seed frequencies f ranging from 1 to 2000, and the crosses correspond to fixed-length seeds with lengths l from 12 to 32.

In general, for alignments of genomic sequences we observe that adaptive seeds clearly outperform fixed-length seeds. This means that, for a given sensitivity, the same alignments are identified faster using adaptive seeds than using fixed-length seeds. For lower sensitivities, the time required for alignment can be reduced by more than an order of magnitude.

2.1.1 Contiguous and spaced seeds

In panels A of Figure S6 and Figure S7, we compare effects of the contiguous seed **1** to three spaced seeds: **111001001001010** (PH-OPT), **110110110000** (DATA-OPT) and **110**. PH-OPT and DATA-OPT are tuned for noncoding and protein-coding DNA [10]. Spaced seeds improve performance of alignment for both adaptive and fixed-length cases. Nevertheless, this improvement is much smaller than one caused by adoption of adaptive seeds instead of fixed-length seeds.

In panel B, we evaluate the effect of performing a single alignment with two different seeds. For all three pairs of spaced seeds shown in panel A we merged the observed alignment scores. For each query sequence we selected the alignment which scored better – either with the first seed of the pair, or with the second one. For computation time we simply summed the individual running time for each seed of the pair. This procedure seems to produce only a small improvement in sensitivity as compared to the individual non-contiguous seeds shown in panel A.

2.1.2 Query masking

In panels C of Figure S6 and Figure S7, we compare performance of LAST on unmasked and repeat-masked query sequences. For the masked case, initial seed matches are not allowed in the repetitive regions of the sequence. In addition, we show the effect of masking during gapless extensions (i.e. counting masked bases as mismatches).

For the two pairs of genomes studied, masking of repetitive sequences leads to similar decreases in sensitivity for adaptive seeds and fixed-seeds when runs with masking and without masking are compared to each other. In contrast, reduction of calculation time for fixed-length seeds is noticeably higher than for adaptive seeds. Still, adaptive seeds with no masking display better performance than fixed-length seeds with any of the tested masking strategies.

2.1.3 Sparse indexes

LAST provides an option to construct the index only for every N^{th} base in the target genome. This parameter strongly influences the size of the index.

The effect on performance of sparse indexing is shown in panels D of Figure S6 and Figure S7. In both studied datasets, sparse indexing for every second, third or fourth base of the target genome leads to curves nearly overlapping the curves for the complete index. Even a slight performance improvement due to sparse indexing is visible for fixed-length seeds of shorter lengths. For the mouse genome, the LAST index requires 13.5Gb of disk space for the complete index and only 5.4Gb when every fourth base is taken into account. For the *P. falciparum* database the size of the index is reduced from 124Mb for the complete index to 49Mb for every fourth base. Therefore, for smaller machines it might be advisable to use sparse indexing.

2.2 Protein sequence comparisons

In this section, we expand on our results using protein sequences.

2.2.1 Adaptive seeds

Adaptive seeds seem promising for protein comparison because amino acids are not equally abundant. For example, a match consisting of two tryptophans in a row might be rare enough to merit an alignment extension, but three alanines in a row might not be. Adaptive seeds can capture this idea.

To test this, we aligned fly (*Drosophila melanogaster*) proteins to worm (*Caenorhabditis elegans*) proteins. These organisms have some highly similar proteins, some extremely diverged proteins, and everything in between. For each fly protein, we aimed to find the highest-scoring alignment to any worm protein.

As expected, adaptive seeds find highest-scoring alignments more often than fixed-length seeds do, for a given running time (Figure S8). From another point of view, adaptive seeds can find highest scoring alignments as often as fixed-length seeds can, but faster. On the other hand, the two methods converge in performance as running time and sensitivity increase.

2.2.2 Low complexity masking

The goal of protein sequence comparison is usually (but not always) to identify evolutionary relationships. In this case, we must be careful of low complexity segments, such as short-period tandem repeats, which are rife in proteins. These lead to strong alignments that do not reflect evolutionary relationships. To avoid such alignments, low complexity segments are typically masked prior to alignment. Therefore, we did some further tests after hard-masking the query (fly) proteins with pseg, mirroring the default behavior of BLAST.

With masking, the advantage of adaptive seeds over fixed-length seeds is reduced (Figure S8B). This is because masking removes the most egregious non-uniformities in sequence composition. Nevertheless, adaptive seeds retain a clear advantage.

2.2.3 Subset seeds

Subset seeds are promising for protein comparison, because amino acid substitutions are not equally common. For example, hydrophobic amino acids replace one another more often than they replace hydrophilic amino acids. Subset seeds can capture this idea [41].

