P-BLASTn: local alignment tool for probabilistic DNA databases

Jonathan Pearce, 260672004

December 11, 2017

Introduction

One of the most well known and highly cited tools in Computational Biology and Bioinformatics is BLAST, introduced in 1990 by Altschul et al.[1] Blast is a local sequence alignment tool that can used to find near optimal alignments of small query sequences in large databases of genomes. Since its original publication there have been plenty of improvements and adaptations of BLAST used for specific scenarios of sequence alignment, however not all scenarios that call for a BLAST derived tool have been addressed. BLAST and its variations are designed to work with databases of deterministic sequences, but there are some biological applications where it is more advantageous to utilize a probabilistic sequence such as computationally inferred ancestral sequences. Therefore a variation of BLAST that works with probabilistic genomes could prove to be very helpful. We present a probabilistic adaptation of BLAST for nucleotide local alignment, P-BLASTn. This algorithm combines greedy and probabilistic selection for local alignments in order to reduce the scoring and ranking complexity of alignments and to make results more comparable to those found in deterministic versions of BLAST. A sample of chromosome 22 from the predicted BoreoEutherian ancestor sequence was used to test and evaluate the algorithm. Correct local alignment position in the probabilistic genome was achieved over 90% of the time with sampled query sequences of up to 20% unexpected noise from substitutions, insertions and deletions.

Methodology

Algorithm Overview

The popularity of BLAST makes it evident that its general framework is very effective, therefore the principle behind each stage of P-BLASTn mirrors that of BLAST. However, P-BLASTn is a probabilistic algorithm that enables it to leverage the probabilities in the database sequence to form more accurate results. Beyond the accuracy of P-BLASTn, we wanted the algorithm to be efficient (1 second per query) and to work for any probabilistic sequence, which meant we could not make any assumptions about the underlying distribution of probabilities. This leaves the door open for others to adapt and optimize P-BLASTn in situations where they know more about the probabilistic sequences.

P-BLASTn ranks potential local alignments using their respective alignment score. This means that for alignment evaluation P-BLASTn does not consider the probability of this alignment occurring in the genome sequence, in fact utilizing the probabilities from the genome to help score the alignment becomes problematic. If you simply calculate the raw probability of a sequence being found in the genome and use this for scoring purposes your penalizing longer sequences since any given probability $<= 1.0$ then making any sequence longer can only lower the probability or keep it the same. Another thought is to take the average probability per nucleotide, this prevents longer sequences from being unfairly penalized. The issue with this solution is that it penalizes regions of the probabilistic genome that have more uncertainty about which nucleotide is most likely to be the true nucleotide at that position. These issues with probability guided the development and concept of P-BLASTn. The greedy nature of the algorithm, specifically stages 2 and 3, ensure that all sequences have a high probability of actually occurring in the genome relative to any other sequence found in that area of the genome.

Input

To begin the probabilistic genome is assembled into a $5*L$ size array (database), where L is the length of the genome. Row 2,3,4 and 5 refer to the probability of the nucleotide at any give position in the genome being an 'A','C','G' or 'T' respectively. Row 1 serves as a pointer to the the nucleotide that has p_i^{max}, which is the highest probability of a particular nucleotide being the correct nucleotide at position i (see appendix 1 for visual description). P-BLASTn commonly needs to find p_i^{max} and its associated nucleotide, Row 1 simply streamlines this process.

Set-up

A key assumption of the P-BLASTn algorithm is to assume a good local alignment will contain w consecutive nucleotides that match w consecutive high probability nucleotides in the genome database. Specifically we assume the w nucleotides in the genome database each have the highest probability at their respective indexes. Therefore we can pre-process the probabilistic genome and store every seed of length w that satisfies this assumption. P-BLASTn finds every seed by moving a window of length w along row 1 of the probabilistic genome one spot at a time. After shifting to index i in the genome, the nucleotide that was taken at index $i-w$ is dropped and the nucleotide at position i that is associated with p_i^{max} is added (e.g. if the probabilities at position i are as follows: {'A': 0.1,'C': 0.1,'G': 0.7,'T': 0.1} then we say G is the nucleotide associated with p_i^{max} and $p_i^{max} = 0.7$) and this new 'seed' of length w is stored with the associated genome index.

