Sapling: Accelerating Suffix Array Queries with Learned Data Models

Melanie Kirsche[1*], Arun Das[1], Michael C. Schatz[1,2,3]

[1]Department of Computer Science, Johns Hopkins University, Baltimore, MD
[2]Department of Biology, Johns Hopkins University, Baltimore, MD
[3]Cold Spring Harbor Laboratory, Cold Spring Harbor, NY

Abstract

Motivation: As genomic data becomes more abundant, efficient algorithms and data structures for sequence alignment become increasingly important. The suffix array is a widely used data structure to accelerate alignment, but the binary search algorithm used to query it requires widespread memory accesses, causing a large number of cache misses on large datasets.

Results: Here we present Sapling, an algorithm for sequence alignment which uses a learned data model to augment the suffix array and enable faster queries. We investigate different types of data models, providing an analysis of different neural network models as well as providing an open-source aligner with a compact, practical piecewise linear model. We show that Sapling outperforms both an optimized binary search approach and multiple widely-used read aligners on a diverse collection of genomes, including human, bacteria, and plants, speeding up the algorithm by more than a factor of two while adding less than 1% to the suffix array's memory footprint.

Availability and implementation: The source code and tutorial are available open-source at https://github.com/mkirsche/sapling.

Contact: mkirsche@jhu.edu

Supplementary Information: Supplementary notes and figures are available online.

1. Introduction

Aligning sequencing reads to a reference genome or collection of genomes is a key component of many genomic analysis pipelines, including variant calling [1], quantifying gene expression levels (RNA-seq) [2], identifying DNA-protein binding sites (ChIP-seq) [3] and several others [4]. Many techniques have been proposed to solve the read alignment problem in ways that are computationally efficient and robust to sequencing errors and true biological differences. Since finding inexact alignments is generally much slower than finding exact matches, a common approach is to use the seed-and-extend heuristic [5]. When using this heuristic, small segments of the read are used as seeds, and exact matches of these seeds are found using an algorithm for exact string matching. Then, the exact matches are used as candidate alignment sites, and each is scored based on how well the whole read aligns in the surrounding region. This heuristic has been shown to perform well in many genomic applications, and is used by a large number of leading short and long reads aligners including Star [6], Bowtie2 [7], BWA-MEM [8], NGMLR [9] and many others. It is also used as a core routine for whole genome alignment [10] and many other applications [11].

The seed-and-extend heuristic relies on being able to quickly search for exact matches of seed sequences in the reference genome. The problem of finding these matches, called the exact substring search problem, has applications both within and outside of genomics [12]. A number of data structures have been proposed to solve this problem by indexing the reference genome in such a way that the

exact substring search problem can be solved quickly. These include suffix arrays [13], suffix trees [14], hash tables [15], and FM-indexes [16]. For genomic applications, suffix arrays are one of the key data structures for seed-and-extend algorithms used by Star [6], BLASR [17], MUMMER4 [10] and others. The suffix array consists of the lexicographically ordered list of suffixes present in a string, and once constructed, a binary-search-like algorithm can be used to quickly locate exact matches of query strings [13].

Learned index structures [18] are a technique for accelerating queries on a variety of data structures by leveraging patterns present in the particular dataset being processed. While classical data structures are asymptotically optimal, these runtime bounds are based on a worst-case analysis where it is assumed that the dataset has no specific patterns that can be exploited. However, many real-world datasets have learnable patterns, and learned index structures have been used in many different applications such as B-trees and Hash-maps [18]. Additionally, learned index structures have previously been considered for read alignment using a modified FM-index [19], although the source or implementation are not available and it was only applied to a single dataset.

Here we present Sapling, an open-source algorithm which leverages learned index structures for accelerated read mapping. At its core, it uses suffix arrays, which we augment with a model of the particular genome that is being indexed. We evaluate two different types of data models - a neural network trained on the suffix array, as well as a compact piecewise linear model. We find that by using a data model, the core suffix array query time is reduced by more than a factor of two while only increasing the size of the data structure by less than 1% across a variety of genome sequences. We offer Sapling as both an open-source library for exact substring search and a standalone read aligner at https://github.com/mkirsche/sapling.

2. Methods

2.1 Suffix Array Search

For a text T of length n, let T[i] be the character in the ith position of T, and define a substring of T, T[i..j], where $0 \leq i \leq j < n$, as a string of characters T[i], T[i+1], …, T[j]. We define the exact substring search problem as follows: Given a text T of length n and a pattern P of length m, report all positions x in T such that T[x..(x+m-1)] is equal to P. A naive algorithm that considers all possible values for P would take O(n * m) operations, which is infeasible for large texts, especially when many queries each need to be evaluated. In genomic applications where the text is a reference genome and the pattern is a genomic read a few properties generally hold: 1) The text is much (multiple orders of magnitude) larger than each query, and 2) The same text is used across multiple queries (typically many millions to billions of sequencing reads for a single genome). In an attempt to exploit these properties, several algorithms have been proposed which index the text on its own before any of the queries are considered, and then this index is used to reduce the number of possible alignment positions for every query.

One popular index is the suffix array. A suffix of T is defined as any substring T[i..n-1]; that is, any substring which ends after the last character of T. Suffixes are related to substring search queries because any occurrence of a length-m pattern P at some position x in T corresponds to a suffix of T, T[x..n-1], whose first m characters are exactly the string P. When the suffixes are considered in

lexicographical order, all such suffixes starting with P will occur contiguously. This property of suffixes serves as the intuition behind the use of suffix arrays for exact substring search queries.

The suffix array is defined as an array of positions corresponding to the lexicographical order of suffixes in a given text. For a text T with n characters, we define the suffix array of T, SA_T to be a permutation of {0, .., n-1} such that $SA_T[i]$ is the start position in T of the ith suffix of T when the suffixes are sorted lexicographically. For example, in the text T = “CAT”, the sorted order of suffixes is {“AT”, “CAT”, “T”}, so SA_T = {1, 0, 2}. For any pattern P, each occurrence of P in T will be the prefix of some suffix of T, and since each such suffix starts with the characters in P, the start positions of the instances of P in T will occur consecutively in SA_T. This reduces the problem of exact substring search to that of finding the range of suffix array positions [i, j] such that $T[SA_T[k]..(SA_T[k]+m-1)] = P$ for all integers k in [i, j]. These positions can be found using a binary search algorithm, which starts with an initial search space of [0, n-1] and repeatedly bisects the search space, querying the middle suffix to decide whether the suffixes starting with the characters in P occur in the first or second half, and recursively searching the half-sized space. The naive binary search algorithm, for a pattern of length m, requires O(log(n) * m) operations since each query requires a string comparison of up to m characters. However, a more efficient binary search algorithm specialized for the suffix array has been proposed which requires O(log(n) + m) operations. This exploits an auxiliary data structure called the longest common prefix array (LCP array) that stores the number of shared characters between the prefixes of consecutive suffixes [13]. Another important property of the suffix array is that it supports queries of any length using a single index. This makes it more flexible and universal than other popular techniques, such as hash tables.

2.2 A learned index structure for suffix arrays

When performing the binary search algorithm, each iteration requires checking the middle of the current search space. For large genomes, this means that consecutive iterations at the start of the algorithm correspond to distant array positions. Consequently, the algorithm has poor spatial locality and results in many cache misses. While the number of iterations is relatively small (~32 for a mammalian-sized genome), most of the memory accesses result in cache misses that are many times slower than memory accesses with cache hits - e.g., approximately 4ns to access from L1 cache vs 100ns to access from main memory on a modern Intel CPU [20–22]. Therefore, we propose a method which uses a data model so that with a single memory lookup into the model and a small number of efficient arithmetic operations, the initial search space for binary search is significantly reduced, and the cache misses which occur at the beginning of the binary search algorithm can be mostly circumvented.

As described above, learned index structures have been used to replace or augment data structures with a data model which models some properties of the particular data being stored. In the case of suffix arrays, we define for a suffix array SA_T a true mapping R(x) which maps a k-mer x to the set of positions of the suffix array that correspond to suffixes starting with x. From the data, we learn a function P(x), a low-memory and arithmetically efficient approximation of R. Then, for a query k-mer Q, P(Q) gives an approximate position of where in the suffix array Q occurs. By performing this query on every k-mer in T, we can obtain a global error bound E on the predictions, which has the property that for any suffix in T, P(x) gives a position which is no more than E positions away from the nearest value in R(x). For a given k-mer x, we can compute P(x), and if x is present in the suffix array, there will be some suffix array position y in [P(x) - E, P(x) + E] such that the suffix starting at position $SA_T[y]$ starts

with x, and this value of y can be computed using a binary search with an initial range of length 2E + 1 instead of length n (**Figure 1**). Therefore, we seek a model with three properties: the ability to perform predictions quickly, a low memory footprint, and a small error bound across genomes.

