Supplementary Material
SMT: A High-Performance Data Structure for Counting Kmers

Jader M. Caldonazzo Garbelini

1 Introduction

This supplementary material is intended to provide a detailed insight into the algorithms utilized throughout the study. Herein, we present the pseudocodes of the algorithms CREATE-SMT, KSEARCH, HMAP, KHMAP, IUPAC-SEARCH, and KDIVE, accompanied by elucidative explanations on each of them. Moreover, we will exemplify the practical application of the Sparse Motif Tree (SMT) in discovering conserved motifs in ChIP-seq data, elucidating the process of biological pattern discovery. The aim is to offer a thorough understanding of the inner workings of the algorithms and demonstrate the effectiveness of SMT in genomics data analysis. The source codes for the respective pseudocodes presented here can be found at: `https://github.com/jadermcg/smt`.

2 Algorithms

In this section, we will detail the core algorithms employed in our analysis. The pseudocodes for the algorithms CREATE-SMT, KSEARCH, HMAP, KHMAP, IUPAC-SEARCH, and KDIVE are presented, providing a clear view of the algorithmic structures and operations. Each pseudocode is accompanied by a description that clarifies its functionality. The exposition of these algorithms aims to offer substantial technical understanding that facilitates the replication and extension of the work presented.

2.1 Create-SMT

The algorithm CREATE-SMT is designed to build the search database that will be used later for search operations. The pseudocode representation provided aims to elucidate the basic functioning of the algorithm. However, it is worth mentioning that some aspects of synchronization/semaphores and multithreading have been abstracted to keep the presentation clear and concise. For a thorough understanding, especially regarding multithread handling and synchronization, it is highly recommended to refer to the complete source code, which is available on GitHub.

Algorithm 1 CREATE-SMT

1: $n \leftarrow$ number of nodes
2: $M \leftarrow \text{zeros}(n, 4)$ ▷ Creates an empty matrix with n rows and 4 columns.
3: root $\leftarrow 1$ ▷ The root is set to 1 and the new node to 2.
4: new_node $\leftarrow 2$
5: **for** $s \in S$ **do** ▷ Processes all sequences $\{s_1, s_2, s_3, \ldots, s_n\}$ starting from the root, each with width `k`.
6: node = root
7: **for** $i = 1$ to w **do**
8: symbol = $s[i]$
9: **if** $M[\text{node}, \text{symbol}] = 0$ **then** ▷ Checks if `M[node,symbol]` equals 0.
10: $M[\text{node}, \text{symbol}] = \text{new\_node}$ ▷ A new node needs to be added at this position, as carried out in lines 10 and 11.
11: new_node = new_node + 1
12: node = new_node ▷ The variable `node` takes the value of the new node and is incremented in line 12.
13: **else**
14: node = $M[\text{node}, \text{symbol}]$ ▷ If `M[node,symbol]` is not 0, the node already exists and the variable `node` is simply updated with the value represented by M[node, symbol].
15: **end if**
16: **end for**
17: **end for**
18: **return** M

In line 3, the root (`root`) is set to the value 1 and the new node (`new_node`)

as 2. In line 5, all the sequences $\{s_1, s_2, s_3, \ldots, s_n\}$ are processed starting from the root, each of which has a width of `k`. In line 9, the algorithm checks if `M[node, symbol]` is equal to 0. If this occurs, it means that there is no edge with the symbol defined by the variable `symbol` leaving from the variable `node` and going towards another node. In other words, it is necessary to add a new node at this position, which is done in lines 10 and 11. Finally, the variable `node` takes the value of the new node (`new_node`) and in line 12, the new node is incremented. In line 13, if `M[node, symbol]` is different from 0, then the node already exists, and in line 14, the variable `node` is simply updated with the value represented by M[node, symbol].

2.2 Ksearch

This algorithm's primary objective is to perform exact searches on the SMT data structure. It stands out for its efficiency, operating with a time complexity of $O(k)$, where k is the size of the KMER to be found. Consequently, the algorithm traverses the fragment only once, conducting direct queries to the SMT structure to find exact matches. Its efficiency makes it particularly useful in scenarios where it is imperative to minimize search time, which is often the case in bioinformatics and computational biology applications. Algorithm 2 illustrates how this functionality was implemented.