In our implementation we utilized a python dictionary to efficiently store the seeds, where every key was a distinct seed that had been found in the genome and the key was a list of indexes where that seed could be found. There are most likely more efficient ways to process and retrieve the seeds but they would most likely be more complicated in implementation.

Stage 1

The first stage of P-BLASTn moves through the query sequence with a window of length w. For every substring of the query sequence of length w we look through the dictionary of seeds to see whether that exact string of length w was found in the set-up procedure. Assuming we find a matching seed, the list of indexes of this seed are retrieved. For each of these indexes we proceed with Stage 2 of the algorithm. After Stage 1 every alignment is maximized in terms of probability of alignment, score of alignment and has length w.

Stage 2

From the left and right edge of the current alignment we do a gap-less linear expansion in each direction. The alignment continues until the proportion of matches in the alignment drops below a threshold, given by equation 1 below.

$$\text{length of alignment} * threshold_2 < \text{number of total matches in alignment} \tag{1}$$

Where $threshold_2$ is the proportion of matches you want in your gap-less alignment. In implementation we added a second condition to equation 1, where we forced Stage 2 to extend at least 5 nucleotides before terminating, this was designed to avoid having one mismatch terminate the linear expansion before we could check its potential. While the inequality given by equation 1 is satisfied, each iteration of Stage 2 works as follows. Suppose we are at index i in the probabilistic genome and index j in the query sequence. Let Q_j be the nucleotide in the query sequence at position j, and let G_i^{max} be the nucleotide at position i of the genome that is associated with p_i^{max}. First we check if $Q_j == G_i^{max}$, if yes, then we add that nucleotide to the alignment, we call this case an efficient match (appendix 2 for description) since this maximizes our alignment score and our alignment probability at the same time. Otherwise we continue, the next step is to probabilistically determine whether to take Q_j or G_i^{max}. we generate r which is a uniform random number between 0 and 1 and if $r < p_i^{max}$, we select G_i^{max} for our alignment (efficient mismatch). If $r >= p_i^{max}$ we select Q_j for our alignment (inefficient match). When equation 1 is no longer satisfied Stage 2 traces back to the previous local maximum (with respect to alignment score), this prevents any rapid decrease in match rate to negatively effect an alignment. Stage 2 is considered greedy because we never consider the two nucleotides at position i that would give us inefficient mismatches. The general principle behind Stage 2 is to maximize the probability of the alignment sequence and alignment score, with the probability given a small priority over score. The pseudocode of Stage 2 is below.

Algorithm 1 Gapless Linear Expansion

```
1: $iterations \leftarrow 0$
2: while $iterations <= 5$ or $l_{alignment} * threshold_2 < n_{matches}$ do
3:     if $Q_j == G_i^{max}$ then //efficient match
4:         $seq$ +=$G_i^{max}$
5:         $n_{matches}$ +=1
6:     else
7:         $r \leftarrow U[0, 1]$
8:         if $r < p_i^{max}$ then
9:             $seq$ +=$G_i^{max}$ //efficient mismatch
10:        else
11:            $seq$ +=$Q_j$ //inefficient match
12:            $n_{matches}$ +=1
13:    $l_{alignment}$ +=1
14:    $iterations$ +=1
15: traceback to previous local maximum score
```

Stage 3

Following the gapless linear expansion of Stage 2, P-BLASTn proceeds to once again expand from the left and the right side of the current alignment using a modified and simplified Needleman-Wunsch algorithm. However, unlike Stage 2, the cost of this expansion is not linear with respect to the length of the expansion, like the Needleman-Wunsch algorithm its cost is O(n^2), if n were the length of the expansion. Due to this increased cost per expansion P-BLASTn prunes the set of alignments found from Stage 1 & 2. P-BLASTn ranks the alignments by alignment score and then keeps at most the top 100 highest scoring alignments. Every remaining alignment, goes through Stage 3. The modification from the original Needleman-Wunsch algorithm is that efficient matches are treated as matches and the other 3 types of nucleotide pairings (appendix 2) are all considered mismatches, thus this procedure finds the alignment of the respective query and genome sections that maximizes the number of efficient matches in a given distance. Similarly to Stage 2, we utilize a threshold ($threshold_3$) to ensure the alignment is being relatively successful. In this case $threshold_3$ represents the ratio at which we want the best alignment to find efficient matches. Therefore after computing each new row in the dynamic programming table we save the maximum value of the row and use that in our threshold calculation. The described inequality is below.

