Alignment Algorithms for Probabilistic Sequences

Marina Zhou 260898607, Jessie Xu 260966881, Juliette Xu 260960059

December 15, 2023

Introduction

BLAST (Basic Local Alignment Search Tool) - a tool introduced in 1990 by Altschul et al.[1], that is often used to find alignments between short query sequences and a long genome. This is rather useful in many genome analyses. However, BLAST does not work when the genome (database) sequence is probabilistic. Therefore, an analog of BLAST that works for probabilistic sequences is necessary. In this report, we present a variation (probabilistic adaptation) of BLAST that can align short (non-probabilistic) queries to long probabilistic sequences. A sample of chromosome 22 from the predicted BoreoEutherian ancestor sequence was used to test and evaluate the algorithm.

Our proposed algorithm takes two inputs: a probabilistic database sequence and a query sequence. It aims to output a local alignment with the highest score. To test the accuracy of the algorithm, we picked a random starting position in the probabilistic database. For each test, 10 random short sequences of 100 nucleotides were generated by selecting nucleotides randomly according to the probabilities at each position. We also implemented additional random substitutions, insertions, and deletions of the nucleotides to test the robustness of the method. The accuracy of the algorithm was evaluated by verifying whether the algorithm could accurately output the alignment between the query sequence and the corresponding segment of the database sequence from which the query was derived.

There are three main differences between our proposed solution and the traditional BLAST algorithm. First, we proposed a new matching scheme based on the given probabilistic database. Second, the gap penalty (linear) is defined as the average confidence of the database. Third, during the scan for hits stage of the algorithm, our algorithm does not seek a perfect match. Our proposed algorithm will consider any matches above a certain threshold (e.g., 95% match between the w-mer from the query and the w-mer from the database). As a result, the algorithm takes into account a wider range of matches, thus improving the accuracy with probabilistic inputs. However, this approach will also significantly increase the running time of the algorithm, making the trade-off between efficiency and accuracy an important factor to consider.

Overall, our proposed solution performed very well. When no random substitutions, insertions, and deletions of nucleotides were used to generate the testing query, it can produce a 100% accurate alignment. When 5 random substitutions, insertions, and deletions of nucleotides were used to generate the testing query, the program can find an alignment 30% of the time with an average search time of 495.49 seconds. The reported alignments were at most 2 positions off from where they were generated. We also discovered that, when compared with the traditional BLAST algorithm, the running time of our algorithm will not be negatively affected by large mer size. The algorithm is more efficient when we use a large w-mer.

Methodology

Input and Output

The algorithm takes two inputs: the sequence database file and the probabilistic database database file. The database sequence contains a single string, where each character represents a nucleotide. The probabilistic database contains a series of decimal values separated by a space. These decimal numbers represent the probability of the corresponding nucleotide appearing in its position in the database sequence. For instance, if the database sequence begins with the nucleotide 'A', and the probability of 'A' occurring at this position is 0.7, then the first decimal in the probabilistic database will be noted as 0.7.

The algorithm outputs 1) the optimal local alignment between the query sequence and the database sequence, 2) the score of the optimal alignment, 3) the running time, and 4) the starting index of the local alignment in the database sequence.

Pre-Processing

Convert Input

We began by converting the sequences in the sequence database file into lists. Then, we aggregated the lists into a larger list named L_1. Similarly, we converted the probability sequences from the probabilistic database file into lists and aggregated them into L_2.

Calculate Gap Penalty

We iterate over L_2 to compute the average value of each probabilistic nucleotide. We then used this average (about 0.92) as our gap penalty. Note that in our current implementation, as the database contains only one sequence, we directly do $sum(L1[0])$. However, in cases where the database contains multiple sequences, we should calculate the average probability across all sequences, as described above.

Find Probability of All Nucleotide

The probabilistic database only contains the probabilities of the corresponding nucleotides appearing in their positions in the database sequence. Therefore, we need to calculate the probabilities of the other three nucleotides occurring at the same positions. First, we create a matrix M with the four nucleotides (A, T, G, C) as rows and the sequence index as columns. Then, we iterate through $L2$ to find the probability of each nucleotide and record it in M. Finally, we subtracted this probability from 1 and divided the result by three to determine the probabilities of the remaining three nucleotides. For example, for a nucleotide "A" at index 1 with a probability of 0.7, we will have "T", "G", and "C" at index 1 with a probability of $\frac{1-0.7}{3} = 0.1$ each.

