DNA Search: Learned Super Maximal Exact Match

Carlos Garcia, Jonathan Kelly

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
6.890 Final Project Report
{carlosga, jgkelly}@mit.edu

Abstract

Analyzing DNA frequently requires finding regions of overlap between large genomic sequences and short sequences of interest. Finding these regions of overlap has large implications in the fields of medicine and biology, particularly when it comes to locating genetic diseases or other genetically dependant human conditions. Although the goal is relatively straight forward, finding the matching regions between two sequences, the reality is that this task can be extremely time and space intensive for computers to perform because of the sheer enormity of the human genome. In this paper, we explore the use of a Lookup Table (LUT) and a Recursive Model Index (RMI) to locate super maximal exact matches (SMEMs) between two DNA sequences rapidly and with minimal space to store the genomic data. We compare our approach to the traditional state-of-the-art methods for finding SMEMs and show that there is a speed-space tradeoff that can be taken advantage of using using a LUT or RMI.

Index Terms: DNA Search, Learned Indexes, SMEM

1 Introduction

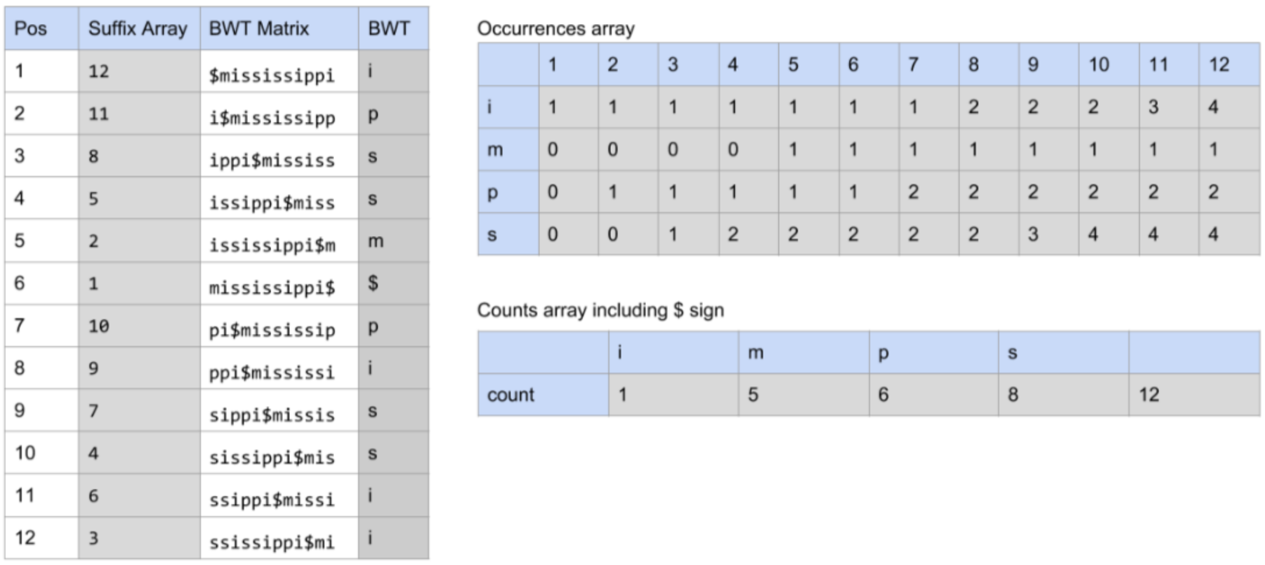
Modern DNA analysis tools allow biologists to search through a large genomic sequence looking for a smaller sequence of interest. The larger sequence, referred to as the reference sequence, is often times many orders of magnitude larger than the sequence of interest, often called the query sequence. The search for a query sequence within a reference sequence can come in many forms. The first, and most common, is exact match search. Exact match search looks for the entire query sequence within the reference sequence, requiring the query sequence to appear in the exact order in its entirety in the reference sequence. A second form of search is an inexact search. As the name would suggest, inexact search allows for slight modifications to the query sequence which can be specified by the user. Normally these modifications take the form of insertions or deletions along the query sequence. A third form of search, the one we focus on in this paper, is super maximal exact match search, or SMEM search. An SMEM is simply the largest maximal exact match, or MEM, passing through a given index in the query sequence. A MEM is a match between the query and reference sequence that cannot be extend further in either direction.

In this paper, we expand upon previous work done towards finding exact matches in order to implement and evaluate SMEM search using a traditional algorithm, an augmented algorithm using a Look up Table (LUT) and an augmented algorithm using a Recursive Model Index (RMI). The LUT approach, although requiring a large space overhead, is able to improve the speed of finding SMEM's in comparison to the traditional approach, by learning all possible substrings of a given size within the reference sequence. The LUT approach, although useful for reference sequences of hundreds of thousands or even a few million, fails when used on massive reference sequences because it is often infeasible to store the entire table in memory. On the other hand, the algorithm that takes advantage of the RMI, is not able to find SMEMs as fast as the traditional or LUT approach, but requires significantly less space. One reason the RMI is not able to locate SMEMs as fast as the other algorithms, is that it often has some degree of prediction error. This error requires the algorithm to perform a "last-mile" search, which can be costly in runtime. We will explore the traditional, LUT, and RMI approaches for finding SMEM's in more depth in the rest of this paper.

