



McGill

BLAST FOR PROBABILISTIC DATABASE GENOME

COMP561 – Computational Biology Methods

December 13th, 2025

Mansour Mohamed-Anis, 260985264

Yilin An, 260967497

Xiaoyi Xu, 261114154

Faculty of Computer Science, McGill University

TABLE OF CONTENTS

TABLE OF CONTENTS.....	1
INTRODUCTION.....	2
METHODOLOGY.....	2
1. Input data.....	2
2. Design overview: BLAST style pipeline.....	3
3. Seed database construction.....	3
Purpose of seeding.....	3
Output of seeding.....	3
Variant seed words.....	4
Data structure choice for the seed index.....	4
Persistence to avoid rebuilding.....	4
4. Score matrix database construction.....	5
Motivation.....	5
Score definition.....	5
Data structure and persistence.....	5
5. Query generation and testing.....	5
6. BLAST-like alignment algorithm.....	6
Seed search stage.....	6
Fallback seeding using variants.....	6
Ungapped extension stage.....	6
Gapped refinement stage.....	7
Output and visualization.....	7
RESULTS.....	7
DISCUSSION.....	8
1. Discussion of failed approaches.....	8
2. Future work.....	9
3. General considerations.....	10
4. Pathway to impact.....	11
CONCLUSION.....	11
REFERENCES.....	12
CODE AVAILABLE ON.....	12

INTRODUCTION

When analyzing ancient genomes, computational biologists often face the challenge of working with completely deterministic databases. In fact, at each genomic position, the presence of a specific nucleotide is frequently represented by a probability distribution rather than a single base. As a result, standard alignment tools, such as BLAST, which assume a deterministic nucleotide sequence, are no longer applicable in the context.

To overcome this critical limitation, our group developed an enhanced heuristic version of the BLAST algorithm specifically adapted for probabilistic genomic data. This novel approach builds upon the use of specialized databases and a modified scoring matrix that accounts for nucleotide uncertainty. Through integrating probabilistic information into the alignment process, our method enables more trustworthy genome alignment for ambiguous sequences.

Probabilistic models have been incorporated into sequence alignment in a number of contexts. Profile hidden Markov models (profile HMMs) and probabilistic scoring frameworks represent sequence families and alignments with explicit probabilities rather than fixed substitution scores, enabling integration of uncertainty and quality information. These models underpin tools such as those based on probabilistic local alignment and Bayesian posterior inference [1].

Other work addresses uncertainty directly in sequence data. For example, recent research introduces probabilistic frameworks that propagate uncertainty from base calls through downstream genomic analyses, employing probability matrices to capture per-base uncertainty [2].

Despite this prior work, most existing alignment methods either assume a deterministic reference or use probabilistic representations for profiles or reads rather than the reference genome itself being probabilistic at every position. Our work differs by adapting the BLAST heuristic to a fully probabilistic genome database, integrating probabilistic seed scoring and fallback mechanisms tailored to uncertainty at each genomic position.

METHODOLOGY

1. Input data

The provided dataset includes:

- `chr22_ancestor.fa`: the dominant (most likely) nucleotide at each genome position (a standard FASTA style sequence).
- `chr22_ancestor.conf`: a confidence list, one value per position, representing the probability of the dominant nucleotide at that position.

This project uses a simplified but consistent probabilistic model. At each position i , the dominant nucleotide has a probability $P_{max}(i)$ given by the confidence file. The three other nucleotides share the remaining probability equally:

$$P(other, i) = \frac{1 - P_{max}(i)}{3}$$

This assumption is computationally convenient and matches the idea that uncertainty is symmetric among the three non-dominant bases when no other information is provided.

2. Design overview: BLAST style pipeline

The algorithm was built as a sequence of components, each solving one part of the full problem:

- Precompute a seed database of short words to enable $O(1)$ lookup.
- Precompute a per-position scoring structure that supports $O(1)$ scoring during extension.
- Generate test queries from the genome with controlled mutations and random extensions.
- Implement a BLAST-like search:
 - seed search using words from the query sequence
 - fallback mechanism using variant seeds if exact seeds are weak
 - ungapped extension first
 - gapped refinement second, only on flanks

Each step was implemented in a separate file to make the pipeline reproducible and modular. The main innovation is the construction of the databases to include probabilistic nucleotides.