Algorithm 2 KSEARCH

```
1: k ← kmer.size()
2: next ← 0
3: for i = 1 to k do
4:     c ← kmer[i]
5:     next ← SMT(next, c)
6:     if next == 0 then return False
7:     end if
8: end for
9: return True
```

The KSEARCH algorithm is quite straightforward and starts by determining the size of the KMER, storing this information in the variable `k`. It then initializes the variable `next` with the value 0. This variable will be used to maintain the current state of the search within the SMT structure. After this, the algorithm enters the main LOOP that iterates through each character `c`

of the KMER. Within this LOOP, the SMT is called with the current state `next` and the character `c` to obtain the next state. If at any point the state `next` becomes 0, the algorithm concludes that the KMER was not found and returns FALSE. The efficiency of this algorithm is highlighted by its time complexity $O(k)$, which is linear with respect to the size of the KMER. This makes KSEARCH an extremely effective tool for exact searches within the SMT.

2.3 Hmap

The HMAP is structured to efficiently process datasets, converting sequences into k-mer representations and accumulating the counts of these k-mers in a hash map. Through a parallel and multithreaded approach, it enables the quick construction of a hash map from a sparse matrix, while managing the reading and writing of data in an optimized manner. The consumer function (Algorithm 3 first procedure) is responsible for the final aggregation of processed data, while the hmap function ((Algorithm 3 second procedure)) oversees the data loading and transformation process, ensuring correct synchronization among parallel operations.

The Algorithm 3 begins with the `consumer` function, which continuously checks whether there are maps to process in the `maps` queue or if the processing is still ongoing (line 2). If there are maps, it takes a map from the queue and processes each key-value pair, aggregating the values to the `final_hash` (lines 4-8). Concurrently, the `hmap` procedure is initiated, iterating through a range of values (line 10), creating a new `hash` map for each value in the range. For each value, it creates a new sparse matrix `S`, loads data from `smtdb` to `S`, extracts columns 4 and 5 to `counts` and `kmers`, respectively, and then discards `S` (lines 12-16). It then identifies the indices where `counts` is non-zero, and for each index, converts the corresponding value in `kmers` to a `kmer` string and accumulates the counts in the `hash` map (lines 17-21). After processing all the values in the range, it places the `hash` map in the `maps` queue (line 22) and, at the end of the procedure, sets `still_processing` to false to indicate that processing has been completed (line 24).

2.4 KHmap

The KHMAP algorithm employs a recursive approach to generate count hash tables from a previously built SMT. If an SMT was initially created with a

Algorithm 3 Quick HashMap Construction

```
1: procedure CONSUMER
2:     while not maps.empty() or still_processing do
3:         map ← new Map()
4:         while maps.try_pop(map) do
5:             for all kv in map do
6:                 final_hash[kv.first] += kv.second
7:             end for
8:         end while
9:     end while
10: end procedure
11: procedure HMAP(nb, k)
12:     for r in (0, nb) do
13:         hash ← new Map()
14:         for i in r do
15:             S ← new SparseMatrix()
16:             Load S from smtdb
17:             counts, kmers ← ExtractColumns(S, 4, 5)
18:             Delete S
19:             nonZeroIndices ← FindNonZeroIndices(counts)
20:             for j in nonZeroIndices do
21:                 kmer ← IndexToKmer(kmers[j], k)
22:                 hash[kmer] += counts[j]
23:             end for
24:         end for
25:         maps.push(hash)
26:     end for
27:     still_processing ← false
28: end procedure
```

`k` value equal to κ, the KHMAP is capable of extracting count tables for any value of k such that $1 \leq k \leq \kappa$. The execution of this process is facilitated by Algorithms 4 and 5.

The algorithm COUNT K-MERS initiates a recursive function that takes five arguments: the data structure SMT, the current node of the tree (`node`), the maximum size of the KMER (`kmax`), a hash map to store the counts (`hmap`), and a counter `j` to keep track during the recursion. The first thing

Algorithm 4 COUNTING

```
1: function COUNTING(SMT, node, kmax, hmap, j)
2: if j == kmax then
3:     count = SMT(node, 4)
4:     hmap(kmer) += count return
5: end if
6: for i = 1 to 4 do
7:     next = SMT(node, i)
8:     if next > 0 then
9:         COUNTING(SMT, node, kmax, hmap, j + 1)
10:    end if
11: end for
```

the algorithm does is to check if the counter `j` is equal to `kmax`. If that is the case, it enters a conditional block to perform the counting. In this block, the algorithm retrieves the count value stored in the current node through SMT(`node`, 4). This count value is then added to the hash map `hmap` associated with the KMER under analysis. Shortly after, the function returns, ending this instance of recursion.

