Computational Molecular Biology and Bioinformatics

X-Mapper

Malay Bhattacharyya

Associate Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2025

Sequence alignment with fixed-length seeds

The state-of-the-art alignment methods split sequences into contiguous, fixed-length seeds, called k-mers (a subsequence of length k). Some of these include BLAST [1] for similarity search, Minimap [2] for read alignment, and Kraken [3] for metagenomic taxonomy annotation.

- The alignment is faster with longer and unique seeds, but is prone to errors.
- The alignment is more accurate with shorter seeds that avoid mutations, but becomes slower.

Sequence alignment with variable-length seeds

The use of variable-length seeds for alignment purposes is motivated by different prior works.

- Variable-length seeds have previously been used for local alignment in LAST.
- Spaced seeds were first introduced in PatternHunter, and later widely adopted by many aligners like SHRiMP2.

The X-Mapper is a fast and accurate sequence alignment approach that uses the concept of dynamically incorporating k-mers of dynamic sizes and gaps, termed as gapped x-mers [4].

Building a pyramid of x-mers for reference

We first construct a pyramid of x-mers to represent the reference. This considers the following properties.

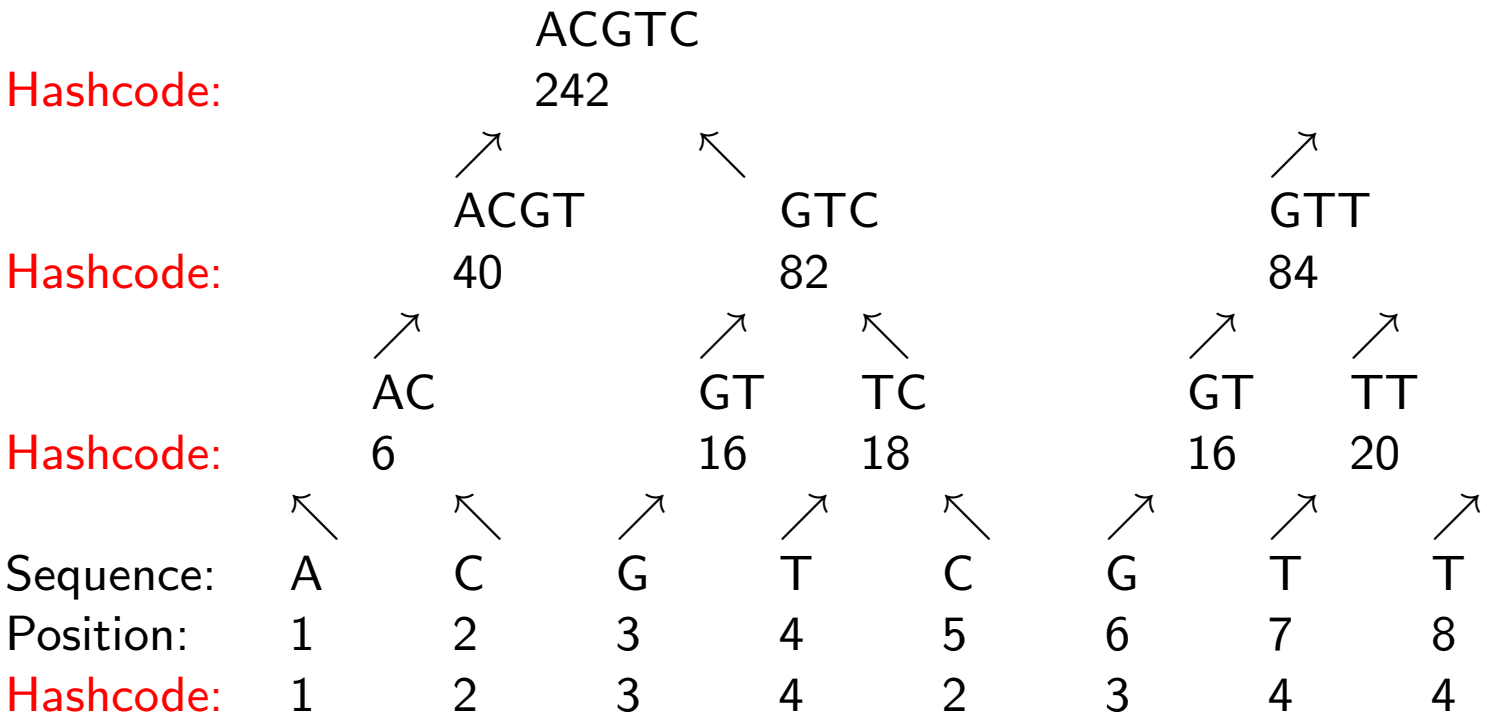
- The construction solely depends on the sequence bases in the pyramid. If the same process is applied to the same sequence in the reference and to a portion of a query, the same pyramid will be produced.
- Every base is covered by an x-mer in each level of the pyramid (different size of x-mers), except for x-mers that are too large and would extend past the ends of the sequence.
- The number of x-mers in each level of the pyramid usually decreases exponentially, which keeps the total number of x-mers in the pyramid less than about 4 times the length of the sequence.