2 Previous Work

A major breakthrough for DNA aligning came from the use of the Burrow-Wheeler Transform (BWT) and its associated data structures, referred to as an FM Index [2] [3]. Using these data structures, and a clever backwards search algorithm, exact matches between two sequences are able to be found efficiently **??**. Building off of this exact match algorithm, a secondary algorithm that finds maximal exact matches (MEMs) is possible, where maximal exact matches are defined to be matches in the query sequence that can't be extended any further in either direction in the reference sequence. In the example below, CT is a MEM since in the reference sequence there is no instance of ACT or CTG and so CT cannot be extended further in either direction.

```
Reference Seq: CTCAATGC
Query: ACTGC
```



Pos	Suffix Array	BWT Matrix	BWT
1	12	$mississippi	i
2	11	i$mississipp	p
3	8	ippi$mississ	s
4	5	issippi$miss	s
5	2	ississippi$m	m
6	1	mississippi$	$
7	10	pi$mississip	p
8	9	ppi$mississi	i
9	7	sippi$missis	s
10	4	sissippi$mis	s
11	6	ssippi$missi	i
12	3	ssissippi$mi	i

Occurrences array

	1	2	3	4	5	6	7	8	9	10	11	12
i	1	1	1	1	1	1	1	2	2	2	3	4
m	0	0	0	0	1	1	1	1	1	1	1	1
p	0	1	1	1	1	1	2	2	2	2	2	2
s	0	0	1	2	2	2	2	2	3	4	4	4

Counts array including $ sign

	i	m	p	s	
count	1	5	6	8	12

Figure 1: An example of the BWT, suffix array, occurrences array, and count array data structures for the sequence *mississippi*. This example is originally used in [1].

While finding MEMs is often useful, what is generally more useful is finding super maximal exact matches or SMEMs [4] [5]. An SMEM is defined about an index in the reference sequence, and is the largest MEM that passes through that index. In a query sequence, every index will be part of an SMEM, and many indices will be part of the same SMEM. In the example above, looking at the third index of the query sequence (the T) there are two MEMs that pass through it, CT and TGC. Since TGC is the longer of the two MEMs, the SMEM for the third index (the T) is TGC. This is also the SMEM for the fourth and the fifth index. The SMEM for the second index is CT, and the SMEM for the first index is simply A.

In recent work done for a final project entitled GENIE in 6.830 in the Fall of 2019, a group of students demonstrated that it is possible to augment the exact match algorithm to be more efficient in space or time with a couple of additional data structures, namely a Look up Table or a Recursive Model Index [1]. In this section we first describe the data structures and algorithm used to do traditional exact matching, as well as an algorithm to find SMEMs. We then discuss the more recent work done using a Look up Table and an RMI.

2.1 BWT and FM Index

The Burrows Wheeler Transform was originally used for compression [2] [6]. Given a sequence, it works by grouping lexicographically similar elements near each other in a reversible way, and in such a way that not all elements need to be transmitted when sending the sequence. This reversible compressible new sequence contains all the elements from the original sequence and is called the BWT. To get the BWT, we first define an alphabet A and a character $ that is lexicographically smaller than every element in A. We append $ to the end of our initial sequence, forming a new sequence X of size n, where the first n-1 characters are the original sequence. For example, in Figure 1 our initial sequence is mississippi and our X is mississippi$.

We then create a BWT matrix. The matrix has n rows, and each row is a different rotation of X. Importantly, rotations just shift the characters to the right, keeping the order the same (except at the trailing edge which wraps to the beginning). In the BWT matrix, the n different rotations of X are sorted lexicographically by first element, so that the first element in the BWT matrix is always the rotation X' where the first element of X' is $ and the n-1 subsequent elements are the initial sequence. The BWT is simply the last column of the BWT matrix. Information on how to recreate the initial sequence from the BWT can be found in the original BWT paper [2] and is out of the scope of the paper, as this is not necessary in our research. An example of a BWT matrix and the associated BWT is in Figure 1.

While the BWT is useful for compression, three other data structures derived from the BWT matrix are useful for sequence matching [7] [8]. These structures are the suffix array, the occurrences matrix, and the count array. These data structures together are called the FM Index [3].

The first of these structures, the suffix array, describes for each row in the BWT matrix how many rotations from X the current row is. Specifically, since we use 1 based indexing in our FM Index, for suffix array SA, SA[2] is defined to be one plus the number of rotations required to get the second row of the BWT Matrix back to X. Coincidentally, it is also n + 1 - position of $ (again using 1 based indexing). The suffix array is vital in determining the original position of matches found

Search	Start	End
	1	12
is**s**	C[**s**]+O(0,**s**)+1 = 8+0+1	C[**s**]+O(12,**s**) = 8+4
	9	12
i**ss**	C[**s**]+O(8,**s**)+1 = 8+2+1	C[**s**]+O(12,**s**) = 8+4
	11	12
iss	C[**i**]+O(10,**i**)+1 = 1+2+1	C[**i**]+O(12,**i**) = 1+4
	4	5

Figure 2: An example of backwards search algorithm locating the suffix array interval of the query *iss* in the reference sequence *mississippi*. This example is originally used in [1].

in the initial sequence.

The occurrences matrix is built from the BWT, and represents how many times each character in A has been seen at every index in BWT. For example, in the example in Figure 2, O(9, s) = 3 because in BWT[:9] = {i, p, s, s, m $, p, i, s} there are three occurrences of s.

The count array, which is really more of a dictionary, provides information on how many of each character in A is in X. If C is the count arrya, then for any character a that is in A and is present in X, C[a] is the number of symbols in X that are lexicographically smaller than a [1]. The lexicographically smallest element in A that is in X will always have a count of 1 since there will always be the $ that is smaller than it.

Examples of the suffix array, the occurrences matrix, and the count array for the reference sequence *mississippi* are all displayed in Figure 1.

2.2 Backwards Search Algorithm

The backwards search algorithm is used to find matches of a query sequence in a larger reference sequence, using the FM Index of the reference sequence. Once an FM Index has been created, the backwards search algorithm will support any number of queries against that reference sequence, of any length. Since the indices of the suffix array are based off of the BWT matrix which is sorted, they have the property that each appearance of a query subsequence in the reference sequence will fall in exactly one interval of the suffix array. For example in Figure 1, all appearances of the query *iss* will be in position 4 to 5 of the suffix array. Importantly, from there the actual positions of the query sequence in the reference sequence are simply the values at the specified indices of the suffix array. In Figure1 position 4 and 5 of the suffix array correspond to the values 5 and 2, which are the positions of the query substring *iss* in the original reference sequence.