If `j` is not equal to `kmax`, the algorithm proceeds to a LOOP ranging from 1 to 4, which are essentially the numerical representations for the four nucleotides in a DNA sequence. For each iteration, the algorithm checks if there is a next valid node in the SMT tree. If there is, the function calls itself recursively, advancing to the next node and incrementing the counter `j`. This strategy allows the algorithm to efficiently explore the SMT tree and extract the counts of all possible KMERS of size up to `kmax`.

The algorithm HASH is a natural extension of the COUNTING algorithm and serves to create a hash table for counting *kmers*. It is also a recursive function and starts with the HASH function, which takes seven arguments: the data structure SMT, the current node in the tree (`node`), the desired KMER size (`k`), the maximum KMER size (`kmax`), the current KMER sequence, the hash map (`hmap`), and a counter `j` to control the recursion. Similar to the previous algorithm, the first step is to check if the counter `j` has reached the maximum size `k`. If true, the algorithm calls the COUNT_KMERS function to perform the counting of the KMER and store it in the hash map. At this point, the function returns, ending the current instance of recursion.

Otherwise, the algorithm enters a LOOP that iterates from 1 to 4. Each

Algorithm 5 HASH

```
1: function HASH(SMT, node, k, kmax, kmer, hmap, j)
2: if j == kmax then
3:     COUNTING(SMT, node, kmax, hmap, j) return
4: end if
5: for i = 1 to 4 do
6:     next = SMT(node, i)
7:     if next > 0 then
8:         HASH(SMT, next, k, kmax, kmer + char(i), hmap, j + 1)
9:     end if
10: end for
```

value in this iteration represents one of the four nucleotides in a DNA sequence. Within the LOOP, the algorithm checks if there is a next valid node in the SMT structure. If one exists, the HASH function calls itself recursively, advancing to the next node. In addition, it also updates the value of the KMER by appending the character corresponding to the next node and increments the counter `j`. Thus, the algorithm keeps a record of all the KMERS in a hash map, providing quick and efficient access to the data. This allows the hash table to be efficiently created, navigating through the SMT tree and collecting the necessary information.

2.5 IUPAC-search

The IUPAC-SEARCH algorithm is a specialized search function designed to identify and count patterns specified in the IUPAC format within an SMT. The IUPAC format is a standard way of representing nucleic acid sequences, making this algorithm particularly useful for bioinformatics and the study of genetic sequences. When an IUPAC pattern is provided to the function, it traverses the SMT to identify all instances of that pattern.

The function then returns all KMERS that match the IUPAC pattern along with their counts, stored in a map where each KMER is mapped to its respective count. This provides a high-level and concise summary of all the locations in the SMT where the pattern of interest occurs, as well as the frequency with which each associated KMER is found. The IUPAC notation for nucleic acids, used for representing genomic sequences, includes letters for the four standard nucleotides and also letters to represent possible combi-

nations thereof. Below are the definitions for IUPAC letters that represent degenerate patterns.

- R: represents a purine, i.e., an abbreviation for a position that could be either Adenine (A) or Guanine (G).
- Y: represents a pyrimidine, i.e., an abbreviation for a position that could be either Cytosine (C) or Thymine (T).
- S: STRONG or *strong interaction*, a 3-hydrogen bond. Represents a position that could be either Guanine (G) or Cytosine (C).
- W: WEAK or *weak interaction*, a 2-hydrogen bond. Represents a position that could be either Adenine (A) or Thymine (T).
- K: KETO, i.e., the nucleotides with ketone groups. Represents a position that could be either Guanine (G) or Thymine (T).
- M: AMINO, i.e., the nucleotides with amino groups. Represents a position that could be either Adenine (A) or Cytosine (C).
- B: represents a position that could be any base except Adenine (A) (i.e., could be C, G, or T).
- D: represents a position that could be any base except Cytosine (C) (i.e., could be A, G, or T).
- H: represents a position that could be any base except Guanine (G) (i.e., could be A, C, or T).
- V: represents a position that could be any base except Thymine (T) (i.e., could be A, C, or G).
- N or X: represents any base, i.e., an abbreviation for a position that could be Adenine (A), Guanine (G), Cytosine (C), or Thymine (T).