Before any design choices were made, we studied the genome sequence to understand the range of probabilities available [0.39 – 1.0]. While this did not impact the overall design, some tuning variables were chosen based on this analysis.

3. Seed database construction

Purpose of seeding

Direct dynamic programming alignment of a query against an entire chromosome is too expensive. BLAST solves this by first finding short exact matches (seeds), then extending only promising regions. This project follows the same principle. We used word length $k = 11$ because it is long enough to reduce random matches, and short enough to still find seeds under moderate mutation. It also matches a common BLAST style choice for nucleotide seeding, which is already optimized.

Output of seeding

For each genome start index i , define the dominant k mer:

$$w_i = G[i:i + k], \text{ while: } i \leq (G - w + 1)$$

We compute the probability that the probabilistic genome would generate that dominant word (all the highest likelihood nucleotides) under the simplified model:

$$P(w_i) = \prod_{t=0}^{k-1} P_{max}(i + t)$$

This value is stored so seeds can later be filtered and ranked by probability.

Variant seed words

In addition to storing the dominant word, we store variants where one position is replaced by '_'. The underscore character does not indicate a gap. It means: "one of the three non-dominant nucleotides at that position". Allowing one error in the seed effectively reduces the number of perfectly matched characters required in a row and increases the likelihood that the seed-finding algorithm finds a match. This is important because the database genome is probabilistic in its nature, and even a single low-likelihood nucleotide within the seed would result in it having a very low probability. For example:

Nuc	A	A	A	A	A	A	A	A	A	A	A	total
Prob	0.99	0.99	0.98	1.0	0.91	0.7	0.85	0.99	1.0	1.0	1.0	0.515

Versus

Nuc	A	A	A	A	A	A	A	A	A	A	A	total
Prob	0.99	0.99	0.98	1.0	0.91	0.40	0.85	0.99	1.0	1.0	1.0	0.294

For each position j within the word, a variant is formed: w_i with the j th base replaced by '_'. Its probability is computed by substituting the dominant base probability with the alternative probability at that position:

$$P(w_{i,j}^{var}) = P(w_i) \cdot \frac{P_{other}(i+j)}{P_{max}(i+j)}, \text{ where: } P_{other}(i+j) = \frac{1-P_{max}(i+j)}{3}$$

This design makes sense because it encodes a single mismatch away from the dominant genome, it avoids enumerating all 3 explicit alternatives, and it allows lookup by a compact pattern string.

Data structure choice for the seed index

The seed index must support fast lookup by word, not by position. Therefore, the natural structure is a hash map keyed by the word string. We store:

- key: word (a string of length 11 containing A/C/G/T or one _)
- value: list of occurrences, each occurrence is (genome_start_index, probability)

This supports an average $O(1)$ lookup for a word, returning all candidate genome locations for that seed and attaching a probability to each hit so the search can be probability aware.

Persistence to avoid rebuilding

Building this index is expensive, but it only needs to be done occasionally. Therefore, it is saved to disk. A Python pickle file was used because it preserves Python-native structures without manual serialization code, loads quickly on repeated runs, and avoids recomputing the index. This is why the database file is written directly into the working directory and then reused by later programs.

Implementation file: `build_db.py`

Output file created: `probabilistic_db.pkl`

4. Score matrix database construction

Motivation

During extension, scoring must be extremely fast. If scoring required recomputing probabilities or performing complex logic at each alignment step, the runtime would increase significantly. Instead, a per-position score vector is precomputed, so that scoring a query base at a genome position takes $O(1)$ time.

Score definition

At genome position i , define scores for matching query base X in $\{A, C, G, T\}$:

$$S(X, i) = 1 - t \cdot (P_{\max}(i) - P(X, i)), \text{ where } (t = 3) \text{ is a tunable parameter}$$

This produces a maximal score for the dominant base at that position, and a reduced score for non-dominant bases, proportional to how confident the dominant call is. Given the simplified probability model, if X is dominant at i , then:

$$P(X, i) = P_{\max}(i). \text{ Otherwise, } P(X, i) = \frac{1 - P_{\max}(i)}{3}$$

The score model is designed so that high-confidence positions strongly penalize mismatches, low-confidence positions penalize mismatches less, and the scoring aligns with the probabilistic interpretation.