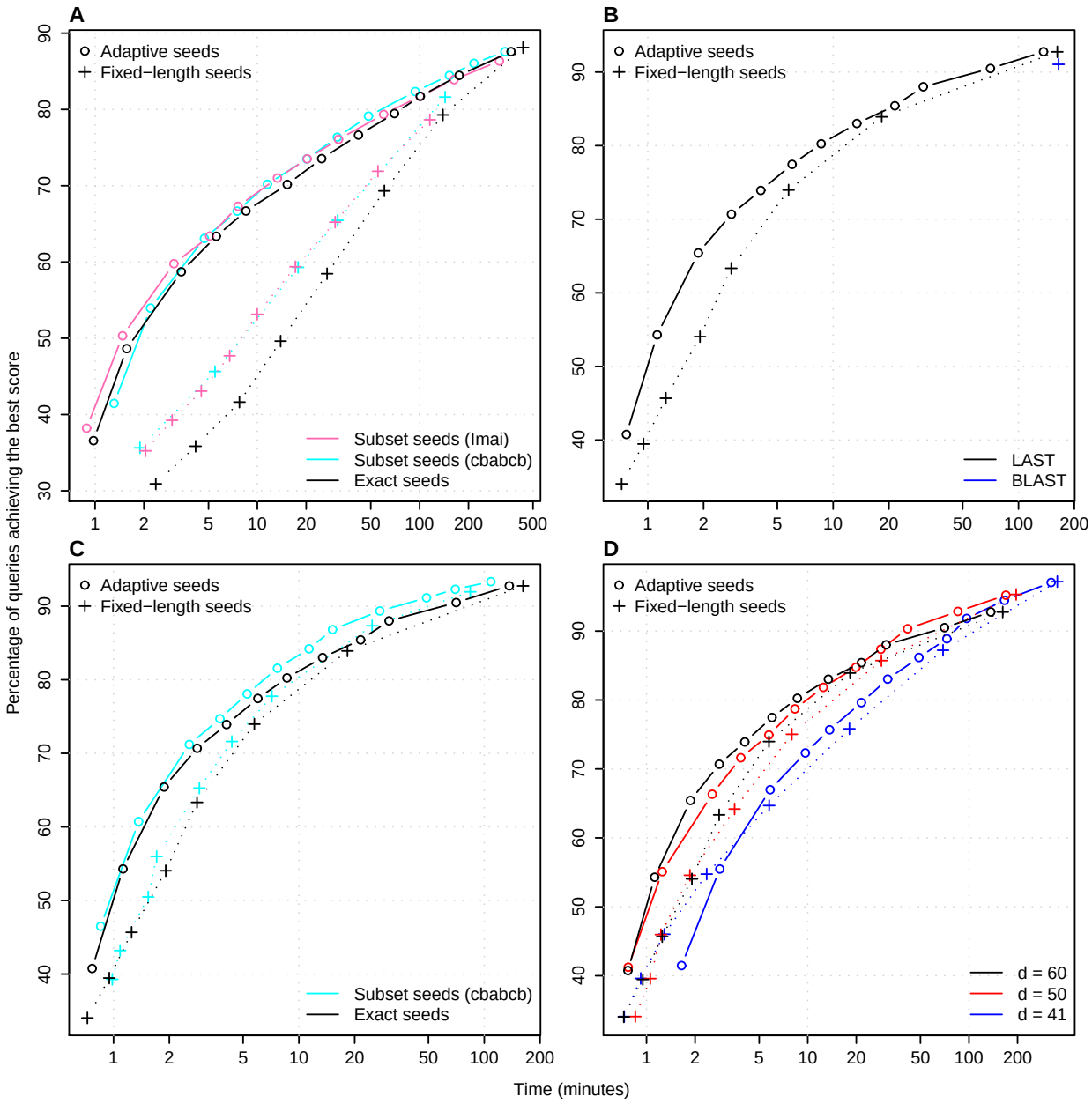

Figure S8: Performance of various methods for aligning proteins. For panel A the proteins were not masked, but for panels B-D, low complexity segments in the queries were hard-masked with `pseg`. (A) Effect of subset seeds. (B) Performance of LAST and BLAST. (C) Effect of subset seeds. (Here, Imai seeds performed worse than cbabcb seeds, and are omitted for clarity.) (D) Effect of the min-score parameter for the gapless extension phase, d. The other panels all use $d = 60$, except that BLAST uses $d = 41$.

On the other hand, there are many possible subset seed patterns, and it is hard to know which is best. We tried just two: "Imai" and "cbabcb". Imai seeds are matches using a reduced alphabet, where these groups of amino acids are considered equivalent: `ILMV`, `WYF`, `P`, `C`, `GA`, `STQN`, `DE`, `KRH`. cbabcb seeds use the Imai alphabet in "b" positions, the full alphabet in "c" positions, and allow any two characters to match in "a" positions (like the don't care positions in spaced seeds). As usual, we obtain seeds of any length by cyclically repeating the pattern.

We found that fixed-length subset seeds perform better than fixed-length exact seeds, and adaptive subset seeds perform better than adaptive exact seeds (Figure S8A,C). (Figure 3 in the main paper shows Imai seeds.) With adaptive seeds, the improvement was very slight for unmasked proteins, but a bit more substantial for masked proteins with the cbabcb pattern. It is likely that more carefully designed patterns would work better.

2.2.4 BLAST

The standard protein comparison tool is `BLAST`, which uses yet another seeding strategy: fixed-length *vector seeds* [6, 11]. These allow inexact matching. Specifically, they are matches whose score exceeds a threshold, the score being the sum of scores for every matched pair of amino acids, given by a scoring matrix. It is not obvious whether vector seeds will perform better than subset seeds: vector seeds capture amino acid relationships more finely, but subset seeds can be found with less computational cost. There is evidence that subset seeds perform competitively with vector seeds [41].

`BLAST` performed slightly worse than `LAST` in comparing our proteins (Figure S8B). It is hard to say why, because these programs differ in many details. (For example: `BLAST` looks for nearby *pairs* of length-3 vector seeds.) We can conclude that `LAST` performs competitively overall. This demonstrates that our main result, the advantage of adaptive over fixed-length seeds, is not an artifact of implementation.

2.2.5 Minimum gapless alignment score

Finally, we examined an algorithm parameter, d: the minimum score for passing a gapless alignment through to the gapped extension stage. This parameter has a big effect on performance. If it is too high we miss gappy alignments, but if it is too low we risk long run times by triggering expensive gapped extensions too often. The default value of 60 gave good performance for fast/low-sensitivity alignment, but lower values were better for slow/high-sensitivity alignment (Figure S8D). `BLAST`'s value of 41 seems appropriate for `BLAST`'s position on the speed/sensitivity curve. In any case, our other results are based on a reasonable setting for this important parameter.