Algorithm 6 illustrates the workings of the IUPAC-SEARCH. The algorithm starts with a condition that checks if the variable i is equal to k in line 1. If both are equal, it indicates that the algorithm has reached the end of the IUPAC pattern it is searching for in the tree. In this case, between lines 2 to 5, the algorithm retrieves the count and the address of the current node

Algorithm 6 KIUPAC-SEARCH

```
1: if i == k then
2:     count ← SMT(node, 4)
3:     address ← SMT(node, 4 + 1)
4:     patterns[address] ← count
5:     return
6: end if
7: c ← iupac[i]
8: symbol ← 0
9: next ← 0
10: if c == A OR c == C OR c == G OR c == T then
11:     symbol ← char2int(c)
12:     next ← SMT(node, symbol)
13:     if next ≠ 0 then
14:         KIUPACSearch(SMT, patterns, iupac, next, i + 1, k)
15:     end if
16: else if c == R(puRine) then
17:     for symbol ∈ {A, G} do
18:         next ← SMT(node, symbol)
19:         if next ≠ 0 then
20:             KIUPACSearch(SMT, patterns, iupac, next, i + 1, k)
21:         end if
22:     end for
23: else if c == Y(pYrimidine) then
24:     for symbol ∈ {C, T} do
25:         next ← SMT(node, symbol)
26:         if next ≠ 0 then
27:             KIUPACSearch(SMT, patterns, iupac, next, i + 1, k)
28:         end if
29:     end for
30: end if
31: Other cases (such as S, W, K, M, B, D, H, V, N, X) follow a similar
    format.
```

in the tree, logs the count in the pattern map corresponding to the address, and then terminates the algorithm's execution.

If i is not equal to k, the algorithm proceeds to get the current character

from the IUPAC pattern and initializes the variables *symbol* and *next* to zero on lines 8 to 9. A new condition is checked at line 10, where if the current character is one of the nucleotide bases (A, C, G, T), the algorithm executes the instructions on lines 11 to 15. In this block, the algorithm transforms the current character into its corresponding integer value (using the *char2int* function), obtains the corresponding child node in the tree and, if such a node exists, recursively calls the IUPAC-SEARCH function to continue the search from that node.

On line 16, if the current character is R (indicating purine), lines 17 to 22 are executed. Here, the algorithm checks each of the possible symbols for purine (A and G). For each symbol, it gets the corresponding child node in the tree and, if that node exists, recursively calls the IUPAC-SEARCH function.

On line 23, if the current character is Y (indicating pyrimidine), the algorithm executes lines 24 to 28. This block is similar to the previous one, but it checks each of the possible symbols for pyrimidine (C and T). Finally, line 31 notes that other IUPAC letters (such as S, W, K, M, B, D, H, V, N, X) should follow a similar format to the one explained above. In the end, the algorithm stores all the KMERS and their respective counts in the *patterns* map.

The primary advantage of the IUPAC-SEARCH over the brute-force algorithm lies in its superior computational efficiency. While the brute-force method proceeds to exhaustively analyze all the KMERS in the dataset, attempting to find matches with the IUPAC pattern, the IUPAC-SEARCH, on the other hand, operates more intelligently and selectively on the SMT. This latter algorithm excels in terms of speed, as it is capable of pruning the search tree. This means that nodes subordinate to patterns that do not meet the search criteria are not even considered, thus optimizing the search process.

2.6 Kdive

KDIVE was created with the objective of performing efficient text fragment searches in the SMT, even if these present up to `d` degenerations. In other words, KDIVE should return true for the search even if the query STRING contains up to `d` MISMATCHES. It is important to highlight that the algorithm assumes as a premise that the probability of a MISMATCH occurring is the same for any position in the sequence.

For example, if the probability of a MATCH is equal to $\frac{1}{4}$, then that of a MISMATCH is $1 - \frac{1}{4} = \frac{3}{4}$. Therefore, the occurrence of a MISMATCH is

three times more likely than that of a MATCH. This characteristic is important because it alters the expected number of computations carried out and consequently modifies the algorithm's complexity. The construction of KDIVE was based on limited depth search [**norvig2013**] and the Algorithm (BRANCH AND BOUND). Algorithm 7 shows the basic functioning of KDIVE.