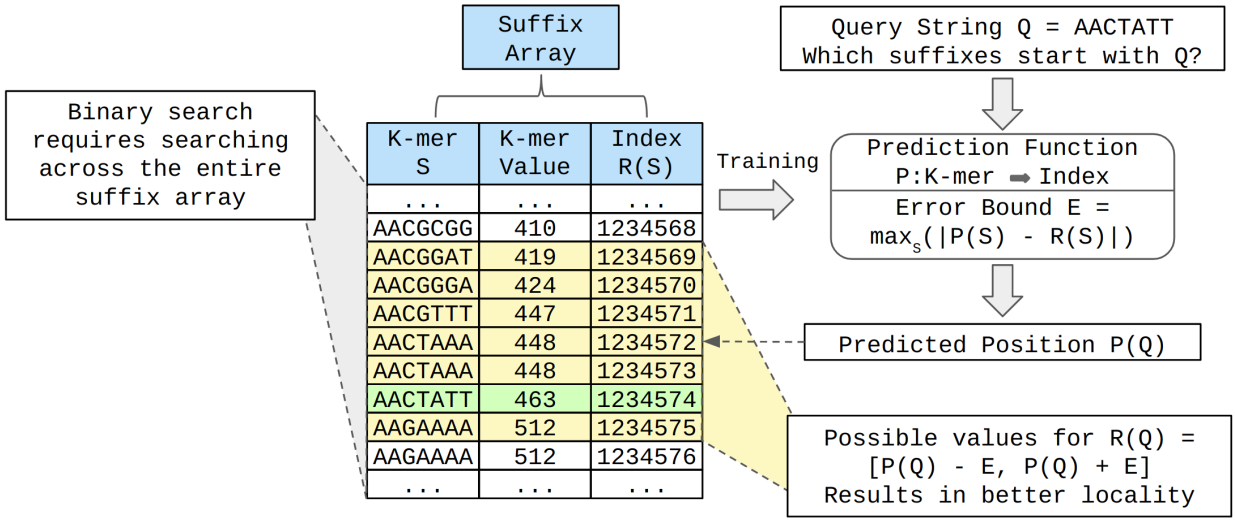


Figure 1. The suffix array lookup can be considered a prediction problem by defining a mapping R(S) which maps a k-mer S encoded as an integer K-mer value to each of the positions in the suffix array corresponding to suffixes starting with S. Learned index structures can approximate this mapping with a function P(S) mapping the k-mer value of each k-mer S to an estimated index, which is trained on the suffix array for a particular dataset using the (S, R(S)) pairs. The maximum error E across all k-mers in the string is computed so that when a particular k-mer Q is queried, if it is present in the string, then at least one of its suffix array positions falls in the range [P(Q) - E, P(Q) + E]. This smaller range can be used for the binary search lookup, resulting in better spatial locality.

2.3 Modeling with Artificial Neural Networks (ANNs)

The first method we explored for modeling the suffix array distribution was using an Artificial Neural Network (ANN) [23] to learn the true mapping R(x). In this approach, we trained ANNs on (k-mer value, suffix array position) pairs, with the goal of using the trained network to predict the approximate suffix array position of a given k-mer **(Figure 2a)**. To ensure that the function being learned is over numeric values, Sapling encodes each k-mer as its k-mer value - an integer with 2k bits. In this conversion, two bits are allocated for each of the k characters, with the two highest-order bits corresponding to the first character and the two lowest-order bits corresponding to the last character. The two bits for a given character are 00 if the character is “A”, 01 for “C”, 10 for “G”, and 11 for “T”. This encoding scheme ensures that any k-mer which comes lexicographically before another will have a smaller integer value, resulting in a simple, monotonically non-decreasing mapping.

For modeling, we first transform the suffix array positions into “residual values” - this detrending is performed by considering a straight line from the first k-mer to the last k-mer (i.e. fitting a linear function to the entire genome, such as plotted in **Figure 2a**), and then computing how each suffix array position differs from this line. The residual values are more easily learned by the ANN since the function will have a smaller range of values to consider. The input data is then unit scaled so that both the k-mers and the suffix array positions are within [0, 1]. We divide the input data into B equal-sized intervals, and

an individual ANN is trained on each of them. For these neural nets, we used a basic “rectangular” architecture consisting of L layers, each with N nodes (aside from the single input node in the first layer and the single output node in the last layer). The networks were fully connected (each node in layer i passed input into every node in layer i+1), with no drop out, with a ReLU activation function [24] applied between layers.

The loss function used was mean squared error (the average of the square of the differences between the predicted suffix array residual position and the true value). We trained the model to minimize this loss function using the Adam optimizer with default PyTorch [25] hyper-parameters (learning rate 0.001, betas = [0.9, 0.999] and epsilon = 1e-8). The training for these models proceeds in epochs, during which the model’s ability to predict the input data is assessed and improved. During each epoch, the current model (using the parameters it has learned up to that point) makes predictions on the input data, and the mean squared error is computed. Based on this error, the parameters in the model are updated through a process called back propagation. To speed up training, we used a batch size of 64 values; this means that the model makes predictions for 64 input values, the mean square error is calculated across these 64 predictions, and the model’s parameters are updated accordingly, before the next batch is loaded. The input data is shuffled at the start, so the batches do not contain consecutive data points.

For training, we set the maximum number of training epochs to be 200. All models were trained for at least 10 epochs, and after this initial period, if a reduction of 10% or more in the value of the loss function was not achieved during the last 10 epochs, the training procedure was terminated to limit wasted work. When the training for a particular neural network ended, the best model across all training epochs was kept and used to predict the suffix array positions for all k-mers in the network’s corresponding interval of k-mer values.

2.4 Modeling with Piecewise Linear Functions (PWL)

An alternative data model we explored is a piecewise (PWL) linear model. In this model, the space of all 4^k possible k-mers is subdivided into a fixed number b equally-sized intervals, where b is a power of 2 to allow fast calculation of which interval each k-mer falls into **(Figure 2b)**. Then, for each interval, the lexicographically earliest k-mer from the genome which is present in that interval is stored along with its corresponding suffix array position. While this idea of “marker” k-mers to limit the range of the suffix array to search has been used previously [6], Sapling improves upon this approach by interpolating the exact suffix array position of the entire k-mer, giving an even smaller interval of candidate positions without further increasing the memory footprint required. If Sapling recognizes that the interpolation mis-predicts the true position of the query, Sapling will dynamically adjust the range to cover a larger range so that the correct result is guaranteed to be computed with only a modest time penalty (see PWL Implementation below).

In the algorithm used by Sapling, the prediction P(s) is computed as follows:

1. Calculate which interval x is in from its $\log_2(b)$ highest-order bits.
2. Look up the pair (x_1, y_1) corresponding to the earliest k-mer in the same interval and the pair (x_2, y_2) corresponding to the earliest k-mer in the next interval.
3. Consider a line segment between (x_1, y_1) and (x_2, y_2), and output the y-value which corresponds to an x-value of s.

This simple model allows very efficient queries consisting of looking up two pairs which are adjacent in memory followed by a small number of arithmetic operations. The memory footprint is parameterized on the number of intervals, storing two 64-bit integers per interval, and we show that even with a relatively small number of intervals, small error bounds can be achieved across different genomes. For these reasons we use this data model in our implementation.

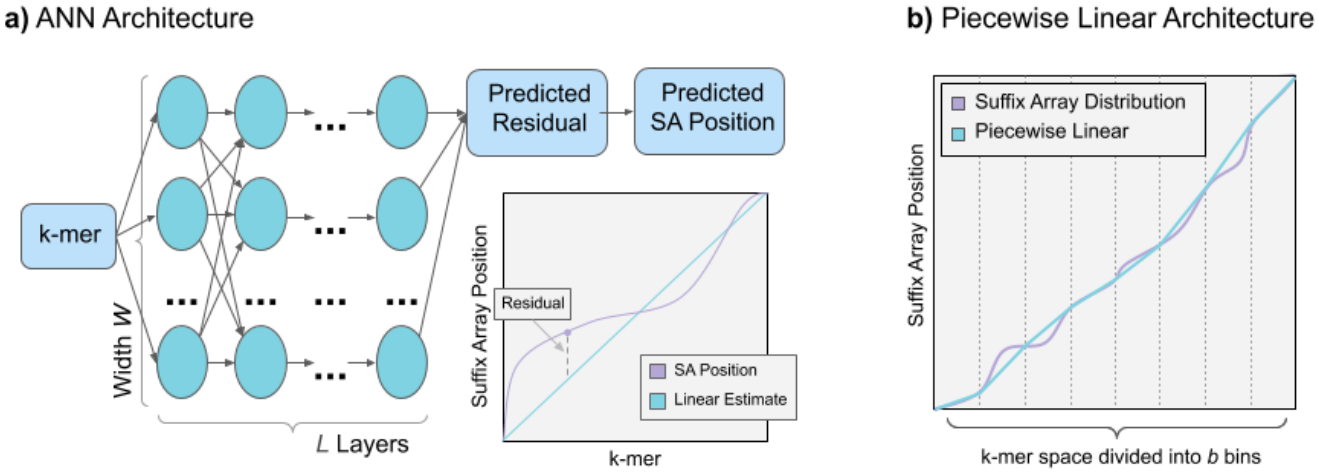


Figure 2. a) Schematic diagram of ANN architecture. An input k-mer encoded using a simple binary encoding scheme is passed to a fully connected ANN with L layers, each with width W. The output value from the ANN is the predicted residual value, which is then projected to the actual suffix array position using a linear transformation. In practice, we use multiple ANNs that each learn the distribution of a portion of the k-mer space (not shown). **b) Schematic diagram of Piecewise Linear model.** The piecewise linear model divides the space of possible inputs (k-mers encoded as integers) into b equal-sized intervals. It stores representative data points from each interval (those with the lowest k-mer values) and connects points in consecutive intervals with line segments. Then, when estimating the suffix array position for a particular k-mer, the linear function between that k-mer's interval and the following interval is used to estimate the suffix array position.

2.5 PWL Implementation

When dividing the space of possible k-mers into buckets (intervals), the partitioning is done in such a way that each group has the same number of possible k-mers. However, in practice, due to varying k-mer frequencies, it is possible for some buckets to have particularly small or large sections of the suffix array contained in them. The buckets with many points, indicative of repetitive sequences in the genome, often have particularly poor predictions, and this causes the maximum errors to be much worse than the median errors or even the 95th percentile errors (see **Results**). To avoid binary searching over a range which is almost always much larger than necessary, Sapling uses an additional cutoff. Once the predictions have been made for every k-mer in the genome, in addition to storing the maximum error in each direction, Sapling also stores the 95th percentiles of the errors in each direction. Then, when searching for a particular k-mer given its predicted position, rather than immediately executing the binary search algorithm, Sapling first checks the position corresponding to an error equal to the 95th percentile in the appropriate direction. Then, in 95% of cases, the size of the search range can be immediately reduced to the 95th percentile error, which is typically much smaller than the maximum error, further improving performance.