2.2.6 Summary

In summary, adaptive seeds are useful for protein comparison. Combining them with subset seeds leads to even better performance. One idea we did not try is adaptive vector seeds: these would extend until the accumulated score reaches some rarity in the target. Our methods are aimed primarily at giga-scale sequence comparison (e.g. the 6-frame translation of the human genome versus all known proteins); for smaller problems, slower methods such as `PSI-BLAST` and hidden

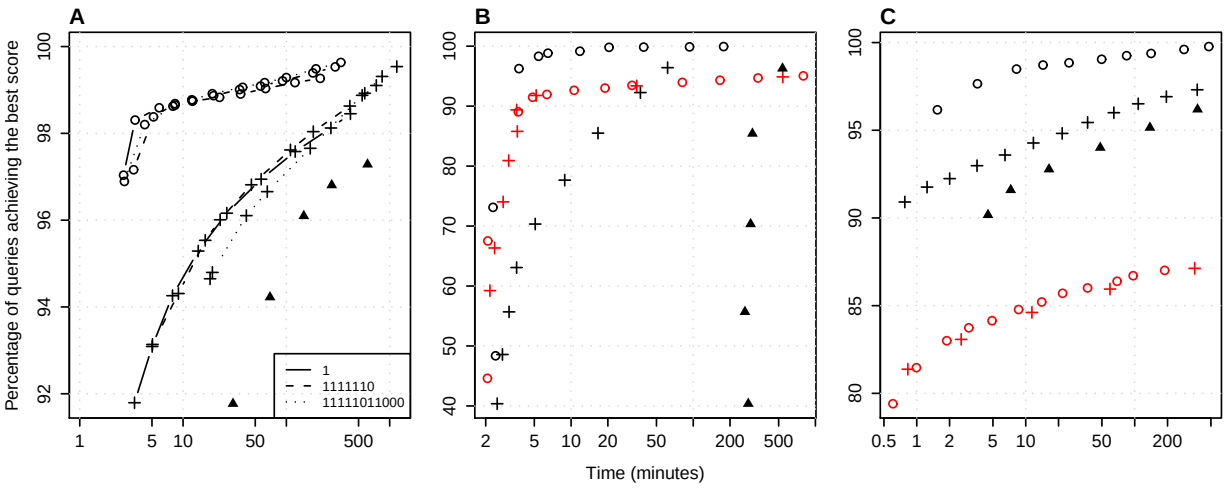

Figure S9: Performance comparison of fixed-length seeds against adaptive seeds for aligning short reads from next generation sequencers to their respective genomes. (A) Illustrates how various spaced seeds can affect fixed-length and adaptive seeds using the same *A. thaliana* dataset from the main paper (SRR014005, median read length 105). Also, the effect from soft repeat-masking with `WindowMasker` (with masking in red; without in black) for (B) *D. melanogaster* (SRR014394) and (C) *P. falciparum* (SRR006912) are shown. Median read lengths are 36 and 279, respectively. In all cases, fixed-length and adaptive seeds are indicated by "+" and "○", respectively. As a comparison, `megablast` results are indicated as "▲".

Markov models are no doubt more sensitive [7]. On the other hand, `LAST` can use position specific scoring matrices [18], so it could be developed along similar lines to `PSI-BLAST`.

2.3 Short read sequence comparisons

The main paper gives results for one short read sequence dataset. Here, we analyze two additional datasets, all three of which are obtained from NCBI's Sequence Read Archive [2]. The dataset used in the main paper is based on *A. thaliana* (SRR014005) using the 454 GS 20 platform. The two additional datasets are based on *D. melanogaster* (SRR014394) and *P. falciparum* (SRR006912) using the Illumina Genome Analyzer and 454 GS 20 platforms, respectively. Further details about all three datasets are given in Section 3, below.

Figure S9A expands on our analysis of the *A. thaliana* dataset by examining the effect spaced seeds have on alignment. The contiguous seeds of Figure 3D for both adaptive and fixed-length seeds are shown as solid lines. The two spaced seeds considered are **1111110** and **11111011000**. No noticeable difference is seen for adaptive seeds. For fixed-length seeds, the seed pattern **11111011000** performs slightly worse for longer seed lengths.

The next two panels (Figure S9B-C) present results for soft-masking using `WindowMasker` for *D. melanogaster* and *P. falciparum*, respectively. These results complement those of Figure 3D in the main paper for *A. thaliana*. In both cases, adaptive seeds outperform fixed-length seeds when masking is not employed.

When masking is used (shown in red), sensitivity drops below that of unmasked adaptive seeds. This is expected since parts of the genome are no longer considered when seeds are sought. Interestingly, the accuracy for adaptive and fixed-length seeds with masking fall below fixed-length

Table S3: Run times of `lastdb` and `formatdb`. The runs were performed on a 2.53 GHz Intel Xeon E5540 CPU.

Sequences	Preprocessing method	Time (sec)
C. elegans proteins	`lastdb`, exact-match seeds	6.35
	`lastdb`, Imai seeds	7.38
	`lastdb`, cbabcb seeds	6.66
	`formatdb`	0.85
D. melanogaster genome	`lastdb`, **1** seeds	98.77
	`lastdb`, **1111110** seeds	100.34
	`lastdb`, **11111011000** seeds	111.38
	`formatdb`	3.59
P. falciparum genome	`lastdb`, **1** seeds	9.53
	`lastdb`, **1111110** seeds	9.79
	`lastdb`, **11111011000** seeds	10.50
	`formatdb`	0.66

seeds without masking for *P. falciparum*. This is because its high A+T content causes 66.8% of its genome to be masked (see Table S4).

It is conceivable that fixed-length seeds are inferior in our results only because our implementation of them is slow. To test this possibility, we also obtained results with the popular tool `megablast`, which uses fixed-length seeds (triangles in Figure S9). We expected `megablast` to perform similarly to our fixed-length seeds. Surprisingly, it performs worse, especially for the *D. melanogaster* data. This shows that our implementation of fixed-length seeds is not slow, compared to a widely used alternative.