Building a pyramid of x-mers for reference

Construction of the pyramid of x-mers occurs as follows.

1. We build an x-mer of length 1 for each position in reference.
2. To build the next level, we first visit each x-mer and based on the content of x-mer it decides whether that x-mer will merge with the x-mer to its left/right. Each x-mer of length 1 either requests to merge left/right, and encourages its descendants to request to merge left/right, for 4 combinations in total.
3. For x-mers formed by merging two other x-mers, each x-mer usually identifies the parent with the larger hashcode and merge left/right based on its request. For the other parent, the child x-mer passes its request along about which direction its descendants should merge with.
4. We visit every pair of adjacent x-mers, and if either x-mer in the pair requests to merge with the other, then we perform a merge and add the resulting x-mer into the next level of x-mers.

Building a pyramid of x-mers for reference



Note: Larger x-mers are built by merging smaller x-mers left if the sum of number of A's and C's is odd, and right if the sum of number of A's and C's is even. Hashcode for a child x-mer = 4 * Hashcode of left parent x-mer + Hashcode of right parent x-mer.

Building a pyramid of x-mers for reference

Interestingly, if x-mers make independent, uniformly random decisions about which directions they request to merge in, then the probability of any particular child x-mer to exist is $(1 - \frac{1}{2} * \frac{1}{2}) = \frac{3}{4}$. Hence, the number of x-mers at each level decreases exponentially as a function of the level.

For ambiguous bases (e.g., N), we check the relevant possibilities among A, C, G, and T. When multiple (default is 3) ambiguous x-mers merge, we evaluate all of the possible combinations.

To reduce memory usage, we choose hash functions and merge directions in a symmetric way such that the existence of an x-mer with a certain hashcode ensures the existence of the reverse complement x-mer with the same hashcode.

Expanding gapped x-mers

1. When we encounter an x-mer that seems long enough, we first expand it into a gapped x-mer unless this is disabled.
2. We add a gap of length equal to approximately half of the length of the x-mer, followed by a k-mer of approximately the same length. Each x-mer extends its gap to the left if it plans to next merge to the right, and extends its gap to the right if it plans to merge to the left.
3. Each x-mer is finally replaced with a new x-mer that ignores the contents of its gap.

Note: The x-mers choose to extend their gaps to the left if the sum of number of A's and G's is odd; otherwise, they choose to extend their gaps to the right.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.
Can we expand the x-mer AC as _AC? No.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.
Can we expand the x-mer AC as _AC? No.
Can we expand the x-mer GT as A_GT?
Yes. Hashcode of A_GT: $16 * 4^1 + 1 = 65$.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.
Can we expand the x-mer AC as _AC? No.
Can we expand the x-mer GT as A_GT?
Yes. Hashcode of A_GT: $16 * 4^1 + 1 = 65$.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of TC_T: $18 * 4^1 + 4 = 76$.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.
Can we expand the x-mer AC as _AC? No.
Can we expand the x-mer GT as A_GT?
Yes. Hashcode of A_GT: $16 * 4^1 + 1 = 65$.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of TC_T: $18 * 4^1 + 4 = 76$.
Can we expand the x-mer GT as T_GT? No.

