Modified BLAST for probabilistic sequences

Qifei Zhao
McGill University
Montreal, Canada
qifei.zhao@mail.mcgill.ca

Lingyin Wang
McGill University
Montreal, Canada
lingyin.wang@mail.mcgill.ca

1 INTRODUCTION

The BLAST (Basic Local Alignment Search Tool) is widely used for finding alignments between a short query sequence and a long genome. It is a heuristic approach that search only for local alignments containing high-scoring gapless pairs (HSPs). However, in certain cases, we may not have full knowledge of the genome sequences of a species. Possible situations include when the sequencing data is ambiguous at certain positions (e.g. at polymorphic sites), or when the genome sequence is predicted from ancestral or descendent sequences. The database to which the short query is aligned may not be available as an exact sequence, but as a most probable sequence with certainty about the identity of the nucleotide at each position.

In our work, we explore Two-Hit method to optimize the process of finding HSPs and develop a modified version of BLAST that works to align a short non-probabilistic query sequence against a long probabilistic genome.

2 DATASET

The dataset is a long sequence of a genome of a species. However, there is uncertainty involved in evolution, so the exact nucleotide at each position is not generated with a hundred percent certainty. There is also a sequence of probabilities in the same length of the genome such that each element in this sequence represents the percentage of certainty of observing this nucleotide at this position in the sequence.

2.1 Pre-processing the data

Since both the genome sequence and the probability sequence are given as text files (http://www.cs.mcgill.ca/ blanchem/561/ probabilisticGenome.html), we conduct the following steps to convert them to lists in python:

(1) Read text file into python as two strings, since there is only one line in both text files.
(2) Split two strings by space and convert them into two lists, supposing they are *SEQUENCE* and *SEQ_PROB* respectively.
(3) Create a new list *NUC_PROB* of the same length as *SEQUENCE*. Each element of *NUC_PROB* is a dictionary of the form $\{'A' : 0, 'C' : 0, 'G' : 0, 'T' : 0\}$.
(4) Then for each element of *NUC_PROB*, assign the probability of this position to the corresponding nucleotide type according to *SEQ_PROB*. For other three nucleotide types, assign $(1-p)/3$ to all of them (assuming p is the probability at this position in *SEQ_PROB*).

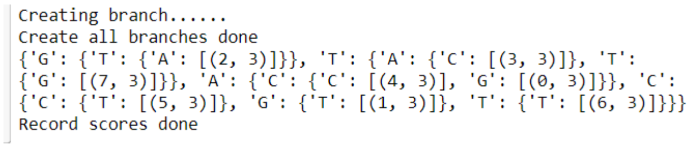
3 METHODOLOGY

In order to identify a range in the data sequence that matches with the query sequence, we use the following steps to first find an initial matching word hit of the query in the sequence and then extend this hit to both ends of the query. The procedure utilizes the same ideas in BLAST with modifications, taking probabilities of nucleotides in each position into consideration.

3.1 Generate a list of initial hits

We begin by finding some initial hits of words that match the query and the sequence in exact orders. A word is defined as a sub-sequence of the genome with length equaling the value indicated by parameter *word_size*. To speedup the entire process, we generate a search trie such that each leaf represents a possible word that has a match in query. This saves plenty of time when searching a hit in database later, since words in the database that does not match any word in query can be skipped directly. We apply the ideas described in a paper of BLAST [2] which is a developed version of the original BLAST paper[1], while we add modifications to include probabilities in calculation. The following steps describe how to find an initial hit in details:

(1) Create a list of words of the query sequence. Starting from the first position of query, create one word from this starting point. Then, slide to the next position and get the word starting from this position, and so on. Suppose this list is *QUERY_WORDS*.
(2) Generate a list of all possible words with the same word size. Suppose it is *DB_WORDS*.
(3) Filter the words in *DB_WORDS*. For each word, if the matching scores between this word and any word in *QUERY_WORDS* are all lower than a threshold, then this word is removed from the list. Comparing two words, add points to matching score if the nucleotides at this position matches and deduct points otherwise.



```
Creating branch......
Create all branches done
{'G': {'T': {'A': [(2, 3)]}}, 'T': {'A': {'C': [(3, 3)]}, 'T':
{'G': [(7, 3)]}}, 'A': {'C': {'C': [(4, 3)], 'G': [(0, 3)]}}, 'C':
{'C': {'T': [(5, 3)]}, 'G': {'T': [(1, 3)]}, 'T': {'T': [(6, 3)]}}}
Record scores done
```

Figure 1: An example search trie in the form of dictionary in python. Built with $word\_size = 3$ and query $ACGTACCTTG$

The remaining list is all the words that match with at least one word in *QUERY_WORDS*.

