Efficient Probabilistic Genome Alignment with one Extension

Mélanie Ghaby MELANIE.GHABY@MAIL.MCGILL.CA
Department of Computer Science
McGill University
Montreal, QC H3A 0C8, CA

1. Introduction

The identification of unknown sequences is a fundamental challenge when it comes to bioinformatics. BLAST is an algorithm that attempts to provide an answer to such questions by aligning short query sequences with long deterministic genomes using a heuristic approach. Although using a heuristic can improve the runtime, some steps of the algorithm, such as the extension phase, can still be time-consuming. Furthermore, BLAST is tailored for deterministic sequences, which limits the range of questions that can be answered using this algorithm. For instance, sequence data resulting from a genome assembly can be ambiguous at certain positions. This can be due to polymorphism at some sites. In such a case, the genome can be represented using a probabilistic model by assigning the probability of each nucleotide occurring at each position. A probabilistic genome can also be used to represent inferred ancestral sequences that are not directly obtained but are estimated computationally. Aligning a short query sequence against a long probabilistic sequence is not handled by the current BLAST algorithm, whose goal is to optimally align two sequences, a concept that has to be defined when it comes to aligning a sequence against a longer probabilistic sequence.
This report presents an algorithm that was designed to address the challenge of aligning a short query sequence against a long probabilistic genome. To do so, finding an optimal alignment is equated to finding the region of the probabilistic genome where the queried sequence of nucleotides would be most likely to belong. Additionally, the algorithm attempts to reduce the time spent on extension phases by only performing one single gapped extension using a variant of the Needleman-Wunsch algorithm.

The following sections will delve into the presentation of the algorithm in the Methodology section, the evaluation of its performance based on speed and accuracy in the Results section, and conclude with an interpretation of the results and suggestions for future improvements in the Discussion.

2. Methodology

The algorithm takes as input a short query sequence, a long predicted sequence of nucleotides, which will be referred to as the database for conciseness, as well as a sequence of probabilities corresponding to the probability of each nucleotide of the predicted sequence occurring, which will be referred to as the probability database for conciseness. It outputs the index of the database where the query sequence optimally aligns as well as the optimal alignment itself. To do so, it goes through the following steps: the initial processing of the data, the subsequence identification & the targeted reprocessing of the subsequence, the prediction of a tight range where the query could optimally align, and the final alignment phase.

2.1 Preprocessing of the Data

This step is implemented in the provided Python script `data_preprocessing.py`.

Other than the query, the algorithm takes two inputs: a predicted sequence and a sequence of confidence values between 0 and 1. For each position in the database, the corresponding value p in the probability database refers to the probability that the nucleotide predicted at this position is correct. The probability of this nucleotide being incorrect and therefore being any of the other three possible nucleotides is the same for all three, $(1 - p)/3$. The database and the probability database provided were analyzed to ensure they were formatted properly and were of the same length. This initial analysis also revealed that the lowest probability in the sequence of confidence values is 0.39. Since $0.39 > (1 - 0.39)/3$, this implies that the database exclusively contains the nucleotide that is most likely to be correct at each position.

Additionally, 16.98% of the predicted nucleotides are predicted with 1.0 confidence, and only 22% of the nucleotides have less than 95% probability of being correct. Although these informations do not impact the steps of the algorithm, they will impact the choice of parameters for the algorithm, which will be mentioned in the Discussion.

After the initial analysis, both sequences were converted into arrays in Python to facilitate using them for the next steps. The predicted sequence was processed into a dictionary of words. The keys were all the combinations of nucleotides of a size w found in the database, and their values were lists of the starting indices where these words could be found in the database. This was done without taking into account any associated probability: since the predicted sequence only contains the most likely nucleotide at every position, the dictionary also contained the most likely word at every position. This dictionary was stored in a JSON file.

One dictionary was processed for every word size between six and ten inclusively for later testing. Using six as a lower bound was a decision made in order to limit the number of hits in the later steps that may occur by chance. The decision to use ten as an upper bound is so that it is large enough to reduce hits occurring by chance while still being lower than $w = 11$, a size frequently used in deterministic databases, which may cause to miss hits in case of predicted nucleotides with less confidence.

2.2 Subsequence Identification

This step is implemented in the provided Python script `subarray_identifier.py`.