Expanding gapped x-mers

Can we expand the x-mer ACGTC as ACGTC__T? No.
Can we expand the x-mer ACGT as ACGT__TT?
Yes. Hashcode of ACGT__TT: $40 * 4^2 + 4 * 4^1 + 4 = 660$.
Can we expand the x-mer GTC as A_GTC? No.
Can we expand the x-mer GTT as GT_GTT?
Yes. Hashcode of GT_GTT: $84 * 4^2 + 3 * 4^1 + 4 = 1360$.
Can we expand the x-mer AC as _AC? No.
Can we expand the x-mer GT as A_GT?
Yes. Hashcode of A_GT: $16 * 4^1 + 1 = 65$.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of TC_T: $18 * 4^1 + 4 = 76$.
Can we expand the x-mer GT as T_GT? No.
Can we expand the x-mer TT as TT_?
Yes. Hashcode of TT_: $16 * 4^1 + 4 = 68$.

Saving x-mers into hashtable

1. Every x-mer is assigned a hashcode based on its content (sequence).
2. The hashcode and the number of bases in the x-mer are used together to identify a bin in the hashtable, and the position of the x-mer is added into the bin if there are not already too many x-mers in this bin. This allows subsequent lookups to quickly identify all of the positions of an x-mer based solely on its hashcode and number of bases used.
3. We impose a limit on the maximum number of x-mers that may be saved into any particular hash bin. This limit is a function of the length of the x-mer.

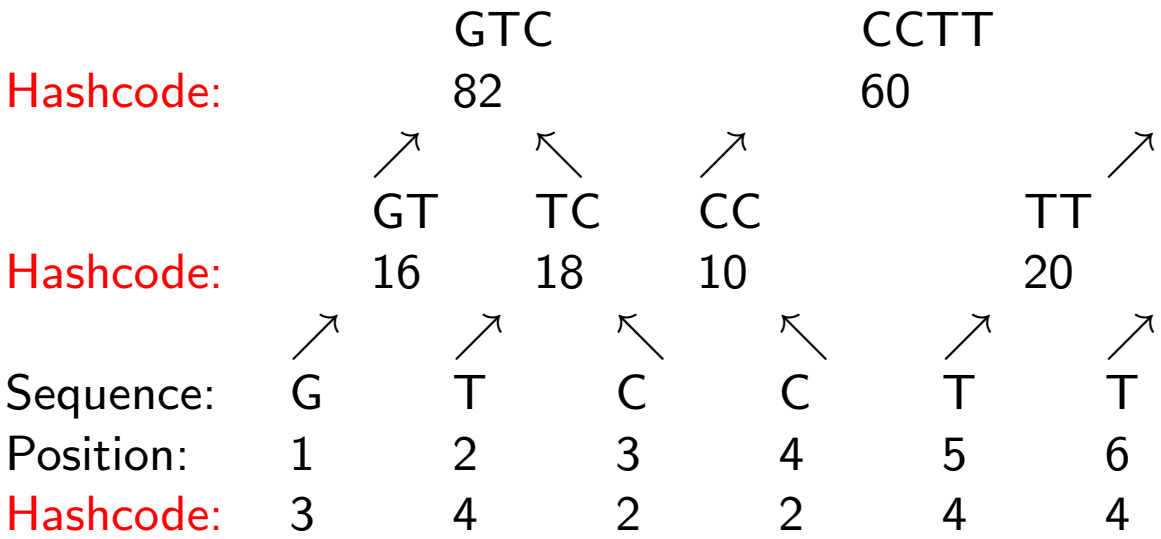
Saving x-mers into hashtable

Hashcode	Position	# Positions	Sequence/x-mer
1	1	1	A
2	2, 5	2	C
3	3, 6	2	G
4	4, 7, 8	3	T
65	1	1	A_GT
68	4	1	T_GT
76	4	1	TC_T
660	1	1	ACGT__TT
1360	3	1	GT_GTT

Building x-mers for queries to align using the pyramid

1. For each query, we build a separate pyramid in the same way.
2. We start at the x-mer in the first position of the first row of the pyramid for the query and perform a lookup in its x-mer database to see in how many positions that x-mer is found. If the number of positions exceeds the maximum allowed, we proceed to the next position of the next row of the pyramid.
3. Otherwise, we save that x-mer and its corresponding alignment offsets (positions of the query in the reference) and return to the next position in the previous level of the pyramid.
4. Repeat steps 2-3 to form a path of x-mers through the pyramid, which dynamically adjusts the level of the pyramid and therefore also adjusts the length of the x-mers based on the number of matches of each in the reference. If gapped x-mers were enabled when indexing the reference, the same are used when analyzing the query.

Building x-mers for queries to align using the pyramid



Note: Larger x-mers are built by merging smaller x-mers left if the sum of number of A's and C's is odd, and right if the sum of number of A's and C's is even. Hashcode for a child x-mer = 4 * Hashcode of left parent x-mer + Hashcode of right parent x-mer.