The performance of `megablast` can be understood by considering its seed search algorithm, as described in [36]. `megablast` first looks up k-mer matches between the query and target sequences, where k is typically *less than* the seed length. It then tries to extend each k-mer match to the left and to the right using the sequence data. In comparison, LAST finds fixed-length seeds by using the bucket table to find k-mer matches, and then extending them up to the seed length by binary search in the suffix array. Our method is more efficient for highly repeated k-mers. `megablast` has the advantage of using less memory: it only needs to index every $w - k + 1^{\text{th}}$ base, where w is the seed length, in order to guarantee finding every exact match of size w.

2.4 Other additional information

All running times reported above and in the main paper exclude the time required by `lastdb` to construct the index. Some of these values are reported for several datasets in Table S3. For comparison, we also report the time for `formatdb`, the preprocessing program used by BLAST. `formatdb` is much faster because it does not construct an index.

Query sequences	Maximum alignment score			Pool
	System A	System B	System C	
1	40	50	50	50
2	50	30	40	50
3	30	60	–	60
4	–	–	–	–
5	70	70	60	70
Sensitivity	50%	75%	25%	

Figure S10: An example illustrating pooling using a set of 5 queries and three systems showing how their results are combined to form the pool. The query results that are used to form the pool are shaded. The last row gives the performance of the three systems.

3 Materials

3.1 Experiment platform

Experiments were performed on two different sets of Linux machines. Therefore, only times within the same data type are considered comparable. Experiments with genomic data were run on a 2.8 GHz AMD Opteron 854 with 64 GB RAM and 1 MB L2 cache, while proteomic and short-read data were performed on a set of identical 2.0 GHz Dual-Core AMD Opteron Processor 246 with 6 GB of RAM. No parallelization across servers was employed.

All running times were obtained from the system `time` function by combining the time spent by the process (user time) with the time spent by the system on behalf of the process (system time).

3.2 Pooling

When evaluating multiple methods which heuristically attempt to optimize some score, it is convenient to compute the actual optimal score for use as a gold standard. In our case this would correspond to computing the full Smith-Waterman alignment of each query to its target sequence. Unfortunately this is difficult to do in practice due to the computation time required.

Fortunately, in order to evaluate relative performance, we do not really need to have an absolute gold standard; in the same way that we do not need to know the absolute limit of human running speed in order to decide who is the fastest runner in a group.

More formally, the evaluation performed in our paper is similar to what is known as *pooling* in the information retrieval community [44]. As illustrated in Figure S10, for each evaluation point, namely query sequence, we adopt the maximum score achieved by any method as the gold standard score for that evaluation point.

3.3 Masking

Three different systems were used for masking: `pseg` [47], `Tandem Repeats Finder` [9], and `WindowMasker` [37]. The correspondence between dataset and masking system is shown in Table S4. The last column gives the percentage of residues that were masked.

Repetitive regions in protein sequences were masked using `pseg` with options `-z 1 -q`.

Table S4: The size of each masked data set and the percentage of residues that are masked. All data sets are genomes except for the first and third. The system used for masking is given in the third column while options used are provided in the accompanying text.

Species	Size ($\times 10^6$ residues)	Masking system	% masked
D. melanogaster (proteins)	12.4	`pseg`	12.3 %
P. yoelii	20.2	`Tandem Repeats Finder`	6.3 %
H. sapiens EPD (5k)	9.2	`WindowMasker`	27.2 %
A. thaliana	119.2	`WindowMasker`	22.6 %
P. falciparum	23.2	`WindowMasker`	66.9 %
D. melanogaster	139.7	`WindowMasker`	23.0 %

The *P. yoelii* query sequences were masked using `Tandem Repeats Finder` with the command line: `trf400.linux.exe 2 7 7 80 10 50 500 -h -m`. These parameters, which mask repeats with period ≤ 500, are simply the ones recommended at the `Tandem Repeats Finder` website.

All remaining datasets were masked using `WindowMasker` in two-pass mode [37]. The `WindowMasker` software is part of the `BLAST+` command-line applications (version 2.2.22+) from NCBI using the NCBI C++ toolkit. The first pass used default options and collected statistics over the entire genome. The second pass was conducted on each chromosome individually with these statistics and `DUST` enabled (`-dust true`). `DUST` is a module within `WindowMasker` for locating and masking low-complexity regions.

3.4 Genomic data

3.4.1 Human and mouse

We used the UCSC Genome Bioinformatics Site as the source of the *M. musculus* genomic sequence (version `mm8`). Based on the complete unmasked mouse sequence, four `LAST` indexes were created, for contiguous seeds and for three different spaced seed patterns: **110**, **110110110000** and **111001001001010**. These indexes were queried with *H. sapiens* promoter sequences obtained from the Eukaryotic Promoter Database, release 100 [42]. The whole downloaded set contained 1870 human promoters extended to the maximum length of 5000nt upstream of the respective genes, leading to the total sequence size of approx. 9.2Mnt.

We calculated alignments using a score of 2 for matching nucleotides, cost of 1 for transitions and cost of 2 for transversions [17]. Moreover, the gap existence was set to 16 and gap extension cost was equal 1. We studied only alignments of score at least 150.

3.4.2 Plasmodium

The Plasmodium genomic sequences were downloaded from the 5.5 release of the PlasmoDB database. As the query sequences, we used 2960 contigs of *P. yoelii* from the file `PyoeliiGenomic_PlasmoDB-5.5.fasta` retrieved on 11/08/2009. The length of the contigs varies from 2000nt to 51,480nt with a mean of 6815nt and 76.1% `A+T` content. The database

was built from 14 chromosomes of *P. falciparum* from the file
`PfalciparumGenomic_PlasmoDB-5.5.fasta` retrieved on 07/01/2009. The `A+T` content of this genome is 79.3%.