Due to these properties of the suffix array, finding an exact match of a query subsequence boils down to simply finding the suffix array interval. To do this, we use a backwards search algorithm published by Ferragina et al [3] that utilizes the suffix array, count array, and occurrence array. The specifics of how the algorithm works can be found in the original paper, and an example of the algorithm in practice can be found in Figure 2.

2.3 BWA-SMEM

The Burrows Wheeler Aligner-SMEM, or BWA-SMEM, search algorithm is the more traditional approach for finding SMEM's [9]. BWA-SMEM utilizes the FM-index created from the reference sequence, described in section 2.1 and the backwards search algorithm, described in section 2.2. However rather than doing a single search for the entire query sequence, it does many smaller searches of substrings of the query sequence and searches for those instead, building up SMEMs as it queries.

The algorithm works by iterating over every base in the query sequence and performing two operations: a forward and backward extension. The forward extension phase finds all the matches between the query and reference sequences starting at the given base in the query sequence. For each substring we use the backwards search algorithm to determine if it exists within the reference sequence. For example, if our reference sequence is "mississippi" and our query sequence is "pissssi" and we're starting at position two in the query sequence (one indexed), then we get a match for "i" in the FM-index, and then for "is", and then for "iss", but then do not return a match when we look up "isss" since that sequence does not exist in the reference sequence. The result of the forward extension is an array containing all the substrings that matched. In our example case that would be ["i", "is", "iss"]. For each of the substrings that we find matches for in the forward extension phase, we must then backwards extend. The process is identical to the forward extension phase except we extend the substrings backwards instead of forwards and instead of returning an array, we return the single longest match. In our example case, we will look up "pi", then "pis" and finally "piss", of which only "pi" returns a match. Thus the final MEM's at position 2 in the query sequence are ["pi", "is", "iss"], of which the SMEM is "iss".

In order to find all the SMEM's we must run this forward and backward extension algorithm at every base in the query sequence, however, there is a way to avoid some unnecessary computation. After finding an SMEM, we can begin looking for the next SMEM at the base just after the end of the last SMEM we found. In our example case that would mean that after finding "iss" at position two, we can move immediately to position five in the query. The reason this is possible is that no index can belong to more than two SMEMs. Therefore since you already extended forward across positions 3 and 4 and you have found one SMEM they belong to, you can be certain that the only other SMEM they can belong to will be found during backwards search from position five. If there is another SMEM found encompassing positions 3 and 4 that is longer than "iss", those positions will simply be assigned to the longer SMEM.

One important note is that the backwards search algorithm does not return the actual positions of the SMEMs in the reference sequence, but rather the positions in the suffix array. For each found SMEM these suffixes can then be stored and later converted to the actual position of the matches in the reference sequence.

2.4 Exact Match LUT

In the work done by the previous students, a look up table was used to try and improve the speed of the backwards search algorithm[1]. The general idea is that a look up table could predict locations in constant time, where as the backwards search algorithm is linear in the size of the query. The look up table used mapped potential queries to the suffix array location of the queries. It was created by first choosing a key size K for a given reference sequence, and then traversing through the suffix array once, mapping every substring of size K in the reference sequence to the suffix array interval that corresponds to that substring.

This method is most useful in the case where the query length q is always of size K, because a constant time lookup in the LUT immediately tells if the query is in the reference sequence. However it can also be extended to be used when the query size is less than, or greater than the LUT key size k.

If the query size $q < K$, then we can pad the query with the lexicographically smallest character in our alphabet until q = K. In the case of DNA, the padding character would be A. Once there is a new padded query of size q, call it Q', the suffix array interval for Q' can be looked up directly in the LUT. Since the suffix array is sorted, a traversal of the suffix array from the returned position can find all matches for Q'.

If the query size $q > K$, then you are able to look at the last K bases in Q, and use that as your Q'. You can then find the suffix array interval (s,e) for Q' using the LUT, and for the remaining q - K bases in Q use the backwards search algorithm to find the suffix array interval for Q. The reason to used the last K bases in Q is to allow for minimal work for the backwards search algorithm, since it is able to pick up right where the K bases ended. This means that the larger K is the fewer iterations of backwards search necessary. This presents a trade off in K, while a very large K might lead to faster look ups, it is often inefficient and impractical since the space required for the index scales linearly in K.

Using the LUT, the previous students were able to demonstrate a much faster exact match search, using about half the CPU cycles of the backwards search algorithm [1]. However, as previously mentioned, a LUT is only useful if it can be kept in memory, and so with larger reference sequences that generally require larger K, a LUT often becomes infeasible. This is unfortunate as the backwards search algorithm is linear in the size of the query, so larger query sizes would be where the LUT is most useful.

2.5 Recursive Model Index

The concept of a recursive model index was first introduced by Kraska et Al[10] in a paper titled "The Case for Learned Index Structures". In this paper, an RMI was used to replace a B-tree, and was recommended for use in a number of other data structures. The previous students that worked on exact match, did so using this RMI in the class that Professor Kraska taught in the fall of 2018.

Applied to the exact match problem, the RMI can be used to take the place of the LUT. This means that the RMI utilizes a hierarchy of models that ultimately predict the location of a substring within the suffix array. When the root model is queried, it will decide which of its child models to visit. These decisions propagate down to the leaf level where a single leaf model ultimately returns a suffix array prediction for the given query sequence. The set of parameters specifying how many models to have at each layer in the RMI we call the models expert level.