Building x-mers for queries to align using the pyramid

Can we expand the x-mer GT as _GT? No.

Building x-mers for queries to align using the pyramid

Can we expand the x-mer GT as _GT? No.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of A_GT: $18 * 4^1 + 4 = 76$.

Building x-mers for queries to align using the pyramid

Can we expand the x-mer GT as _GT? No.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of A_GT: $18 * 4^1 + 4 = 76$.
Can we expand the x-mer CC as CC_T?
Yes. Hashcode of TC_T: $10 * 4^1 + 4 = 44$.

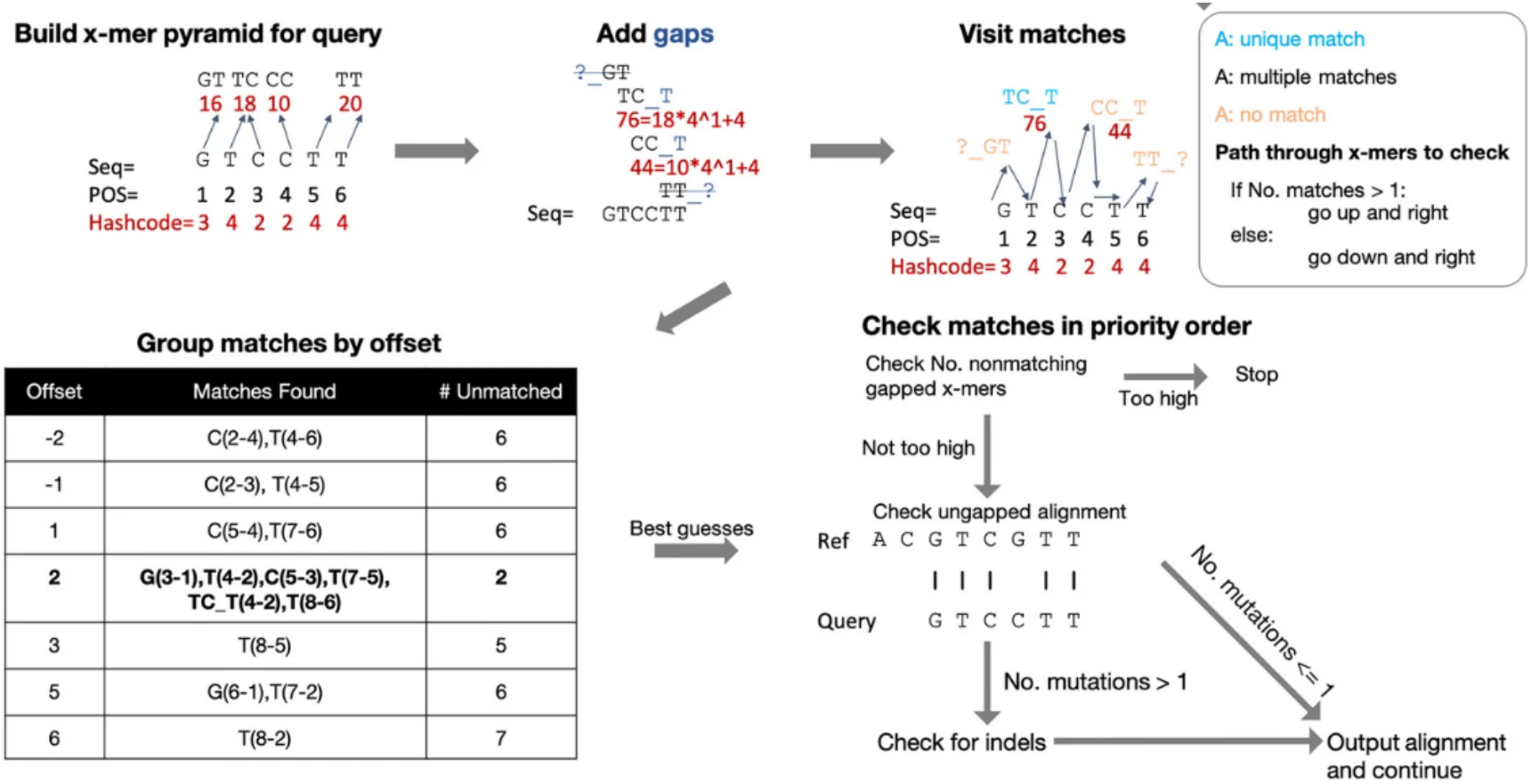
Building x-mers for queries to align using the pyramid

Can we expand the x-mer GT as _GT? No.
Can we expand the x-mer TC as TC_T?
Yes. Hashcode of A_GT: $18 * 4^1 + 4 = 76$.
Can we expand the x-mer CC as CC_T?
Yes. Hashcode of TC_T: $10 * 4^1 + 4 = 44$.
Can we expand the x-mer TT as TT_? No.