(4) Build a *search trie* from *DB_WORDS*. Use dictionaries in python. Each node is represented by a key in the dictionary. Each leaf would be a list which contains pairs such that each pair represents a hit. The first element in the pair represents the index of the starting point of this matching word in the query, and the second element represents the score of this match. An example of the trie is displayed. This is built with word size 3 and query $ACGTACCTTG$.

(5) Check word matches from each position p of the database. First, get the word starting from index p. Since uncertainty is combined with each position, we obtain all the possible words starting from p. Then, search in the trie for a match in the query. If there exists a match, create a *Hit* object to store related information. A *Hit* object contains the following fields: *db_start, db_end* representing the start and end positions of the match in the genome sequence; *query_start, query_end* representing the start and end positions of the match in the query sequence; *diagonal* representing the difference in the start position of hit in query sequence and database sequence (used in Two-Hit method later); and finally *prob_score* representing the matching score, which is calculated by the following equation:

$$\sum_{i=0}^{word\_size-1} p_i \times score_{match/mismatch} \tag{1}$$

where p_i represents the probability of observing this nucleotide at this position in database sequence, and *score* is positive if it is a match or negative otherwise.

(6) Return a list of Hit objects.

3.2 Two-Hit Method

After single hits are found, we do not directly start extension for each hit. Instead, we first try to find pairs of hits that are highly probable to belong to the same alignment, and hence would include each other in the extension step. The incentive to do so is that the extension step using dynamic programming is the most computationally costly process in the original BLAST algorithm, so it is desirable to reduce the number of extension trials needed[2]. Since the query is usually much longer than the chosen word size, we can assume the final best alignment of the query to the database sequence contains more than one initial hits.

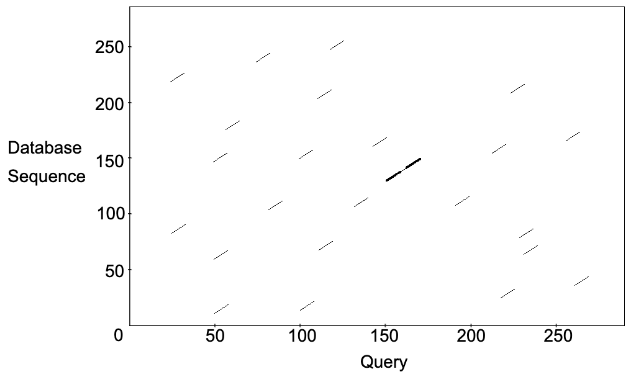


Figure 2: Sample BLAST comparison of a query to a database sequence. The initial hits are represented as thinner lines, which are of same length equivalent to the word size. The Two-hit method is to find pairs of hits that are on the same diagonal and within a certain distance between each other. In this example, one pair of hits are found, and the two hits are indicated by thicker lines, with thinner line joining them representing the part to be aligned one-to-one.

We implement the two-hit method proposed in [2] with modification to fit our algorithm. As demonstrated in Figure 2, we are looking for a pair of non-overlapping hits that lie on the same diagonal. The attribute *diagonal* of the *Hit* object makes this process easier. It is defined as the difference between the *Hit*'s start position on the database sequence and its start position on the query. Since no gap has been considered yet, the start positions provide us with full knowledge of the *Hits*' positions given the word size. For example, given the word size equal to 10, a *hit* aligning position 10 to 19 in the database sequence to position 5 to 14 will have *diagonal* equal to 5.

