Accelerated pattern search in DNA sequences

Laia Rubio Castro

July 1, 2024

Resum– La bioinformàtica utilitza mètodes informàtics per operar amb moltes dades biològiques. Ens ajuda a entendre com funciona el DNA per crear i mantenir vius els éssers vius. Una tasca important és trobar patrons en seqüències de DNA que controlen els gens. Aquest mètode consisteix a analitzar grans conjunts de dades genòmiques i trobar posicions amb patrons. Aquest projecte vol accelerar el procés de recerca d'aquestes seqüències de genòmica mitjançant un enfocament look-ahead. Les millores s'afegiran a BioPython, un conjunt d'eines gratuïtes que ajuden a estudiar la computació biològica. El projecte contribueix a estudiar els programes genètics dins de les cèl·lules i accelerar el procés de recontrucció de xarxes de regulació amb presència genètica.

Paraules clau– Bioinformàtica, binding de proteïnes, ADN, motiu seqüències d'ADN, BioPython, xarxes de regulació

Abstract– Bioinformatics uses computer methods to handle large amounts of biological data. It helps us understand how DNA works to create and keep living beings alive. One important task is find instances of patterns in DNA sequences that control genes expression. This method consists on looking through large genomics datasets and identifying likely pattern positions. This project seeks to accelerate the process of finding these genomics sequences using heuristic approaches. The improvements will be added to BioPython, a set of free tools which helps studying biological sequence analysis. The project contributes to the inference of genetic programs within cells and to accelerate the process of reconstructing transcriptional regulatory networks.

Keywords– Bioinformatics, protein binding, DNA, patterns, DNA sequences motifs, biological computation, BioPython, regulatory networks

1 INTRODUCTION

BIOINFORMATICS is a scientific discipline that uses computer technology to gather, store, analyze, and distribute biological data and information, such as DNA sequences, as well as annotations related to these sequences. Scientists and medical professionals rely on databases to organize and index this biological information, increasing our understanding of health and disease and, in some cases, integrating them into medical practices. Nowadays, the challenge is no longer obtaining this information but knowing how to understand and interpret it, as bioinformatics works with large genomic datasets seeking practical insights into their complexity.

A gene is a segment of DNA that provides instructions for creating a specific protein. While the majority of humans share the same genes in a similar sequence, with over 99.9% of DNA being identical across individuals, variations exist. On average, there are 1-3 letter discrepancies per gene among individuals. These differences can alter protein structure and function, as well as affect its production timing, quantity, and location [14].

Transcription is the process by which a DNA sequence is converted into RNA, which can then be translated into a protein. It plays a crucial role in regulating genes and responding to changes in the cellular environment. This process is tightly regulated to ensure that the right genes are expressed at the right time and in the right amounts. Proteins, in turn, respond to changes in the cellular environ-

• Contact e-mail: 1600830@uab.cat
• Specialization: Computació
• Work tutored by: Ivan Erill Sagales (Department of Information and Communications Engineering)
• Course 2023/24

ment by interacting with other molecules, modifying gene expression, and carrying out specific functions within the cell.

As related to gene analysis, sequence motifs are becoming increasingly significant in determining genetic regulatory networks and understanding the regulatory mechanisms of individual genes. DNA sequence motifs are short, repetitive patterns found in DNA, believed to serve a biological purpose. They frequently indicate specific binding sites, where proteins such as transcription factors attach to regulate transcription [9][10].

Consensus sequences are used to describe the typical binding sequence for DNA-binding proteins and are important in understanding gene regulation and protein-DNA interactions. Creating a consensus sequence is one way of representing the most common base at each position within the motif. It is derived from aligning multiple instances of the motif and determining the most frequently occurring base at each position.

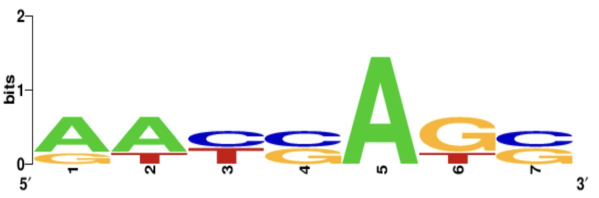
However there's a more informative depiction of the motif, sequence logos. Unlike consensus sequences, they display the frequency of each base at each position within the motif, as well as the degree of conservation at each position. This leads to a more detailed understanding of the motif, including the variability and importance of different bases at specific positions, offering a more comprehensive and visually informative representation of DNA sequence motifs [9][21].

```
CON aatgAgg  7
seq1 AATCAGG
seq2 ATTCAGC
seq3 AACGAGC
seq4 GACGATG
```