We used a scoring system adjusted for the high `A+T` content of the Plasmodium sequences. The match score for `A-A` and `T-T` pairs was set to 3, and for `C-C` and `G-G` pairs to 9. We used a mismatch cost of 4, a gap existence cost of 15 and a gap extension cost of 3. We considered alignments scoring more than 200.

3.4.3 Hits Between Plasmodium Genomes

The example of Figure 2 in the main paper which depicts the positions of hits between a *P. yoelii* contig and the *P. falciparum* genome employs the same datasets as above. In the *P. falciparum* genome, the MB2 gene is on chromosome 5 at positions 686,802 to 691,640 (4838 nts in length). Its homologous gene in *P. yoelii* has an accession ID of PY03311 and is located at positions 1 to 3342 of contig MALPY00946. The dashed box in blue highlights this area. These locations were identified using the PlasmoDB web interface (`http://plasmodb.org/plasmo/`).

In the graph for adaptive seeds, another diagonal line appears beyond the top left-hand corner of the homologous region. This is another homologous region, unrelated to MB2.

3.5 Protein data

Protein sequences were taken from the files `flyBasePep.txt` and `sangerPep.txt`, downloaded from the UCSC genome database on 07/08/2009. Sequences with non-standard amino acids (e.g. `X`) were excluded. This yielded 21,228 fly proteins and 23,770 worm proteins.

We aligned the proteins using the Blosum62 matrix, with a gap existence cost of 11, a gap extension cost of 1, and a minimum gapped alignment score of 100. This is a reasonable threshold, because $\sim$10 alignments with score ≥ 100 would be expected between random protein sequences of the size of these datasets. (We determined this using `ALP1.1` [43]). We set the max-drop parameter for the gapped extension phase to 51 (one of the limited number of values `BLAST` allows for this parameter).

Some contortions were needed to fix parameters for NCBI `BLAST` (version 2.2.20). To request alignments with raw score ≥ 100, we used options `-C F -Y 1000000000 -e 0.00012`. To set the gapped max-drop parameter to 51, we used options `-X 20 -Z 20`. We also used `-F F` to turn off `BLAST`'s internal repeat-masking.

The "cbabcb" subset seed pattern was suggested by the subset seed design tool `IEDERA` [30]. We did not use the patterns derived by Roytberg et al. [41], because they used multiple simultaneous patterns, but we desired single patterns.

3.6 Short reads sequence data

Three short read datasets were downloaded from the NCBI Sequence Read Archive [2], as summarized in Table S5. The first four columns indicate the species, accession ID, sequencing instrument, and median read length for each dataset. Duplicate sequences were removed and the number processed are reported in the fifth column. The last column gives the size of the pool – the difference between these two columns is the number of reads whose best alignment score for any method

Table S5: Datasets selected for our experiments with short reads sequencing data. All data was obtained from the NCBI Sequence Read Archive [2], with their accession numbers given in the second column. The median lengths of the reads, the number of queries processed, and the size of their respective pools are given in the last 3 columns.

Species	Accession ID	Instrument	Median read length	Unique queries	Pool size
A. thaliana	SRR014005	454 GS 20	105	133,420	130,500
D. melanogaster	SRR014394	Illumina Genome Analyzer	36	898,275	673,682
P. falciparum	SRR006912	454 GS 20	279	7,923	7,903

failed to exceed our minimum threshold. The accuracies reported in the graphs are percentages with respect to this last column's values.

The genome for *A. thaliana* was downloaded from the NCBI ftp site [1] on 06/29/2009. Version 3 of the *D. melanogaster* genome was obtained from the UCSC Genome Browser [3] on the same date, but with `chrUextra` excluded. The same *P. falciparum* dataset as the one used for our genome-based experiments was used (version 5.5 of PlasmoDB).

`LAST` was compiled using `gcc` version 3.3.3 with compiler optimizations turned on (`-O3`).

Local alignment with `lastal` was performed with match and mismatch scores of 1 and -1, respectively. A gap existence cost of 2 and a gap extension cost of 1 were also used. The minimum alignment score was set at 30.

Experiments with various spaced seed patterns were also performed, with accuracy results reported for the *A. thaliana* dataset in Figure S9C. The spaced seed patterns chosen were obtained from the `LAST` manual where various patterns were calculated based on the tag size and the number of allowable mismatches. These patterns were obtained using other software [13, 29]. The two chosen ones (**1111110** and **11111011000**) correspond to a tag length of 36 and one and two mismatches, respectively.

For comparison to `LAST`'s fixed-length seed implementation, We selected `megablast` (version 2.2.22+) [48], which is part of the NCBI `BLAST+` family of programs. In order to improve its running time, an index was created using the `makembindex` tool [36], after building the database with `makeblastdb`.

The same parameters as `LAST` were chosen. As with `LAST`, the running time includes only the time for alignment and excludes both the database and index creation times. Several values for the length of the fixed-length seed were chosen, starting with its default of 28.

For each of the three datasets, its respective pool was formed by combining all runs that appear in the graphs (Figure 3, and Figure S9). This includes all runs of fixed-length and adaptive seeds, regardless of whether or not masking and spaced seeds were applied. The scores from all indicated runs of `megablast` were also included. Any other runs not shown in the graphs but reported elsewhere (such as spaced seed runs of `lastdb` for SRR014394 and SRR006912 in Table S3) were excluded from the pool.

Parameters for fixed-length and adaptive seeds (l and f, respectively) were chosen in increments indicated in Figure 3D. In some cases additional lengths were also computed and shown to reduce the amount of unused space in the figures.