Given the large size of the database, the algorithm works by progressively refining the range of the predicted sequence that is most likely to correspond to the query sequence. In this second step, it predicts a range that may potentially capture the query sequence and uses it for the next steps.

The database is divided into a number of subarrays that depend on the length of the query:

$$\text{number of subarrays} = \left\lfloor \frac{\text{database length}}{5 \times \text{query length}} \right\rfloor$$

This formula leads to subarrays of size approximately five times the length of the query sequence. This is done as an attempt to prevent a subarray from only containing part of the region in the database corresponding to the query (in which case the region corresponding to the query would be included in two subarrays). However, this does not perfectly control for this edge case, which is better addressed towards the end of this step during the subarray refinement.

Inspired by the FASTA algorithm (Pearson, 1990), the number of hits of word size w between the query and every subarray is counted. A uniqueness reward is applied to prevent high hits occurring due to subarrays simply having a lot of repetitive patterns. To reward unique hits, the number of hits for a given subarray is multiplied by the number of unique hits it contains, thereby rewarding subarrays with a greater variety of common words with the query. In Figure 1, a visualization of hits based on subarray number for some randomly generated query with a random seed of 100 using NumPy revealed a global maximum in the number of hits for one subarray. This subarray may correspond to the query range, as such a range is most likely to contain many hits.

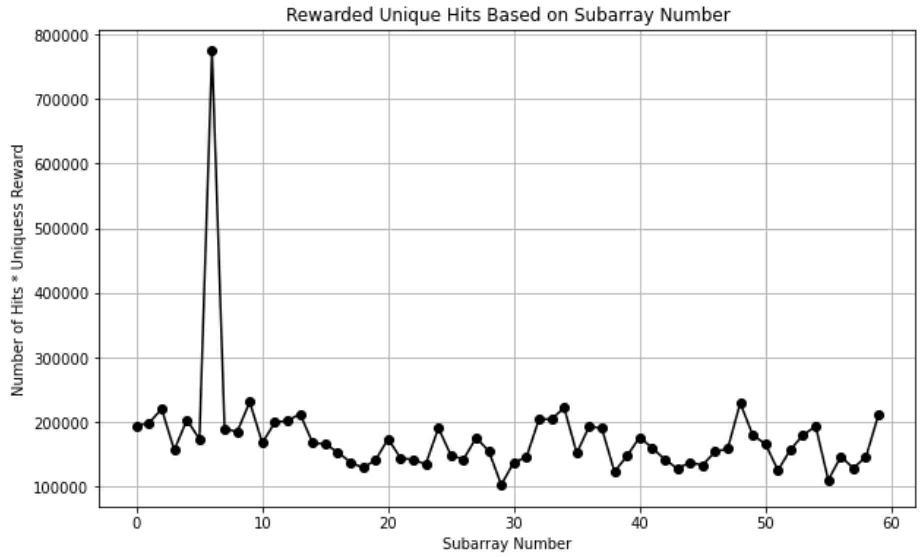

Figure 1: Example of maxima reached in subarray range containing query

In some cases, other subarrays also reflected this phenomenon, but with local maxima that were much lower than the global maximum suspected to be the corresponding query range. After counting the number of hits and applying the uniqueness reward (a product which will then be referred to as rewarded hits), each subarray is defined as a tuple:

$$(\text{database starting index},\ \text{database ending index}\ ,\ \ \text{rewarded hits})$$

The subarray with the maximum number of rewarded hits, the one immediately to its right, and the one immediately to its left (when applicable) are combined into a new tuple with the starting index of the leftmost subarray, the ending index of the rightmost subarray, and a number of rewarded hits corresponding to the sum of their rewarded hits. This step helps control for the possibility of the subarray with the maximum hits not encompassing the entirety of the query sequence, in case the query sequence is split between two subarrays. This new tuple is then further refined to reduce the range encompassed by the subarray, making the next steps less computationally expensive. The range covered by the tuple is divided into subarrays of size (query length) $\times$ 2.5. The subarray with the highest number of rewarded hits with the query, as well as its leftmost and rightmost subarrays (when applicable), are combined into a new subarray of size (query length) $\times$ 7.5. The starting and ending indices of this range in the database, referred to as the *targeted subsequence*, are then returned for the next step.

2.3 Probabilistic Reprocessing of the Targeted Subsequence

This step is implemented in the provided Python script `targeted_reprocessing.py`.