[1]

	1	2	3	4	5	6	7
A	0.75	0.75	0.00	0.00	1.00	0.00	0.00
C	0.00	0.00	0.50	0.50	0.00	0.00	0.50
T	0.00	0.25	0.50	0.00	0.00	0.25	0.00
G	0.25	0.00	0.00	0.50	0.00	0.75	0.50

[2]



[3]

Fig. 1: [1]Consensus sequence [2]PSFM [3]Sequence logo

In order to generate sequence logos we can use as basis a Position-Specific-Frequency Matrix (PSFM) by converting the frequency information in a PSFM into graphical representations [Fig.1]. A PSFM is derived from a list of binding sites for a specific protein, where the relative frequencies of each nucleotide at each position within the binding motif are measured. These frequencies denote the conditional probabilities of observing a particular base at a specific position in the motif given the protein's binding. Mutual information, calculated from PSFM and background genome frequencies, represents the reduction in uncertainty about base occupancy in a sequence upon observing protein binding as shown in sequence logos [Fig.1] [8][18].

As each cell of the PSFM is the frequency *P(ji)* of the nucleotide *j* at position *i*, if we assume all nucleotides are equiprobable, therefor, the background frequency *P(j)* is 0.25. We can compute the log-likelihood as

$$\log_2(\frac{P(ji)}{P(j)})$$

for each cell of the PSFM, obtaining the Position Specific-Scoring Matrix (PSSM), which we'll use for the motif site search methods.

2 Objectives

DNA sequence motifs enable biomedical researchers to reconstruct transcriptional regulatory networks by scanning genome sequences and predicting putative binding sites for transcription factors, inferring which genes are regulated by each transcription factor in a given genome. However, the analysis of large volumes of genome data is predicated on the availability of methods that enable fast scanning to predict instances of a particular DNA sequence motif. Here's a structured approach of the main objectives:

- Understanding DNA sequence motifs: Identifying and characterizing DNA sequence motifs is fundamental for understanding gene regulation, evolutionary relationships, and functional genomics.
- Exploring different site search methods: Various methods are available for identifying instances DNA sequence motifs, each employing distinct strategies and algorithms. Naïve methods involve straightforward approaches like Position-Specific Scoring Matrix (PSSM) sliding windows, systematically scanning the genome sequence for potential motifs instances. Look-ahead methods enhance efficiency by stopping the search if a window cannot yield a score above a specified threshold, optimizing the search process. Permuted look-ahead techniques prioritize positions by importance before initiating the look-ahead process, further optimizing motif search. Enumerative methods enumerate and score all N-mers above a threshold, followed by string searches on the genome sequence to identify motifs instances.
- Implementing an accelerated algorithm: Developing an algorithm with a lookahead perspective involves considering future steps or potential outcomes during the search process. Prioritize optimizing critical components of the algorithm, such as motif scoring and alignment, to accelerate the overall search process to validate the efficiency and effectiveness of the accelerated approach against existing algorithms
- Integration with BioPython: The new algorithm or method developed for motif searching should align with Biopython's modular architecture and coding standards in order to integrate it into the project, ensuring compatibility and usability.

3 State of the art

3.1 Motif site search

The identification of motif instances is a crucial aspect of genomics analysis, aiming to identify motif instances within DNA sequences that are indicative of functional elements such as transcription factor binding sites or regulatory regions [17][11]. Here's a more detailed explanation of some of the existing methods:

- Naïve PSSM sliding window: This method involves scanning the genome sequence using a Position-Specific Scoring Matrix (PSSM) within a sliding window. The PSSM is a scoring matrix representing the binding affinity of a protein for DNA. The sliding window technique involves moving a fixed-size window along the genome sequence and scoring each position based on its similarity to the motif represented by the PSSM [17].

- Look-ahead enhanced PSSM sliding: Similar to the naïve approach but with an added optimization. The search stops if the window being scanned cannot submit a score above a certain threshold. This optimization reduces unnecessary computations by avoiding scoring windows that are unlikely to contain significant matches [24][17].

- Permuted look-ahead: Before performing the look-ahead search, positions within the genome sequence are sorted by importance. This sorting allows prioritizing more relevant regions, potentially improving search efficiency [17].