We then compare the *diagonal* among the initial hits. If two *hit*s have the same diagonal, we can infer that the part between these two short alignments would be aligned without gap. For example, given word size equal to 10, for hits with a certain *diagonal* equal to x, if there is one hit with starting position *a* on the database sequence and *a-x* on the query and another hit with starting position *b* on the database sequence and *b-x* on the query (*a*<*b*), position *a+10* to *b-1* on the database sequence will be of the same length (*b-a-10*) and aligned without gap to position *a-x+10* to *b-x-1* on the query.

Since database sequences are scanned sequentially, this coordinate always increases for successive hits and we only need to look at the hits after the current hit. A window size is applied to restrict the distance between the two hits. Overlapped hits are invalid to be combined and ignored.

We use score to decide whether to further extend the *two-hit*s found. The score of a *Two-Hit* is the sum of the scores of the two initial single hits and the score of the inter-hit region. An equation similar to the one used for initial hits' score calculation (Equation 1) can be used to calculate the score of the inter-hit region:

$$\sum_{i=0}^{length-1} \sum_{n \in \{A,T,C,G\}} p_n \times score_{match/mismatch} \qquad (2)$$

where $length$ is the length of the inter-hit region; p_n represents the probability of observing this nucleotide at this position in database sequence; $score$ depends on whether the nucleotide is in match with the nucleotide at the respective query position or not.

The *Two-Hit*s with scores higher than a threshold given will be outputted as a list *Hit* objects and to be extended in the following steps.

3.3 Ungapped extension

Now extend the *Two-Hit*s gapplessly to generate a *High Scoring Pair (HSP)*. First extend to the right of the second hit. Calculate the matching or mismatching score at each position and keep tracking of the total matching score of this alignment. Stop extending as soon as the total matching score falls lower than the original matching score. Then extend to the left side of the first hit in the same way. Stop if the total matching score began to decrease. Return the list of hits after this ungapped extension and begin the gapped extension.

3.4 Gapped extension

Using dynamic programming (Smith-Waterman algorithm [3]), extend the hit to both the right side of the second hit and the left size of the first hit until the entire query is matched. Since we are only looking for one best alignment, the method only outputs one *Hit* object with the highest score. Given a fixed scoring system, the higher the score is, the lower the probability that the alignment occurs by chance and the more probable that it is the correct alignment. The pseudo-code is shown on the right.

Concerning the running time and space, it is not necessary to do back-tracking in this step. Only the starting and the ending position of the hit in the database would be sufficient to explain the performance of the algorithm.

```
Algorithm 1: Gapped extension on HSP
  Input: A HSP as a Hit object
  Output: A new Hit object with query completely
          aligned
  Data: The $NEUC\_PROB$ list with probability of
        nucleotides at each position, a hit object $hit$ and
        the query sequence.
1 $ext\_hit$ ←new hit object
2 if $Not\_Matched(hit.query\_end)$ then
3     endl ← $length(query) - hit.query\_end$
4     $M\_r$ ← matrix of size $endl \times endl$
5     gp ← gap penalty
6     for i=0 to endl do
7         for j=0 to endl do
8             if i==0 and j==0 then
9                 $M_r[i][j] \leftarrow 0$
10            else if i==0 and j>0 then
11                $M_r[i][j] \leftarrow M_r[i][j-1] - gp$
12            else if j==0 and i>0 then
13                $M_r[i][j] \leftarrow M_r[i-1][j] - gp$
14            else
15                $match\_s \leftarrow \sum_{i=0}^{word\_size-1} p_i score_{match/mis}$
16                $query\_gap\_s \leftarrow M\_r[i-1][j] - gp$
17                $db\_gap\_score \leftarrow M\_r[i][j-1] - gp$
18                $M\_r[i][j] \leftarrow max\{match\_s, query\_gap\_s, db\_gap\_score\}$
19            $best\_s \leftarrow max\{M\_r[endl-1]\}$
20            update $ext\_hit.db\_end$
21            update $ext\_hit.query\_end$
22    Do the same for the left side of first hit
23    update $ext\_hit.db\_start$
24    update $ext\_hit.query\_start$
25    return $ext\_hit$
```