In this RMI implementation, the individual models are fixed as linear models, whose inputs are the output of the previous model. Each model only needs to store the coefficients for its linear model, so the RMI takes up very little space in memory and has the potential to work quite fast since it only needs to perform a few additions, multiplications and/or comparisons at every level.

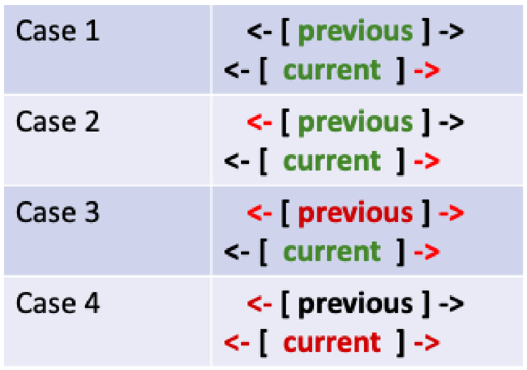
Since the RMI needs to take in integers to train, in the work done in GENIE the substrings used to train and the queries passed to the RMI were encoded. This encoding is done by mapping each base pair of the sequence to an integer, A to 0, C to 1, T to 2, and G to 3. Then this number is read as a base 4 number and converted to a base 10 number. Importantly, this mapping only holds as 1:1 if a fixed query size is used. The RMI is trained by iterating through the suffix array, and mapping every K length query to its position in the suffix array. Since the suffix array indices correspond to the sorted BWT matrix, and the encoded queries increase numerically as the substrings they represent increase lexicographically, increasing suffix array positions map to strictly increasing encoded substrings. This is what makes it possible for the RMI to predict the location in the suffix array index.

However, the location the RMI returns will not necessarily be correct. The hope is that once the RMI has learned the BWT matrix and suffix array, it can at least get reasonably close to the right suffix array location and that a "last mile" search around the predicted location will provide the correct result in reasonable time. The "last mile" search can be any form of search, and the one used in GENIE was an exponential search in both directions from the predicted suffix array location to find upper and lower bounds, and then a binary search from there. The binary search finds the actual suffix array location and then returns that value as the final output.

The functioning RMI can be directly swapped for the LUT but since the LUT is in memory and will never make a mistake, the LUT will always be faster. However, the RMI takes up very little space and so is still practical for large reference sequences. While the previous work with GENIE was able to use the RMI for exact match query prediction, they did not have a Python implementation of the backwards search algorithm and so did not compare the run time of the RMI to that of the backwards search algorithm or to the LUT.

3 Experimental Methods

As we were in collaboration with the students, Tony Peng, Ashwath Thirumalai, and Elizabeth Wei, who had done the previous learned index work aforementioned, we had access to their code as well as the resources that they had acquired when doing their project. Namely we had access to a real world dataset from Sanchit Misra (Intel), of a single human chromosome consisting of approximately 250 million



Case 1	<- [previous] -> <- [current] ->
Case 2	<- [previous] -> <- [current] ->
Case 3	<- [previous] -> <- [current] ->
Case 4	<- [previous] -> <- [current] ->

Figure 3: The four cases when doing the backwards substring lookup in the SMEM LUT algorithm.

nucleotides. This was invaluable since DNA has been demonstrated to have an underlying structure, and therefore has the potential to be learned more easily than a random sequence of nucleotides. We also had access to an RMI implementation, originally provided by Jialin Ding (MIT) that given training data fits a recursive model index using linear regression. However, the code used for the previous project was written in a mixture of C, C++, and Python. This made it impossible to compare the run times of their original algorithm, the augmented algorithm using the LUT, and the augmented algorithm using the RMI. To avoid this, we decide to not build off their previous work, but instead implement each part for ourselves in Python, so we are then able to compare our different approaches. We choose Python as the RMI is built in Python and it is the most difficult to replicate in a different language.

In the rest of this section, we describe our implementations and design decisions when building the original BWA-SMEM algorithm, an augmented SMEM algorithm using a look up table, and an augmented algorithm using a recursive model index.

3.1 BWA-SMEM Algorithm

Our first task was implementing the BWA-SMEM algorithm described in section 2.3. To do this, we first built infrastructure to create the BWT matrix, and from that the FM Index, given a reference sequence. It is important to note that for each reference sequence that queries are run against, the FM Index needs to be built only once, so while it is often a lengthy process, it is a one time cost.

After creating and storing the FM Index, we implemented the backwards search algorithm described in Section 2.2. This takes in a query sequence of any size, and outputs the suffix array interval for that query. In the previous work done regarding exact match, the query was always assumed to be in the reference sequence. In our work, we can not use that assumption, as the BWA-SMEM algorithm won't terminate until it finds a query that can not be extended, or in other words, is not in the reference sequence. If our backwards search algorithms determines the query sequence is not in the reference sequence, it returns -1.

Once the backwards search algorithm was complete, we implemented the BWA-SMEM algorithm as described in 2.3. This algorithm takes in a query sequence, and returns the SMEM associated with each index of the query sequence.

3.2 Look Up Table

After implementing the BWA-SMEM algorithm to use as a baseline, we wanted to see if it was possible to improve the algorithm with a look up table in a similar way as was done in the exact match setting described in 2.4. This is also an important step towards using an RMI, because ideally the RMI can eventually take the place of the LUT.

There are a number of key differences in the SMEM algorithm compared to the exact match algorithm that cause it to be more difficult setting to use a look up table. The first issue is that we don't know what the SMEM sizes are going to be, and so it is a challenge to choose a good key size K to use in the LUT. With a randomly generated query sequence, the key size can be estimated however based on the probabilistic length of the average SMEM. The equation below can be used to estimate approximately this key size:

$$K * \frac{refseqsize}{4^K} \approx 1$$

The leading K in the statement is because at each index there are K different substrings able to be looked up in the look up table. However, this only holds true for query sequences that are randomly generated, but often in practice the query sequence will have large subsequences that are from the reference sequence. This can potentially lead to suboptimal runtime with the LUT.