New Scoring Scheme

In this algorithm, we introduced a new scoring scheme based on the given probabilistic database. If two nucleotides match, we plus the probability as the match score of the nucleotide at position i in the database sequence. If the two nucleotides mismatch, we minus the probability of the nucleotide at position i in the database sequence. The alignment score is calculated by summing up the score of each nucleotide. Table 1, Table 2, and Table 3 gave an example of a scoring scheme concerning the nucleotide A and T at the first position in the database.

Nucleotide k at Position 1	**Probability**
A	0.7
T	0.1
G	0.1
C	0.1

Table 1: Probability of nucleotides at position 1.

Nucleotide in Dataset	**Nucleotide in Query**	**Score**
A	A	+0.7
A	T, G, C	-0.7

Table 2: Score of matches and mismatches regarding nucleotide A at position 1.

Nucleotide in Dataset	**Nucleotide in Query**	**Score**
T	T	+0.1
T	A, G, C	-0.1

Table 3: Score of matches and mismatches regarding nucleotide T at position 1.

Probabilistic BLAST

Creating W-mers

This step is the same as the traditional BLAST algorithm. We started with a given mer size w, pre-processed lists L_1 containing sequences from the database, and the input query. We extracted all w-mers and their starting index from the database and the input query. These w-mers were then stored in two separate dictionaries: D_1 for the database and D_2 for the query. In these dictionaries, each w-mer serves as a key, and its value is the list of indices where this w-mer appears in the sequence. For example, a dictionary { "ATG": [1, 10] } indicates that the substring "ATG" appears at both the first and tenth indices of the sequence.

Finding Hit

For finding hits, we used a probabilistic approach instead of simply identifying identical pairs of w-mers in the database sequence. We iterated over each w-mer in D_1 and compared it to every w-mer in the database of D_2. If the pair has a similarity score more or equal to $0.95 \times len(query)$, where $len(query)$ represents the length of the query, we will proceed with ungapped extensions. This similarity threshold is slightly higher than the expected score for the query (the expected score is defined as the average probability of the nucleotide, which is about 0.92). This similarity score is calculated by comparing the alignment score of a w-mer in the query and a w-mer in the database, using the new scoring scheme. Suppose the similarity score is greater than or equal to $0.95 \times len(query)$. In that case, we will take the current w-mer as the key, and a list of sorted tuples, (matched-index, similarity score), as the values to create a new dictionary D_3. The tuples are sorted from highest similarity score to lowest similarity score. Here is an example of D_3 with $w = 4$: {"ATCG": [(5, 4.0), (8, 3.8)]}. It represents that the w-mer "ATGC", has a hit with the database word starting at index 5 with a similarity score of 4.0, and also has a hit with the database word starting at index 8 with a similarity score of 3.8.

Ungapped Extension

We loop through each alignment in D_3. These initial alignments are termed 'seeds'. We first extend the sequence to the right. As we add each nucleotide, we record the current similarity score and position of the newly added nucleotide. We keep expanding the alignment and updating the highest similarity score while recording the index that produces this similarity score. We stop expanding the alignment when the current similarity score of the current alignment is below the highest similarity score by $\delta = 5$. The same procedure is applied to the left side. As figure 1illustrates, we started the extension with a seed that has a similarity score of 2.7. We gradually reached a similarity score of 10.3 at the 5^{th} nucleotide, then dropped to a similarity score of 5.3 at the 11^{th} nucleotide. At this point, we must stop the extension because the new score is below the threshold $\delta = 5$. In the range from no extension to extending 11 nucleotides, we select the highest similarity score of 10.3, therefore extending our sequence by 5 nucleotides.