4 EXPERIMENT AND RESULT

We generate query of different lengths from the data set for the experiments. Each query with a given length is generated by first choosing a random start position on the database sequence. The query is built by filling each position by randomly, according to the probabilities at corresponding position on the database, choosing a nucleotide. The query then

Step Name	Time used (sec)
Building Trie	30.95
Finding one hit	723.89
Two hit	12.54
Finding HSP	0.71
Gapped extension	355.05
Find best hit	0.01

Table 1: Time used by each step, tested using the first 2500 nucleotide of the dataset and a 100 nucleotide query generated from that.

goes through possible mutation with probability of substitution equal to 0.05, probabilities of deletion and probability of insertion both equal to 0.01.

We use a simple scoring system that consists of a "reward" for a match and a "penalty" for a mismatch. According to [4], the (absolute) reward/penalty ratio should be increased as one looks at more divergent sequences. We choose 1/-2 because our sequences are approximately 95% conserved[4]. We use a linear gap penalty of -5.

We first test the time used by each step and check which step is the most time consuming. Next, both accuracy and running time are compared under different parameter settings, including different *word_size, window_size or query_length*. We are using Google Colab to test these cases. The laptop is plugged in while testing to ensure the best performance.

4.1 Time used by each step

Due to the large length of the genome database sequence, we test on the first 2500 neucleotides. Since the respective portion of the time used on each step would remain the same under the same parameter setting, using a smaller dataset would give similar result as using the original dataset while reducing the total running time. The result is as shown in Table 1.

From this table, it is clear that finding the initial matching hit and doing gapped extension cost considerable time. It is probably because finding one hit requires scanning through the entire database, one position after another. As for the gapped extension, since we choose $word\_size = 9$, $window\_size = 30$ and $query\_length = 100$ in this case, the two matrices created for dynamic programming would consume both huge memory and plenty of time. In minimum, a matrix would have size $[(100 - 9)/2] \times [(100 - 9)/2]$, which is large to manipulate. This could be improved by choosing a direction of path during dynamic programming and only expanding part of the matrix instead of the whole.

Word size	Accuracy (%)	Time used avg. (sec)
9	98.5	448.8
8	99.3	153.4
7	99.1	58.5

Table 2: Time and accuracy of prediction on the position of hit, using different word size, given $query\_length = 50$ and $window\_size = 30$.

4.2 Different $word\_size$ settings

Test the algorithm with word sizes 6, 7, 8 and 9. Although the default word size for DNA sequence alignment in BLAST is 11, it turns out that both building the search trie (128.61s with word size 10, compared to 30.95s with word size 9) and searching for a word inside the trie take a great amount of time. Thus, we only test using word size less than 10. We test on 50 randomly generated query sequence from the database sequence and record both the accuracy of predicting the position of match and the running time.

We define the accuracy as the rate of total length of the aligned part of query over the original length of the query. The aligned part of the query is the part that matches the position in the database sequence where the query is generated. It is reasonable that a hit position is regarded as matched if the predicted ending (or starting) point of hit is within 1 or 2 positions of the original position. Because the ending (or starting) position, assuming index *j*, may be a mismatch, while the positions before the ending position (index *j-1*) are matches. In this case, the dynamic programming algorithm would return *j-1* instead of *j* since the maximum score occurs at position *j-1*. As the word size decreases, the average time used for calculating hits on a query decreases exponentially. As the word size decreases, the number of words decreases and the time needed to search through the words decreases. Although the number of initial hits tend to increase, the number of HSPs to be extend by the modified S-W algorithm may not increase as well, because of the two-hit method filtering the hits. Hence the total running time decreases.

4.3 Different $window\_size$ or $query\_length$ settings

Same experiments are performed on different $window\_size$ from 20 to 50 at intervals of 10 and test $query\_length$ from 40 to 70 at intervals of 10. Table 3 and Table 4 displays the time used in average and the accuracy of predicting hit positions, respectively.