Algorithm 7 KDIVE

1: **if** $k > d_{\max}$ **then**
2: **return** ▷ The search string exceeded $d_{\max}$ mismatches and the tree will be pruned.
3: **end if**
4: **if** $i \geq k$ **then**
5: $resp = 1$ ▷ Found the search string with up to d mismatches.
6: **end if**
7: $symbol = s[i]$
8: **for** $j = 1$ to 4 **do**
9: $next_{\text{node}} = M[\text{node}, j]$
10: **if** $next_{\text{node}} \neq 0$ and $symbol = j$ **then**
11: kdive$(M, s, d_{\max}, d, i+1, k, next_{\text{node}}, resp)$ ▷ Match!
12: **else**
13: kdive$(M, s, d_{\max}, d+1, i+1, k, next_{\text{node}}, resp)$ ▷ Mismatch!
14: **end if**
15: **if** $resp = 1$ **then**
16: **Break**
17: **end if**
18: **end for**

Algorithm 7 was implemented recursively and takes as its main parameters the SMT matrix, the fragment `s`, and the total number of allowed degenerations `d`. If $s \in M$ with at most `d` mismatches, the algorithm returns the boolean value TRUE, indicating the presence of the fragment with up to `d` mutations.

The algorithm performs its task by traversing the SMT to identify matches between the search STRING and the stored *kmers*, taking into account the number of allowed MISMATCHES. It is worth noting that this algorithm can be easily adapted to return the complete set of degenerate elements, instead of just a boolean value (TRUE or FALSE). This modification can be useful in certain applications, as it allows for more detailed information about the

matches found. For example, it is possible to identify which *kmers* in the SMT correspond to the search fragment, considering the allowed mutations.

The parameters $d_{\max}$, `d`, `i`, and `k` allow for the control of searching for exact or approximate matches. The recursion enables the algorithm to traverse the tree in depth and check all possible paths until it finds a match or reaches the maximum depth defined by the parameter $d_{\max}$. A possible call to Algorithm 7 could be `kdive(M, s,` $d_{\texttt{max}} = 2$`, d = 0, i = 1, k = 10, node = root, resp = FALSE)`. In this case, the algorithm expects to find a match even if up to $d = 2$ mutations are detected in sequences of size $k = 10$.

Algorithm 7 has two base cases, shown on lines 1 and 4. The first checks if the number of mutations has exceeded the limit $d_{\max}$, that is, if $d \geq d_{\max}$. If this condition is met, a pruning will occur in the search. The second base case checks if the algorithm has completely analyzed the query string, which will happen if $i \geq k$. If this condition is true, then the pattern has been found and the *resp* variable is updated to the value TRUE.

The time complexity of KDIVE can be measured through the Negative Binomial distribution. Consider a SMT with ν nodes, d MISMATCHES, and fragments of width k. Also consider the random variable $X \sim NB(d, p)$, which counts the number of comparisons that the KDIVE algorithm performs. It's easy to verify that $p = \frac{3}{4}$, since if the probability of a MATCH is $\frac{1}{4}$, the probability of a MISMATCH will be $1 - \frac{1}{4} = \frac{3}{4}$, therefore $X \sim NB(d, \frac{3}{4})$. The PMF $P_x(d, p)$ is given by Equation 1 and the expectation by Equation 2.

$$P_x(d, p) = \binom{n-1}{d-1} p^d (1-p)^{n-d} \tag{1}$$

$$E(X) = \sum_{x \in X} x P_x(d, p) \tag{2}$$

The Negative Binomial distribution is a generalization of the geometric distribution, which is defined as $X \sim GEOM(p) = NB(1, p)$. Therefore, the geometric distribution is the Negative Binomial distribution with $d = 1$, making it possible to use it for calculating the expected value. For example, consider $X \sim NB(1, p)$, $Y \sim NB(1, p)$, and $Z = X + Y$. If X and Y are independent, then $Z = X + Y \sim NB(2, p)$. Therefore, we can calculate $E(Z) = E(X) + E(Y) = \frac{1}{p} + \frac{1}{p} = \frac{2}{p}$. In this way, $X \sim NB(d, p)$ can be written as $Z = X_1 + X_2 + X_3 + \cdots + X_d$, where $X_i \sim NB(1, p)$. Calculating $E(Z) = E(X_1) + E(X_2) + E(X_3) + \cdots + E(X_d) = \frac{1}{p} + \cdots + \frac{1}{p} = \frac{d}{p}$.

With this result, we can write that the time complexity of KDIVE is $O\left(\frac{d\times 4}{3}\right) = O(d)$. It is easy to verify that the larger the size of d, the more operations will be necessary. In particular, the complexity grows linearly with d. It is important to highlight that MOTIFS rarely exceed a width of 20 nucleotides, and the number of mutations is generally no more than 20% of their size. Considering this, we can expect to find an average of $d = 5$ MISMATCHES, which results in $\frac{5\times 4}{3} \approx 6.666$ comparisons per fragment.

