Alignment algorithms for probabilistic sequences

Ozan Ciga, **Mansha Imtiyaz**
School of Computer Science
Computational Biology Methods and Research: Final project report
McGill University
{ozan.ciga,mansha.imtiyaz}@mail.mcgill.ca

Abstract

Although there are fast methods for sequence mining and alignment between a short query sequence and a large database (e.g. BLAST), these methods do not account for possible inaccuracies in the nucleotide sequences that render each nucleotide not a value, but rather a set of values with different probabilities. In this paper, we present a method that is analogous to the BLAST, but accounts for the probabilistic nucleotides in the database to be queried. The designed system can work with different *w-mers* (7, 11, 15 etc) and sensitivity parameters can be adjusted to any threshold to capture any possible substring with nonzero probability. Ungapped and gapped extensions are also modified, since we have to consider the fact that there is no direct *match* or *mismatch* for the probabilistic entries, so a "match" can only be scored in terms of its probability of occuring in that location. System is tested by picking multiple random short sequences from the database, modifying them (via indels and substitutions) slightly and seeing if the system can locate these short sequences. Performance is evaluated by checking if the system can locate the sequences in *low confidence regions*, where the assigned nucleotide probabilities in the database are not high. Also, we will check if the system can detect the high confidence regions with small modifications in the sequence, thereby proving that the system is flexible and can work on both extremes.

1 Introduction

Sequencing errors and other phenomena that can't be controlled lead to low confidence nucleotides or regions in the genome. To account for this, sequencing is done with an additional step: We add a confidence value associated with the each sequenced nucleotide in the genome. Since an individual nucleotide sometimes can take more than a single value, using the standard alignment algorithms and databases will surely miss some valid cases under this scheme. There are different ways to approach probabilistic sequence alignment. For instance, Li et. al (2009), propose a dynamic programming algorithm that takes O(mn) time, yet this is not feasible for a huge database (e.g. a genome versus a small search query), so we turn our attention to the much simpler BLAST type online hashed databases.

Probabilistic sequences introduce many questions in sequence alignment and querying of a short sequence in a database. How can we score a match? What does a "mismatch" mean for a probabilistic sequence? How to determine E-values for probabilistic sequence alignments, which introduces another layer of randomness on top of the one we already try to estimate?

All of these questions are addressed in this paper under the methodology section. In the same section, we also describe a method that directly mimics BLAST for probabilistic genome (note that if the short search query was probabilistic and genome deterministic, we would just iterate over the original BLAST with all the possibilities and this problem wouldn't be as interesting), from finding hits to scoring and extending matches. Results section includes the conducted tests with multiple queries and how well the system performs under different queries. In discussion and future work section we argue the strong and weak points of our method, and discuss possible extensions of the work and conclude the paper.

2 Methodology

The system is built of two parts: Generating the probabilistic BLAST database and performing queries on it to extend the hits. Former is pretty straight-forward, although there are certain trade-offs will be made for the heuristic system to have acceptable computational cost and space. In the latter, we describe how we can modify the standard ungapped and gapped extensions to account for the probabilistic genome sequence, and how can we score them to list the most meaningful matches in the database in correct order.

2.1 Designing a probabilistic BLAST lookup database

Main trick BLAST uses is to set up lookup table and hash the indexes of each possible w-mer to this lookup table. We've thought about many ways to mimic this behavior with minimal effort, but with a deterministic database, this is impossible. For any query substring to be searched for hits, we have to check each w-mer combination, since in the database, the substring may occur probabilistically. For example, assume "ACT" is the substring, then we'll have to look for all substrings "***", and check whether it assigns nonzero probability for each position. We may eliminate some substrings, yet this process will have $O(4^{\text{substring size}})$ complexity, **not** including the remaining work (selecting most probable hit as candidate, ungapped and gapped extensions, calculating E-values etc).

In contrast, dynamic programming algorithms are also not feasible, both in terms of time and space. Even if we had found a method which has constant multiplier additional cost, i.e. O(c*m*n), this is not acceptable.

Instead, we turn our attention to hashing all *not unlikely* combinations to a lookup table. The term not unlikely" indicates, whether a substring $\{S_1, .., S_n\}$ may not have a large probability, it is not less than a predetermined threshold. For example, if all nucleotides in positions S_i have ≥0.9% probability of being their corresponding nucleotides, any other combination will have $(0.1)^{\text{w-mer size}}$, which, for w=7 is *one in 10 million*. Although the system theoretically allows for any threshold value to keep in the database, the database sizes blow up pretty quickly: For *probability threshold* of 10^{-5} (i.e., any possibility than can happen 1 in 100 thousand), the very small sequence database given to us for the project is estimated to require ~**415 GB's**, which is highly infeasible.