Once the LUT size is chosen, we can then create the LUT. We did this by iterating through the suffix array, and mapping every K length substring of the reference sequence to the suffix array interval that it resides at. To make it easy to later switch in the RMI, we actually stored the encoded substring rather than the alphabetic sequence. As an implementation aside, since Python dictionaries are able to more quickly lookup strings than integers, we cast the encoded substring integers to strings and used that as the key.

After the LUT is created and stored in memory, the next question becomes how to use the LUT to most effectively reduce the number of iterations of the backwards search algorithm. We tried a number of different ways and designed an algorithm that we believe best utilizes the look up table. For an index i, the algorithm works by looking at each of the size K substrings that pass through i. There will K of these substrings. The first substring that is checked in the lookup table is the substring that has i as its first character. The next substring has i as its second character, and it works back from there. After checking each substring, we learn things about the previous substring that was checked. There are four main cases when checking a substring: (1) the current substring is in the LUT and the previous substring is in the LUT and the substrings are consecutive in the actual reference sequence (not the suffix array), (2) the current substring is in the LUT and the previous substring is in the LUT but they are not actually consecutive in the reference sequence (and therefore not a MEM), (3) the current substring is in the LUT and the previous one is not, and (4) the current substring is not in the

LUT – doesn't matter if the previous substring was or wasn't in the LUT.

These four cases are displayed in Figure 3. In each of the four cases you are able to learn and eliminate potential forward or backward iterations from some of the substrings, avoiding unnecessary work. In Figure 3, these eliminations are denoted by red arrows, meaning that search in that direction is no longer necessary. If the search is necessary, than forward or backward search begins at the end of the LUT match. If there are no matches for any of the K substrings, meaning that the SMEM at the index must be smaller than size K, then the normal BWA-SMEM algorithm is applied from that index.

Similar to BWA-SMEM, once an SMEM is found, search begins again at the index after that SMEM. This allows for another early stop condition; if you have already found a MEM and it is larger than the distance between your current backwards search and the start of the previous SMEM, you can stop early, as there is no way it will be larger than the current MEM.

After designing an algorithm to take advantage of the LUT, we implemented it in Python. The code is available at [1] in the SMEM.py file of the SMEM directory.

3.3 Recursive Model Index

As mentioned at the beginning of Section 3 (this section), we were provided with code to fit an RMI, and then make predictions. Also since we were in collaboration with the group who built GENIE, we had access to their code. While this group had not ever compared the RMI exact match with the backwards search algorithm, we were able to use their code for the last mile search, a combination of exponential and binary search. With this, to get the RMI running we simply had to hook up the correct data structures and format our data in such a way that the RMI could train on it. We did this in the same way as described in section 2.5, encoding each K length substring in the reference sequence and using those as inputs with the targets being the position of the substrings in the suffix array. Since the suffix array positions are sorted, the RMI simply has to predict the starting suffix array position of substrings that appear multiple times in the reference sequence. While it seems like a difficult problem to predict where a substring is, since the encoded substrings are strictly increasing with the suffix array positions, the prediction ends up being a simple probability problem.

One important modification we had to make to the last mile search is to return quickly if the substring is not in the reference sequence. This is an issue not necessary in the exact match case, as in that case there is the assumption that the query is always in the reference sequence. If the query is not in the reference sequence, the search algorithm will return a lower position that is larger than the upper position.

After we had the exact match RMI prediction working, we were able to quickly use it to predict SMEMs by swapping it in for the LUT algorithm described in Section 3.2. Since the RMI and LUT take the same input and produce the same output, anywhere that had been making a call to the look up table, we now make a call to the RMI.

[1]https://github.com/jgkellymit/GENIE-SMEM

4 Experimental Setup

4.1 Dataset

As mentioned in the beginning of section 3, we had access to a full human chromosome of 250 million bases. However, when creating the FM Index for a reference sequence it is necessary to first create the BWT matrix which takes up n^2 space where n is the size of the reference sequence. This is a significant constraint since when we create the BWT matrix it is done in memory. One million bases can be stored in about 500KB of memory, but this means that the BWT matrix would require approximately 260 GB of space to be stored. Since this then needs to be sorted, it becomes a very tricky problem to create FM indexes for large reference sequences. While there are optimized algorithms that are able to create these FM Indexes for large reference sequences, we decided that it was unnecessary for our project and evaluation to work with full size reference sequences. With this in mind, and with the size constraints imposed, we chose a reference sequence that consists of the first 100,000 bases of the original full data.

4.2 Experimental Description

The first algorithm that we implemented was the exact match search using standard backwards search. We then tested this on multiple query sizes. From there we implemented the BWA-SMEM algorithm, and again tested it on multiple query sizes. After that we built a LUT and the augmented SMEM algorithm that takes advantage of the LUT, described in section 3.2 We ran experiments sweeping the LUT size parameter, and also comparing the LUT-SMEM algorithm to the BWA-SMEM algorithm. From there we took the previous code written for GENIE and the RMI provided, and finished building a working exact match algorithm using the RMI. We then compared the runtime of the RMI exact match algorithm to the backwards search algorithm. As the RMI can have many different expert levels (the number of models at each layer in the reference sequence), we compared how different expert levels performed for a fixed query size. We then switched out the RMI for the LUT and created a RMI-SMEM algorithm. Similar to the LUT-SMEM, the RMI-SMEM algorithm also has a key size K that we swept and optimized in a similar way to that of the LUT. Finally we compared the BWA-SMEM, LUT-SMEM, and RMI-SMEM algorithms on two types of queries. The first type of query is a randomly generated query, which would have an average SMEM size given by K in the equation in section 3.2. The second type of query is created by pulling various sized pieces from the reference sequence. The average SMEM size for this query is the average size of the pieces pulled from the reference sequence, which we are able to tune. We set the K for the LUT and the RMI as close as we can to what we predict the average SMEM size should be.