We can calculate the complexity of the KDIVE algorithm in relation to the size of the input sequences. Consider a dataset with n sequences of width t. We have a total of $m = t-k+1$ fragments of size k in each sequence of size t. Let X be the number of comparisons made for each fragment. We know that X follows a negative binomial distribution with parameters $p = \frac{3}{4}$ and d, and that $E(X) = \frac{d}{p} = \frac{4d}{3}$. Therefore, the number of comparisons per sequence will be $m\frac{4d}{3}$. If we consider, for example, a fragment of size $k = 20$ with $d = 5$ mutations, then $E(X) = \frac{5\times 4}{3} \approx 6.666$, which will result, on average, in $6.666\ldots \times m = O(m)$. It is interesting to highlight that this value will be even lower in practice because KDIVE uses the branch and bound algorithm to prune the search tree.

Even in the worst-case scenario, the algorithm will still exhibit linear asymptotic behavior. In a less detailed analysis, we arrive at the following complexity: $k(m-k+1) = km - k^2 + k$, in which the dominant term is k^2, and therefore the complexity of this algorithm will be of the order of $O(k^2)$. However, k is often much smaller than m, and in this way we can make a more refined analysis. Given that $k \ll m$, the expression $k(m-k+1)$ can be approximated as: $k(m-k+1) \approx km$.

The complexity of the algorithm is generally dominated by the term km and can be expressed as $O(km)$. However, considering that k is not greater than 20 and m can grow indefinitely, we can adjust the complexity analysis. As k has a constant upper limit ($k \leq 20$), the complexity of the algorithm is mainly influenced by the term m. Therefore, we can express the complexity of the algorithm as $O(m)$.

3 Real world application: improve DNA biological motifs finding

In this section, we will demonstrate how to utilize SMT for discovering conserved motifs in ChIP-seq data. Through a practical example, we aim to elucidate the application of the outlined algorithms in a real-world scenario, showcasing the effectiveness and efficiency of the SMT in processing genomic data to unveil biologically relevant patterns.

For the performance test, we employed the MA0047.3 dataset available in the JASPAR 2022 repository. The file containing the genomic positions (.bed) was obtained directly from this repository. Using the hg38 reference genome, it was possible to construct a dataset composed of 262,152 sequences, each containing 100 bases. This dataset served as a guide for subsequent analyses and experiments, demonstrating the applicability of SMT in the discovery of conserved motifs in ChIP-seq data.

The steps conducted for the analysis were as follows: 1) Acquisition of the genomic positions file (.bed) from the JASPAR repository; 2) Construction of the dataset from this file, resulting in 262,152 sequences of 100 bases each; 3) Preprocessing of the sequences, converting all bases to uppercase and employing the REPEAT MASKER and DUST utilities for the elimination of spurious sequences; 4) Kmer counting through the execution of SMT; 5) Extraction of the 10 most frequent kmers; 6) Application of the KDIVE algorithm with a value of `d = 3` to the most frequent kmers, generating 10 initial PWM (Position Weight Matrices) models for subsequent analysis.

Lastly, similar models were discarded, resulting in 3 final models that were used as input for the Expectation-Maximization algorithm (EM). The results of this experiment were compared to those obtained by the MEME algorithm [**bailey1995**], which also employs EM. Both algorithms were executed with the value of `k = 11`. The aim of this analysis was to demonstrate that the use of SMT for kmer counting, followed by passing these kmers as initial seeds to EM, can improve accuracy in the search for biological motifs, offering an effective alternative to the standard procedure.

Table 1 illustrates the results of the frequency analysis conducted by the SMT algorithm, highlighting the 10 most frequent kmers found in the dataset. Notably, all of these kmers share the core of the motif **GTAAACA**, suggesting a significant presence of this motif in the analyzed sequences. These kmers will serve as input for the KDIVE algorithm, which will identify

#	KMERS	COUNTS
1	AAAA**GTAAACA**	2055
2	AAA**GTAAACA**A	2024
3	AAAT**GTAAACA**	1877
4	TTAT**GTAAATA**	1723
5	AAT**GTAAACA**A	1692
6	TTAA**GTAAACA**	1652
7	TTAT**GTAAACA**	1644
8	AA**GTAAACA**AA	1614
9	TTAA**GTAAATA**	1612
10	TAA**GTAAACA**A	1598

Table 1: Results of the frequency analysis performed by the SMT algorithm. This table shows the 10 most frequent kmers. It is interesting to note that they all display the core motif **GTAAACA**. These kmers will be used by kdive to find sister kmers with up to `d = 3` mutations and later the models will be used in the EM optimization algorithm.

sibling kmers with up to `d = 3` mutations. Subsequently, the generated models will be used in the EM optimization algorithm for a more in-depth analysis in the search for biological motifs.