Data structure and persistence

The score database is stored as a list: `score_db[i] = (S(A, i), S(C, i), S(G, i), S(T, i))`

This supports $O(1)$ access by genome index and $O(1)$ access by base using a simple mapping dictionary. The database is saved as a pickle file for fast reuse.

Implementation file: `build_score_matrix_db.py`

Output file: `score_matrix_db.pkl`

5. Query generation and testing

To evaluate the algorithm, we generate queries from the genome so that the true location is known. The query generator does the following:

- Choose a random genome start index
- Extract a window, for example, ~20 bases or longer, depending on testing needs
- introduce random point mutations, at least 3, to ensure the query is not identical to the genome
- Write a FASTA output with headers that store start and end indices

This enables objective evaluation because the algorithm output can be compared to the known location.

Implementation file: `generate_samples.py`

6. BLAST-like alignment algorithm

All search and alignment logic is implemented in `BLAST.py`. The program uses the two precomputed databases:

- `probabilistic_db.pkl` for seeding
- `score_matrix_db.pkl` for fast scoring during extension

Seed search stage

The query is split into all overlapping k mers $q[pos:pos + k]$. Each word is looked up in `seed_db`, and all hits are collected. Each hit provides: the query seed position, the genome seed position, and the seed probability from the index.

Seeds are ranked by probability, and the top 200 are retained. This is consistent with BLAST's strategy of limiting extension work to the most promising candidates. This reduces cost because extension is the expensive stage.

Fallback seeding using variants

`max_prob` is defined as the variant seed word (containing one '`_`') with the highest probability anywhere in the database. This value is used as a threshold because it represents the best possible "single mismatch away from dominant" seed that the database can offer. If our exact seeds are worse than this, it suggests the query may not match the dominant genome well enough at the seed level, and variant seeding becomes justified. Fallback runs in two cases:

- No exact matches are found for words without '`_`'.
- Among the top 200 exact seeds, if at least one has $P(w_i) < \text{max_prob}$.

When fallback triggers, variant seeds are searched and appended to the existing seed list. The original exact seeds are never deleted. To do this, it is important to have efficient variant word generation. Naively inserting '`_`' everywhere and rebuilding all k mers for the query sequence would be expensive. Instead, for each query index i , only words whose window contains i are affected. That produces at most k new words for that index. For word length k , index i affects word starts:

$$s \in [i - (k - 1), i]$$

clipped to valid word start positions. Each affected word is generated by replacing the corresponding character with `_`. These new words are looked up, and any hits are appended. This keeps the fallback cost controlled and predictable with a complexity of $O(|q| \cdot k)$.

Ungapped extension stage

For each selected seed, ungapped extension is performed first. The seed defines a diagonal alignment between query and genome. Extension proceeds left from the seed and right from the seed. At each step, the score is updated using the precomputed per-position scores from `score_db`. An X drop rule is used, meaning that extension stops in a direction when the running score falls more than `UNGAPPED_X_DROP` below the best score seen so far. This is a standard BLAST idea that reduces wasted extension into low-scoring regions. The best ungapped high-scoring segment pair is chosen across all seeds.

Gapped refinement stage

Gapped alignment is not run across the whole region because that can reduce the score by introducing gaps inside a strong ungapped core. Instead, gapped alignment is restricted to the flanks: a left segment outside the ungapped core and a right segment outside the ungapped core. Needleman-Wunsch dynamic programming is used on each flank, using a match score from the per-position scoring matrix database and a constant gap penalty $c = -2$. The final alignment is chosen as the ungapped core alone or the ungapped core plus gapped flanks. Only if the gapped refinement increases the total score is it accepted. This prevents the algorithm from replacing a strong ungapped match with a weaker gapped alignment.

Output and visualization

The program prints the number of seeds found, the best seed and its position, the ungapped and final alignment scores, and an alignment visualization graph.