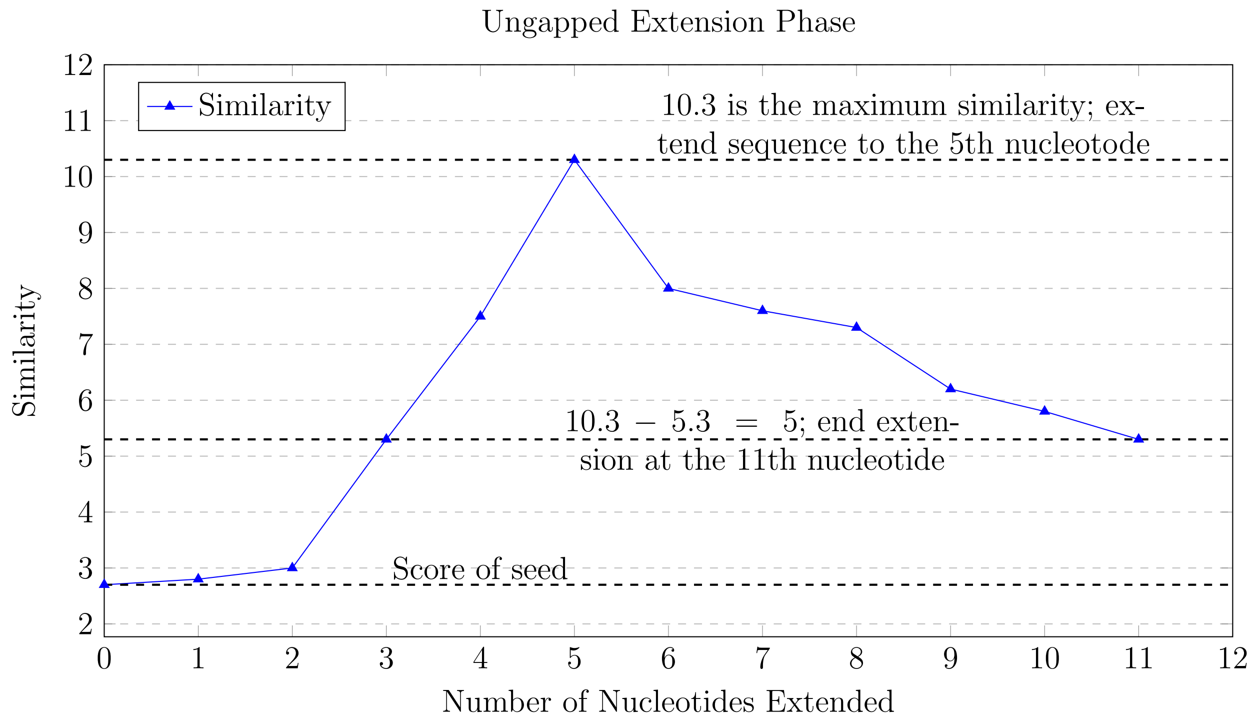

Figure 1: Plot of the right ungapped extension Phase.

Calculating E-value

After we get the ungapped extended alignments, we calculate the E-value of each alignment to increase computational efficiency. We first set a threshold for the E-value and then used a formula to calculate the E-value of the current alignment. If the E-value exceeds this threshold, we discard the alignment. For alignments with small E-values, we proceed with the gapped extension. The formula is $E = Kmne^{-Ls}$, where s is the alignment score, m is the length of the database, and n is the length of the query. L and K depend on the scoring scheme. In our algorithm, we take L and K equal to 1.

Gapped Extension

For the gapped extension stage, we used an alternative Needleman-Wunsch algorithm. This algorithm takes two aligned sequences — one from the query and the other from the database, along with a probabilistic matrix and a gap penalty. The algorithm begins by initializing a score matrix and a backtrack matrix. The scoring matrix is the same as the scoring matrix in the traditional Needleman-Wunsch algorithm using the scoring scheme we defined earlier. The backtrack matrix records the decisions we made (match, mismatch, gap in seq1, or gap in seq2). We start filling our two matrices following the new scoring scheme like what we do in the traditional Needleman-Wunsch algorithm. We will also use the same backtrack procedure to get the optimal alignment.

Results

Overview

To evaluate our algorithm, we used a section of chromosome 22 from the predicted BoreoEutherian ancestor sequence as the probabilistic database. The input sequences we used were generated by selecting an index in the chromosome 22 sequence randomly and then creating a query of n nucleotides. To determine the nucleotide at a given position in the random query, we generated it based on the two given files. As such, the likelihood of a nucleotide being chosen for the query sequence was directly proportional to the probability of that nucleotide being the true nucleotide at that position. Random substitutions, insertions, and deletions were performed on the query sequence at various specified rates.

To test our algorithm, 10 queries were generated with different combinations of nucleotide substitution, insertions, and deletions at varies rates (randomly substitute 5, insert 5 and delete 5 nucleotides). The queries are generated independently and each query is made of 100 nucleotides. The accuracy of the results were evaluated based on the difference between the starting index s of the database query, which the random query was generated based on, and the starting index of the local alignment in the database query returned by the algorithm. In theory, the "optimal" alignment should start at index s since that was used to generate the random sequence used as input. Thus, the difference in the starting index could suggest how "off" (inaccurate) the alignment/match found is to the actual alignment.