Fortunately however, even with thresholds in the order of 10^{-3}, we can capture the most unconfident regions which no difficulty. For example, smallest 7-mer probability for our dataset is 0.0029551, and since the individual probabilities are low, i.e. each (or multiple) nucleotide can be switched to some other nucleotide, and the probability will not decrease drastically. This is because we use $\bar{p}=\frac{1-p_i}{3}$ for the switched nucleotide, and when the p_i is low, $\bar{p}$ will be large.

In contrast, this method does not allow multiple switches in very high confident regions. For example, assuming we have only p=0.99 and 1 in a 7-mer, observing a drastically different combination is very low, because (1-0.99)/3 = 0.003, and multiplying these will lead to lower probabilities than threshold. But, this is not a huge problem, since we *don't expect* to see that much deviation in these high confidence regions. The model still allows for some of this, if the remaining high confidence regions are preserved. For example, for ACAG**A**AT, where **A** has p=0.99, and remaining positions are 1 or 0.99, with current thresholds we can still observe ACAG**G**AT, and the database will have this position in the lookup table.

As with all approaches that are approximate, we are not guaranteed to find the optimal answer under this scheme. We can get the optimal solution by setting the threshold to a very value, but this will increase the cost in space and in time. In fact, system will exhibit exponential time (since there are exponential number of combinations for a substring), which is much worse than a dynamic programming approach that takes pseudo-linear time. The database generation algorithm is sketched below.

Algorithm 1. Generate entries of BLAST database.

Input: Probabilistic Genome database DB
Output: Hash table B

1: for each position **i** in database DB
2: substr = DB[**i**:**i**+(w-1)] = $\{S_i..S_{i+(w-1)}\}$
3: permute for each S_j in substr with $p_j<1.00$
4: $\bar{p_j} = \begin{cases} \frac{(1-p_j)}{3}, & \text{DB[j]} \neq \text{Sj} \\ p_j, & \text{DB[j]} = \text{Sj} \end{cases}$
5: calculate w-mer probability P = $\prod_{k=i}^{i+w+1} \overline{p_k}$
6: if P>*threshold*

7: add (P, **i**) to look-up table B

Note that the "permute" part can take $O(4^w)$ time, trying each {A,C,G,T}. Indeed, even for a small genome, it takes around an hour to scan the whole database, yet this is done only once, so it is acceptable. Overall, the time complexity is $O(n*4^w)$, where n is the length of the genome. The space complexity is the same, yet it is dependent on the *threshold*, which in fact reduces the space requirement tremendously, with the trade-off that the system will miss most of the hits with really low probability (in the order of one in a million).

2.2 Ungapped extension phase and scoring

Once we have a list of hits for the input query with the probabilities, we proceed towards second part of pipeline, ungapped extension, that sifts through exact word matches between query and the probabilistic database and decides whether to perform a more accurate comparison between them. Ungapped extension attempts to grow a larger but possibly inexact word by allowing mismatches to appear in the extended word. The extended word, called *high-scoring segment pair(HSP)* in Blast system is considered worthwhile if it has few enough mismatches. In case of matches, exceeding beyond a threshold, the HSP is discarded.

The scoring scheme has to be designed keeping in mind the varying probabilities. Probability of a nucleotide equal to 0.25 means each nucleotide is equally probable where as probability less than 0.25 would likely result in a mismatch in that position. Probabilities [0.25 to 1] with a match are likely to have higher score.

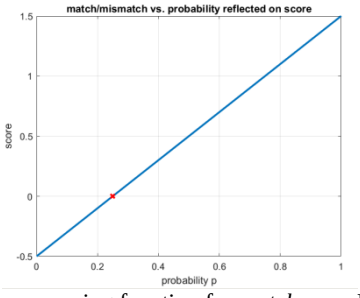

Figure 1. Linear scoring function for *match* case. For each p>0.25, score is positive, and for p<0.25, it is negative since assigning the predicted value will be more likely than not wrong. As confidence of the nucleotide increases, we increase the score of the alignment.

Our system takes this into account by designing a mixture model for two probability sets (high and low probabilities) and setting up the scoring scheme for the two sets. For scoring each of the positions in the input query to the probabilistic nucleotide in the database, we take three constants for scoring high probability matched nucleotides, low probability matched nucleotides and high probability mismatches. The scoring has been such that the high probable matches get the highest score where as the positions where the probability of a match is low get a comparatively lower score. The algorithm to score the nucleotides is described below.

Algorithm 2. Scoring the positions.

Input: Nucleotide a of input query, b from DB and probability p_b .
Output: Score s.

1: if a = b
2: s = match_score*prob(b) // if high probability
3: match_low_score*prob(b) // if low probability
4: if a ≠ b
5: s = 0 // if high probability
6: match_low_score*$\frac{1-p_b}{3}$ // if low probability

In case of mismatches with high probability, no score is given so that we get only increasing scores with a constant line in the graph showing the area of mismatches as shown in the figure below.