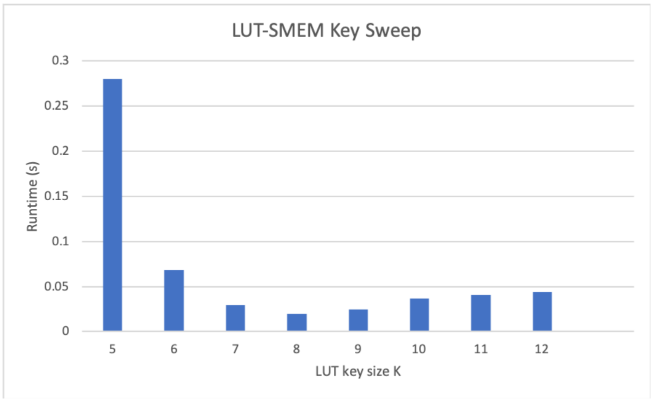


Figure 4: A sweep of the LUT key size K, running the LUT-SMEM algorithm. Run on a reference sequence of size 100,000, and a query size of 1000. Queries were randomly generated, values are averages over 100 queries.

5 Results

In this section we discuss the results from the various experiments described in section 4.2.

5.1 Look up Table SMEM

After creating a look up table algorithm that accurately produced SMEMs, we ran experiments to measure the affect of the look up table key size K. We did this by training multiple different look up tables on the same reference sequence of length 100,000, each having a different key size. We fixed a query size, and then randomly generated 100 queries, and recorded the average time it took each of the different LUTs to find all the SMEMs of the queries. The results for this experiment are in Figure 4. From this figure we see that the key size K can make an order of magnitude of difference if chosen well vs if chosen poorly. We also see that the best key size K aligns with the predicted key size K from the equation in section 3.2. Again, this makes sense as we want the key size to be as close to the average SMEM length as possible, so that the LUT algorithm is able to take advantage of the quick look up speed that the LUT provides.

After choosing an optimal key size for our LUT, we ran an experiment comparing the run time of the LUT-SMEM algorithm to the standard BWA-SMEM algorithm. In this experiment we randomly generated queries of different sizes, 100 of each size, and then recorded the average time it took each algorithm to find the SMEMs associated with the query. The results are in Figure 5. We see what we had anticipated: the LUT algorithm is indeed faster than the original BWA-SMEM algorithm. This is a positive result as it means that if the RMI is able to predict quickly, when swapped out for the LUT it has the potential to be faster than the standard algorithm. Also it is important to note, that the LUT is faster than the standard algorithm, but that is only the case with a well chosen key size K. If for example we had naively chosen the key size K to be 6 instead of 8, it would have been twice as slow as the BWA-SMEM algorithm.

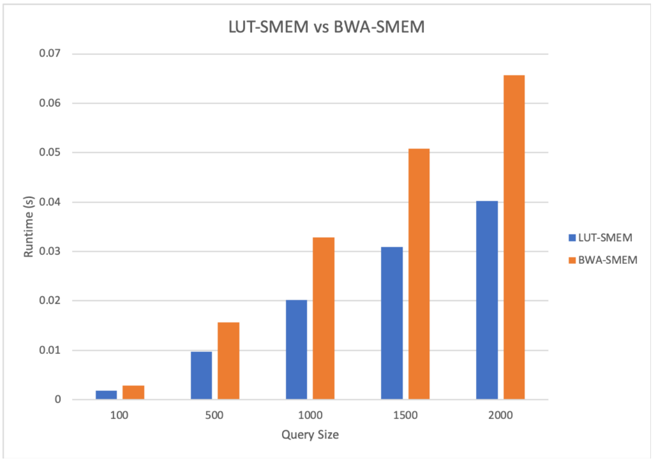


Figure 5: A run time comparison of the LUT-SMEM and BWA-SMEM algorithms, across multiple query sizes. Run on a reference sequence of size 100,000, with an LUT key size K of 8. Queries were randomly generated, values are averages over 100 queries.

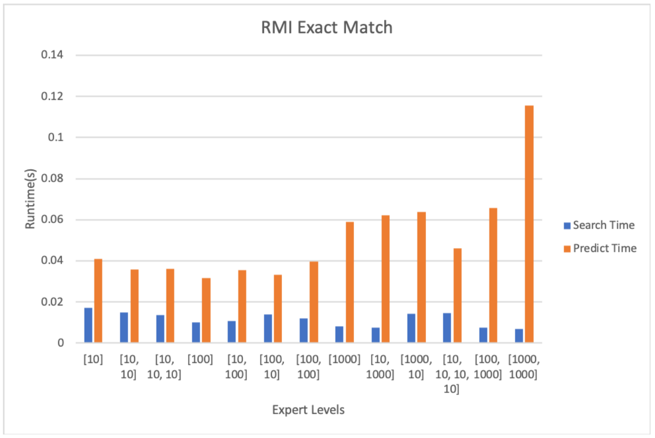


Figure 6: RMI exact match with different expert levels (number of models at each layer). Run on a reference sequence of size 100,000, with a model key size and query size of 500. Queries were generated by taking random segments of the reference sequence, values are averages over 100 queries.