When using Sapling, it is assumed that the size of k-mers used when constructing the index is equal to the length of the k-mers being queried (k = 21 in our experiments). However, for some applications, the index will be searched for queries of alternative or varying lengths (both smaller or larger values). The suffix array prediction function can be evaluated with similar speed for such strings without rebuilding the model as follows (**Figure 4**):

- ➢ If the query length q is less than the Sapling k-mer size k: Pad the end of the query with A's (the lexicographically smallest value). The k-mer value can be padded in this way quickly by bit-shifting the k-mer value 2*(k-q) bits to the left.
- ➢ If the query length q is greater than the Sapling k-mer size k: Let the k-mer value of the length-k prefix of the query be v. Then, set the k-mer value of the query as a floating-point value between v and v+1 based on the remaining characters and evaluate the piecewise linear function at that value.

Sapling is available as open-source software on Github (https://github.com/mkirsche/sapling), and provides a succinct library for constructing the piecewise linear data model and using it to perform suffix array lookups. We also implemented a simple seed-and-extend aligner as a proof-of-concept which uses Sapling for seeding and the Striped-Smith-Waterman algorithm [26] for extending seeds into full alignments. This aligner accepts fasta and fastq formatted files as input and outputs alignments in SAM format [27].

3. Results

3.1 Suffix Array Distribution

We tested Sapling on six diverse reference genome sequences: *E. coli*, *C. elegans*, *S. lycopersicum* (tomato), human (both chromosome 1 in isolation and the full human reference), and *T. aestivum* (wheat) (**Supplemental Table 1, Supplemental Figure 1)**. While the function we are trying to approximate is monotonically non-decreasing, there are many potential functions that can emerge based on the composition of the suffix array. While the suffix array for a random string will result in approximately a straight line, repetitiveness and biological selection against certain sequences [28] can drastically affect the nature of the function (**Supplemental Figures 2-3**). Therefore, we investigated the true suffix array position functions for each of these genomes to ensure that the functions are learnable across species. **Figure 3** shows the true Suffix Array Distributions for each of the six reference genomes listed above.

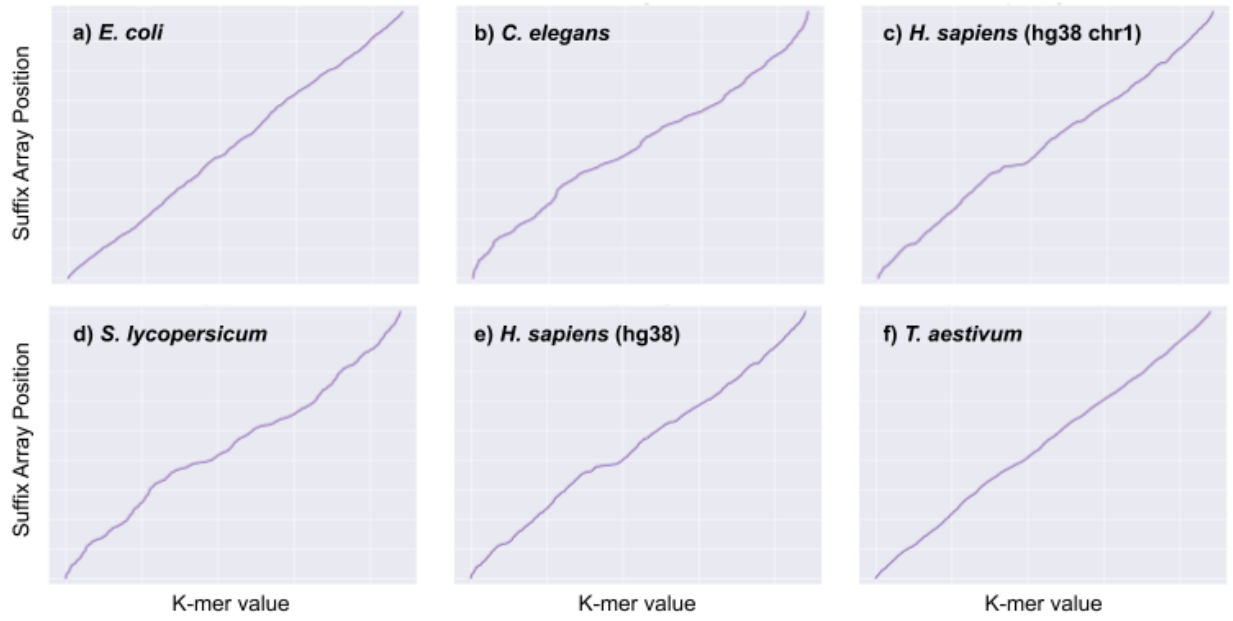


Figure 3. Suffix Array Distribution for 6 genome sequences: *E. coli*, *C. elegans* (nematode), *H. sapiens* (chr1), *S. lycopersicum* (tomato), *H. sapiens* (all of hg38), and *T. aestivum* (wheat).

3.2 Model Training and Accuracy

In testing the feasibility of different models, we measured the prediction accuracy of several potential PWL and ANN models on human chromosome 1 (**Supplemental Tables 2 & 3)**. **Table 1** describes the characteristics of a selection of model architectures as well as their memory footprints. For the ANN, most bins are trained within 40-60 epochs, although a few particularly complex bins require up to 180 epochs until convergence requiring more than 1 day of training on an NVIDIA Quadro P5000 GPU (**Supplemental Figure 4**). For each model, we calculated the prediction error for every k-mer present in the genome, defined as the absolute difference between the predicted suffix array position and the nearest position which corresponds to a suffix starting with the query. The mean, median, and maximum errors were computed both within each bin and genome-wide. By studying each bin individually, we were able to highlight cases where the learned function modelled the suffix array position function particularly well or poorly (**Supplemental Figures 5 & 6**). In particular, for all of the genomes we studied, the first and last bins had particularly high prediction errors caused by the high relative frequencies of homopolymer A and T sequences in the genomes that challenged the PWL model.

We found that increasing the width of the ANN used for each bin in the model resulted in improved performance, without adding much overhead. However, we found that while increasing the depth (number of layers) of each ANN in the model resulted in performance increases, it added significant memory overhead. This leads us to conclude that utilizing shallower, wider nets is the most efficient way to approach this problem. Overall, the PWL model had improved median and 95% percentile accuracy compared to the ANN model, especially when considering the memory overhead involved, although the ANN model had a lower maximum error.

Table 1. Summary of performance and model complexities for several PWL and ANN architectures analyzing human chromosome 1 (length 230 Mbp). Memory overhead refers to the amount of space required for the data model and is in addition to the requirements for a standard suffix array lookup (i.e., the genome, suffix array, and LCP array).

Model Type	Piecewise Linear	Piecewise Linear	Piecewise Linear	Neural Network	Neural Network	Neural Network
Number of Buckets	16k	256k	2m	1k	16k	16K
Width x Depth	N/A	N/A	N/A	32 x 1	32 x 1	128 x 2
Median Error	899	68	14	900	131	56
95th Percentile Error	7,658	1,579	653	4,238	853	463
Maximum Error	263,165	180,453	135,664	45,839	24,081	13,264
Memory Overhead	256 KB	4 MB	32 MB	8 MB	131 MB	1245 MB

3.3 Runtime analysis

Based on the accuracy results above, along with the very fast numerical computations for the PWL, we implemented Sapling to use the PWL data model to accelerate suffix array queries. We then compared the performance of Sapling using different numbers of intervals to a number of existing alignment algorithms (**Figure 4, Supplemental Tables 4 & 5, Supplemental Note 1**). For this, we implemented a string-optimized binary search, the asymptotically optimal algorithm for searching a suffix array [13]. We also ran the widely-used Bowtie [29] and Mummer4 [10] short read aligners in their exact-matching modes to obtain a fair comparison to Sapling's performance. For each aligner, we measured the amount of time needed to perform 50 million queries on the human genome, where each query is a random 21-mer which is known to occur in the genome, ignoring the time required for indexing. This indexing time was 47 minutes for PWL, but is amortized across all queries and experiments which use the index so can be effectively ignored. For consistency, all tools were run to only consider the forward strand of the genome. See **Supplemental Note 1** for the exact commands used, and **Supplemental Figures 7-8** for an in-depth comparison of Sapling to Bowtie with different sampling frequencies and the memory usage of each tool.

For this analysis, we trained Sapling to also use 21-mers to focus the analysis on the advantages of the data model without the interpolation across kmer lengths. For the runtime experiments, we used a single core of an isolated 2.5 GHz Haswell node with 128 GB of RAM to minimize variation in runtime, except for the experiments on the larger *T. aestivum* genome, which were run on a tmpfs ramdisk with 1 TB of RAM using a single core of an Intel(R) Xeon(R) CPU E7-8860 server at 2.20 GHz. As expected, we see the runtime performance of Sapling improves as the number of intervals increases. In an ideal case, with a perfect prediction function, the number of suffix arrays lookups would be decreased from

$\log_2(n)$ - approximately 32 for the human genome - to a single lookup at the predicted position. Our model is able to reduce the search range to a few thousand rows, reducing the number of lookups to about 10 for most queries. This results in an algorithm which is more than 3 fold faster than the string-optimized binary search and nearly 6.5 fold faster than bowtie when used with the largest number of intervals.