$$\text{row index} * threshold_3 < \text{maximum value in previous row} \tag{2}$$

In our implementation we added a buffer (similar to Stage 2) to ensure we did not terminate this extension too quickly. For clarity the the recursive definition of the Stage 3 Needleman-Wunsch algorithm is below.

$$match(i, j) = \begin{cases} 1, & \text{if } Q_j == G_i^{max} \text{ // efficient match} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$stage3(i,j) = \begin{cases} 0, \text{ if } i{==}0 \text{ or } j{==}0 \\ max(stage3(i-1,j), stage3(i,j-1), stage3(i-1,j-1) + match(i,j)) \end{cases} \tag{4}$$

Evaluation

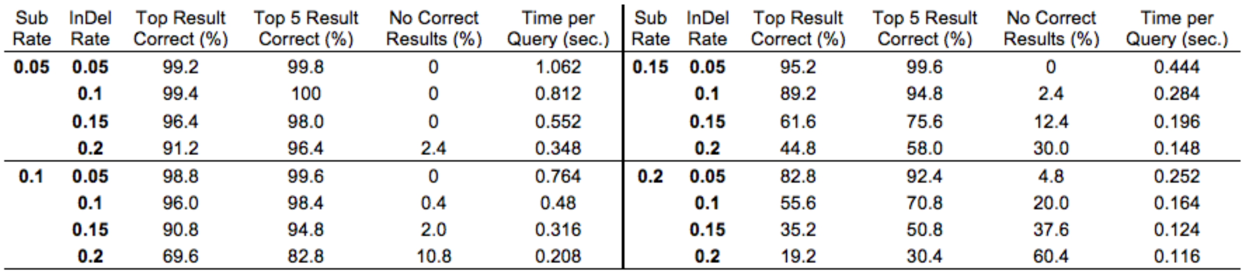
To evaluate P-BLASTn we used a section of chromosome 22 from the predicted BoreoEutherian ancestor sequence as the probabilistic database, this section was roughly 600,000 nucleotides long. We also generated our queries from the same sequence. To generate our queries we choose an index in the chromosome 22 sequence randomly and then created a query of n nucleotides. To determine the nucleotide at a given position in the query we sampled a random number between 0 and 1 and used the probability of each nucleotide at the position in the database to determine which we should use. Such that 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 were injected into the query sequence at a desired rate, similarly insertions and deletions were accounted for in the query at a specified rate as well. The deletion procedure was designed to create deletions of different lengths to more accurately represent real biological sequences. Due to the randomness of the insertions and deletions, as well as the different lengths of deletions the query sequences would vary in length around the desired query length n.

To test P-BLASTn, 16 query data sets were generated with different nucleotide substitution rates (5%,10%,15%,20%) and different insertion/deletion rates (5%,10%,15%,20%). The desired length of the query sequences used was 1000 nucleotides and each dataset contained 250 independently sampled queries. Scoring procedure for alignments was as follows, any match scored +1, any mismatch scored -1 and a linear gap penalty of -1 was used in Stage 3. In order to check whether a query had computed a correct result, we checked whether the seed index (i.e the index in the database where the seed came from) was within the start and end index of the query sequence. Because of the insertion and deletions causing the query lengths to fluctuate around the desired length n it was not possible to confirm that the seed came from the exact spot it was supposed to, only checking within the window of length n would work. It is unlikely that one query sequence would contain the same series of w nucleotides twice. Assuming that each nucleotide in the database is independent of any other nucleotide we can roughly approximate the number of false positives I would expect to find while testing my 16 data sets. Equations 5 and 6 express this approximation and we find that the expected number of false positives will be less than 1, given that the seed length will be 13 for my trials. Therefore this method of evaluation is considered perfectly accurate.