3.7 LAST fixed-length seeds compared with LASTZ and BLASTN

Panel A of Figure 4 demonstrates the similar performance of LAST's fixed-length seeds and the ones used by LASTZ. LAST was run with parameters `-a15 -b3 -e200 -d120 -x90 -y90` and a scoring matrix adjusted for high A+T content. Additionally, the `-l` option was used to specify seed lengths. For LASTZ we used the following options: `--gap=15,3 --xdrop=90 --ydrop=90 --hspthresh=120 --gappedthresh=200` and the same scoring matrix. Moreover, we added the `[multiple]` modifier to the database sequence and both input sequences were unmasked with `[unmask]`. Finally, we used the following options: `--noentropy --recoverseeds --notransition`. We varied seed lengths with the `--seed=match` or `--seed=half` options.

Since BLASTN does not support arbitrary scoring matrices, we had to follow another strategy in order to calculate performance data for panel B of Figure 4. First we defined match ($+1$), mismatch (-1), gap existence (-7) and gap extension scores (-1) which could be processed by BLASTN. Executing the program manually for a certain E-value threshold (`-e1e-16`), we scanned the output and identified the score thresholds which BLASTN finds optimal for aligning provided query and target sequences. Consequently, BLASTN was executed with arguments: `-p blastn -e1e-16 -r1 -q -1 -G7 -E1 -X28 -Z28 -y20`. Moreover, we used `-Fn` to disable masking of repetitive sequences and we used `-W` to specify length of the seed. These parameters, once converted to the units used by LAST, resulted in the following arguments: `-e55 -d15 -r1 -q1 -a7 -b1 -y13 -x17`.

3.8 Y chromosome comparison

Here, we aimed to find homologous regions between the chimpanzee and human Y chromosomes. We used the human sequence from GRCh37, and the chimpanzee sequence from Supplementary File 2 of [25] (Table S6).

One difficulty is that the sequences have many simple repeats such as ATATATATATATAT, leading to regions that are similar but probably not homologous. To deal with this, we identified simple repeats in both sequences, using Tandem Repeats Finder 4.04 with options `2 5 5 80 10 30 200 -h -m -r` [9]. We used these options, which mask repeats with period ≤ 200, because they have been shown to supress spurious alignments of mammalian DNA [17]. We marked these repeats using lowercase letters. We did not identify or mark other kinds of repeats, such as LINEs and SINEs.

Next, we indexed the human sequence using `lastdb -c`, and aligned the sequences using `lastal -u1 -q3 -e30`. Option `-c` excludes lowercase letters from seeds, and option `-u1` treats them as mismatches during gapless extensions. Option `-q3` sets the mismatch cost to 3: together with the default match score of 1, this scoring scheme is appropriate for sequences with $\sim$99% identity [45]. Finally, `-e30` selects alignments with score ≥ 30. This score threshold is high enough to avoid chance similarities: the expected number of alignments with score ≥ 30 in random sequences of the same size and A+T composition is only 0.001. (We determined this using `ALP1.1` [43].)

To quantify the amount of chimpanzee sequence with homology to human, we counted all bases that lay within any LAST alignment. These alignments do not contain huge gaps: the default setting of the gapped X-drop parameter limits the gap size to 20.

Chromosome	Sequenced bases	A+T (%)	Simple repeats (%)
Human Y	25 653 566	60	11
Chimpanzee Y	26 041 480	60	12

Table S6: Statistics of chimpanzee and human Y chromosomes.

As a negative control, we reversed the repeat-marked chimpanzee sequence (without complementing it), and aligned it to the human sequence, using the same options as above. Because DNA never evolves by reversal, there are no true homologies in this test. The alignment produced zero output, suggesting that our procedure is effective at avoiding false-positive (non-homologous) alignments.

Note that our method is unable to distinguish orthologous regions from paralogous regions. It seems extremely difficult to distinguish orthologs from paralogs, especially if gene conversion is rife in Y chromosomes [25].

3.9 Expected frequencies of adaptive seeds

Figure 1C-D in the main paper shows the expected frequencies of adaptive seeds between uniformly random sequences. We obtained these frequencies by generating pseudorandom sequences of the appropriate lengths, and counting the adaptive seed hits.

4 Analogy with Text

For our analogy, the story "Alice's Adventures in Wonderland", by Lewis Carroll, was taken from the Canterbury corpus [8], a publicly-available corpus developed for the evaluation of text compression systems. The story was pre-processed by converting all letters to lower case (case-folding), reducing all words to their root form (stemming) [39], and ensuring that the story is a sequence of words, irrespective of the interleaved punctuation and white space. It is from this post-processed form where we obtained the given statistics. While a system was used to do this pre-processing [46], a simpler system could have been used given the story's small size.

There are significant differences between this analogy and actual sequence alignment:

1. Case-folding and stemming are problems specific to natural languages and different problems would arise for sequence alignment.

2. Given that natural language text can be re-arranged with minimal loss in meaning, alignment is perhaps inappropriate for text searching.

3. The string being sought ("The Queen of Hearts, she made some tarts") is known to exist (exactly) in the story. In practice, more efficient text searching algorithms could have been deployed.

Despite these differences, the analogy is meant only as an example to introduce the main paper since text management is a skill that many readers would already be familiar with.

5 Related Work

Here we describe related work by other researchers. Despite our efforts we have likely neglected some relevant work, for which we apologize. To put our method in historical perspective: we made a working prototype in 2007 and published it on the Web in 2008.

5.1 Variable length seeds

Adaptive seeds are similar to the "variable length seeds" described earlier by Csűrös [16]. Variable length seeds start with standard K-mer lookup tables for finding fixed-length seeds of length K. Shorter seeds are then created by merging lookup table entries for K-mers with common prefixes. This merging is guided by K-mer counts in the target sequence (like our method) and by letter frequencies in the query sequence (unlike our method). The key limitation is that the seeds can only become shorter: they cannot become longer than K. (And K cannot be arbitrarily large, because the memory requirement increases exponentially with K.) Thus, the number of seed matches remains $O(QT)$ in the worst case, just as for fixed-length seeds.