It makes a dictionary for the targeted subsequence while taking into account the confidence values associated with its range. In order to do so, for every word of size w, for every predicted nucleotide with less than 0.95 probability of correctness, the four possible nucleotides are considered, producing $4^{(\#\text{ positions with probability } <0.95)}$ word combinations for every word.

The probability of correctness of every word is defined to be the product of the probability of correctness of every nucleotide, assuming a simplified model of independence between nucleotides. The words with < 0.01 probability of correctness are not added to the dictionary. Therefore, the threshold to produce other word combinations cannot exceed 0.95: if a nucleotide has at least 0.96 probability of being correct, then the other three nucleotides each have approximately 0.01 chance of correctness and would therefore be a part of words that will be later filtered out. The dictionary produced from this step has as keys the words that are produced and not rejected due to a low probability of correctness, and as values their starting indexes in the database and their probability of correctness.

The code that produces this dictionary uses parallel processing with `ThreadPoolExecutor` in order to concurrently produce the combination of many words that will be added to the dictionary. The threads process different sections of the database and its corresponding probabilities sequence since these can happen independently, which reduces the time spent building the dictionary.

2.4 Tight Range Prediction Using Diagonal Dictionary

The implementation is found in the Python script `diagonal_dictionary_recomposer.py`.

The two-hit method aims to reduce the number of extensions performed by only extending if two hits in the same diagonal are within a specific window of distance (Althschul et. al., 1997). Inspired by this, this step operates under the heuristic that a diagonal from which the query is produced is more likely to contain more hits. Therefore, this step returns a range of diagonals that always encompasses the diagonal with the most hits as well as some other diagonals with high hits within that range. To do so, the probabilistic dictionary produced in the previous step is used to compute the hits with the query. A tuple of this form is created for every hit:

$$(\text{database start index}, \text{query start index}, \text{probability of word correctness}).$$

Using this list of tuples, a new dictionary is made. Its keys correspond to diagonals computed by the formula:

$$\text{diagonal} = \text{database start index} - \text{query start index},$$

and its values are tuples of the form

$$(\text{database start index}, \text{query start index}, \text{word}).$$

The fifteen diagonals with the most hits are then further evaluated. The average and the median distance between two consecutive diagonals are computed. The diagonals whose distance to the immediate next diagonal is smaller than the average distance and the median distance are kept and ordered in ascending order. The diagonal with the maximum number of hits is always kept, regardless of its distance to immediate diagonals. This is done based on the heuristic that the range of diagonals with the highest number of hits is most likely to contain the starting index in the database of the most likely alignment between the query and the database.

Finding a range is necessary instead of simply relying on the diagonal with the maximum number of hits: the diagonal with the maximum number of hits may not correspond to the exact starting index in the database of the optimal alignment due to insertions and deletions in the query. Therefore, a range of diagonals is returned instead.

In order to ensure that this range isn't too big due to some diagonals having a high number of hits by chance, they are controlled so that they stay within the average and median distance of one another. Reducing the range is important to reduce the running cost of the next steps. This range of diagonals is returned in the form of a tuple containing the smallest diagonal to the left and the biggest diagonal to the right + the query length. These are meant to represent a range of the database containing an unknown number of gaps where the query would optimally align. This range of prediction is extended by 10 by reducing the starting position by 5 and adding 5 to the ending position to make sure not to miss a diagonal within a few integers of the range.

2.5 Alignment

This step can be found implemented in the provided Python script `alignment.py`.

The query sequence is finally aligned against the range from the database returned in the previous step. In order to account for the fact that this returned range is bigger than the query, the Needleman-Wunsch has been modified to return an optimal alignment of a substring of the selected range from the database with the query sequence. In order to do so, when building the dynamic programming matrix for the Needleman-Wunsch algorithm, the nucleotides of the query sequence represent the rows, and the nucleotides of the previously selected range from the database represent the columns. The first column is initialized to be 0, and the traceback to recover the optimal alignment is made from the maximum value stored in the last row. This modification was suggested to solve a COMP561 Final Examination 2016 available on MyCourses. The algorithm also needs to take into account the probabilistic nature of the predicted sequence. In order to do so, a simplified independent probability model has been assumed, where each nucleotide is independent from the other.