- Enumerative: This method involves enumerating and scoring all N-mers (sequences of length N) above a certain threshold. After scoring, a string search for selected N-mers is performed on the genome to identify matches. This approach is exhaustive but may be computationally expensive, especially for larger motifs [15].

- Look-ahead enumerative: Similar to the enumerative approach but with a look-ahead optimization. Only valid N-mers are enumerated using the look-ahead technique. The Aho-Corasick search algorithm may be utilized for efficient string searching [12].

- Suffix tree/array: This method involves constructing a suffix tree or array from the sequence. Suffix trees are data structures that allow fast substring search and motif identification [17].

- Super-alphabet search: Increases the alphabet size to include k-mers (sequences of length k). This approach decreases computation with increasing alphabet size [17] [16].

- Fourier transform: Transforms the sequence-motif alignment problem into the frequency domain using Fourier transform techniques. This approach can provide insights into periodic patterns within the sequences [17].

- Compression-based: Uses classical run-length compression techniques to compress the sequence. Motifs are identified based on the compressed sequence and scoring is performed accordingly [17].

3.2 BioPython

Biopython is an open-source set of Python tools for computational biology and bioinformatics, created by a global team of developers. Offering a range of classes for representing biological sequences and annotations, it offers versatility in reading and writing different file formats. It also helps with programmatic access to online repositories of biological data.

When BioPython was first launched in 2000, its primary focus was accessing, indexing, and processing biological sequence files. Although this remains a central aspect, over the following years the integration of additional modules, extended its functionality over other domains within biology [23][7].

We can access its development tools, tutorials and the repository in the official web page [4].

4 METHODOLOGY AND PLANNING

To carry out this project, we'll use the Scrum methodology [19], which is an Agile project management framework that focus on iterative progress, collaboration and flexibility to overcome possible changes. It divides the project into iterations called sprints.

To develop this project, we identified a sequence of tasks to outline the most crucial steps to be taken, represented into three primary phases:

- Initial Phase: In this initial stage, we will analyse and explore the subject matter to establish objectives, planning, and the methodology to be followed.

- Development Phase: During the development phase, we will implement the proposed models, improvements and additional functionalities to get the final results.

- Final Phase: Once we obtain the results, we will present and complete the project documentation.

We will carry out the events in 2-week Sprints, in which each Sprint will have defined objectives with fixed tasks to be performed.

5 DEVELOPMENT

5.1 Setup

This phase was important to understand BioPython and identify necessary packages and the procedures to implement changes and publish updates.

We followed the BioPython tutorial [15], using their github page [5], to get practical knowledge.

To get a better knowledge base, we practiced using self-generated data and real-world datasets extracted from the JASPAR database, a repository of transcription factor binding profiles [20], as well as datasets from CollecTF, a database of transcription factor binding sites in the Bacteria domain implement more topic-oriented examples [6].

BioPython provides simplicity and accessibility, but some bioinformatics applications require high-speed execution. Tasks like large-scale pattern search often require optimizations and can only be achieved by low-level languages like C.

C, as a compiled language, tends to be faster because it's translated directly into machine code by the compiler, while Python as an interpreted language is executed line by line by the interpreter. In tasks where performance is essential, such as pattern search in large DNA sequences, the speed advantage can make a difference.

In addition, C provides direct control over memory management, enabling efficient memory allocation and deallocation. This level of control can be important when working with large data sets, as it enables more accurate memory optimization compared to Python's automatic memory management system.

A C wrapper is a piece of C code that provides an interface to C library, so it can be called from other programming languages. It acts as a bridge between C code and higher-level languages, allowing to easily integrate C functions into Python scripts [13][22].

As Python and C have complementary strengths, by combining them and integrating C functions into Python workflows via a C wrapper we enable access to their functionalities without sacrificing performance.

5.2 Background

Our main goal is accelerate pattern search in DNA sequences. To obtain it, we took as a starting point the previously reported advantages of the Look-ahead, Permuted look-ahead and Super-alphabet algorithms [17][16]:

- Look-ahead algorithm: At each step of the score computation, there is a maximum score that can be added based on the matrix entries. For a segment *s* to be considered as a match, its final score *Sc(s)* must meet or exceed the given threshold T. This implies that if the partial score reached at the *(m-1)th* step is less than *T-Vmax[m]*, where *Vmax[m]* is the maximum value in the *m-th* row, then there is no need to perform the final comparison because the segment will never reach the threshold. This is the same for all intermediate positions between 1 and *m*, and it involves computing the minimum threshold score *Ti* that must be reached at the *i-th* comparison step.