5.2 Sequence comparison with suffix arrays

QUASAR is an early sequence comparison method using a suffix array [12]. This method enabled much faster sequence comparison, but is restricted to searching for strongly similar DNA sequences. It uses a suffix array to find fixed-length exact matches ("q-grams"): the only advantage of the suffix array over a standard hash table is that different match lengths can be used without rebuilding the index from scratch.

5.3 Spaced suffix arrays

The perM algorithm uses periodic spaced seeds to map DNA reads to genomes [15]. Although the publication does not use the term "suffix array", it describes a spaced suffix array very similar to ours. They do not use adaptive seeds, however: instead, they lengthen each seed until it reaches the end of the DNA read. This method fulfils their objective of finding all alignments with up to three mismatches, but is not a general approach to finding low-similarity alignments.

BFAST also uses spaced seeds for mapping DNA reads to genomes, and the publication does use the term "suffix array" [23]. On the other hand, it uses fixed-length seeds. So it does not make full use of suffix arrays, and it could have used a standard approach of hashing with collision resolution.

5.4 Maximal unique matches

The MUMmer system compares large sequences by finding maximal unique matches [31]. It can find exact matches that are unique in either both, or just one, of the sequences. The latter is similar to adaptive seeds with a maximum frequency of 1 (except that adaptive seeds are right-minimal instead of maximal). This approach has been useful for comparing highly similar (but large) sequences, but the strict uniqueness requirement limits its sensitivity. To overcome this

limitation, MUMmer has an option to find all (not just unique) maximal matches longer than some threshold, but this is then similar to fixed-length seeds, with potentially $O(QT)$ matches.

5.5 Rare multiple exact matches

Also related is an algorithm to find rare maximal exact matches (RMEMs) [38]. This method finds maximal exact matches that occur at most t_1 times in one sequence and at most t_2 times in the other sequence. Furthermore, the method was generalized to multiple sequence comparison.

RMEMs are more general than adaptive seeds, but also require a more complex algorithm. Specifically, they require $O(t_2Q+t_1t_2T)$ time and $O(t_2T)$ space [38]. So this algorithm is efficient only when t_1 and t_2 are small, whereas adaptive seeds are similar to RMEMs with $t_2 = \infty$.

An interesting difference between RMEMs and adaptive seeds is that RMEMs allow symmetric matching, by choosing $t_1 = t_2$. In some cases, however, asymmetric matching is desirable. Here are three examples:

- Mapping short DNA reads to a genome. It makes sense to limit matches between one part of a read and many parts of the genome, but not to limit matches between one part of the genome and many reads. With adaptive seeds, we use the genome as the "target" and the reads as the "query".
- Annotating repeats in genomes, which is often done by comparing the genome to a library of repeat sequences. In this case, we do not wish to limit matches between one repeat and many parts of the genome. Using adaptive seeds, we make the genome the "query" and the repeat library the "target".
- Annotating a set of proteins by comparing them to all other known proteins. It makes sense to limit the number of reference proteins aligned to one query protein, but not vice versa.

In general, asymmetric matching seems desirable whenever we wish to "annotate" query sequences by comparing them to reference sequences.

As far as we know, RMEMs have not been combined with a subsequent BLAST-like extension step. Instead, they have been combined with subsequent chaining and filling steps in CoCoNUT [4]. This system can find chains of colinear non-overlapping RMEMs, and then globally align the sequences between the RMEMs in each chain. Compared to BLAST-like extension (dynamic programming with X-drop), chaining and filling may be faster for long alignments, and it can be used for multiple sequence comparison. On the other hand, BLAST-like extension can find subtle alignments that contain just one seed. In particular, the alignment of a short DNA read to a genome may include only one RMEM, and chaining does not help in this case. Also, chaining involves an extra scoring scheme to define scores for chains, and it is not clear how this can be chosen to get optimal nucleotide-level alignments. The filling step is questionable, because it forces alignments between sequences no matter how dissimilar they are. In short, chaining and filling is appropriate for long, strong alignments, but not for short or weak alignments.

5.6 Longest prefix matches

Finally, adaptive seeds have some similarity to the use of longest prefix matches in segemehl, for mapping short DNA reads to genomes [22]. Taking each position in the read as a starting point,

segemehl finds the longest exact match to any place in the genome, using an enhanced suffix array. If this match occurs fewer than (say) 500 times in the genome, segemehl calculates an alignment with optimal edit distance for each occurrence. Furthermore, segemehl can allow a limited number of differences within prefix matches, by enumerating all possible differences at certain positions.

The sensitivity of segemehl with exact matches is inherently limited, because these matches are *longest*. Each match is expected to have about two occurrences in the genome, because matches with more than two occurrences are unlikely to be longest. The exception is matches that start near the end of the read, which are limited by segemehl's maximum occurrence parameter (e.g. 500). Adaptive seeds, in contrast, need not be longest matches, and so their sensitivity is arbitrarily tunable by their maximum frequency parameter. Moreover, segemehl is specialized for short read mapping, and cannot be used for other alignment tasks.

5.7 Summary

From a theoretical computer science perspective, adaptive seeds are a minor variant of ideas that have been published before. Adaptive seeds are deliberately defined in such a way that it is trivial to find them with suffix tree (or similar) algorithms, compared to, say, RMEMs. On the other hand, we are the first to unify this approach with spaced and subset seeds, which enables state-of-the-art sensitivity.

Our main contribution is practical. By adapting and combining previous ideas, we have the first BLAST-like method that can find similar regions in giga-scale sequences, with high sensitivity and without repeat-masking. All previous methods (including short read mappers) either required repeat masking or had limited sensitivity. Although it often remains desirable to mask simple repeats (in order to avoid non-homologous alignments), this is not always desirable (e.g. for read mapping), and these are not the only kind of repeat.

References

[1] NCBI genomes. ftp://ftp.ncbi.nih.gov/genomes/.