Results with No Substitutions, Insertions, and Deletions

The initial experiments we ran the algorithm against was the default setting, where we only used the random sequences we generated, with no additional substitutions, insertions, or deletions of nucleotides. The mer size is 11 and the threshold for E-value is 0.001. The matching threshold used requires there to be at least a 95% match between the w-mer in the database and the w-mer in the query. This requires the match score to be at least $0.95 \times len(query) = 95$. Unless otherwise specified, these conditions were used for all later experiments. For these experiments, we were able to get an alignment for 8 out of the 10 input queries. For the queries that we were able to find an alignment, the accuracy was promising as we were able to align with the exact starting indexs that were used to generate the random queries. The average score for this was 77.7563 and the standard deviation for the scores was 4.1879.

Results with Only 5 Substitutions

Next, we tested our algorithm with different variating combos of substitutions, insertion, and deletion. Initially, we only added 5 random substitutions in our input queries and the algorithm was pretty successful at finding alignments. It was able to find an alignment for 10 out of the 10 input queries used. As for alignment accuracy, similar to the previous experiments, it was also able to find exact matches for all the alignments. The average score was 80.52533 and the standard deviation for the scores was 5.15049.

Results with 5 Substitutions, 5 Insertions, and 5 Deletions

Then, we tried to find alignments with 5 substitutions, 5 insertions, and 5 deletions. For our E-value, we used 0.001 and we had the threshold for word matching in the ungapped phase to be 95%, similar to before. With this, our algorithm struggled to find alignments, as only 1 out of the 10 input queries was able to produce an alignment. The accuracy of the alignment was acceptable, as it was able to find a starting index very close to the actual starting index. The calculated starting index was 77807, compared to the actual starting index which was 77810, which is only 3 nucleotides difference in index.

Results with Relaxed E-values and Matching Threshold

In an attempt to improve the results of the alignment with this combination of substitution, insertions, and deletions, we tried to relax the constraints (i.e. E-value, and matching threshold). For the modifications to the constraints, we increased the E-value from 0.001 to 10 and lowered the matching threshold to 90% instead of the previous 95%. However, the results we got did not present significant improvements, but rather a lower accuracy in comparison. Again, only one input query was able to produce an alignment output. Even though the region was still fairly close to the actual region, the difference in position for the starting index is 8 (actual = 4388, calculated = 4396), which is a lower accuracy.

Results With Only Insertions and Deletions

We focused on either only inserting or deleting nucleotides from the sequence next, without any substitutions and using the same constraints as before (an E-value of 0.001 and a threshold of 95%). The insertion/deletion was performed on 4 sets of testing queries, 2 being the insertion/deletion of 5 nucleotides and 2 being the insertion/deletion of 10 nucleotides. The results for insertion/deletion had a similar pattern based on the number of nucleotides changed. The algorithm was able to find more alignments for when there are only 5 modifications in nucleotides, with gaps in both query and database sequences. The table below shows the average accuracy and the standard deviation of accuracy for the datasets. As shown, the algorithm performed better with a modification of only 5 nucleotides, finding alignments 9 out of the 10 times it was run. In comparison, for insertion/deletion of 10 nucleotides, the algorithm found fewer alignments and performance varied.

Note: the accuracy below refers to how many nucleotide differences there are for the calculated starting index (for example, a database query starting index of 2, and a calculated starting index of 3 will give an accuracy score of 1 (3-2)).

Setting	**# Matches**	**Accuracy Average**	**Accuracy Standard Deviation**
Insert 5	9/10	1.142857	1.355262
Insert 10	7/10	3.7142857	2.5475078
Delete 5	9/10	1.222222	1.396645
Delete 10	5/10	2.8	2.13542

Table 4: Statstics of insertion and deletion

Results with Varying W-mer Sizes

Lastly, we tried various w-mer sizes for our dataset to see if it impacts the accuracy of our algorithm. For these experiments, the constraints were standard like before, with a match threshold of 95% and an E-value of 0.001. The results are shown below in the table. As seen, because we only used a 95% match threshold instead of requiring an exact match, w-mer size does not really affect our accuracy.