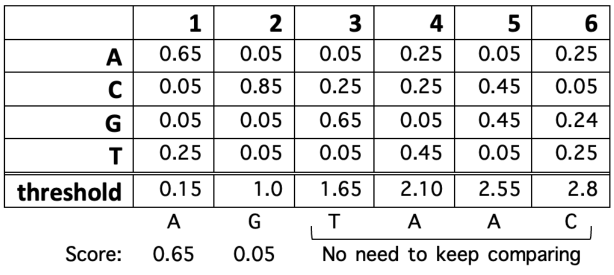
- Permuted look-ahead algorithm: As the purpose of lookahead is to eliminate a non-matching segment as soon as possible and there is no technical limitation on computing the score in a different order, by arranging the matrix rows differently it may result in higher intermediate thresholds during the initial comparison steps. This can lead to an earlier drop of a segment.

- Super-alphabet algorithm: This technique is used to accelerate algorithms which are often slow in bioinformatics applications due to the small size of the original alphabet. To create the superalphabet, we fix an integer width *q* for the alphabet and define each *q-tuple* of the original alphabet as a superalphabet symbol. The search time is *O(nm/q)*, which provides a theoretical speedup by a factor of *q* independent of the threshold.

5.3 First Steps

Following the planned schedule we focused on the operation of the previous algorithms for their subsequent implementation:

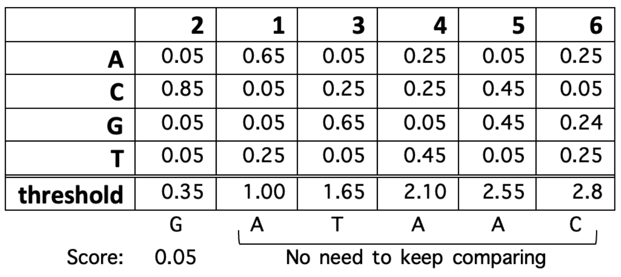
- Look-ahead algorithm:
    1. Max score estimation: Maximal score computation for the maximal value that can be added in every scoring step, which is based on the cells of the scoring matrix. This maximal score is determined by finding the largest number from each column of the matrix.
    2. Calculation of threshold: A minimum threshold score should be determined at every iteration of comparison and this may be done by calculating the maximum possible scores that can be obtained starting from any given position up to the last cell.
    3. Threshold comparison: To be considered as a match, a segment of this sequence must have a final score greater than or equal to the threshold value. At each similarity check stage, we have to establish several intermediary thresholds.
    4. Segment assessment: It calculates partial scores for each segment at every step. Wherever there exists a part-score less than an intermediate threshold, it allows cutting off such segments without completing the full comparison.



	1	2	3	4	5	6
A	0.65	0.05	0.05	0.25	0.05	0.25
C	0.05	0.85	0.25	0.25	0.45	0.05
G	0.05	0.05	0.65	0.05	0.45	0.24
T	0.25	0.05	0.05	0.45	0.05	0.25
threshold	0.15	1.0	1.65	2.10	2.55	2.8
	A	G	T	A	A	C
Score:	0.65	0.05	No need to keep comparing			

Fig. 2: Look-ahead example from C.Pizzi, DEI -Univ. Of Padova (Italy)

- Permuted look-ahead algorithm:
    1. Permutation matrix: To optimise the performance in scoring segments the order of matrix columns is permuted based on a specific criteria.
    2. Calculation of scores: The algorithm calculates the maximal and the expected score based on background frequencies and the matrix entries for each step of the scoring process.
    3. Permutation criteria: The rows of the scoring matrix are sorted based on the difference between the maximal and expected scores to prioritise rows that have a higher impact on the score calculation.
    4. Segment scoring: It evaluates the partial score of each segment using the permuted matrix. If the partial score is under a certain threshold, the algorithm can discard the segment early without completing the full comparison.