In this probabilistic BLAST, an E value was intentionally excluded because the theoretical basis that makes E values meaningful in classical BLAST does not hold. Karlin Altschul statistics assume a fixed substitution matrix, independent and identically distributed letters, and position-independent background frequencies [3]. Here, scores depend explicitly on genomic position through position-specific probabilities, so the scoring system is non-stationary and violates these assumptions. As a result, the analytical distribution of maximal alignment scores is unknown, making any classical E value mathematically invalid. While an empirical E value could be estimated using Monte Carlo simulations, doing so would require repeatedly running the complete seeding and extension pipeline on randomized queries, which is computationally far more expensive than the search itself ($O(|q| \cdot k \cdot \text{samples})$). Additionally, because scores depend on local genome context, they are not directly comparable across positions or genomes, making an E value difficult to interpret even if computed. Instead, the algorithm relies on strong internal statistical controls such as probability-based seed ranking and probability-based extension scores. For the intended use case of targeted biological inference rather than large-scale database screening, the relative ranking of alignments is more meaningful than a global E value.

RESULTS

We evaluated the proposed method using a set of queries generated by applying random mutations to known intervals of the reference genome. For each query, the algorithm reports a single best alignment obtained by extending the top-200 probability-ranked seeds with ungapped X-drop extension, followed by local gapped refinement. Alignment performance was assessed in terms of localization accuracy, boundary precision, interval overlap, and computational efficiency.

- $Accuracy = \frac{\text{number of queries correctly localized}}{\text{total number of queries}} = \frac{1}{N} \sum_{i=1} [overlap_i]$
 - $Groud_truth_interval = [db_start, eb_end]$
 - $Predicted_interval = [ps, pe]$
 - $Overlap = \max(0, \min(pe, E) - \max(ps, S))$
- Boundary precision:

- Start error = predicted start – true start
- End error = predicted end – true end
- Interval overlap:
 - $IoU = \frac{|pred \cap truth|}{|pred \cup truth|}$

	Localization accuracy	Start error (mean,bp)	End error (mean,bp)	IoU(mean)	CPU Running time per query(s)	Resource Usage(MB)
100 queries	1.0	-10.7	15.8	0.82	2.99	~1994

1. Localization accuracy

Overall, the method achieves high localization accuracy under moderate sequence uncertainty. Accuracy is interpreted as a localization metric rather than a measure of exact boundary correctness. All predicted alignment intervals overlap with the ground truth regions, resulting in an accuracy of 1.0. This indicates that the algorithm consistently localizes the correct genomic region, although small boundary offsets remain, as reflected by non-zero start and end position errors.

2. Boundary Precision

Analysis of boundary errors reveals a mild systematic bias in alignment boundaries. The median start error is -10.7 bp, indicating that predicted alignments tend to begin slightly earlier than the true region. In comparison, the median end error is 15.8 bp, indicating a tendency to extend beyond the actual endpoint. This behavior is consistent with the sensitivity-oriented nature of ungapped X-drop extension.

3. Interval Overlap

The mean IoU is approximately 0.82. These values indicate substantial overlap between predicted and true alignment intervals for most queries. Even when boundary offsets occur, the algorithm generally recovers the correct core alignment region, resulting in high overall overlap.

4. Computational Efficiency

From a computational standpoint, the algorithm processes each query in approximately 3 seconds on average. Peak memory usage is approximately 2 GB, reflecting the storage requirements of probabilistic databases and dynamic programming during gapped refinement. Overall, these results demonstrate that the method remains computationally feasible for a BLAST-style prototype despite the added complexity of probabilistic alignment.

DISCUSSION

1. Discussion of failed approaches

At first, we considered a “greedy” BLAST-based heuristic that reduces the probabilistic database to a deterministic sequence and iteratively refines it based on alignment feedback. Specifically, we construct a representative genome by selecting, at each position, the

nucleotide with the highest probability, thereby forming a deterministic database sequence. Using this database, we plan to perform a standard BLAST search to identify all perfect-matching seeds between the query and the genome. Each seed is then extended with an ungapped extension. In this procedure, if a significant drop is observed, the corresponding position is considered unreliable. The nucleotide at this position is then replaced with the second-likely one, and the effect of the modification on the alignment score is evaluated. The method is applied iteratively to all positions to detect any substantial drop in score, progressively refining the alignment until the end of the query sequence is reached.