[2] NCBI sequence read archive. http://www.ncbi.nlm.nih.gov/Traces/sra/.

[3] UCSC genomes. http://hgdownload.cse.ucsc.edu/downloads.html.

[4] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. CoCoNUT: an efficient system for the comparison and analysis of genomes. *BMC Bioinformatics*, 9:476, 2008.

[5] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.

[6] 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.

[7] S. F. Altschul, T. L. Madden, A. A. Schäffer, 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.

[8] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. 1997 IEEE Data Compression Conference*, pages 201–210, 1997.

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

[10] B. Brejová, D. G. Brown, and T. Vinař. Optimal spaced seeds for homologous coding regions. *J Bioinform Comput Biol*, 1:595–610, 2004.

[11] B. Brejová, D. G. Brown, and T. Vinař. Vector seeds: An extension to spaced seeds. *Journal of Computer and System Sciences*, 70:364–380, 2005.

[12] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proc. 3rd Annual Conference on Research in Computational Molecular Biology*, pages 77–83, 1999.

[13] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae XXIII*, pages 1001–1018, 2003.

[14] M. Cameron, H. E. Williams, and A. Cannane. Improved gapped alignment in BLAST. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:116–129, 2004.

[15] Y. Chen, T. Souaiaia, and T. Chen. PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, 25:2514–2521, 2009.

[16] M. Csűrös. Performing local similarity searches with variable length seeds. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 373–387, 2004.

[17] M. C. Frith, M. Hamada, and P. Horton. Parameters for accurate genome alignment. *BMC Bioinformatics*, 11:80, 2010.

[18] M. C. Frith, R. Wan, and P. Horton. Incorporating sequence quality data into alignment improves DNA read mapping. *Nucleic Acids Res.*, 38:e100, 2010.

[19] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ASM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.

[20] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge Press, New York, NY, 1997.

[21] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7:576–577, 2010.

[22] S. Hoffmann, C. Otto, S. Kurtz, C. Sharma, P. Khaitovich, J. Vogel, P. Stadler, and J. Hackermuller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Comput. Biol.*, 5:e1000502, 2009.

[23] N. Homer, B. Merriman, and S. Nelson. BFAST: an alignment tool for large scale genome resequencing. *PLoS One*, 4:e7767, 2009.

[24] P. Horton, S. M. Kiełbasa, and M. C. Frith. DisLex: a transformation for discontiguous suffix array construction. In *Proc. of the Workshop on Knowledge, Language, and Learning in Bioinformatics (KLLBI)*, pages 1–11, 2008.

[25] J. Hughes, H. Skaletsky, T. Pyntikova, T. Graves, S. van Daalen, P. Minx, R. Fulton, S. McGrath, D. Locke, C. Friedman, B. Trask, E. Mardis, W. Warren, S. Repping, S. Rozen, R. Wilson, and D. Page. Chimpanzee and human Y chromosomes are remarkably divergent in structure and gene content. *Nature*, 463:536–539, 2010.

[26] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *Proc. 15th International Symposium on String Processing and Information Retrieval*, volume 5280 of *LNCS*, pages 3–14, 2008.

[27] W. J. Kent. BLAT – the BLAST-like alignment tool. *Genome Research*, 12:656–664, 2002.

[28] Z. Khan, J. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25:1609–1616, 2009.

[29] G. Kucherov, L. Noé, and M. Roytberg. Multiseed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2:51–61, 2005.

[30] G. Kucherov, L. Noé, and M. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *Journal of Bioinformatics and Computational Biology*, 4:553–569, 2006.

[31] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12, 2004.

[32] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25, 2009.

[33] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760, 2009.

[34] R. Li, C. Yu, Y. Li, T. Lam, S. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25:1966–1967, 2009.

[35] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6:5–27, 1993.

[36] A. Morgulis, G. Coulouris, Y. Raytselis, T. L. Madden, R. Agarwala, and A. A. Schäffer. Database indexing for production MegaBLAST searches. *Bioinformatics*, 24:1757–1764, 2008.

[37] A. Morgulis, E. M. Gertz, A. A. Schäffer, and R. Agarwala. WindowMasker: window-based masker for sequenced genomes. *Bioinformatics*, 22:134–141, 2005.

[38] E. Ohlebusch and S. Kurtz. Space efficient computation of rare maximal exact matches between multiple sequences. *J. Comput. Biol.*, 15:357–377, 2008.

[39] M. F. Porter. An algorithm for suffix stripping. *Program*, 14:130–137, 1980.

[40] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, 2007.

[41] M. Roytberg, A. Gambin, L. Noé, S. Lasota, E. Furletova, E. Szczurek, and G. Kucherov. On subset seeds for protein alignment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6:483–494, 2009.

[42] C. D. Schmid, R. Perier, V. Praz, and P. Bucher. EPD in its twentieth year: towards complete promoter coverage of selected model organisms. *Nucleic Acids Res.*, 34:D82–D85, 2006.

[43] S. Sheetlin, Y. Park, and J. Spouge. The Gumbel pre-factor k for gapped local alignment can be estimated from simulations of global alignment. *Nucleic Acids Res.*, 33:4987–4994, 2005. `http://www.ncbi.nlm.nih.gov/CBBresearch/Spouge/html.ncbi/index/software.html`.

[44] K. Spärck Jones and C. V. Rijsbergen. Report on the need for and provision of an "ideal" information retrieval test collection. Technical report, Computer Laboratory, University of Cambridge, 1975.

[45] D. J. States, W. Gish, and S. F. Altschul. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods*, 3:66–70, 1991.

[46] R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, University of Melbourne, Australia, Dec. 2003.

[47] J. C. Wootton and S. Federhen. Analysis of compositionally biased regions in sequence databases. *Methods in Enzymology*, 266:554–571, 1996.

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