$$\text{false positives} = p(\text{duplicate seed}) * l_{\text{query}} * n_{\text{queries per dataset}} * n_{\text{dataset}} \tag{5}$$

$$\text{false positives} = (\frac{1}{4})^{13} * 1000 * 250 * 16 = 0.0596 \tag{6}$$

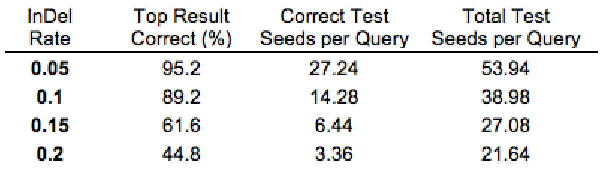
We expected the dataset with substitution rate = 0.05 and indel rate = 0.05 to garner the best results as the rate of substitution and insertions/deletions was lowest here. We tuned the parameters $w, threshold_2, threshold_3$ for this dataset until it was producing sufficient results and queries were being answered at around 1 second each. For the other 15 datasets, $w, threshold_2, threshold_3$ were kept the same and the performance of P-BLASTn with each dataset was recorded. Figure 1 presents the results for the 16 query datasets. There are four measures used to evaluate each dataset. 'Top Result Correct' is the percentage of the queries in that dataset that found it's number one scoring result to be correct (i.e the seed was from the correct portion of the database). Similarly 'Top 5 Result Correct', is the percentage of the queries in that dataset that found at least one of its top 5 scoring results to be correct, this included the top scoring result, therefore 'Top 5 Result Correct' >= 'Top Result Correct'. 'No Correct Results' is the percentage of the queries in the dataset that failed to find a result within the correct portion of the database. Finally, 'Time per Query' is the average amount of time it took to answer a single query in the dataset.



Sub Rate	InDel Rate	Top Result Correct (%)	Top 5 Result Correct (%)	No Correct Results (%)	Time per Query (sec.)	Sub Rate	InDel Rate	Top Result Correct (%)	Top 5 Result Correct (%)	No Correct Results (%)	Time per Query (sec.)
0.05	**0.05**	99.2	99.8	0	1.062	**0.15**	**0.05**	95.2	99.6	0	0.444
	0.1	99.4	100	0	0.812		**0.1**	89.2	94.8	2.4	0.284
	0.15	96.4	98.0	0	0.552		**0.15**	61.6	75.6	12.4	0.196
	0.2	91.2	96.4	2.4	0.348		**0.2**	44.8	58.0	30.0	0.148
0.1	**0.05**	98.8	99.6	0	0.764	**0.2**	**0.05**	82.8	92.4	4.8	0.252
	0.1	96.0	98.4	0.4	0.48		**0.1**	55.6	70.8	20.0	0.164
	0.15	90.8	94.8	2.0	0.316		**0.15**	35.2	50.8	37.6	0.124
	0.2	69.6	82.8	10.8	0.208		**0.2**	19.2	30.4	60.4	0.116

Figure 1: $w = 13$, $threshold_2 = 0.9$, $threshold_3 = 0.8$

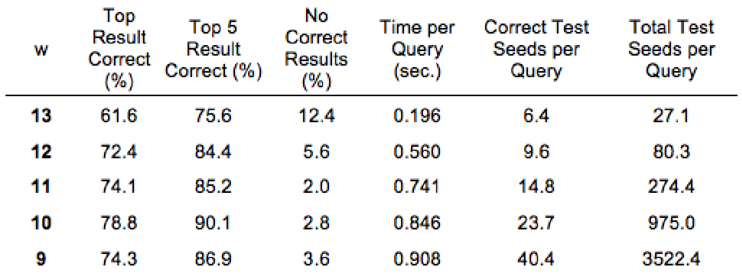
The results from these tests are quite promising. For the three data sets where substitution rate + insertion/deletion Rate <= 0.15, P-BLASTn performed nearly perfectly and for 9/10 data sets where substitution rate + insertion/deletion rate <= 0.25, P-BLASTn's top result was correct over 89% of the time. All queries were answered in just over 1 second, and it was clear that the average time required to answer a query was inversely proportional to how much the query was effected by substitutions and insertions/deletions. We were curious what was causing the aggressive drop in performance for the data sets with high substitution and insertion/deletion rates. We choose to do a follow up and examine the 4 test sets where substitution rate = 0.15 because they had a nice spread between good results and poor results. Our follow up was concerned with the number of seeds from within the target window that were actually being retrieved in Stage 1 and being run through the algorithm. In figure 2 the total number of seeds retrieved in Stage 1 is compared to the number of these seeds that came from the correct portion of the genome (i.e the section that would trigger a correct answer).