As stated in the introduction, an optimal alignment in a probabilistic context is defined to be the alignment with the highest likelihood of occurring. In the case of an independent probability model, this is equivalent to identifying the region of the probabilistic genome where, when aligned with the query, the probabilities of each nucleotide and gap composing the alignment are maximized, such that their product, which corresponds to the joint probability of the alignment, is also maximized.

In order to maximize the joint probability in the frame of the Needleman-Wunsch algorithm, the following observation has been made: maximizing the product of probabilities is equivalent to maximizing the sum of their logarithms. This is due to the property of logarithms that the logarithm of a product is equal to the sum of the logarithms of its factors. Using this property allows one to take into account the probabilistic nature of the predicted system without applying major modifications to the original Needleman-Wunsch algorithm. It also allows to prevent the numerical instability and loss of information that could happen by multiplying too small probabilities.

When aligning the query sequence against a nucleotide of the database whose associated probability of correctness is p, the following scoring scheme is used:

- When aligning two nucleotides, if the nucleotide in the database is the same as the one in the query, add the log of the probability p of that nucleotide being correct to the current score.

- When aligning two nucleotides, if the two nucleotides differ, add $\log\left(\max\left(0.002, \frac{(1-p)}{3}\right)\right)$ to the score. This is done to prevent computing $\log(0)$ in case $p = 1.0$, as that would cause the score to become $-\infty$. 0.002 is chosen as a penalty, since the highest value smaller than 1.0 in the confidence values sequence is $p = 0.99$, for which case $q = \frac{(1-p)}{3} \approx 0.003$. Therefore, 0.002 is chosen as a value that is smaller than 0.003 to penalize more harshly mismatches with certain nucleotides than with uncertain ones, without having to manage $-\infty$.

- Gaps also have a penalty score of log(0.002) for the same reason cited above.

Using this scoring scheme, the Needleman-Wunsch algorithm remains the same as described in the first paragraph of this subsection. The algorithm finishes with his step and returns the alignment as a string, with gaps represented as '-', and the starting position of this alignment in the database.

3. Results

The predicted sequence as well as its corresponding confidence values sequence were provided by Prof. Mathieu Blanchette.

The evaluation of the performance was done in the provided script `code.py`, which imports and uses the scripts corresponding to every step mentioned in the Methodology. It also imports `query_generator_and_tester.py`, responsible for generating and testing the query.

To evaluate the performance of the algorithm, the NumPy library and its random module were used, as well as the `time` library. The query is generated according to the probabilities of the predicted sequence. The starting index of the query is randomly selected, as well as its length, which ranged from 150 to 2510 nucleotides (excluding indels). Mutations were randomly introduced at a rate of 0.1, while indels were added at a rate of 0.05 randomly through the sequence. A simulation of 1000 iterations was run, each time generating a random query and a random word size between 6 and 10 inclusively. To ensure the reproducibility of results, the random seed value in NumPy was set to 100 for any use of the random function. To simplify visualizations and group the data, the query lengths were binned into subgroups.

Table 1: Number of simulations conducted for different Word Sizes and Query Lengths

Word Size	**250–500**	**500–1000**	**1000–1500**	**1500–2000**	**2000–2510**
6	28	36	45	41	42
7	26	41	38	38	42
8	31	32	40	49	50
9	31	46	49	50	36
10	26	47	55	38	43

Several metrics were tracked and stored in the `Time Matrix` and `Correctness Matrix`, providing insights into the time taken by various steps and their success rates. These metrics were saved in an Excel file at the end of the simulation, referred to as `official_dataset`, and analyzed in the R script `analyse.R`. The files are also provided to ensure transparency. The time was tracked at every step.

Table 2: Processing and Alignment Times

Step	**Median Time (s)**	**Average Time (s)**
Targeted Reprocessing	6.96	33.6
Diagonal Range Finder	0.249	0.296
Alignment	11.4	13.9
Total Time	26.1	47.8

The algorithm had an average runtime of 47.8 seconds across the 1000 simulations, with the subarray identification and targeted reprocessing being the most time-consuming step, averaging 33.6 seconds. The high average runtime, compared to the median of 6.96 seconds, indicates the presence of outliers where the time taken to build the probabilistic dictionary was significantly longer than at least half of the other simulations. The time-consuming nature of building the probabilistic dictionary is expected: with a subarray size 7.5 times the length of the query, the combinatorial complexity of combining words made of nucleotides whose probability of correctness fell below a threshold leads to an exponential increase in runtime as the query length grows, as further demonstrated in the following graphs.