The average time used grows slowly as the $window\_size$

Window size	Accuracy (%)	Time used avg. (sec)
20	98.1	239.3
30	99.0	244.8
40	98.3	260.4
50	98.6	282.7

Table 3: Time and accuracy of prediction on the position of hit, using different window size, given $word\_size = 8$ **and** $query\_length = 70$

Query length	Accuracy (%)	Time used avg. (sec)
40	94.5	123.6
50	97.0	170.7
60	97.8	198.3
70	99.0	244.8

Table 4: Time and accuracy of prediction on the position of hit, when tested on queries with different lengths, given $word\_size = 8$ **and** $window\_size = 30$**.**

increases, rising only around 10 seconds when increasing the $window\_size$ by 10. So $window\_size$ does not effect the running time significantly.

In general, the average running time increases as the query length increases. This is because when keeping the $window\_size$ unchanged, shortening the query saves time when doing gapped extension because there are fewer positions left in query to be aligned. The accuracy is high for every query length and slightly improves as the query length increases.

5 DISCUSSION AND FUTURE WORK

In conclusion, we did generate a match of query in database sequence with satisfying accuracy. Among all the steps in the algorithm, finding an initial hit in the database and gapped extension on HSP are the two steps that use the most of the time. By comparing the performance under different parameter values, increasing any of $query\_length$, $word\_length$ or $window\_size$ would cause a considerable rise in total running time. However, the accuracy of predicting positions of match is not effected while changing parameters.

One defect of this algorithm is long running time. Although part of the reason is the length of the database sequence and the query sequence, some steps in the algorithm consist of too many calculations which slow down the entire process. A solution to this is to do these calculations over distributed system. This is applicable since the calculation of different hits do not interfere with each other. By running several computations at the same time, the total time could be reduced.

After discussion, there are two reasons that the accuracy is not high enough when $query\_length$ is large. The sequence we generated may have more than one hit in the database. Sometimes, the hit that are not at the original positions would achieve a higher matching score. This may be an essential reason of our query sequence not matching to the place where it is generated. Another reason is the thresholds that we use in calculation. Suppose there are actually more than one hit in the database sequence. Then if the threshold is not large enough, the hit with correct positions may be eliminated by the threshold. This is confirmed by listing all the hit objects after each step (but we can only test with a smaller sub-sequence of database, in case that the IO output crashes). Besides, if the threshold is large, then the number of candidates of hits are relatively small. So, the time used by gapped extension also decreases, which would in turn result in huge reduce in total time since gapped extension is the portion that consumes the most of the time.

In this situation, we choose the probability of three less probable nucleotides to be $(1-p)/3$. However in reality, the probability is not equally distributed among the three less probable nucleotides. More information of the genome sequence would be needed to build a complete matrix of probabilities in each position.

One major modification in our work to the BLAST is how we calculate the alignment scores. We adopt the match/mismatch score matrix and use the sum of the scores multiplied by the probabilities for each position. It is shown to be a valid scoring system as the accuracy of our algorithm is generally high.

While affine gap penalty is more used in real alignment problems, we choose linear gap penalty because deletion or insertion is generated independently from neighbor positions when we build our test queries. Another reason we avoid affine gap penalty is that it would be difficult to define scores for the alignments if we do not calculate score for each position independently. We would like to explore how to generate deletion/insertion with regard to neighbor positions and to design respective modified affine gap penalty in the future.

As we explore the two-hit method used to reduce the times the modified SW algorithm is called, it might be interesting also to try *multiple-hit* method instead of only combining two hits before expanding to HSPs.

The running time of the algorithm is largely increased compared to the original BLAST, since the probabilistic sequence is processed as multiple possible sequences with different probabilities in our algorithm. We find it hard to reduce the running time under the frame of BLAST. However, when we were reading related works, we noticed that pair HMMs can be also used to solve alignment, and that it might be more efficient when applying to probabilistic sequence. We would like to look into it in more depth in future works.

REFERENCES

[1] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of molecular biology* 215, 3 (1990), 403–410.

[2] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research* 25, 17 (09 1997), 3389–3402. https://doi.org/10.1093/nar/25.17.3389 arXiv:http://oup.prod.sis.lan/nar/article-pdf/25/17/3389/3639509/25-17-3389.pdf

[3] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[4] David J States, Warren Gish, and Stephen F Altschul. 1991. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods* 3, 1 (1991), 66–70.