InDel Rate	Top Result Correct (%)	Correct Test Seeds per Query	Total Test Seeds per Query
0.05	95.2	27.24	53.94
0.1	89.2	14.28	38.98
0.15	61.6	6.44	27.08
0.2	44.8	3.36	21.64

Figure 2: sub rate = 0.15, $w = 13$, $threshold_2 = 0.9$, $threshold_3 = 0.8$

From figure 2 it is a little more clear why the data sets with insertion/deletion rate = 0.15 and 0.2 and substitution rate = 0.15 perform rather poorly. It appears that the query has become so distorted from the original sampling of the genome that very few seeds from the correct area in the genome are found and expanded into potential alignments. P-BLASTn has a minor fix for this situation, if w is lowered (shorten the seed length) Stage 1 should retrieve more total seeds for expansion and possibly more from the target area in the genome. We investigated this strategy for tuning PBLAST-n with the data set with substitution rate = 0.15 and insertion/deletion rate = 0.15. The results from this test are below in Figure 3.



w	Top Result Correct (%)	Top 5 Result Correct (%)	No Correct Results (%)	Time per Query (sec.)	Correct Test Seeds per Query	Total Test Seeds per Query
13	61.6	75.6	12.4	0.196	6.4	27.1
12	72.4	84.4	5.6	0.560	9.6	80.3
11	74.1	85.2	2.0	0.741	14.8	274.4
10	78.8	90.1	2.8	0.846	23.7	975.0
9	74.3	86.9	3.6	0.908	40.4	3522.4

Figure 3: sub rate = 0.15 inDel rate = 0.15, $threshold_2 = 0.9$, $threshold_3 = 0.8$

Changing the seed length w proved to generate more seeds in the correct area which in turn led to better performance of P-BLASTn in general. There is a performance gain in all metrics from $w = 13$ to $w = 10$ especially in 'No Correct Results', however there is a small performance drop from $w = 10$ to $w = 9$, which shows that w became so short and found so many total seeds in the entire genome that P-BLASTn struggled to filter through all of them and some of the correct seeds were unable to distinguish themselves as better options which explains the small performance drop. It's possible that tuning other parameters such as $threshold_2, threshold_3$ could have brought the performance back up. This shows that the parameters in P-BLASTn must be selected carefully because it is very possible to make the threshold's in any of the 3 stages too low which allows low quality sequences to proceed through the algorithm and diminish final results.

Conclusion

As a first generation local alignment tool for probabilistic DNA databases P-BLASTn has proven to be very effective. recording over 95% accuracy in the metric 'Top Result Correct' for 6/16 datasets is very impressive and demonstrates the ability of P-BLASTn in its current state. Keeping the metric 'No Correct Results' below 5% in 10/16 datasets shows that the general framework of the P-BLASTn algorithm is reliable and further work on developing the different stages could have a significant impact. The few datasets that P-BLASTn struggled with were not particularly surprising since the distortion of the sample queries was so great and since we had tuned the parameters for a much easier data set. We did a sufficient job proving that the algorithm can be tuned and improved quickly if provided very messy query sequences. Looking ahead, P-BLASTn was designed to be used for any kind of probabilistic sequence regardless of the distribution of probabilities, it would be great to have an opportunity to work with a few other probabilistic genome segments to tune the algorithm to work more generally and to possibly experiment with automating parameter selection. Manual parameter selection is one of major downsides to P-BLASTn especially since there are 3 critical parameters ($w, threshold_2, threshold_3$) it takes a little time to find good values for all 3 parameters. Of the 3 stages of the algorithm I feel Stage 2 is the most effective and reliable, there is room for improvement in stage 1 and 3. Stage 1 could be made to be slightly more flexible, perhaps we should select any seed of length w that has probability greater than p. In this case processing the genome would take longer and there would be more alignments going into Stage 2, but if $threshold_2$ remains high then stage 2 should be able to quickly toss out low quality alignments. Further it would be interesting to see the effect of adding inefficient matches and efficient mismatches into the Needleman-Wunsch adaptation in Stage 3, its possible that only looking for efficient matches is too strict. Similarly to what has been done to BLAST in the past 27 years, I see plenty of opportunities to optimize and improve P-BLASTn.