Checking optimistically for the single best alignment

1. We follow the x-mer path until one reference offset is found having more x-mer matches than any other, and at least two x-mer matches in total.
2. If step 1 yields exactly one position, we check it first and attempt to determine (using the nonexistence of x-mer matches at other offsets) that it is the best alignment.
3. If we are unable to determine that this optimistic alignment is optimal (which can often be demonstrated via the nonexistence of enough x-mers matching at other offsets), then we continue to follow this x-mer path.

Checking optimistically for the single best alignment



Offset	Matches Found	# Unmatched
-2	C(2-4),T(4-6)	6
-1	C(2-3), T(4-5)	6
1	C(5-4),T(7-6)	6
2	**G(3-1),T(4-2),C(5-3),T(7-5), TC_T(4-2),T(8-6)**	**2**
3	T(8-5)	5
5	G(6-1),T(7-2)	6
6	T(8-2)	7

Visiting offsets in priority order

1. If we observe that the optimistic best alignment is not optimal, we continue to follow the x-mer path and look for new matching offsets. For each discovered offset, we attempt an alignment at that position. Positions that are discovered earlier involve fewer initial mismatched x-mers, and are given higher priority while checking.
2. We generally disregard offsets until two supporting x-mers are found for the same position.
3. The process of checking candidate offsets continues until either of the following happens.
   - We find an alignment and can demonstrate that no other candidate offset can provide a lower penalty (due to the nonexistence of x-mer matches).
   - We can demonstrate that no candidate offset can provide a satisfactory penalty.
   - The x-mer path reaches the end of the query.

Refining bound on alignment penalty

1. Once we find an offset that seems worthwhile to check for a possible alignment, we attempt to compute a bound on the maximum possible alignment penalty at this position, to improve the performance of the subsequent search for indels.
2. We check an ungapped alignment and if its penalty is less than the penalty of a single gap, then that alignment is used.
3. We spot the region around the candidate alignment and extract short pieces (length $\sim \frac{\log(3*QueryLength+1)}{\log(4)+1}$) from it.
4. We also extract short pieces from the query and searche for the maximum number of nonoverlapping, nonmatching pieces that can be detected. If this penalty is higher than what we are interested in, the offset is rejected.
5. Finally, we compute the maximum length of extensions that can exist in an optimal alignment at this offset based on the maximum interesting penalty and the minimum number of mismatched pieces.

Splitting the query and joining alignments

1. If we are unable to determine the nonexistence of indels in the best alignment at the offset, we split the query into several pieces and for each one, separately refine the bound on the alignment penalty (see the previous step) and searche for indels (see the next step).
2. If the best alignments for two adjacent pieces of the query are adjacent, those alignments are re-joined into one contiguous alignment. If neighboring pieces yield best alignments that are nonadjacent, we proceed to thoroughly check for indels (see the next step).