Figure 1 displays the sequence plots of biological motifs generated by SMT+EM. It is noticeable that the predominant sequence in the reference model resembles those identified by SMT+EM, indicating that this technique was able to identify the motif similarly to the reference standard. The predominant sequence identified by the three SMT+EM models aligns significantly with the reference model. Such congruence highlights the accuracy and effectiveness of the SMT+EM technique in motif discovery, offering a promising tool for future applications in bioinformatics and molecular biology.

Figure 2 shows the sequence logo generated by the MEME tool. In it, we observe distinct features compared to the results obtained by the SMT+EM technique. While MEME 1 highlights a predominance of cytosine in the central positions and guanine at the ends, MEME 2 displays a significant pattern of adenine, strongly resembling the JASPAR reference model, albeit with a shift of two nucleotides to the right. In contrast, the motifs generated by SMT+EM, as previously mentioned, maintained a more compact, homogeneous structure

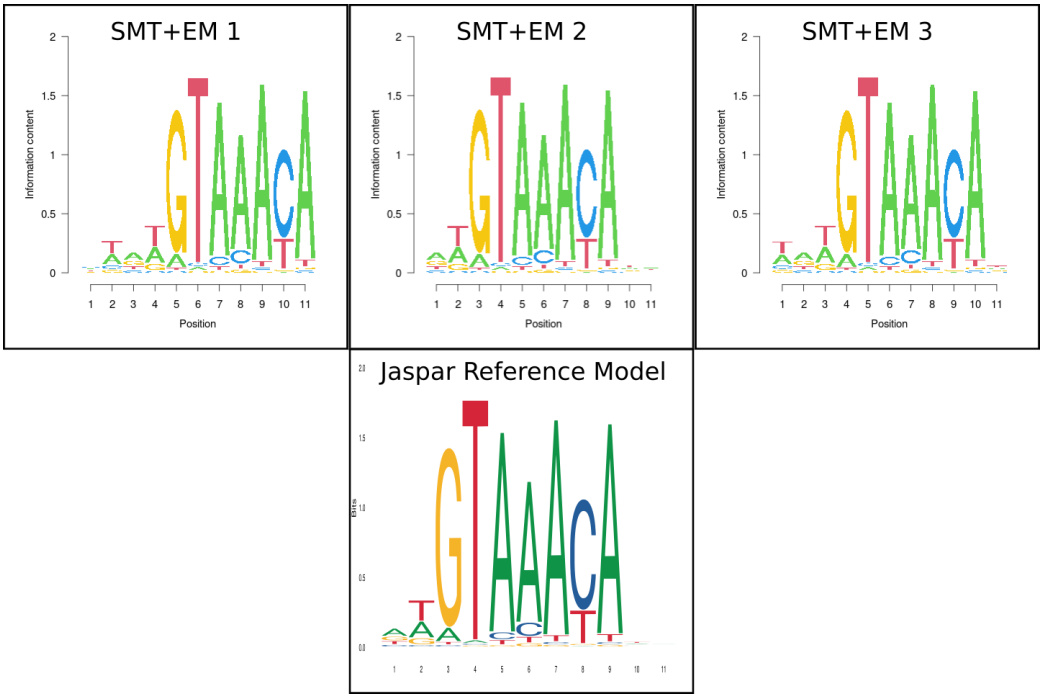

Figure 1: Sequence logos of the 3 best models generated by SMT. It is possible to visually verify that model 2 was the one that showed the greatest similarity with the reference model.

without shifts.

Table 2 presents a comparative analysis between the results obtained by the SMT+EM and MEME algorithms when applied to a specific dataset. The metrics used for this comparison are Manhattan distance (MAN), Euclidean distance (EUC), Kullback-Leibler divergence (DKL), and Hellinger distance (HELL). In Table 2a, which does not consider the shift relative to the reference model, the SMT+EM set surpasses MEME in all metrics, displaying lower values, indicating less divergence from the JASPAR 2022 reference model.