References

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

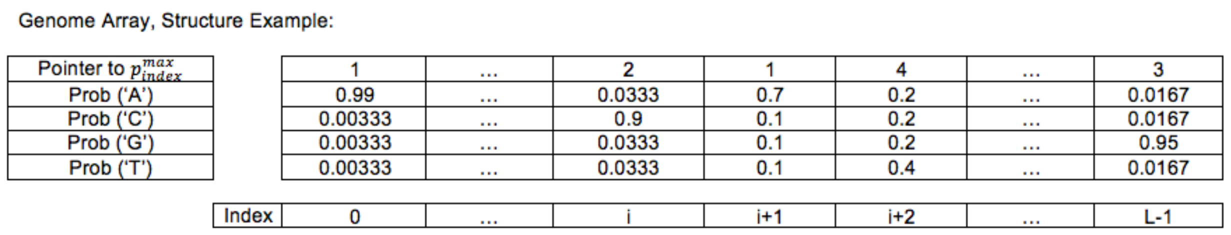
2. Jones, Pevzner ed. *An Introduction to Bioinformatics Algorithms*, MIT Press 2004

3. John, Benos ed. *Sequence Alignment Algorithms*
https://www.cs.cmu.edu/ 02710/Lectures/SeqAlign2015.pdf

4. Thomas, ed. *Sequence comparison: Significance of similarity scores*
http://elbo.gs.washington.edu/courses/GS˙559˙11˙wi/slides/
5A-Sequence˙comparison-Significance˙of˙similarity˙scores.pdf

5. Schatz ed. *Sequence Alignment*
http://schatzlab.cshl.edu/teaching/2011/2011.Lecture3.Sequence%20Alignment.pdf

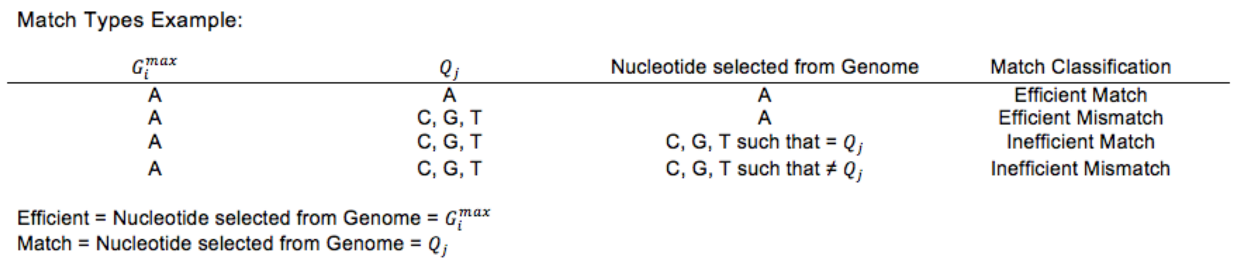
Appendix 1



Genome Array, Structure Example:

Pointer to p_{index}^{max}	1	…	2	1	4	…	3
Prob ('A')	0.99	…	0.0333	0.7	0.2	…	0.0167
Prob ('C')	0.00333	…	0.9	0.1	0.2	…	0.0167
Prob ('G')	0.00333	…	0.0333	0.1	0.2	…	0.95
Prob ('T')	0.00333	…	0.0333	0.1	0.4	…	0.0167
Index	0	…	i	i+1	i+2	…	L-1

Appendix 2



Match Types Example:

G_i^{max}	Q_j	Nucleotide selected from Genome	Match Classification
A	A	A	Efficient Match
A	C, G, T	A	Efficient Mismatch
A	C, G, T	C, G, T such that = Q_j	Inefficient Match
A	C, G, T	C, G, T such that ≠ Q_j	Inefficient Mismatch

Efficient = Nucleotide selected from Genome = G_i^{max}
Match = Nucleotide selected from Genome = Q_j