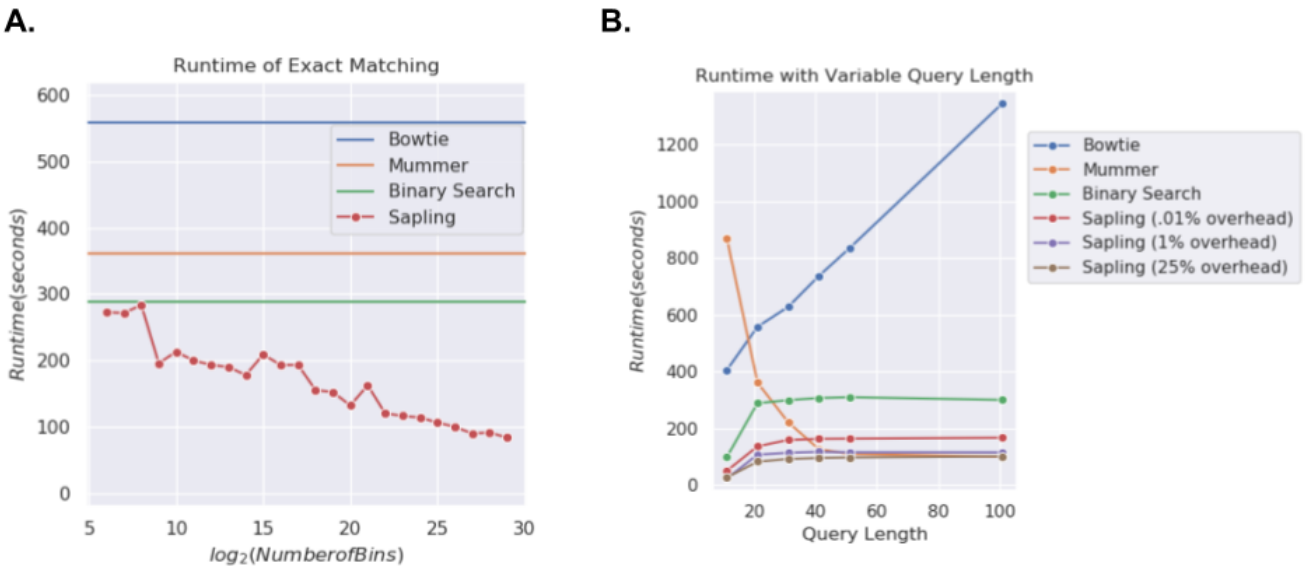


Figure 4. Runtime of different methods to locate 50 million k-mers in the human genome. The queries were sampled randomly from those which occur at least once in the human genome, and the same queries were used for all methods. In addition to running Bowtie (exact matching only), Mummer4 (exact matching only), and a string-optimized binary search algorithm, Sapling was run with several different settings, limiting the number of buckets (and therefore the memory overhead) to various proportions of the size of the human genome. In (**A**), the size of the query k-mers was always set to 21. In (**B**), the query length varied while the same model was used for Sapling (trained on 21-mers in the human genome). This illustrates that Sapling performs well even on queries whose lengths differ from that of the training set. Note that as the query length increases, the runtime of Bowtie scales approximately linearly with length due to its use of an FM-index that processes the query one character at a time, while Mummer grows faster due to the increasing uniqueness of longer queries.

In addition to measuring across model architectures and between different aligners, we also measured how well the runtime of Sapling scales when the genome size is increased. To measure this, we ran Sapling on six different reference genomes of different sizes, and for each genome measured the amount of time required to query five million random k-mers which are present in the genome. We performed a similar experiment for the string-optimized binary search. We find that as the genome size increases, the reduction in runtime from using a data model increases substantially (**Figure 5, Supplemental Tables 6 & 7**).

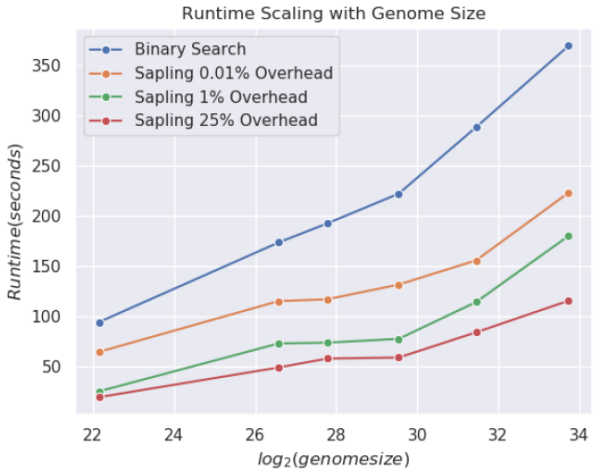


Figure 5. Runtime of Sapling and binary search across six different genomic sequences using 0.01%, 1% or 25% space overhead.

4. Discussion

In this paper, we presented Sapling, a novel algorithm for quickly performing suffix array lookups for use within read alignment and genome alignment algorithms. Sapling uses learned index structures to model the contents of the suffix array as a function rather than as a data structure, and uses a practical piecewise linear model to efficiently approximate this function. Using this method shows significant improvement in the runtime of querying many different genomes, demonstrating that even a simple low-memory piecewise linear approximation of the suffix array position function is sufficient for achieving several-fold improved performance compared to existing tools with modest space overhead. As read and genome alignment is performed on even larger genomes and larger collections of genomes, the need for efficient substring search algorithms becomes even more pressing, and Sapling will be able to scale better to large reference sizes than existing query algorithms.

While this work demonstrates the potential for learned index structures in a very important and widely used genomic application, there remain many possible avenues for future development. Presently, the prototype read aligner uses a basic seed-and-extend implementation that requires additional development to make it competitive with existing aligners for inexact alignment. There are also possible avenues for improving the core algorithm of Sapling, such as by using a different prediction function or non-uniform intervals for the piecewise linear function. In addition, Sapling could be used for modeling other full text index data structures, especially sparse versions of the suffix array [30] or the FM-index, or other data structures entirely. Finally, read alignment is just one of the many data-intensive problems in genomics that requires the efficient use of large data structures. We are investigating other genomic applications of the learned index structures paradigm, including optimized graph representations for genome and pan-genome assembly, optimized variant databases, and other data intensive problems.

Acknowledgements

We would like to thank Alex Dobin and Benjamin Langmead for their helpful discussions. This research project was conducted using computational resources at the Maryland Advanced Research Computing Center (MARCC).

Funding

This work was supported by the National Science Foundation [DBI-1350041, IOS-1445025, IOS-1732253, and IOS-1758800 to MCS].

Conflict of Interest: none declared.

References

1. Nielsen R, Paul JS, Albrechtsen A, Song YS. Genotype and SNP calling from next-generation sequencing data. Nat Rev Genet. 2011;12:443–51.

2. Wang Z, Gerstein M, Snyder M. RNA-Seq: a revolutionary tool for transcriptomics. Nat Rev Genet. 2009;10:57–63.

3. Park PJ. ChIP-seq: advantages and challenges of a maturing technology. Nat Rev Genet. 2009;10:669–80.

4. Soon WW, Hariharan M, Snyder MP. High-throughput sequencing for biology and medicine. Mol Syst Biol. 2013;9:640.

5. Baeza-Yates RA, Perleberg CH. Fast and practical approximate string matching. Inf Process Lett. 1996;59:21–7.

6. Dobin A, Davis CA, Schlesinger F, Drenkow J, Zaleski C, Jha S, et al. STAR: ultrafast universal RNA-seq aligner. Bioinformatics. 2013;29:15–21.

7. Langmead B, Salzberg SL. Fast gapped-read alignment with Bowtie 2. Nat Methods. 2012;9:357–9.

8. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM [Internet]. arXiv:1303.3997 [q-bio.GN]. 2013. Available from: http://arxiv.org/abs/1303.3997

9. Sedlazeck FJ, Rescheneder P, Smolka M, Fang H, Nattestad M, von Haeseler A, et al. Accurate detection of complex structural variations using single-molecule sequencing. Nat Methods. 2018;15:461–8.

10. Marçais G, Delcher AL, Phillippy AM, Coston R, Salzberg SL, Zimin A. MUMmer4: A fast and versatile genome alignment system. PLoS Comput Biol. 2018;14:e1005944.

11. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215:403–10.

12. Charras C, Lecroq T. Handbook of Exact String Matching Algorithms. King's College; 2004.

13. Manber U, Myers G. Suffix Arrays: A New Method for On-Line String Searches. SIAM J Comput. Society for Industrial and Applied Mathematics; 1993;22:935–48.

14. Weiner P. Linear pattern matching algorithms. 14th Annual Symposium on Switching and Automata Theory (swat 1973). 1973. p. 1–11.

15. Karp RM, Rabin MO. Efficient randomized pattern-matching algorithms. IBM J Res Dev. 1987;31:249–60.

16. Ferragina P, Manzini G. Opportunistic data structures with applications. Proceedings 41st Annual

Symposium on Foundations of Computer Science. 2000. p. 390–8.

17. Chaisson MJ, Tesler G. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. BMC Bioinformatics. 2012;13:238.

18. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The Case for Learned Index Structures [Internet]. arXiv:1712.01208 [cs.DB]. 2017. Available from: http://arxiv.org/abs/1712.01208

19. Ho D, Ding J, Misra S, Tatbul N, Nathan V, Vasimuddin, et al. LISA: Towards Learned DNA Sequence Search [Internet]. arXiv:1910.04728 [cs.DB]. 2019. Available from: http://arxiv.org/abs/1910.04728

20. Brett B. Memory Performance in a Nutshell [Internet]. Intel. 2016 [cited 2020 Mar 20]. Available from: https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell

21. Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual [Internet]. 2016. Available from: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

22. 7-Zip LZMA Benchmark [Internet]. [cited 2020 Mar 20]. Available from: https://www.7-cpu.com/

23. Cybenko G. Approximation by superpositions of a sigmoidal function. Math Control Signals Systems. 1989;2:303–14.

24. Ramachandran P, Zoph B, Le QV. Searching for Activation Functions [Internet]. arXiv:1710.05941 [cs.NE]. 2017. Available from: http://arxiv.org/abs/1710.05941

25. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach H, Larochelle H, Beygelzimer A, d\textquotesingle Alché-Buc F, Fox E, Garnett R, editors. Advances in Neural Information Processing Systems 32. Curran Associates, Inc.; 2019. p. 8024–35.

26. Zhao M, Lee W-P, Garrison EP, Marth GT. SSW library: an SIMD Smith-Waterman C/C++ library for use in genomic applications. PLoS One. 2013;8:e82138.

27. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, et al. The Sequence Alignment/Map format and SAMtools. Bioinformatics. 2009;25:2078–9.

28. Herold J, Kurtz S, Giegerich R. Efficient computation of absent words in genomic sequences. BMC Bioinformatics. 2008;9:167.

29. Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10:R25.

30. Vyverman M, De Baets B, Fack V, Dawyndt P. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. Bioinformatics. 2013;29:802–4.

Sapling: Accelerating Suffix Array Queries with Learned Data Models

Melanie Kirsche, Arun Das, Michael C. Schatz

Supplemental Materials

Supplemental Table 1. Genome sequences analyzed in this study.