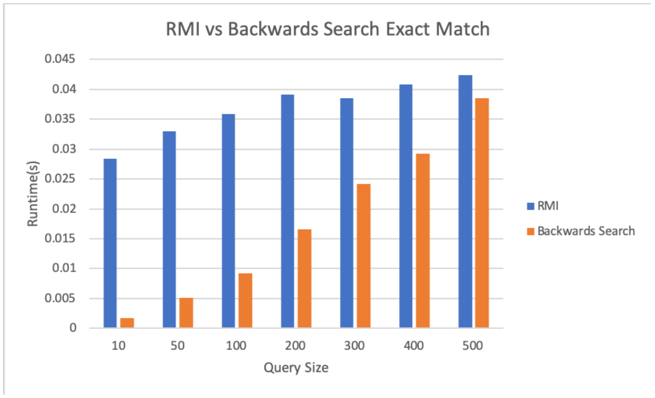


Figure 7: A run time comparison of the RMI and backwards search exact match algorithms, across multiple query sizes. Run on a reference sequence of size 100,000, a model key size equal to the size of the query, and expert level of [10, 100]. Queries were generated by taking random segments of the reference sequence, values are averages over 100 queries.

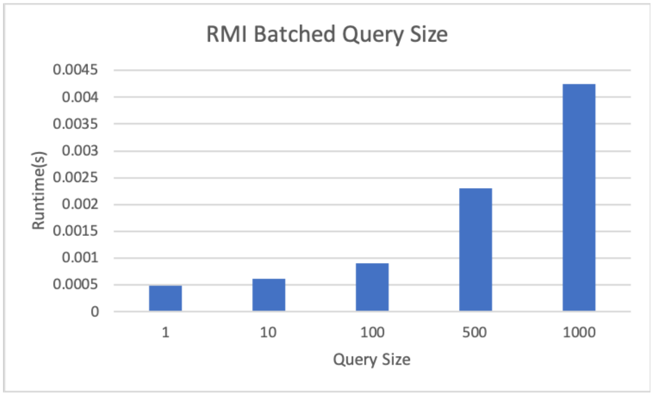


Figure 8: A comparison of the RMI exact match algorithm when batching queries. Only predict time is shown.

5.2 RMI Exact Match

Our next set of experiments came after creating a working RMI exact match algorithm. The RMI is made up of different levels of models, so choosing an appropriate number of layers and how many models for each layer is important. To try and determine a good configuration for our RMI we ran experiments where we used a constant query size of 500 and many different RMI configurations. There are infinite possible configurations, so in our experiments we focused on models between 1 and 4 layers with the max number of models per layer 1000. We measured the average run time of each of these RMIs on 100 different queries that were generated by choosing random parts of the reference sequence. The RMI was trained with substrings equal to the query length of 500. The results from these experiments are in Figure 6. This figure shows the average prediction time, as well as the average search time for each of the different configurations. We can see that the configuration has a significant impact on the run time, and that larger models generally take longer to predict. Also, there appears to be an inverse correlation between the prediction time and the search time. This makes sense in that the larger models take longer to predict, but they then predict more accurately and so less last mile search is necessary. We found that the configuration of [10, 100] worked reasonably well and so used this configuration in future experiments.

Since the previous work done in GENIE had not compared the RMI exact match to the standard backwards search algorithm, our next step was to compare these algorithms. For the RMI we used the expert level configuration of [10, 100], and trained the model with substrings equal to the length of the queries. We varied the query size from 10 to 500, generated 100 queries of each size from random parts of the reference sequence, and then compared the run time of the two algorithms which each determined the suffix array location of the queries. The results are in Figure 7. The results for this experiment are slightly surprising and not exactly what we wanted to see. We new that the backwards search algorithm was an O(n) algorithm, with n being the size of the query, and we see that that is the case in the run time. We had predicted that the RMI would be a mostly constant time algorithm, since the size of the query should not affect the prediction time. While we new it might affect the search time, we had anticipated it to be minor. The results show that it is closer to constant time than the backwards search algorithm, but it definitely does increase with the size of the query.

Aside from the run time of each algorithm individually, there are a couple very important results that we learn from comparing the algorithms. The first is that the RMI would be a viable run time alternative to the backwards search algorithm when running on large queries. One important drawback though is that the RMI is trained on substrings that are the same size as the query, which is not a problem in itself, but can be problematic in our implementation. In our implementation we convert substrings to integers via converting the substring to a base 4 number, and then converting that base 4 number to base 10. This means that for a 500 length substring in the worst case it will be represented by a base 10 number of length 302. The RMI is not designed to handle numbers much larger than this as they can not be cast to floats, so an optimized version of the RMI would need to be designed to be able to handle this

The second thing we learn is that for small query sizes, the backwards search algorithm performs approximately an order of magnitude better than the RMI. This is not a good sign for our idea to apply the RMI to the SMEM problem, since SMEMs rarely get very large and so we generally only make small queries to the LUT.

There is another advantage to the RMI however, in that it has the ability to batch queries and make predictions for multiple queries at once. To see how effective the batching is, we ran a set of experiments with the same expert size for the model and the same query size, but evaluating multiple queries at once. The results from this experiment are in Figure 8. We can see that doing 10 queries can be done in roughly double the time it takes to do a single query. It is important to note though that this doesn't apply to the last mile search time, which would still need to be run on each queries predictions. While in this work, we did not take ad-