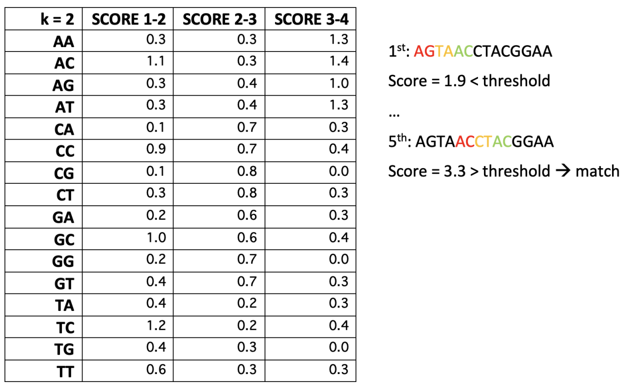


	2	1	3	4	5	6
A	0.05	0.65	0.05	0.25	0.05	0.25
C	0.85	0.05	0.25	0.25	0.45	0.05
G	0.05	0.05	0.65	0.05	0.45	0.24
T	0.05	0.25	0.05	0.45	0.05	0.25
threshold	0.35	1.00	1.65	2.10	2.55	2.8

G A T A A C
Score: 0.05
No need to keep comparing

Fig. 3: Permuted look-ahead example from C.Pizzi, DEI - Univ. Of Padova (Italy)

- Super-alphabet algorithm:
  1. Super-alphabet creation: It groups multiple symbols from the original alphabet to create a new alphabet that represents multiple symbols as a single one in order to reduce the search space complexity.
  2. Preprocessing: The algorithm uses the super-alphabet and the original matrix to obtain an equivalent scoring matrix for the super-alphabet symbols.
  3. Searching process: It searches for patterns or matches using the symbol sequences and scoring matrices.
  4. Matching pattern: It applies pattern matching techniques to identify specific patterns within the sequences.



k = 2	SCORE 1-2	SCORE 2-3	SCORE 3-4
AA	0.3	0.3	1.3
AC	1.1	0.3	1.4
AG	0.3	0.4	1.0
AT	0.3	0.4	1.3
CA	0.1	0.7	0.3
CC	0.9	0.7	0.4
CG	0.1	0.8	0.0
CT	0.3	0.8	0.3
GA	0.2	0.6	0.3
GC	1.0	0.6	0.4
GG	0.2	0.7	0.0
GT	0.4	0.7	0.3
TA	0.4	0.2	0.3
TC	1.2	0.2	0.4
TG	0.4	0.3	0.0
TT	0.6	0.3	0.3

1st: AGTAACCTACGGAA

Score = 1.9 < threshold

...

5th: AGTAACCTACGGAA

Score = 3.3 > threshold → match

Fig. 4: Super-alphabet example from C.Pizzi, DEI -Univ. Of Padova (Italy)

5.4 IDE Configuration

As this project requires the ability of working with two different programming languages we looked for an environment that could support that.

Our first option was to use PyCharm or CLion with the proper extensions as we were familiar with the environments. However, despite it seemed intuitive we couldn't find the proper official extensions for each language and therefore we couldn't confirm its effectiveness and use.

Our final choice was the IDE Visual Studio Code [2] as we had already worked with it, it had the Microsoft official extensions and it seemed intuitive and easy to configure. Once the program was correctly installed we had to install the extensions that fulfilled our requirements from the extensions tab.

On one side we had C programming, for which we needed C/C++, C/C++ extension pack and CMake Tools to get a useful workflow.

On the other side, we had Python programming, for which we needed Python and Python debugger extensions. All of them from Microsoft as a verified publisher.

Despite the functions we implemented were in C, we needed to make sure we had Python installed in our computer as we used a Python interpreter to execute the program. Moreover, we had to modify from the .vscode folder the c_cpp_properties.json file to include the path where the python libraries were located to use the header Python.h in the C source file to be able to execute it with Python. A guided tutorial of the process is presented in the readme file of the repository [3].

We faced several challenges while setting up our project on a Mac with M1 SoC. The primary issues were detecting the paths for Python libraries and handling autogenerated files from the modules with incompatible extensions for the GCC compiler. Our objective was to execute the program by addressing the path detection problem and ensuring compatibility of the generated files with the compiler.

We suspected that the incompatibility issues were caused by the differences between Clang, which was the compiler used by the computer, and the GCC compiler that the project expected. To test this hypothesis, we adopted a try-and-test approach. We analyzed the build process, identified the specific areas where Clang's behavior differed from GCC and experimented with various configuration adjustments.

Upon reconfiguring the environment from scratch on a Windows computer and following the same steps, we observed a smooth setup process and successfully executed the program. This confirmed our initial hypothesis that the incompatibility issues were indeed caused by the differences between Clang and GCC compilers on the Mac.