Note that all 'N' characters were removed from the sequences prior to indexing.

Species Name	Genome Size	Accession
E. coli K-12 substr. MG1655	4,641,652	GCA_000005845.2
C. elegans	100,286,401	GCA_000002985.3
S. lycopersicium	782,475,302	SL4.0 (https://www.biorxiv.org/content/10.1101/767764v1)
H. sapiens (hg38 chr1)	230,481,012	GCA_000001405.15
H. sapiens (hg38 whole genome)	2,934,876,451	GCA_000001405.15
T. aestivum	14,271,578,887	GCA_900519105.1

Supplemental Table 2. Prediction errors for the piecewise linear data model.

This table shows a number of error statistics for different numbers of bins when using the piecewise linear model to predict the suffix array positions of all 21-mers in human chromosome 1.

Number of Bins	Max Error	95th Percentile	Median Error	Mean Error	Median of medians	Max of medians
2^0	12,987,798	11,299,286	6,899,176	6,502,090	6,899,176	6,899,176
2^10	426,608	59,211	11,590	19,578	9,429	273,216
2^14	263,165	7,658	899	2,533	537	163,013
2^18	180,453	1,579	68	627	27	126,588
2^20	150,267	828	19	395	7	108,158
2^21	135,664	653	14	335	4	103,207
2^22	124,792	486	6	270	2	88,840
2^23	111,458	392	4	234	1	84,460
2^24	102,668	314	2	194	1	71,826
2^25	90,667	257	2	170	1	67,470
2^26	83,653	221	1	142	0.5	56,666
2^27	72,889	187	1	127	0	52,476

Supplemental Table 3. Predictions errors for ANN modeling.

This table shows a number of error statistics for different ANN architectures to predict the suffix array positions of all 21-mers in human chromosome 1.

# Buckets	# Nodes	# Layers	Median of Means	95th %tile of Means	Max of Means	Median of Medians	95th %tile of Medians	Max of Medians	Median of Maxes	95th %tile of Maxes	Max of Maxes	Total Size (MB)
2^10	8	1	3,082	9,259	27,850	2,432	8,425	26,627	10,709	26,532	91,606	8
	32	1	1,194	2,559	11,042	892	1,792	4,752	5,434	13,088	45,839	8
	128	1	713	1,879	10,165	520	1,102	4,065	3,775	12,551	50,428	9
	8	2	1,888	5,353	235,792	1,406	4,433	229,429	7,611	19,372	523,256	9
	32	2	683	1,674	9,174	492	1,082	3,977	3,799	12,120	38,627	13
	128	2	435	1,318	8,868	308	696	3,677	3,257	12,144	36,495	76
	8	4	1,197	4,904	106,727	835	4,190	102,676	5,730	21,838	250,781	10
	32	4	435	1,318	8,868	308	696	3,677	3,257	12,143	36,495	22
	128	4	332	1,336	9,790	208	622	4,976	3,217	12,332	51,049	204
2^14	8	1	302	1,228	3,494	245	1,116	3,516	967	3,394	34,965	131
	32	1	164	757	3,458	128	494	3,452	603	2,699	24,081	131
	128	1	105	641	3,463	29	334	3,452	432	2,475	22,944	147
	256	1	92	625	3,491	70	325	3,452	394	2,453	9,079	180
	512	1	86	630	3,817	66	370	3,813	373	2,439	23,573	229
	8	2	211	987	35,024	163	826	34,404	736	3,002	67,349	147
	32	2	113	586	3,456	85	324	3,453	467	2,341	8,687	212
	128	2	71	505	3,448	52	252	3,452	325	2,224	13,264	1,245
	8	4	154	1,015	43,595	115	808	47,065	597	3,111	83,259	163
	32	4	72	463	3,460	53	205	3,452	349	2,199	7,218	360
	128	4	70	390	3,453	30	183	3,452	221	1,937	7,061	3,342
2^18	128	1	128	129	223	2	18	220	61	127	642	2,359

Supplemental Table 4. Runtime of different methods querying the human genome.

Each method was used to locate 50 million 21-mers that are known to occur in the genome. Speed up is computed relative to the suffix array binary search. **Supplemental Table 5** displays additional results for Sapling. All tools were tested using the forward strand only.

Tool	Runtime (s)	Speed up over binary search
Bowtie (exact match only)	558	.52x
Mummer4 (exact match only)	361	.80x
Suffix Array Binary Search	288	1x
Sapling (0.01% memory overhead)	155	1.86x
Sapling (1% memory overhead)	114	2.53
Sapling (25% memory overhead)	84	3.43x

Supplemental Table 5. Runtime of Sapling on the human genome.

Each experiment was repeated 3 times to minimize the impact of contention on the server. The minimum recorded value is highlighted in green. Each run evaluates the time required to query 50 million randomly selected 21-mers known to occur in the genome.

$\log_2$ (number of bins)	Trial 1	Trial 2	Trial 3
6	316.336	272.504	323.805
7	271.494	296.254	316.643
8	304.494	283.534	300.19
9	195.717	225.044	222.746
10	214.702	213.098	213.15
11	204.405	212.146	199.751
12	196.17	193.502	194.178
13	190.024	190.223	190.647
14	178.008	178.424	180.041
15	213.982	208.957	223.68
16	214.863	193.096	250.618
17	266.878	208.551	193.263
18	155.578	156.305	155.422
19	152.37	152.434	152.319
20	132.156	144.372	132.068
21	192.875	174.58	162.492
22	153.798	161.847	120.197
23	144.868	155.88	116.291
24	132.176	150.189	113.859
25	145.011	136.265	106.215
26	171.781	129.758	99.927
27	131.532	95.7156	89.7132
28	136.927	91.2241	91.4228
29	106.476	84.4354	83.7056

Supplemental Table 6. Runtime of binary search on six genomic sequences.

Each experiment was repeated 3 times to minimize the impact of contention on the server. The minimum recorded value is highlighted in green. **Supplemental Table 7** shows the results for these experiments with Sapling.

Genome	Runtime Trial 1 (seconds)	Runtime Trial 2 (seconds)	Runtime Trial 3 (seconds)
E.coli	94.0599	93.9207	93.8439
C. elegans	173.31	174.973	174.904
H. sapiens (hg38, chr1)	198.168	192.317	196.619
S. lycopersicum	222.014	222.038	222.762
H. sapiens (hg38)	300.853	288.3	290.255
T. aestivum	369.465	384.347	372.748

Supplemental Table 7. Runtime of Sapling on six genomic sequences.

Each experiment was repeated 3 times to minimize the impact of contention on the server. The minimum recorded value is highlighted in green.

Genome	Sapling Overhead	Runtime Trial 1 (seconds)	Runtime Trial 2 (seconds)	Runtime Trial 3 (seconds)
E. coli	0.01%	73.147	72.86	64.1779
	1%	25.3867	24.7783	24.7871
	10%	19.6079	19.5705	20.4445
	25%	21.6279	19.0606	19.007
C. elegans	0.01%	115.703	114.843	118.653
	1%	72.6475	72.6565	72.6055
	10%	64.3481	53.6948	53.525
	25%	48.9586	48.6434	48.772
H. sapiens (hg38, chr1)	0.01%	122.171	122.2	116.667
	1%	79.8895	74.3822	73.4093
	10%	65.5379	60.7652	60.0477
	25%	58.9982	58.8076	57.5504
S. lycopersicum	0.01%	146.421	138.54	131.352
	1%	78.8507	77.2295	77.7648
	10%	60.7451	60.2346	61.7996
	25%	58.7461	61.1307	58.5787
H. sapiens (hg38)	0.01%	155.578	156.305	155.422
	1%	132.176	150.189	113.859
	10%	136.927	91.2241	91.4228
	25%	106.476	84.4354	83.7056
T. aestivum	0.01%	222.989	273.322	232.971
	1%	179.87	187.326	281.909
	10%	123.117	132.654	123.919
	25%	136.917	181.883	115.208

Supplemental Table 8. Runtime of Sapling on variable length queries in human

Each experiment was repeated 3 times to minimize the impact of contention on the server. The minimum recorded value is highlighted in green. **Supplemental Table 9** shows the results for these experiments with other methods.

Memory Overhead	Query Length	Training Length	Runtime Trial 1 (seconds)	Runtime Trial 2 (Seconds)	Runtime Trial 3 (Seconds)
0.01%	11	21	49.4984	50.1662	50.3691
0.01%	21	21	155.966	142.324	135.788
0.01%	31	21	163.403	158.525	159.401
0.01%	41	21	162.852	162.508	162.19
0.01%	51	21	163.817	162.996	163.493
0.01%	101	21	167.868	166.339	166.227
0.01%	31	31	153.798	153.085	153.381
1%	31	31	118.959	119.075	120.235
25%	31	31	92.9942	93.4445	91.0275
1%	11	21	23.1026	22.0612	22.8821
1%	21	21	109.52	108.92	106.697
1%	31	21	113.088	114.4	114.152
1%	41	21	117.975	116.462	116.37
1%	51	21	116.881	118.691	114.941
1%	101	21	114.533	117.298	118.299
25%	11	21	25.8727	27.5915	29.9953
25%	21	21	82.1219	81.8653	81.9577
25%	31	21	91.8358	91.4418	91.242
25%	41	21	94.8901	94.8738	100.267
25%	51	21	97.1682	96.5893	99.3514
25%	101	21	102.804	100.689	100.62