We soon noticed that the method is not viable due to several fundamental drawbacks. First, it collapses the probabilistic genome into a deterministic sequence, eliminating many likely-valid seeds. As a result, no perfect-matching seeds may be identified, even when a correct seed is in the underlying probabilistic genome. Second, the iterative refinement of extension can generate a large set of ambiguous positions. For each position, there are multiple plausible nucleotide substitutions, leading to an exponential growth of candidate modifications. Consequently, the computational cost becomes prohibitively high. Third, it ignores the ungapped extension by over-relying on local probabilistic nucleotide substitution. However, biologically significant alignments often involve gaps, and the greedy strategy cannot model gapped extension. Finally, as a greedy heuristic, the approach optimizes only local decisions during extension. Without a globally defined objective across the entire query, it cannot guarantee a globally optimal extension.

The second idea we considered was to construct the top k (e.g., top 10) most likely deterministic genome databases based on the product of nucleotide probabilities across positions. For each database, BLAST and the Needleman-Wunsch algorithm could then be applied to find the query alignment. However, this idea was quickly shown to be infeasible in practice.

First, there are no top k most likely databases. Due to the probabilistic nature of the genome, there likely exist many deterministic databases with very similar, or even identical, overall probabilities. For instance, consider a genomic position with the following nucleotide probability.

Nuc	A	C	T	G
Prob	0.70	0.10	0.10	0.10

If any of the other three nucleotides replaces the most likely nucleotide A, this single modification already generates three distinct databases. When multiple such uncertain positions are considered, the number of candidate databases grows exponentially. Second, assuming we have a manageable set of likely databases constructed, there is no criterion for comparing alignments obtained from different databases. In particular, it is unclear how to trade off a high-scoring alignment derived from a low-probability database against a lower-scoring alignment obtained from a high-probability database.

2. Future work

From the perspective of optimization of algorithms, the future world could explore:

- **Adaptive word lengths:** In the current implementation, a fixed length is used uniformly across the genome during the seeding stage. It may be suboptimal in probabilistic genomes where confidence levels vary substantially across positions. In high-confidence regions, longer seeds are likely to have reliable probability estimates and can provide strong discriminatory power, reducing the number of spurious seed hits. A potential improvement is to adopt adaptive word lengths that adjust seed size based on local seed probability. For example, longer seeds could be used when the maximum or average nucleotide confidence within a region exceeds a predefined threshold. In comparison, shorter seeds could be employed in regions with lower confidence to preserve sensitivity. By dynamically balancing seed length with local uncertainty, the algorithm could reduce false positives in confident regions while avoiding false negatives in uncertain regions.
- **Two-hit and Diagonal-Based Filtering:** In the current implementation, each candidate's seed is extended independently using ungapped X-drop extension, which may lead to redundant or low-quality extensions in noisy regions of the genome. A natural improvement is to incorporate a two-hit or diagonal-based filtering strategy. Instead of extending every individual seed, the algorithm can require at least two high-probability seeds on approximately the same alignment diagonal before triggering extension. Since true alignments generate multiple nearby seeds with consistent query-genome offsets, this two-hit criterion effectively filters out spurious seeds while preserving sensitivity to genuine homologous regions.
- **Adaptive X-drop Thresholds:** The ungapped extension phase currently relies on a fixed X-drop threshold to determine when to terminate the extension. However, in probabilistic genomes, confidence levels vary substantially across genomic regions. Future work could explore adaptive X-drop thresholds that depend on local uncertainty, allowing more permissive extensions in low-confidence regions and stricter terminations in high-confidence regions.
- **Affine Gap Penalties in Gapped Refinement:** The current gapped stage uses a constant gap penalty for both gap opening and gap extension. While this simplifies implementation, it does not reflect the biological reality that insertions and deletions typically occur as contiguous events rather than isolated single-base gaps. A more realistic extension would be to adopt affine gap penalties into the local gapped alignment, which would allow the algorithm to better model short indels commonly introduced by mutation processes.

Several extensions could further improve the accuracy and applicability of the proposed method. A possible next step is to replace the current expected-score framework with a full probabilistic alignment model, such as a Hidden Markov Model [4], which treats both matches and gaps probabilistically. This would allow the algorithm to explicitly model insertion and deletion uncertainty, rather than relying on fixed gap penalties.