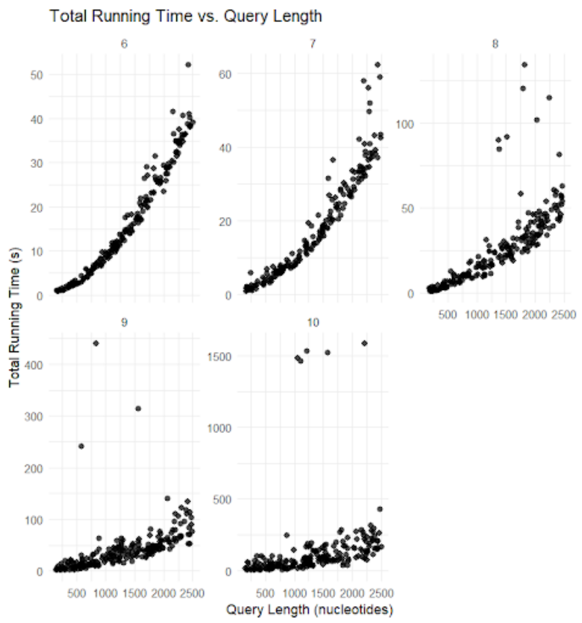

Figure 2: Total Runtime Plots by Word Size

The exponential growth in runtime is visible in Figure 2. There is a sharp increase in running time for longer queries, primarily due to the recombination process triggered when the confidence value of a nucleotide fell below 0.95. This threshold was chosen to ensure accuracy, but it came at a computational cost. However, in this specific sequence, where only 22% of nucleotides had less than 95% probability of correctness, this trade-off was deemed acceptable. This approach may not be suitable for sequences predicted with lower confidence or a different distribution of probabilities. The decision to recombine will be further discussed in the Discussion section. The plots also reveal that both the running time and the variability of the data increased with larger word sizes. This is because larger word sizes tend to have more variability in the number of recombinations they require during the targeted reprocessing.

The two heuristics used were tested for accuracy by verifying after every iteration of the simulation whether the predicted subarray range encompassed the actual query start in the database, and whether the range of predicted diagonals also encompassed the actual query start.

Table 3: Success Ratio of Subarray Identification by Word Size and Query Length

Word Size	**250-500**	**500-1000**	**1000-1500**	**1500-2000**	**2000-2510**
6	0.964	1.000	1.000	1.000	1.000
7	0.962	1.000	1.000	1.000	1.000
8	1.000	1.000	1.000	1.000	0.980
9	1.000	1.000	0.959	0.980	0.972
10	0.962	1.000	0.982	0.974	1.000

As shown in this Table 3, identifying an initial subarray that encompasses the query in the database has a very high success rate, being perfect for over half of the unique word size/query length bin combinations. Accuracy decreases slightly with larger word sizes, which may be explained by the fact that larger word sizes may result in missed hits. This makes it more challenging to identify the subarray with the highest number of hits.

Table 4: Success Ratio of Tight Range Prediction by Word Size and Query Length

Word Size	**250-500**	**500-1000**	**1000-1500**	**1500-2000**	**2000-2510**
6	1.000	1.000	1.000	0.976	0.976
7	1.000	1.000	1.000	0.947	1.000
8	1.000	1.000	1.000	1.000	1.000
9	1.000	1.000	1.000	1.000	0.971
10	1.000	1.000	1.000	0.973	1.000

The tight range identification, based on the diagonals with the most hits, was also perfect for over half of the unique word size/query length bin combinations. However, a few inaccuracies were observed with longer query lengths, which could be explained by the fact that longer queries may be harder to match in the database, as they tend to contain more diagonals with a high number of hits. This makes it more difficult to identify the correct starting index, suggesting the need to adjust this step to take into account query length.

When the range of the alignment was accurately predicted by the previous steps, two metrics were computed on the outputted final alignment to assess the accuracy of the algorithm: the difference between the outputted starting index of the alignment in the database and the true starting index, as well as the Levenshtein distance, which was used to compare the similarity between the aligned query and the intended alignment.

The alignment algorithm was highly accurate in identifying the starting index of the query in the database. There was no particular trend depending on word size or query length. The overall average nucleotide difference was 0.197, and the worst-case scenario involved a three nucleotides difference, which can be explained by insertions or deletions (indels) at the beginning of the query sequence, making it more challenging to accurately determine the start of the alignment.