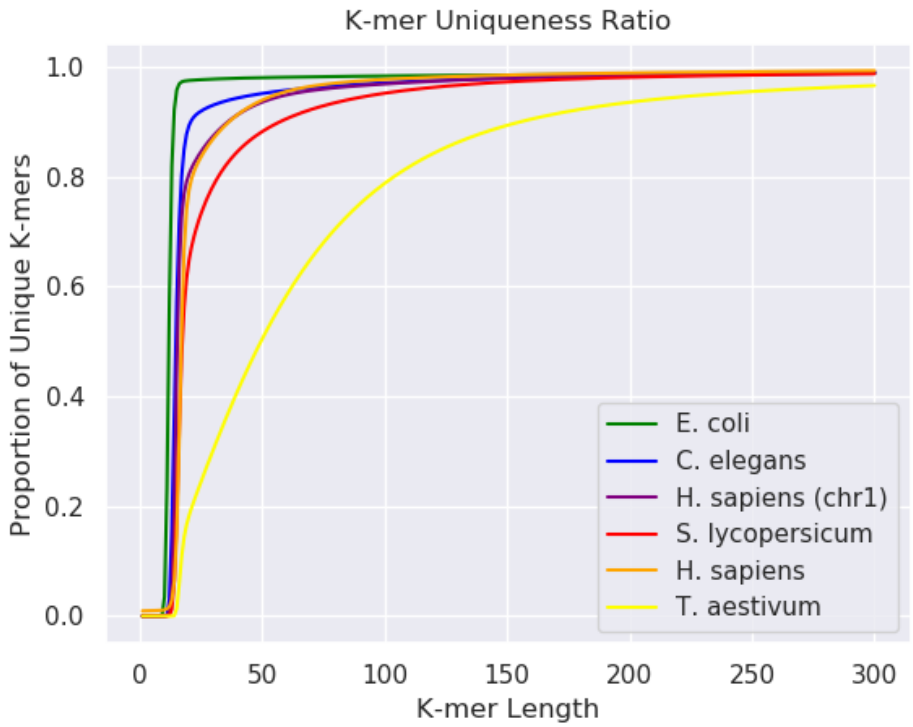
Supplemental Table 9. Runtime of non-Sapling methods on variable length queries in human

Each experiment was repeated 3 times to minimize the impact of contention on the server. The minimum recorded value is highlighted in green. **Supplemental Table 8** shows the results for these experiments with Sapling.

Software	Query Length	Runtime Trial 1 (seconds)	Runtime Trial 2 (Seconds)	Runtime Trial 3 (Seconds)
Bowtie	11	405	415	414
Bowtie	21	649	575	558
Bowtie	31	628	646	629
Bowtie	41	737	743	744
Bowtie	51	839	835	886
Bowtie	101	1351	1365	1347
Mummer	11	872.973	870.457	906.93
Mummer	21	377.273	369.436	360.728
Mummer	31	239.556	220.997	221.86
Mummer	41	166.536	136.299	126.001
Mummer	51	128.036	124.269	109.334
Mummer	101	123.418	121.913	99.256
Binary Search	11	103.214	98.23	97.93
Binary Search	21	300.853	288.3	290.255
Binary Search	31	298.761	301.723	301.78
Binary Search	41	307.782	308.138	306.561
Binary Search	51	310.951	309.613	311.527
Binary Search	101	315.675	299.606	317.609

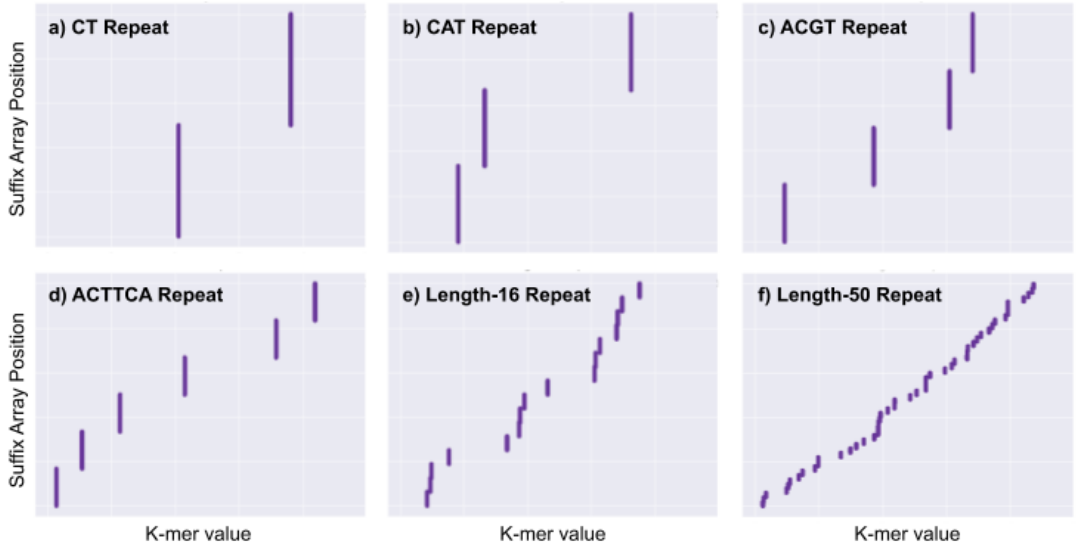
Supplemental Figure 1. K-mer Uniqueness Ratios of Genomes.

For each of the genomes we used, we measured their repetitiveness by plotting how their k-mer uniqueness ratio (proportion of k-mers on the forward strand which occur exactly once) changes as a function of the k-mer length k. If the proportion of unique k-mers grows slowly as k increases, this indicates the presence of long repeats.



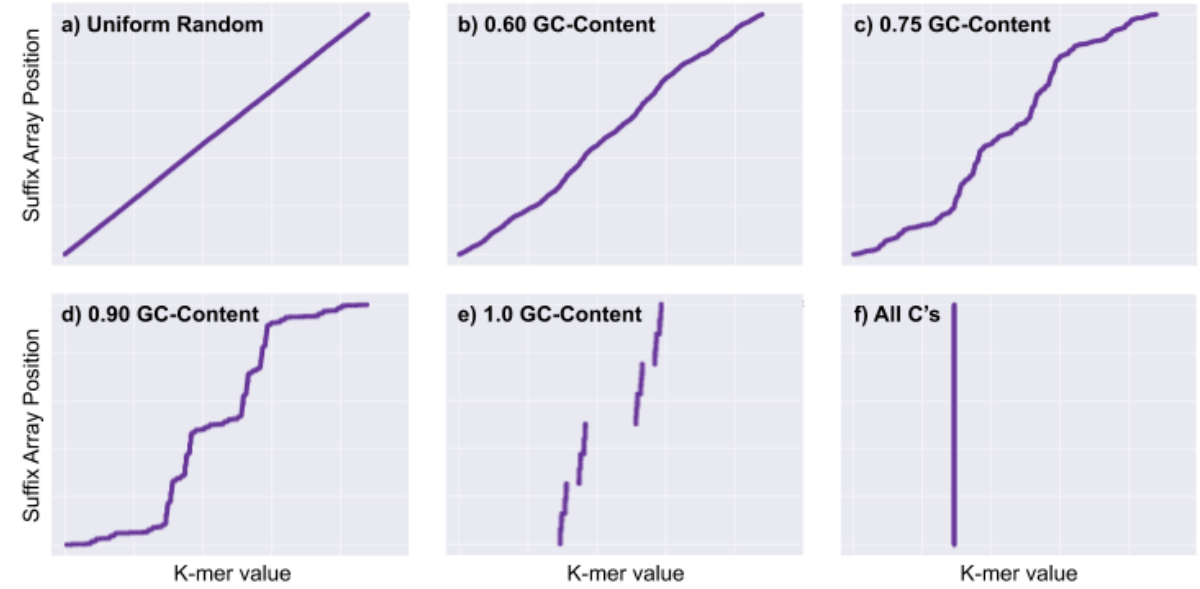
Supplemental Figure 2. Suffix array distributions for simulated repetitive genomes.

To illustrate the effects of repetitive sequence on the suffix array distribution, we computed the suffix array for a number of simulated sequences consisting entirely of repeats of varying length. Each sequence consists of some repeat occurring consecutively for the entirety of the sequence length (set to 100 kbp). **a)** A repeat of the sequence “CT”. **b)** A repeat of the sequence “CAT”. **c)** A repeat of the sequence “ACGT”. **d)** A repeat of the sequence “ACTTCA”. **e)** A repeat of a random length-16 sequence. **f)** A repeat of a random length-50 sequence.



Supplemental Figure 3. Suffix array distributions for simulated genomes with different nucleotide compositions.

To illustrate the effects of variation in sequence content, particularly GC-content, on the suffix array distribution, we computed the suffix array for a number of simulated sequences. Each sequence consists of 100k basepairs, with each basepair independently selected from a fixed probability distribution. **a)** A sequence with 50% GC-content corresponding to an equal probability of each basepair. **b)** A sequence with 60% GC-content. **c)** A sequence with 75% GC-content. **d)** A sequence with 90% GC-content. **e)** A sequence with 100% GC-content. **f)** A sequence consisting of only C's

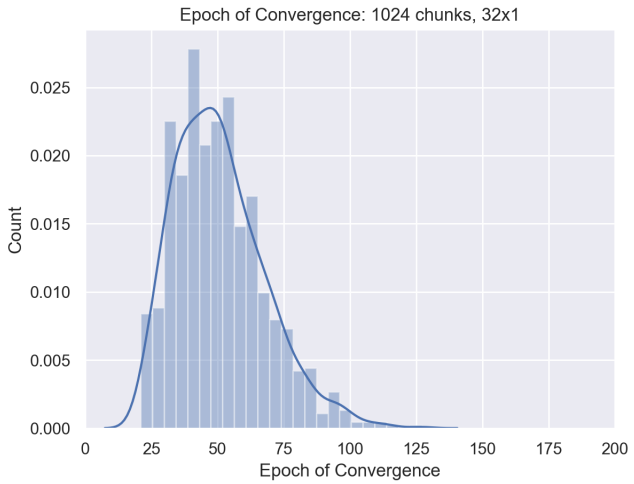


.