3. General considerations

Several general considerations arise when designing alignment algorithms for probabilistic genome sequences. Firstly, parameter selection plays a critical role in probabilistic alignment. Choices such as word length, X-drop thresholds, and gap penalties directly influence the balance between sensitivity and specificity. In probabilistic genomes, these parameters also interact with the genome's underlying confidence distribution, meaning that values optimal for

high-confidence regions may not be optimal in low-confidence regions. Adaptive parameterization based on local confidence levels is beneficial.

Besides, evaluation of probabilistic alignment methods requires carefully constructed benchmarks. Standard alignment accuracy metrics assume deterministic ground truth, whereas in probabilistic settings, the “true” sequence may itself be uncertain. In this project, we address this challenge by using queries derived from controlled random mutations of the reference genome, enabling quantitative evaluation while still reflecting realistic uncertainty. More generally, future benchmarking efforts should consider uncertainty-aware evaluation metrics that account for both alignment correctness and confidence calibration.

4. Pathway to impact

The ability to align sequences against probabilistic genomes has important implications for evolutionary genomics and comparative biology. Computationally inferred ancestral genomes are increasingly used to study long-term evolutionary processes. By explicitly incorporating positional uncertainty into the alignment process, our approach provides a principled way to analyze these genomes without collapsing them into a single, potentially misleading deterministic sequence.

In practical terms, this methodology could improve downstream analyses such as ancestral gene identification, conserved element detection, and mutation rate estimation [5]. Alignments that account for uncertainty are less likely to overcommit to incorrect nucleotide predictions, reducing false confidence in inferred evolutionary events. Beyond ancestral genomics, the same ideas apply to other settings where sequence uncertainty is unavoidable, such as low-coverage sequencing, metagenomic assemblies, and consensus sequences derived from heterogeneous populations.

CONCLUSION

This project adapts the BLAST design pattern to a probabilistic genome by introducing:

- a probability-aware seed database that supports both dominant words and controlled single mismatch patterns
- a precomputed per-position score database enabling constant-time scoring
- a two-stage extension strategy where ungapped extension identifies strong cores and gapped alignment refines only the flanks
- a principled fallback mechanism, ensuring variant seeding is used only when justified

The final pipeline achieves fast lookup and practical alignment while preserving a probabilistic interpretation of uncertainty in the genome.

REFERENCES

- [1] Eddy S. R. (2008). A probabilistic model of local sequence alignment that simplifies statistical significance estimation. PLoS computational biology, 4(5), e1000069. <https://doi.org/10.1371/journal.pcbi.1000069>
- [2] Becker, D., Champredon, D., Chato, C., Guban, G., & Poon, A. (2023). SUP: a probabilistic framework to propagate genome sequence uncertainty, with applications. NAR genomics and bioinformatics, 5(2), lqad038. <https://doi.org/10.1093/nargab/lqad038>
- [3] National Center for Biotechnology Information. (n.d.). The statistics of sequence similarity scores. National Institutes of Health. <https://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html>
- [4] Do, C. B., Mahabhashyam, M. S., Brudno, M., & Batzoglou, S. (2005). ProbCons: Probabilistic consistency-based multiple sequence alignment. Genome research, 15(2), 330–340. <https://doi.org/10.1101/gr.2821705>
- [5] Blanchette, M., Green, E. D., Miller, W., & Haussler, D. (2004). Reconstructing large regions of an ancestral mammalian genome in silico. Genome research, 14(12), 2412–2423. <https://doi.org/10.1101/gr.2800104>
- [6] Database .(2025). Mcgill.ca. <https://www.cs.mcgill.ca/~blanchem/561/probabilisticGenome.html>
- [7] Blanchette, M. (2025). BLAST. Mcgill.ca. <https://mycourses2.mcgill.ca/d2l/le/lessons/807533/topics/8600373>
- [8] Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. [https://doi.org/10.1016/0022-2836\(70\)90057-4](https://doi.org/10.1016/0022-2836(70)90057-4)
- [9] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J. (1990) “Basic local alignment search tool.” J. Mol. Biol. 215:403–410.

CODE AVAILABLE ON

<https://github.com/Distiner13/COMP561---Probabilistic-BLAST.git>