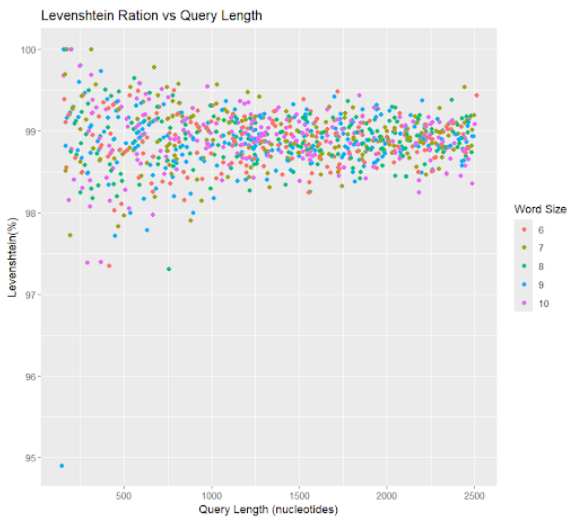

Figure 3: Comparison of Similarity between the Aligned Query and the Intended Alignment

Figure 3 shows that as the query length increases, the Levenshtein distance converges and becomes more stable. Initially, there is some variability in the ratio, with some perfect predictions, but over longer query lengths, the data points become more clustered around 99%, suggesting that the algorithm's performance is high and stable for long queries. The lower scores at smaller query lengths could be due to the fact that a smaller query may align with more regions of the database than a longer one, especially in case of repetitiveness in the database, which may cause alignments that differ from the ones generated for testing. The higher scores could be explained by the fact that a shorter query would require fewer steps to align.

4. Discussion and Future Work

The exponential running time of the algorithm, due to the probabilistic dictionary processing, was computationally expensive. However, it allowed to address predicted nucleotides with unsatisfactory confidences. While triggering a recombination when the confidence fell below 0.95 was acceptable in a sequence where this occurred for only 22% of the nucleotides, it may not scale well when tested on other predicted sequences. Although adjusting the threshold according to the distribution of the confidence values is always an option, the exponential nature of the algorithm's running time needs to be addressed by modifying the algorithm itself. One potential approach involves mapping query lengths to subarrays of predefined sizes for which the targeted processing has already occurred, so that the exponential time required to build the dictionary is no longer part of the running time of the algorithm. Alternatively, implementing wildcards that replace nucleotides with low confidence could prevent the combinatorial nature of recombination.

For future work, it would be interesting to test the algorithm on smaller word sizes, as the runtime seems to increase with larger word sizes, and so does the variability. Smaller word sizes also led to more success in the identification of the initial subarray. Although the precision of the algorithm was acceptable, it can still be improved: the probabilistic model assumed independence between nucleotides, which does not necessarily apply to all regions of a chromosome. Additionally, when choosing diagonals with the most hits to determine the range of alignment, it would have been pertinent to consider the query length, as the results showed that the range encompassed the true start of the query with less accuracy for larger query sizes. The probability of the words occurring for the hits in a given diagonal, as well as their distance, could also be taken into account to reward or penalize certain hits, since a diagonal with high hits but very unlikely words, or hits that are too spread apart, may simply be a product of chance. The alignment algorithm, although extremely accurate, penalized mismatches as severely as it did gaps, which does not accurately reflect the fact that indels are less common than mismatches.

Although there is still work to be done to improve the algorithm, it successfully enabled the alignment of a query with a probabilistic genome, achieving this with a single alignment and a high precision rate. The key takeaway from this experiment is that, in the context of probabilistic genome alignment, the algorithm's parameters should be tailored to the distribution of probabilities and confidence regions. For reproducibility, the tools used for the analysis were Python 3.9 (Spyder) and R Studio version 4.4.1. The scripts and the data produced are available with this submission.

References

[1] Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, **25**(17), 3389–3402. `https://doi.org/10.1093/nar/25.17.3389`

[2] Berger, B., Waterman, M. S., & Yu, Y. W. (June 2021). Levenshtein Distance, Sequence Comparison and Biological Database Search. *IEEE Transactions on Information Theory*, **67**(6), 3287-3294. `https://doi.org/10.1109/TIT.2020.2996543`

[3] Pearson, W. R. (1990). Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, **183**, 63–98. `https://doi.org/10.1016/0076-6879(90)83007-v`