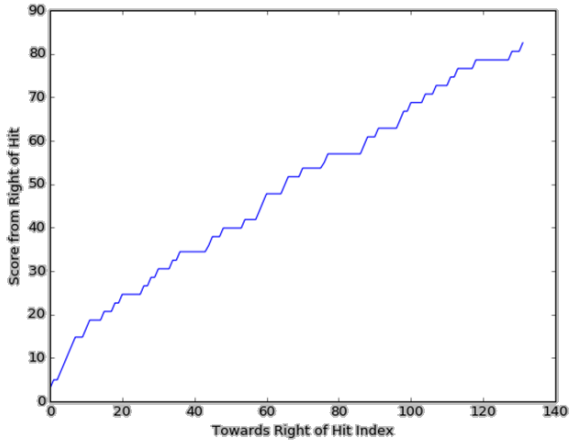

Figure 2. Ungapped Extension towards right of hit w-mer

Each hit we get from the hash-table is extended, to test whether it is contained within the most probable alignment. Because only a small fraction of these hits are expanded, the average amount of computation required is less. The hits are extended to the left and and the right by some distance. Starting from the score of the hits, we keep track of

the running score of the alignment.

Algorithm 3. Left and right extensions of the alignment.

Input: w-mer h and input query.
Output: Ungapped extension from both ends.
1: For each nucleotide **a** in input query and **b** in the DB:
2: Calculate score(a,b)
3: Add Scores to the list

If aligning the next nucleotide from the input query Q and the sequence S increases the running score above its best value, the alignment is enlarged to include the nucleotide. If adding the next nucleotide keeps the score constant for more than a threshold X, the extension should stop. Large threshold values run the algorithm for longer before it gives up finding an extension. The algorithm processes every nucleotide till we encounter the end of input query or the database ends and saves the running score for each point. To weed out the HSPs with less probability, we then find the rate of change of score and stop when the score is constant for more X times and add to the output list the sequence up to that point.

Algorithm 4. w-mer extension.

Input: Hash table B.
Output: Most probable matched alignments in sorted order.
1: for each **w-mer** in table B
2: Find right and left extensions for each **w-mer**
3: for nucleotides **n** in both right and left extensions
4: while no. of mismatches < threshold T
5: add **n** to the output sequence
6: Display the Output

2.3 Gapped extension phase

The gapped alignment is straight-forward. We used the Needleman-Wunsch algorithm for doing gapped alignments (Bao, 2016), and we did only minor modifications on a Python implementation. The recursion and the scoring schemes are almost the same, but we don't extend the alignment towards each end, instead we do local searches. The gapped extensions stops after we go down by the maximum score achieved so far by a certain threshold (10% from the maximum). Even when the extension is performed until the query sequence is exhausted (every nucleotide matched with a nucleotide or aligned with a gap), the time complexity is O(n), where n is the length of query sequence.

$$M_{ij} = max \begin{cases} M_{i-1,j} - b, & \binom{s_i}{-} \\ M_{i,j-1} - b, & \binom{-}{t_j} \\ M_{(i-1,j-1)} + \delta(s_i, t_j), & \binom{s_i}{t_j} \end{cases}$$

Although we use distinct match and mismatch scores in the scoring, we can just use a single value for it, since defining "match" and "mismatch" under this probabilistic scheme is a bit fuzzy.

$$\delta(s_i, t_j) = \begin{cases} p_i * match\ score, & s_i = DB[i] \\ \frac{1-p_i}{3} * mismatch\ score, & s_i \neq DB[i] \end{cases}$$

But using this scoring gives the system the flexibility under certain scenarios: If the genome (or a region of the genome) is sequenced with very high confidence, this method allows for penalizing mismatches even harsher than the affect the $\frac{1-p_i}{3}$ coefficient will have on the score. This scoring still allows us to use the E-values we did before in the ungapped phase, since the scoring will still go up when we get good matches and go down otherwise.

2.4 Calculating the E-value

In BLAST, E-value is used to describe the number of hits one can expect to see by chance when searching a database of particular size.

As the score increases, E-value should decrease exponentially. Our system has been designed such that the score increases for more matches with higher probability. So, we are calculating the e value in the same way as Blast does. That is, $E = Kmne^{(-\lambda S)}$, where m is the database size, n is the length f input query, K and lambda are for scaling the search space size and the scoring system.

3 Results

Version	Found/Queried	% accuracy
High conf.	148/150	0.986
Mid. conf.	143/150	0.953
Low conf.	109/150	0.726

Table 1. Matching performance of the probabilistic system.

We conducted multiple tests using the genome to extract sample points. The accuracy measure here is that if we observe the actual search sequence in the top 10 results returned by our algorithm.