Once we had everything configured and knew the steps we had to follow we started with the implementation of each algorithm. As we mentioned before, our project contained code in C from each searching algorithm therefore, we created 3 C files. To incorporate this files to python, we created a setup.py file which was called from Pyhton to use the algorithms and finally a main.py file to test the proper execution and performance of each function.

The full implementation with comments for a better understanding can be found in the repository [3].

5.5 Setup file

This file is essential for creating Python modules that wrap around C functions, enabling the efficient execution of algorithms in Python.

This script uses Setuptools, a popular library for packaging Python projects, to build and distribute these extensions. Its primary purpose is to configure and automate the process of compiling and installing C extensions for Python, which allows the execution of C-implemented algorithms within a Python environment, making it easier for developers to distribute their algorithms as Python packages. Additionally, it ensures that the C extensions can be easily installed and used by others, following Python's packaging and distribution conventions.

It starts by importing the Setup and Extension functions from Setuptools. The Setup function is used to specify the configuration for the package, and Extension is used to define the C extensions.

The script begins by creating an Extension object named look_ahead with the source file lookAhead.c to define the look_ahead module. The setup function then configures the package with metadata such as the name: look_ahead, version: 1.0, and a description of the package.

Similarly, it defines the permuted_look_ahead and super_alphabet modules, each with their respective Extension objects and source files, and configures the package with appropriate metadata.

5.6 Look-ahead implementation

The planned design was successfully implemented, with the algorithm being coded, tested, and integrated into a Python module.

The primary tasks involved designing the algorithm, implementing it in C, wrapping the C implementation into a Python module, and testing and validating the module with sample data.

It begins by defining the function and parsing its arguments. The function look_ahead takes a DNA sequence (s), a scoring matrix (scoring_matrix_obj), the width of the matrix (m), and a threshold (t) as inputs, which are parsed using PyArg_ParseTuple. Dynamic memory allocation is then performed for the scoring matrix, which is a 2D array where each row represents a position in the motif, and each column represents one of the four nucleotides (A, C, G, T).

The algorithm calculates the maximum possible score at each position of the scoring matrix and stores these values in an array (max_scores).

It then creates an array (Z) to hold the cumulative maximum possible scores from each position to the end of the scoring matrix, which helps in determining the intermediate thresholds (T_intermediate) used to decide the early drop off of sequence evaluation.

The algorithm processes each segment of the DNA sequence, calculating the total score for each position in the segment. If the cumulative score doesn't reach the intermediate threshold at any point, the evaluation for that segment stops early, providing a speedup over the basic scoring approach. Segments that meet or exceed the score threshold are added to the result list, which is returned at the end.

5.7 Permuted look-ahead implementation

As both algorithms use the same basis, once we checked the look-ahead algorithm was working properly we implemented the additional functions to get the permuted look-ahead.

The main tasks involved creating the remaining parts that differ from the look-ahead, implementing them in C, converting them into a Python module, and testing the module with sample data. The algorithm's permuted_look_ahead function takes a DNA sequence (s), a scoring matrix (scoring_matrix_obj), the matrix width (m), and a threshold (t) as inputs, which are parsed using PyArg_ParseTuple.

Dynamic memory allocation followed the same process. The permute_scoring_matrix function is used to permute the scoring matrix, calculating maximum scores for each column. Columns are sorted based on these scores to optimize the scoring process.

The algorithm computes the maximum possible score at each position of the permuted scoring matrix and stores these values in an array (max_scores). Another array (Z) is created to store the cumulative maximum possible scores from each position to the end of the scoring matrix to determine intermediate thresholds (T_intermediate) to decide early termination of sequence evaluation.

The algorithm processes each segment and return a list following the same methodology previously explained.

Testing with various DNA sequences and scoring matrices confirmed that the algorithm correctly identifies motifs exceeding the score threshold. The permuted look-ahead approach significantly reduces computational load by dropping evaluations early when further calculations cannot meet the threshold.

5.8 Super-alphabet implementation

The previous task of setting up the project IDE took significantly longer than expected due to multiple errors and challenges.

We needed to not only resolve these setup issues but also ensure the accurate implementation of our algorithms. However, the complexities involved in the setup made our focus goes towards error fixing rather than algorithm development and testing.

During this process, we encountered some errors that required time and effort to resolve. Additionally, the implementation of the algorithms had to be reviewed several times because some internal components were not well defined. This process was necessary to make sure the algorithms were correctly implemented but added to the delays.

