

APPENDIX C: AN EXAMPLE

The subsequent sections delve into an in-depth elucidation of the proposed algorithm, employing an illustrative example to illuminate its intricacies. Consider an input dataset denoted as $In_seq(\mathcal{S})$ comprising five distinct protein sequences, namely $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5\}$. Within this dataset, variations in sequence length, $|\mathcal{S}_i|$, are acknowledged, and for simplicity, the protein alphabet \mathcal{A} is constrained to eight amino acid letters ($\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_8\}$). Upon establishing the $In_seq(\mathcal{S})$ dataset, essential parameters such as $kmer$, $Max(n)$, and $Min(Sim_thr)$ are user-defined to facilitate candidate motif identification. In this illustrative instance, the goal is to pinpoint five candidate motifs from the input dataset, each with a maximum length of three amino acids and a minimum similarity threshold of 80%. Accordingly, the parameters for $kmer$, $Max(n)$, and $Min(Sim_thr)$ in Algorithm 1 (Appendix A) are set at 3, 5, and 0.8, respectively.

Fig. 1 visualizes the initial stages of the TFEM algorithm. It introduces an array termed *Presence* for navigating protein sequences. The size of this array aligns with the number of elements in protein alphabet \mathcal{A} , in this case, 8. The *Primary_Candidates* function is invoked to meticulously assess individual sequences, marking the presence of each observed amino acid letter with a value of 1 (Lines 4-11 of Algorithm 2 in Appendix A). Notably, recurring amino acids within a sequence do not influence array element values. Following comprehensive amino acid assessment in each protein sequence, the algorithm calculates the participation sum of each identified character, $\mathcal{S}_{[i][j]}$, within the *Sum* array. This operation is carried out for each sequence within the input dataset (Lines 12-14 of Algorithm 2).

Line 17 computes character similarity and compares it with the predefined minimum similarity threshold. Thus, among amino acids participating in the $In_seq(\mathcal{S})$ set, only those with cumulative similarity exceeding the threshold, $\sum Sim(u) \geq Min(Sim_thr)$ (Lines 16-18 of Algorithm 2), are selected. Upon processing the $In_seq(\mathcal{S})$ set, the


Input (\mathcal{S})			A C D E F G H I									Presence Array
\mathcal{S}_1	A I C G C E H C G		\mathcal{S}_1	1	1	0	1	0	1	1	1	
\mathcal{S}_2	C E H A I G C G E D F		\mathcal{S}_2	1	1	1	1	1	1	1	1	
\mathcal{S}_3	A C E H G H		\mathcal{S}_3	1	1	0	1	0	1	1	0	
\mathcal{S}_4	I C E H G A C G		\mathcal{S}_4	1	1	0	1	0	1	1	1	
\mathcal{S}_5	C E H G C G F		\mathcal{S}_5	0	1	0	1	1	1	1	0	
		Sum Array	4	5	1	5	2	5	5	3		
		<i>Sim</i>	0.8	1	0.2	1	0.4	1	1	0.6		

Fig. 1. An example. The presence or absence of any amino acid in each input sequence.

algorithm designates five amino acids (A, C, E, G, and H) as primary characters for the candidate motif, owing to their higher similarity exceeding the stipulated 80% threshold. Subsequently, Algorithm 1 generates a hypothetical tree with nodes representing the five selected characters. Traversed using a BFS approach, the algorithm expands each node by appending an additional character (Fig. 2). This process continues until user-defined parameters for each node are met. All candidate nodes in the tree adhere to a predetermined expansion threshold. Employing BFS traversal, pruning nodes based on a score threshold ($similarity < threshold$), and eliminating duplicate nodes significantly enhances search efficiency, especially for extensive sequences.

Fig. 3 illustrates the sorting process by Algorithm 1. The *Extend_Motif* function is invoked for each candidate during protein sequence analysis. Upon processing sequences, the *Extend_Motif* function aggregates all developed candidates within the *SubCandidates* array. Let's consider a scenario where the initial candidate in *SubCandidates* is CE, displaying 100% similarity. Comparison with the last candidate in *AminoQ* follows, and if the last *AminoQ* element is smaller, TFEM treats it as a distinct non-duplicate element, adding it to the *AminoQ* queue.

Next, the *Extend_Motif* function introduces another extended candidate, CG. Comparing its size to the last *AminoQ*