For *high confidence samples*, we used query sequences ranging between 50-100 in length from

regions of the genome that have very high probability (lowest probability for a nucleotide in a w-mer is 0.8) assigned to the sequenced nucleotide. We picked 150 random high confidence sequences and ran the database on these sequences. Not surprisingly, the hits found in the database come up at the top, and extending these favors the high probability extensions. In this test, we performed substitutions of nucleotides to see if the system still can detect the correct subsequence in the genome in face of minor changes. The results are almost perfect, yet this is expected due to the biased way we conducted these tests.

Interestingly, we observe very similar behavior from the mid. confidence (all probabilities are ≥ 0.6) and low confidence (we deliberately pick the sequences that have more than 4 nucleotides with probabilities ≤ 0.5) samples. We did some substitutions and indels here as well (but not so extreme, in the order of 5-15 nucleotides). After couple of experiments, we realized why we were observing such high accuracies: Although there are low probability regions, a sequence with length 150 nucleotides will surely have confident regions (w-mers with nucleotides that have high probability values), hence even when we expand from a (i.e., pick the starting point) low probability regions, the search is actually done as described in the BLAST lecture notes (Cameron), and in that 150 nucleotides (about 140 7-mers), we are very likely to come across some high scoring regions.

We considered artificially manufacturing completely low confidence regions, but then decided not to because it would be an unrealistic scenario. But this result leads to possibly the most important conclusion from this project: *As query sequence length increases, the probability of getting correct matches increase because we'll get strong evidence from somewhere along the sequence, and expand on that*. For example, assume that our search sequence is as follows (this is from the genome we used):

GGTCCAGGCCGCGTAGTGGCCCGACGGG
GCGGGGCCGGCGGGGGGCGCGTAGCC
CGGCGGGGCCCAGGGGCAGGCGTCGA
GCAGCGCGGGCAAGTCCTC**TGGGGGC**
CGCGGGGCCCGCGCCGGGAGGGTCAG
CGGG

This sequence is one of the least likely (product of the probabilities in the sequence is the lowest) sequences of length in range of 100-150. The bold one is the least likely w-mer. But there are inevitably (since it is very unlikely to have such long consecutive sequences with no high confidence regions) w-mers that have high confidence values, and then we use those to expand on the genome and align locally. Of course, there might be cases where we look for TFBS and similar very short sequences in the genome, and if the probabilities assigned to those are not high in the genome, this method will likely miss those. Yet for finding relatively long and conserved regions in the ancestral sequence, we will get good results.

4 Discussion and future work

Trying to increase the sensitivity of the system by capturing lower probability sequences does not work, simply because of the need to assign a slot for each of the very unlikely sequences in the genome. The space requirements exponentially increase without the threshold, and then since we have clunky and large database, search complexity will go up because of having to look for each of the entries in the database (unless we put a cap on that as well). One can argue that putting these sequences to database is pointless, since even if there are matches for it, due to the low probability and how we designed our scoring function, we will not observe the results coming from those sequences in the top 10 results anyway. And if the sequence is long enough, some other higher probability w-mers will be detected in the genome and expansion through these more likely hits will lead to alignment of the sequences anyway.

As described before, this method will not perform as well on short sequences such as transcription factor binding sites. On the other hand, we did our experiments by extracting small fragments from the genome and making minor alternations on them (indels and substitutions). This is a biased experiment, since we are not sure if these reflect the actual cases that occur in the course of evolution.

We used w=7 for our experiments. Although this is traditionally considered as slow, using larger hit sizes might be problematic due to the observation we made before: In low confidence regions, we try finding high confidence snippets that can

help us expand in the right location in the genome. For example, using w=15 with a search sequence that corresponds to a very low confidence region in the genome may miss the actual spot.

In the future, these things may be put into consideration, to see if the system really performs as good as we observed here. A side note would be to put a more compact package, since our codes for the project are dispersed and not in a single, end-user product form. For example, right now our database generation, search, gapped alignment, ungapped alignment are all in different pieces, yet we present the results as if they are extracted all at once.

References

Li, Y., Bailey J., Kulik L., Pei J. (2009). *Efficient Matching of Substrings in Uncertain Sequences*. SIAM.

Bao, F. (2016). *Python implementation to Smith-Waterman Algorithm*.

NCBI, *The Statistics of Sequence Similiarity Scores*. Retrieved December 9, 2016. URL: https://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html

Cameron D., *Fast alignment heuristics*. McGill Bioinformatics course (561), page 13. Retrieved December 9, 2016, from myCourses.

Appendix

We uploaded our codes and some logs generated by our system to the following Google Drive folder. The codes need the BLAST database we generated to work (around 1 gigabytes). Further information how to run the codes are in the readme file provided in the folder:
https://drive.google.com/drive/folders/0B-7HtboGDodqYnlGUThEbzVhUnM