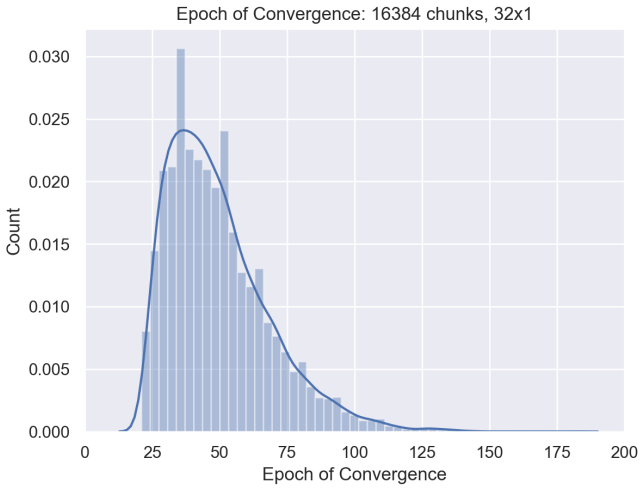
Supplemental Figure 4. Distribution of the number of epochs until convergence.

This figure shows the distribution in the number of epochs until convergence for three ANN model architectures for human chromosome 1.

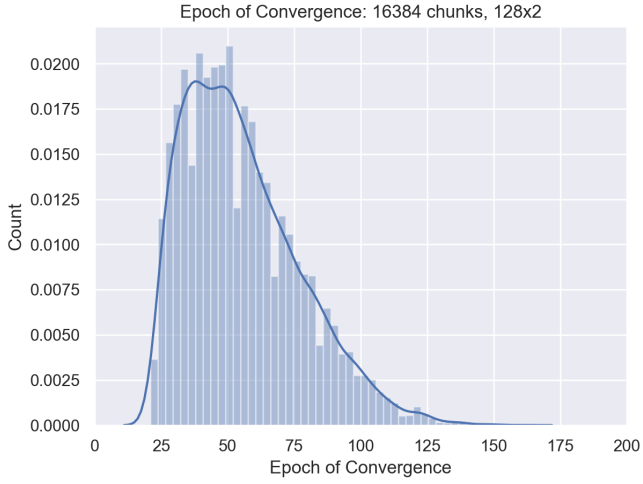
a) 1024 chunks, 32Wx1L



b) 16,384 chunks, 32Wx1L

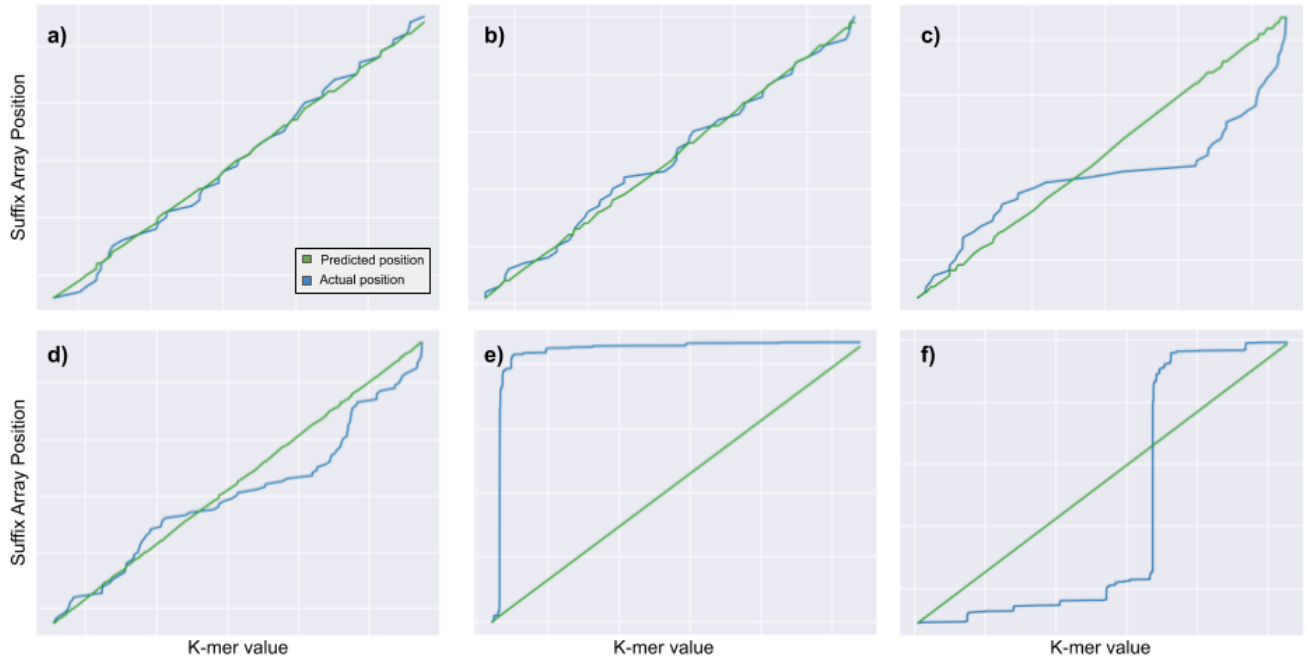


c) 16,384 chunks, 128Wx2L



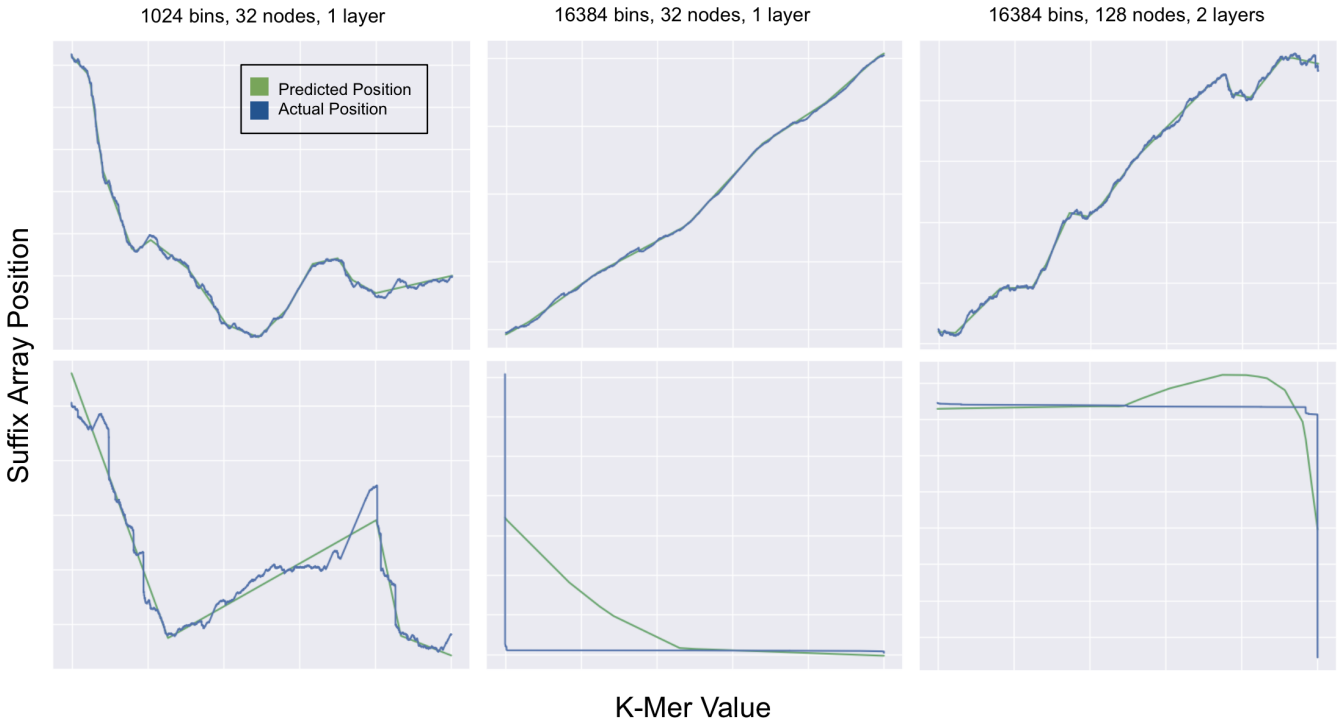
Supplemental Figure 5. Examples of PWL models for selected bins in human chr1.

This figure shows the functions learned by the piecewise linear data model within a few individual segments of human chromosome 1. The total number of segments in this experiment was ~16 million. Panels **a)** and **b)** show bins with the lowest mean error; Panels **c)** and **d)** show bins with average mean error; and Panels **e)** and **f)** show bins with the highest mean error. Blue shows the actual suffix array distribution and green shows the piecewise linear function for this bin.



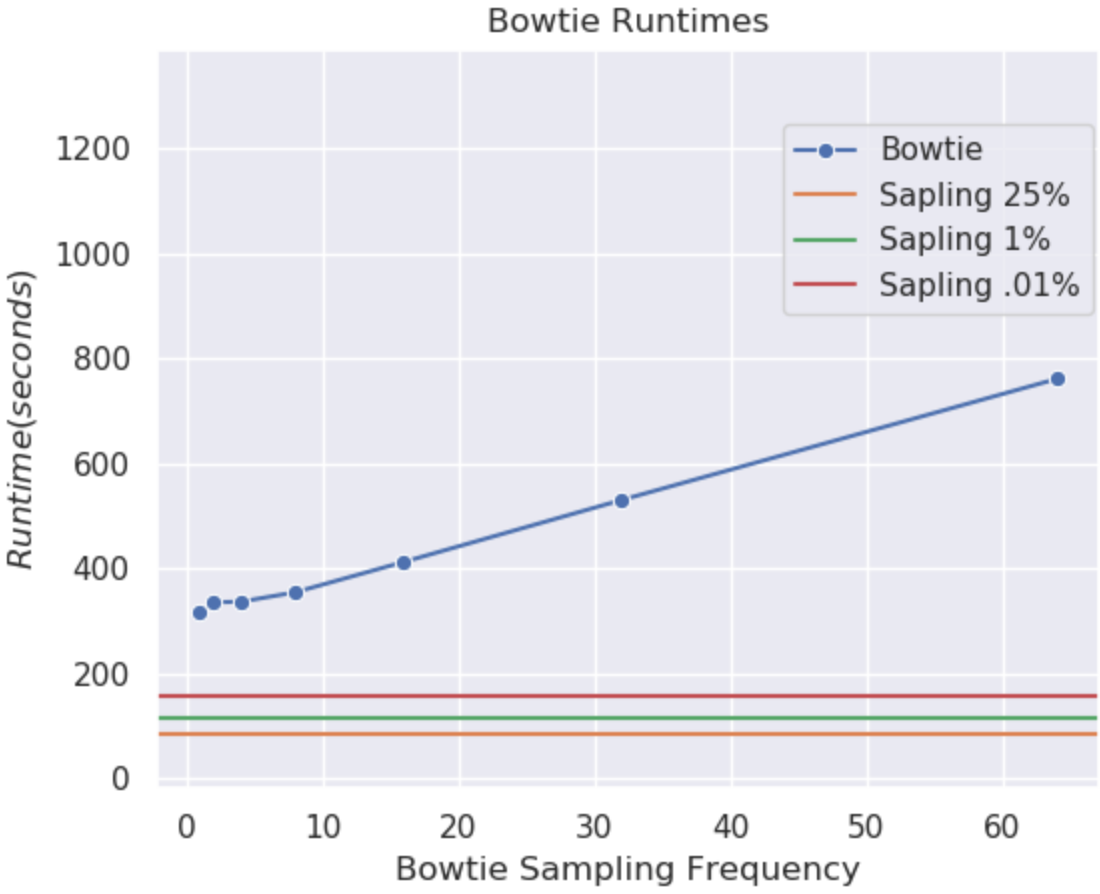
Supplemental Figure 6. Examples of ANN performance for selected bins in human chr1.