Note: the accuracy below is calculated by the same method that was mentioned above for Table 4

w-mer Size	**# Matches**	**Accuracy Average**	**Accuracy Standard Deviation**
11	8/10	0	0
5	10/10	0	0
25	8/10	0.25	0.433013

Table 5: Statistics of varying w-mer sizes

Results with Exact Match During Finding Hits

During the finding hits stage of the algorithm, we used a 95% threshold instead of an exact match. We believe this approach can find more accurate alignments when the input queries have more randomness. When compared to the exact match method, we observed a significant improvement in accuracy. For example, when 10 random nucleotides were deleted from the input query, our program was able to find a match for 5 out of 10 experiments. However, the exact match only found a match once. We also observed similar patterns in other experiments conducted.

Results with Different Penalty Score

Unlike traditional approaches, we introduced a new scoring scheme that is based on the given probabilistic database. The details of it were described in the previous section. If the two nucleotides mismatch, we minus the probability of the nucleotide at position i in the database sequence even if the nucleotide in the query has a higher probability than the nucleotide in the database. For example, as described in Table 6 and Table 7, we would use a score of -0.1 when A has a probability of 0.7 in the database at position 1. Therefore, we also proposed an alternative scoring scheme. This alternative scoring scheme would use a score of -0.7 in this senrio. After conducting a series of experiments, we did not observe significant changes in accuracy.

Nucleotide k at Position 1	**Probability**
A	0.7
T	0.1
G	0.1
C	0.1

Table 6: Probability of nucleotides at position 1.

Nucleotide in Dataset	**Nucleotide in Query**	**Score**
T	A	-0.7
T	A	-0.1

Table 7: Score of matches and mismatches regarding Nucleotide A at position 1. The top row is the new scheme and the bottom row is the old scheme.

Discussion and Future Work

Discussion

The algorithm we implemented was able to successfully perform the alignment of a short (non-probabilistic) query to a long probabilistic sequence. The algorithm can reliably return an accurate local alignment in most scenarios. However, as the randomness (random number of substitutions, insertions, and deletions of nucleotides) increases, the accuracy of the program decreases and can even fail to find an alignment.

One advantage of our implementation is that the mer size does not affect the matching results. This is because, during the finding hits stage of the algorithm, we only require the w-mer in the query to be a 95% match to the w-mer in the dataset instead of an exact match. In comparison to the traditional BLAST algorithm, using a larger w-size will increase the efficiency of our algorithm. This is because using a larger size results in less number of w-mers to compare, reducing the number of seeds to extend in the gapped extension stage. For example, if the database sequence is of length 100 nucleotides, and we use a w-mer size is 99, then there will only be a maximum of 2 hits during the gapped extension stage. Comparing this to a w-mer size of 10, the maximum number of hits is significantly higher. As a result, this will increase the overall running time during the gapped extension phase.

However, this approach also has one significant limitation. Since we only require a match between the w-mer in the query and the w-mer in the dataset to be 95%. Therefore, during the finding hit stage, we have to compare every w-mer in the query and every w-mer in the dataset while the traditional BLAST algorithm can finish the comparison in a linear time. Therefore, making the trade-off between efficiency and accuracy an important factor to consider. In our case, we chose accuracy because we believe it is more important to return a result after expanding more possible matches than to return a failed alignment quickly while only expanding a few options.

Future Work

In our algorithm, we used a basic (brute force) method to produce all the w-mers. The method is such that we use a for loop to iterate through the whole input sequence, indexing the [i: i+w] slice of the input sequence at each iteration. However, there are known optimizations that can be implemented. An example of such is to use the Biopython library which is an open-source Python library specifically designed to enable bioinformatics and computational biology tasks. By using the Seq class in the Biopython library, we can efficiently parse through the input sequence and return all the w-mers found. The implementation of such will be able to further improve our algorithm's efficiency.

Another optimization that we can do for our algorithm, is in the gapped extension phase. Currently, the algorithm uses the traditional Needleman Wunsch (NW) algorithm for the gapped extension phase. However, a modified version of NW could increase the efficiency of the algorithm. The idea of the modification is suggested by Altschul et al.[2] where the alignments produced are confined to a predefined strip of the dynamic programming path graph by heuristic methods. Then the algorithm "considers only alignments that drop in score no more than Xg below the best score yet seen. The algorithm is able thereby to adapt the region of the path graph it explores to the data." (Altschul et al.[2])

References

1. Altschul,S.F., Gish,W., Miller,W., Myers,E.W. and Lipman,D.J. (1990) J. Mol. Biol., 215, 403–410.

2. Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, David J. Lipman, *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acids Research*, *Volume 25*, *Issue 17*, 1 September 1997, Pages 3389–3402, https://doi.org/10.1093/nar/25.17.3389