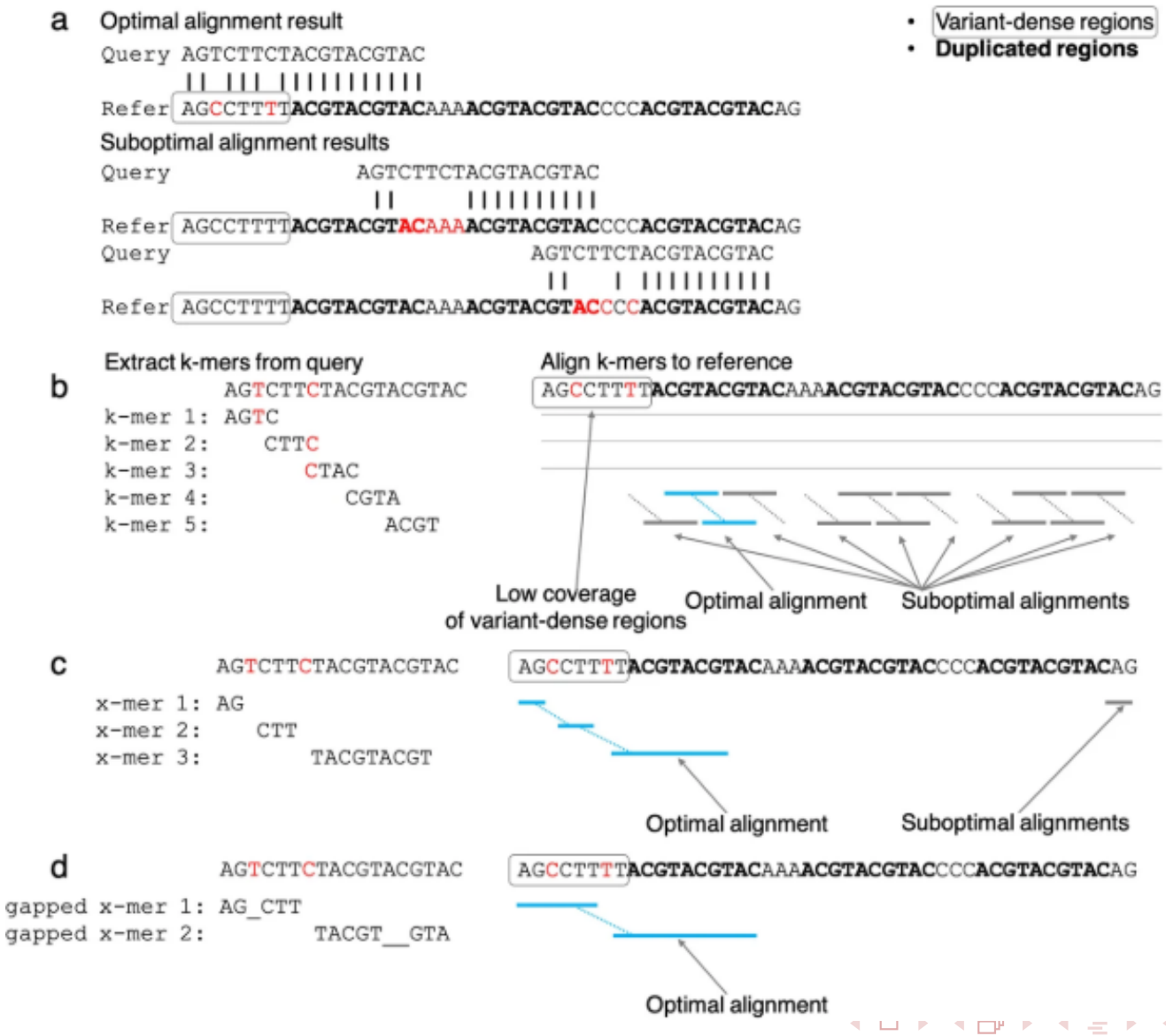
Checking for indels

The last step is to identify the optimal alignment overlapping a given offset via dynamic programming as follows.

1. We first generate a lazy Needleman-Wunsch grid, which can determine the penalty of an individual alignment. This allows each box in this grid to be aware of the minimum penalty required to reach that box from the start of the grid.
2. We search this Needleman-Wunsch grid for the path having minimum penalty. This is done using the A* search algorithm, plus pruning any branches that have exceeded the maximum indel extension length calculated previously for this offset.
3. When the A* search completes, it produces a path that can be transformed into an alignment, and the alignment process for that particular query offset is done.

Note that if the maximum possible indel length is 0, an ungapped alignment is reported.

An illustrative example



References

1. Camacho, C., Coulouris, G., Avagyan, V., Ma, N., Papadopoulos, J., Bealer, K. and Madden, T. L., BLAST+: architecture and applications. BMC Bioinformatics, 10(1):421, 2009.
2. Li, H., Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics, 34(18):3094-3100, 2018.
3. Wood, D. E., Lu, J., Langmead B., Improved metagenomic analysis with Kraken 2. Genome Biology, 20:257, 2019.
4. Gaston, J. M., Alm, E. J. and Zhang, A. N., X-Mapper: fast and accurate sequence alignment via gapped x-mers. Genome biology, 26(1):15, 2025.

All code is publicly available on:
`https://github.com/mathjeff/Mapper`