This figure shows some of the functions learned by the ANN models within a few bins in human chr1. The first row contains functions that are learned well (low mean error), the second row contains functions that are learned poorly (high mean error). The results shown come from the three ANN architectures highlighted in **Table 1**. Blue shows the actual suffix array distribution and green shows the ANN prediction for this bin.



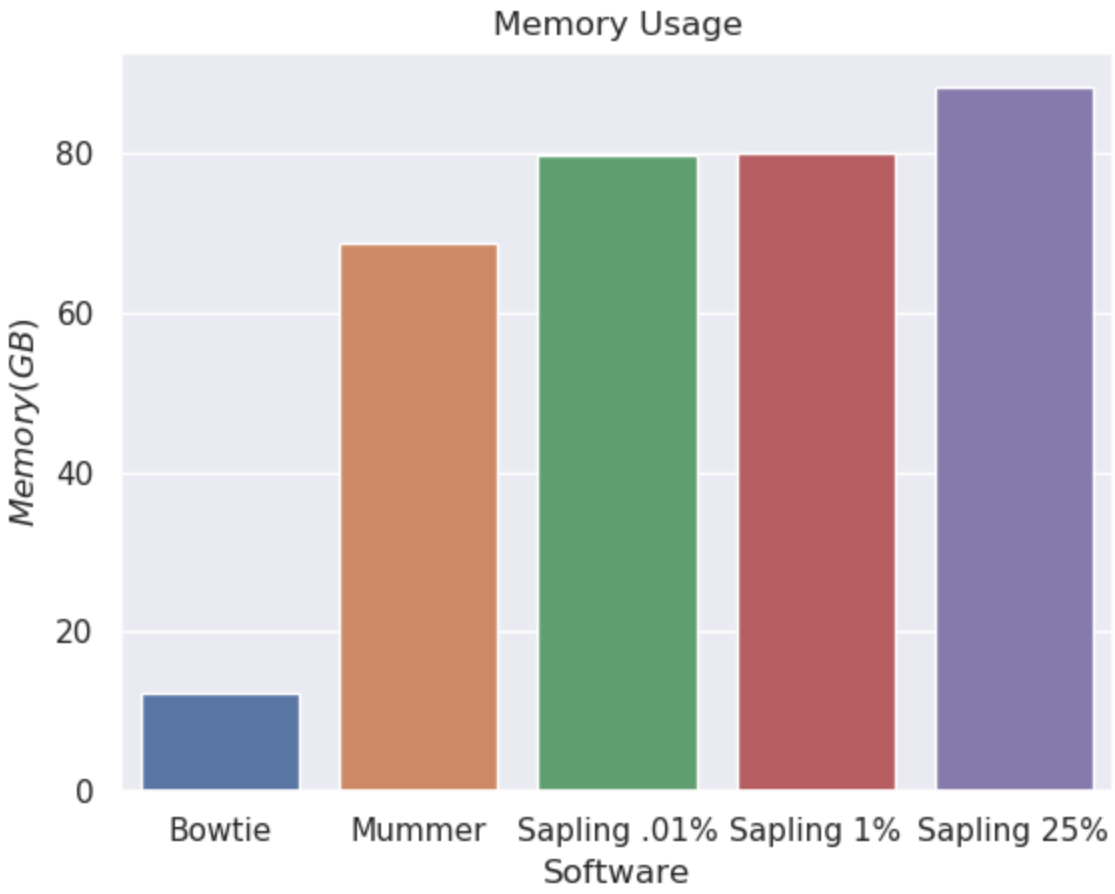
Supplemental Figure 7. Runtime of Bowtie with different sampling frequencies

This figure measures the runtime of Bowtie when different values for the suffix array sampling frequency are used; the default value is 32 and the "*o* =" parameter controls the logarithm (base 2) of this value. For each run we measured the time (not including indexing) required to search the human genome for 50 million 21-mers.



Supplemental Figure 8. Memory usage for querying the human genome

This figure compares the peak memory usage of Sapling, Mummer, and Bowtie when querying the human genome for 50 million 21-mers. For running Bowtie, we set the sampling frequency to 1 to ensure that all tools were consistently using the entire suffix array.



Supplemental Note 1. Commands used for running different aligners

Sapling exact matching:

```
$ suffixarray/refToSuffixArray.sh human.fa
$ src/sapling_example human.fa k=21 saFn=human.fa.sa nb=26 nq=50000000
```

Bowtie exact matching:

```
$ bowtie-build human.fa human
$ bowtie --norc -v 0 -k 1 -t human queries.fastq
```

Mummer 4.0.0 exact matching:

```
$ mummer -threads 1 -save human.mummer -maxmatch -l 21 human.fa \
  queries.fastq
$ time (mummer -threads 1 -load human.mummer -maxmatch -l 21 human.fa \
  queries.fastq | tee mummer.sam)
```

Supplemental Note 2. Algorithm for encoding k-mers as integers

Algorithm 1: Encode Kmers

This function encodes k-mers as integers with 2*k bits such that the numerical ordering of the outputs of the function is equivalent to the lexicographical ordering of the input k-mers.

```
EncodeKmer(kmer):
  Result = 0
  K = kmer.length
  For i = 0 to ( K - 1 ):
    Result = Result * 4
    Char = kmer[i]
    if( Char == 'A' ) Result += 0
    else if( Char == 'C' ) Result += 1
    else if( Char == 'G' ) Result += 2
    else if( Char == 'T' ) Result += 3
  Return Result
```

Supplemental Note 3. Algorithm for querying piecewise linear index

This algorithm queries a piecewise linear index for a particular X-value given that the index divides the space of possible x-values into b equal-sized intervals where FirstX and FirstY are the points with the lowest x-value within each bucket.

```
QueryPiecewiseLinearIndex(FirstX, FirstY, X, b):
  BucketIndex = X / b
  PrevX = FirstX[BucketIndex]
  PrevY = FirstY[BucketIndex]
  NextX = FirstX[BucketIndex + 1]
  NextY = FirstY[BucketIndex + 1]
  Slope = (NextY - PrevY) / (NextX - PrevX)
  Return RoundToInt(PrevY + Slope * (X - PrevX))
```

Supplemental Note 4: Algorithm for building piecewise linear index

This algorithm builds the piecewise linear index for a genome for a given k-mer length and number of buckets to divide the k-mer space into. It defines each k-mer as a point with x-coordinate equal to the encoded k-mer and a y-coordinate equal to the suffix array position corresponding to the start of that k-mer. It then finds the first point in each bucket (logic for handling empty buckets is omitted here), and creates a piecewise linear function by connecting those points. Finally, the maximum over-prediction and under-prediction error are computed.

```
BuildPiecewiseLinearIndex(genome, k, buckets):
  SA = BuildSuffixArray( genome )
  N = genome.length
  FirstX[buckets]
  FirstY[buckets]
  For i = 0 to (N - k + 1):
    X = EncodeKmer( genome.substring(i, i + k))
    Y = SA[i]
    BucketIndex = X / buckets
    If FirstX[BucketIndex] == null || FirstX[BucketIndex] > X:
      FirstX[BucketIndex] = X
      FirstY[BucketIndex] = Y

  MaxOverError = 0
  MaxUnderError = 0

  For i = 0 to (N - k + 1):
    X = EncodeKmer(genome.substring(i, i + k))
    Y = SA[i]
    PredictedY = QueryPiecewiseLinearIndex(X)
    if(PredictedY > Y):
      MaxOverError = max(MaxOverError, PredictedY - Y)
    else:
      MaxUnderError = max(MaxUnderError, Y - PredictedY)

  Return new Sapling(FirstX, FirstY, buckets, MaxOverError, MaxUnderError)
```

Supplemental Note 5: Algorithm for querying Sapling

This algorithm finds the location of a query k-mer in a genome's suffix array given a Sapling piecewise linear index. The runtime of it is O(1) or the Sapling prediction plus O(log([max overprediction] + [max underprediction]) + k) in the worst case, compared to O(log(genome length) + k) in the case of a standard binary search algorithm. Here the standard binary search algorithm is called as a subroutine of the Sapling query with the function signature BinarySearch(suffix array, query string, start of range, end of range).

```
QuerySapling(sapling, Query, SuffixArray, RevSuffixArray):
  X = EncodeKmer(Query)
  PredictedY = QueryPiecewiseLinearIndex(sapling.FirstX,
                                         sapling.FirstY,
                                         X, sapling.buckets)
  MinSaPos = PredictedY - sapling.MaxUnderError
  MaxSaPos = PredictedY + sapling.MaxOverError
  SaPos = BinarySearch(SuffixArray, Query, MinSaPos, MaxSaPos)
  If SaPos == -1:
    Return "Query not Found"
  Return SaPos
```