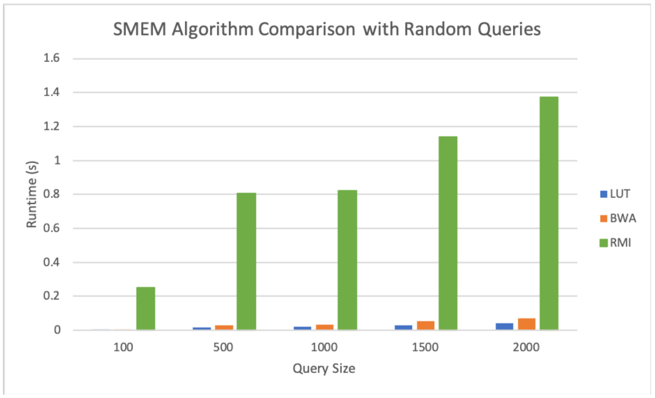


Figure 9: A run time carl

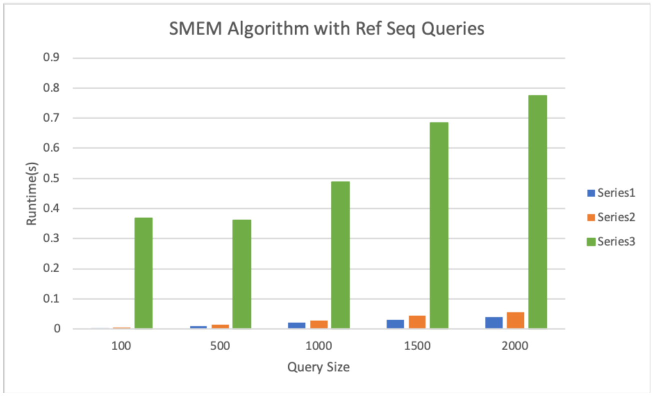


Figure 10: A comparison of the RMI exact match algorithm when batching queries. Only predict time is shown.

vantage of the batching property in the SMEM algorithm, this is one possible improvement left for future work, and makes it a more viable alternative when making multiple exact match predictions.

5.3 RMI SMEM

In our final sets of experiments, we plugged in the RMI in place of the LUT in the SMEM algorithm. Once we verified that it was correctly returning the SMEMs, we compared its run time to the LUT-SMEM and BWA-SMEM algorithms. We ran it with randomly generated queries first, and used a key size for both the LUT and the RMI of 8. We then ran a comparison using queries generated by pulling multiple pieces from the reference sequence, thereby generating queries with longer average SMEMs. We set the LUT and RMI key size in these experiments to 15, what we predicted the average SMEM size would be. In both sets of experiments we varied the size of the query and took the average run time over 10 different runs. The results are displayed in Figure 9 and Figure 10 respectively. From these figures we can see that our suspicions from the exact match algorithms are confirmed, the RMI-SMEM algorithm runs significantly slower than BWA-SMEM or the LUT-SMEM algorithm. It does slightly better on the queries that have an average SMEM size that is larger, but is still about an order of magnitude slower.

6 Conclusion

In this paper we developed a new algorithm that utilizes a look up table to quickly calculate SMEMs, evaluated a recursive model index on the exact match problem, and then used the recursive model index to accurately identify SMEMs. While the look up table was able to more quickly identify SMEMs then the traditional BWA-SMEM algorithm, it did so at the cost of storing a large data structure that would likely become infeasible for large datasets. We established that the recursive model index could be a viable replacement to the standard backwards search algorithm on large queries for the exact match search problem, and that it could potentially be used with batching to outperform on smaller queries as well. The RMI also does not need the occurrence matrix to be stored, which leads to approximately an 81% space reduction in comparison to the backwards search algorithm, as calculated in GENIE [1]. Thus, in the cases of large exact match queries, the RMI is more efficient in both space and run time. While the RMI was able to be used to accurately predict SMEMs, the run time was unfortunately much worse than that of the traditional algorithm. Until the RMI is able to predict more quickly, or at least comparably, with the backwards search algorithm for small queries on the exact match problem, the RMI-SMEM algorithm will always be slower.

In future work we leave two main tasks. The first is to attempt to optimize the RMI prediction algorithm. Theoretically it should be able to predict in constant time once trained, so working to get closer to that goal would be a valuable step. A second task would be to adapt our SMEM algorithm to use batched query look ups, as this would allow for more efficient use of the RMI. While this would be tricky since it is impossible to know the positions of the SMEMS and thus it is impossible to know all the queries you will need to call, it might be possible to predict the length of the SMEMs based on the reference sequence size and then use that to choose position indexes to make queries. Then at each of these positions you could batch every substring that passes through that position and call predict on many substrings at once. A final task left for future work is to implement the RMI algorithm in C, which would allow for comparison with the optimized BWA algorithms and would be a more realistic tool for use in industry.

CODE

As mentioned in a footnote above, our repository can be found at the following link:
https://github.com/jgkellymit/GENIE-SMEM.

ACKNOWLEDGMENTS

We would like to thank the authors of GENIE, Tony Peng, Ashwath Thirumalai, Elizabeth Wei, as well as Sanchit Misra, Nesime Tatbul, Jialin Ding, Kapil Vaidya, and Professor Kraska, for guidance, ideas, and code to help us get started.

References

[1] T. Peng, A. Thirumalai, and E. Wei, “Genie: Gene indexer,” *6.830 Final Project*, 2018.

[2] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” 1994.

[3] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 390–398.

[4] R. Patro, G. Duggal, and C. Kingsford, “Accurate, fast, and model-aware transcript expression quantification with salmon,” *BioRxiv*, vol. 21592, 2015.

[5] J.-H. Choi, H.-G. Cho, and S. Kim, “Game: a simple and efficient whole genome alignment method using maximal exact match filtering,” *Computational Biology and Chemistry*, vol. 29, no. 3, pp. 244–253, 2005.

[6] P. M. Fenwick, “The burrows–wheeler transform for block sorting text compression: principles and improvements,” *The computer journal*, vol. 39, no. 9, pp. 731–740, 1996.

[7] H. Li and R. Durbin, “Fast and accurate short read alignment with burrows–wheeler transform,” *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[8] D. Adjeroh, T. Bell, and A. Mukherjee, *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Science & Business Media, 2008.

[9] H. Li, “Aligning sequence reads, clone sequences and assembly contigs with bwa-mem,” *arXiv preprint arXiv:1303.3997*, 2013.

[10] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 489–504.