These delays affected the overall timeline of the project. Although we implemented the super-alphabet algorithm, we were unable to fully test it due to the unexpected time spent on setup and redefining. As a result, we cannot confirm its correct performance and, therefore, it won't be presented in this version of the project. This approach ensure that only tested and verified components are included in the project.

5.9 Results and Discussion

First, we ensured that the results we obtained were accurate by verifying that the scores and positions matched those calculated using the BioPython method [**??**].

We then evaluated the performance of our implemented algorithms Look-ahead (LA) and Permuted look-ahead (PLA) by measuring their execution times across various real test cases. These test cases use the motifs defined by the binding sites of LexA, CRP, H-NS, ArcA, and RutR and the *E.coli* genome as the sequence to search, downloaded in FASTA from NCBI [1].

The execution times for each algorithm are presented in this graph. It illustrates the execution times for the BIO PSSM, which is the method provided by BioPython that also uses a PSSM matrix to search matches, the LA and the PLA algorithms across the five test cases, with each algorithm represented by a distinct color.

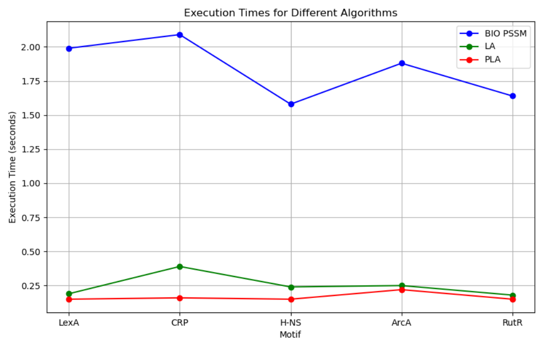


Fig. 5: Comparative analysis of execution times across different cases

The BIO PSSM algorithm consistently showed the highest execution time across all test cases, ranging from 1.58 seconds (H-NS) to 2.09 seconds (CRP), indicating that it is less time efficient compared to the other algorithms.

The LA algorithm showed a significant reduction in execution time compared to BIO PSSM, with execution times ranging from 0.18 seconds (RutR) to 0.39 seconds (CRP).

The PLA algorithm achieved the lowest execution times in most cases, ranging from 0.15 seconds (LexA, H-NS, RutR) to 0.22 seconds (ArcA).

The average execution times of each algorithm across all test cases are 1.836 seconds for BIO PSSM, 0.25 seconds for LA and 0.166 seconds for PLA. We used these averages to calculate the improvement percentages.

We first calculated the improvement percentage of the LA compared to the existing searching method BIO PSSM, which is approximately 86.41%. This proves that our implementation indeed accelerated the pattern search process.

As the LA method shares the same basis as the PLA and the additional steps made on the latter are theoretically made to speed up the performance, we then compared the improvement percentage of the PLA compared to the LA, which is approximately 33.6%. This confirms that our implementation meets the expectations with an overall improvement percentage compared to BIO PSSM of approximately 90.95%.

A more comprehensible way to understand the results is by calculating the speed up, which is a measure of how much faster an improved algorithm performs compared to a reference algorithm. It's calculated by the following formula:

$$SpeedUp = \frac{ExecutionTimeOfReferenceAlgorithm}{ExecutionTimeOfImprovedAlgorithm}$$

We obtained that the LA is approximately 7.344 times faster than the BIO PSSM algorithm. And that the PLA algorithm is approximately 11.06 times faster than the BIO PSSM algorithm and 1.506 times faster than the LA algorithm.

Additionally, we evaluated the performance of these algorithms on datasets with longer and shorter self-generated motifs to further validate the theoretical independence between motif length and execution speed. This is represented in the following graphic.

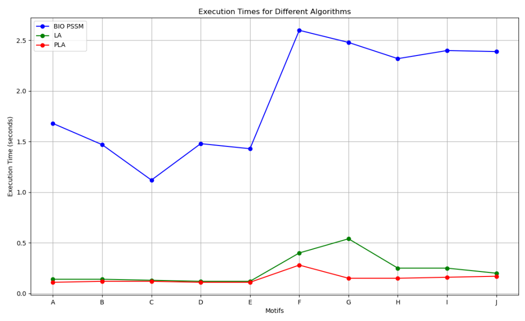


Fig. 6: Comparative analysis of execution times across different self-generated cases

For longer motifs, the BIO PSSM algorithm again shows the highest execution times. In contrast, LA and PLA algorithms maintain significantly lower execution times, demonstrating their efficiency.