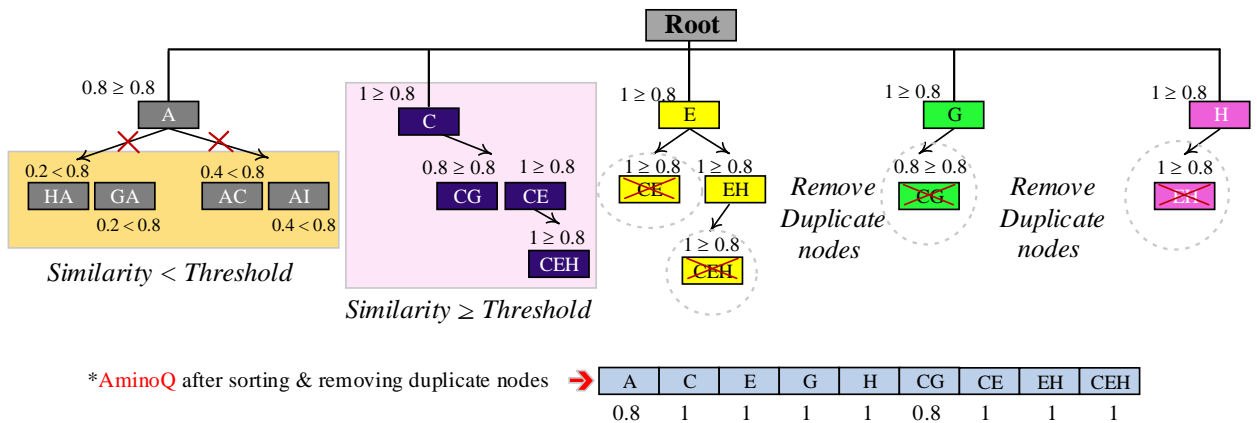


Fig. 2. Search for extensible letters of candidate characters using the branch and bound approach.

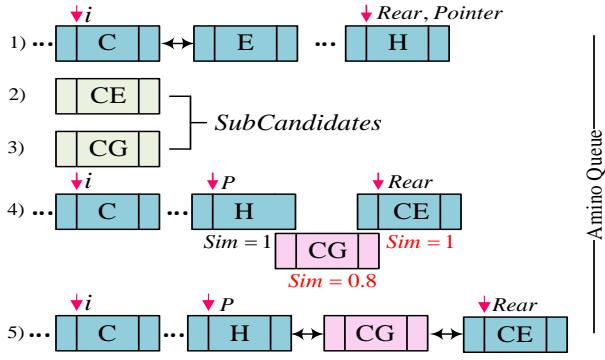


Fig. 3. Sorting of identified candidate motifs using pointer P.

candidate, Algorithm 1 inserts CG into the preceding CE location due to equal lengths ($|CG| = |CE|$). Algorithm 1 further compares CG's similarity with other two-character candidates in *AminoQ*, inserting it at the beginning when similarity is lower than CE (Algorithm 1, Lines 5-19), as depicted in Fig. 3. In the worst case, the comparison may continue until the first candidate, whose size is equal to that of the new candidate (candidates with the same length). In Fig. 3, the new CG candidate is inserted as the first two-letter candidate in the *AminoQ* queue because it exhibits lower similarity than the other two-letter candidates developed. Furthermore, Fig. 4 illustrates the algorithmic exploration of the C branch. Utilizing two 8-letter arrays, the algorithm identifies neighboring characters adjacent to candidate C. The algorithm then records the corresponding values in *Rpresence* and *Lpresence* arrays (Lines 1-13 of Algorithm 3 in Appendix A). Neighboring characters with total attendance exceeding the threshold are stored in the *SubCandidates* array. Algorithm 1 subsequently examines developed candidates in *SubCandidates*, inserting accepted candidates into appropriate *AminoQ* positions. In this example, the algorithm identifies a single motif composed of three amino acids.

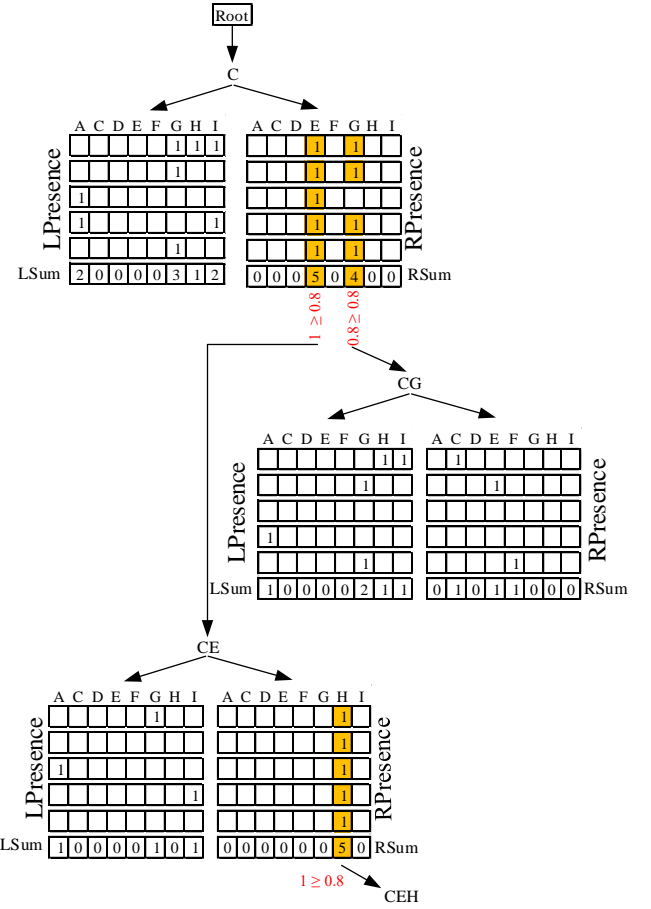


Fig. 4. Search for candidate subsequences utilizing *Lpresence* and *Rpresence* Arrays.