In Table 2b, which considers the shift, SMT+EM's performance remains superior, especially in the DKL metric, where SMT+EM shows a much lower value (0.001172194) compared to MEME (1.123026421). This suggests that SMT+EM, even when the shift is considered, maintains lower divergence from the reference model, potentially indicating higher accuracy in identifying biological motifs due to the k-mer counting provided by SMT. This superior performance of SMT+EM reinforces the efficacy of the k-mer counting process

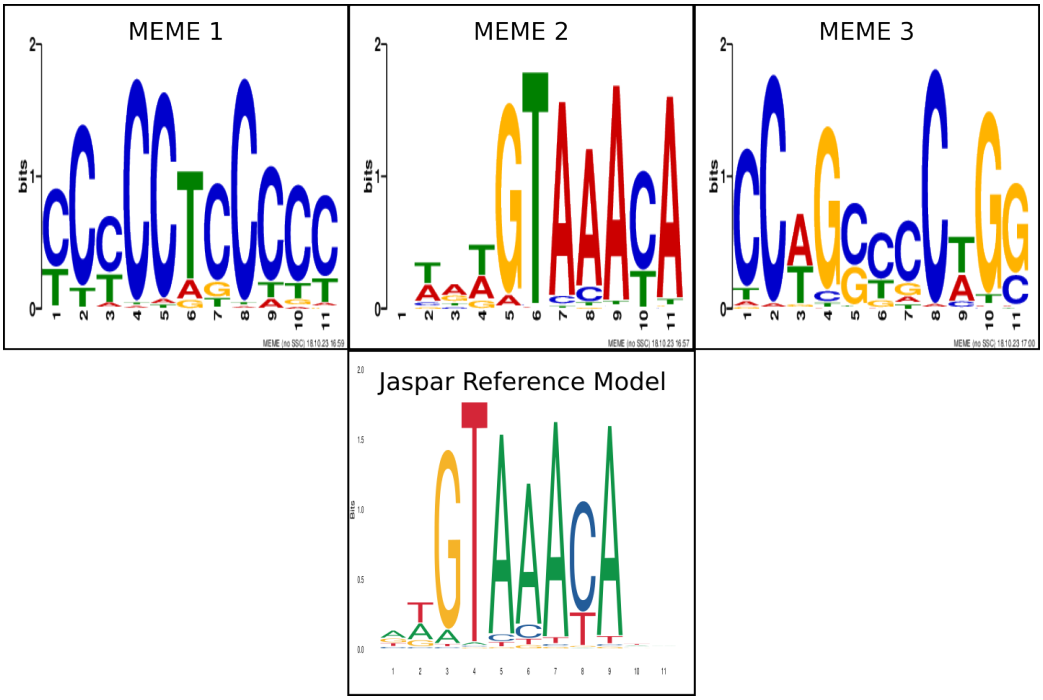

Figure 2: Sequence logos of the 3 best models generated by MEME. It is possible to visually verify that model 2 was the one that showed the greatest similarity with the reference model.

and the use of these as initial seeds for the EM algorithm, contributing to improving the accuracy of biological motif discovery.

Motif discovery is an important application, yet only one among many that can potentially be improved through efficient k-mer counting. The ability to count k-mers quickly and accurately facilitates a wide range of genomic and bioinformatic analyses, including but not limited to diversity analysis, variant detection, and metagenome analysis. This study demonstrates how effective k-mer counting, provided by the SMT algorithm, can be a fundamental step to improve precision and efficiency in complex computational tasks in bioinformatics.

	MAN	EUC	DKL	HELL
SMT	0.02570197	0.01108325	0.001172194	0.01789216
MEME	0.05009735	0.02045907	0.004809305	0.03531271

(a)

	MAN	EUC	DKL	HELL
SMT	0.02570197	0.01108325	0.001172194	0.01789216
MEME	0.97073591	0.44246565	1.123026421	0.42889992

(b)

Table 2: Performance comparison between the SMT+EM and MEME algorithms in several distance metrics. The values are derived from comparing the models generated by each algorithm with the reference model available in JASPAR 2022. Caption: MAN = Manhattan distance, EUC = Euclidean distance, DKL = Kullback-Leibler distance and HELL = Hellinger distance. (a) Metrics computed without considering the displacement in relation to the reference model. (b) Metrics computed taking into account the displacement in relation to the reference model.

4 Final considerations

In this supplementary material, we meticulously detail the algorithms employed, providing a clear view of their structures and operations. We illustrate, through a practical example, how efficient k-mer counting, facilitated by the SMT algorithm, can enhance the discovery of biological motifs. By comparing our results with the MEME algorithm, we observed significant improvement, highlighting the potential of our approach in providing more accurate and faster insights for complex genomic analyses. This exercise emphasizes the importance of effective k-mer counting and how it can be a valuable resource in bioinformatics.