For shorter motifs, both LA and PLA algorithms show even lower execution times, with minimal variation, further supporting the theoretical independence between motif length and execution speed.

In conclusion, the comparative analysis shows that the LA and PLA algorithms are more efficient than the BIO PSSM method. This validates that our implementations not only match but exceed the efficiency of existing solutions, ensuring faster and more effective DNA sequence motif analysis.

5.10 Future Work

We have fully accomplished the first two objectives, allowing us to implement and validate the motif search algorithms (LA and PLA).

Due to the delays we encountered, we prioritized ensuring the fully correct performance of these two algorithms and comparing them to the existing method in BioPython to evaluate the results. As a result, we have declared the development and testing of the super-alphabet (SA) algorithm out of our scope for the moment, and it can be considered for future work.

Despite the challenges and delays, these experiences provided us with valuable learning opportunities in unexpected areas. We will focus on integrating the LA and PLA algorithms into BioPython, thereby expanding the toolset available for DNA sequence motif analysis and achieving our primary objective.

Acknowledgments

I am very thankful to my tutor, Ivan Erill, for providing assistance and guidance during this project. His insights, encouragement, and support were crucial to the successful completion of this project.

References

[1] Ncbi: National center for biotechnology information, Accessed: 10th of may of 2024.

[2] Visual studio code, Accessed: 21th of april of 2024.

[3] Github repository, Accessed: 26th of may of 2024.

[4] Biopython, Accessed: 5th of march of 2024.

[5] Biopython github, Accessed: 6th of abril of 2024.

[6] Collectf, Accessed: 9th of abril of 2024.

[7] B. Chapman and J. Chang. Biopython: Python tools for computational biology. *ACM Sigbio Newsletter*, 20(2):15–19, 2000.

[8] DataVersity. Motifs and mutations: The logic of sequence logos, Accessed: 5th of march of 2024.

[9] P. D'haeseleer. What are dna sequence motifs? *Nature biotechnology*, 24(4):423–425, 2006.

[10] I. Erill. A gentle introduction to... information content in transcription factor binding sites. 2010.

[11] L. Gao, W. Bao, H. Zhang, C.-A. Yuan, and D.-S. Huang. Fast sequence analysis based on diamond sampling. *Plos one*, 13(6):e0198922, 2018.

[12] S. Hasib, M. Motwani, and A. Saxena. Importance of aho-corasick string matching algorithm in real world applications. *Journal Of Computer Science And Information Technologies*, 4:467–469, 2013.

[13] G. K. Kloss. Automatic c library wrapping ctypes from the trenches. 2009.

[14] National Human Genome Research Institute. National human genome research institute, Accessed: 5th of march of 2024.

[15] C. G. Nevill-Manning, K. S. Sethi, T. D. Wu, and D. L. Brutlag. Enumerating and ranking discrete motifs. In *ISMB*, volume 5, pages 202–209, 1997.

[16] C. Pizzi, P. Rastas, and E. Ukkonen. Fast search algorithms for position specific scoring matrices. In *International Conference on Bioinformatics Research and Development*, pages 239–250. Springer, 2007.

[17] C. Pizzi and E. Ukkonen. Fast profile matching algorithms—a survey. *Theoretical Computer Science*, 395(2-3):137–157, 2008.

[18] RCSB PDB. Sequence motif search, Accessed: 5th of march of 2024.

[19] S. Sachdeva. Scrum methodology. *Int. J. Eng. Comput. Sci*, 5(16792):16792–16800, 2016.

[20] A. Sandelin, W. Alkema, P. Engström, W. W. Wasserman, and B. Lenhard. Jaspar: an open-access database for eukaryotic transcription factor binding profiles. *Nucleic acids research*, 32(suppl_1):D91–D94, 2004.

[21] T. D. Schneider and R. M. Stephens. Sequence logos: a new way to display consensus sequences. *Nucleic acids research*, 18(20):6097–6100, 1990.

[22] K. W. Smith. *Cython: A Guide for Python Programmers.* " O'Reilly Media, Inc.", 2015.

[23] Wikipedia contributors. Biopython, Accessed: 5th of march of 2024.

[24] T. D. Wu, C. G. Nevill-Manning, and D. L. Brutlag. Fast probabilistic analysis of sequence function using scoring matrices. *Bioinformatics*, 16(3):233–244, 2000.