Indian Institute of Technology, Gandhinagar

EE 299 - PROJECT COURSE REPORT

SOFTWARE ALGORITHM AND HARDWARE ACCELERATION FOR GENOMICS ANALYSIS

---

21 February, 2025

AUTHOR:

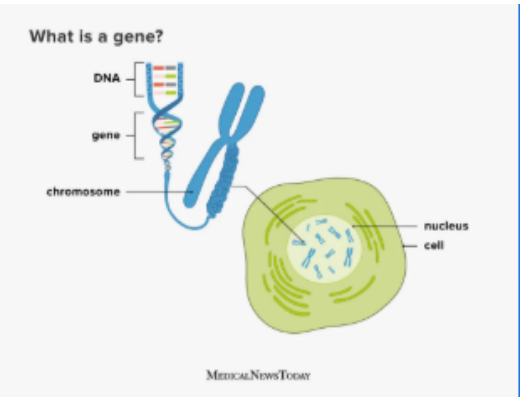
Seemanshi Mall Sia Jariwala Ginisha Garg Anuja Chaudhari

Under the guidance of:
Prof. Joycee Mekie

INTRODUCTION

Genome:

It is the complete set of an organism's genetic material, including all its DNA or RNA, containing the instructions for growth, development, and reproduction.



Genome Analysis:

Genome analysis is the study of the sequence of nucleotide bases(A, G, C, T/U) in an organism’s genome. Since these sequences vary from person to person, they carry the genetic instructions for traits, behaviors, and biological functions.

Need for Intelligent Genome Analysis:

By studying genomes, scientists can understand how genes work, what causes diseases, and how traits are passed down through generations. This research is also crucial for medical advancements and helps in developing personalized treatments for individuals. However, traditional genome analysis methods have several limitations, such as slow processing time, privacy risks, and high resource demands.

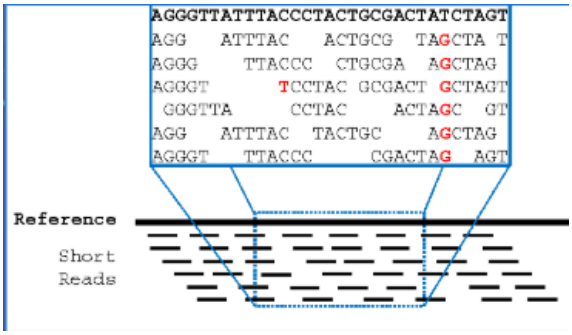
Hence, we need an intelligent genome analysis that has the following features:

1) **Portability:** Making genome analysis accessible on various platforms.
2) **Faster Results:** Reducing the time required for processing.
3) **Scalability:** Enabling efficient analysis of larger datasets.
4) **Precision and Accuracy:** Improving the reliability of genome interpretation.
5) **Efficient Architecture:** Uses computational resources more effectively for better performance.
6) **Data Privacy:** Enhancing security measures to protect genetic information.

SEQUENCING OF HUMAN GENOME

Read:

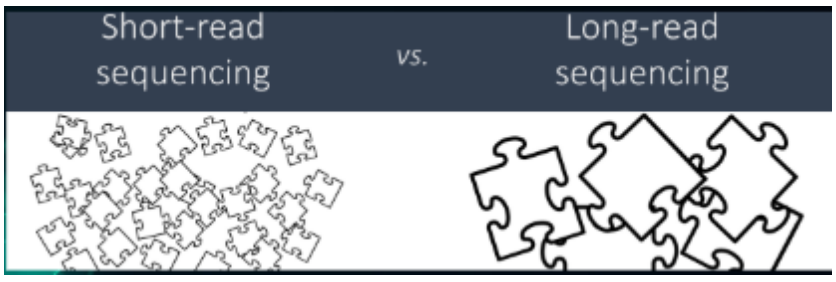
A read is a short fragment of DNA that is generated by sequencing machines. Since human genomes contain billions of base pairs, it is impossible to sequence the entire genome in one go. So, instead, sequencing machines break the genome into small, readable fragments called reads, which are later assembled to reconstruct the original genome.



Technologies Used to Obtain Reads:

Sanger Sequencing	**Illumina Sequencing**	**Nanopore Sequencing**
Very high accuracy (99%)	98% accuracy	92-97% accuracy
$1000 USD per million base	$0.05-0.15 USD per million base	Lower than Illumina
Long read, low throughput	Short read, High throughput	Long read, low throughput
Used when small parts of the genome are targeted	Deals with huge amounts of data	Used because of portability and versatility

The technologies mentioned above primarily produce two types of DNA reads: **short-reads** and **long-reads**. Each method has its own advantages and limitations, making them suitable for different applications.



Short-Read Sequencing

Short-read sequencing involves breaking DNA into small fragments, typically between 50 to 300 base pairs long. This method is widely used due to its high accuracy and scalability. The most common technique for short-read sequencing is Illumina sequencing, which is known for its precision and reliability.

Short reads are useful for:

- **Resequencing:** Identifying variations in DNA sequences.
- **RNA sequencing:** Studying gene expression.
- **Variant detection:** Detecting mutations in DNA.

Challenges with Short-Read Sequencing:

Difficulty in Analyzing Repetitive Regions:

- Short reads may look identical, making it hard to determine their exact position in the genome.
- This can lead to incorrect genome assembly, gaps, or missing sequences in the reconstructed genome.
- Example: If a sequence ATGCC appears 100 times in the genome. Ideally, all 100 copies should be retained, but short-read sequencing struggles to determine the true count.
- Possible issues:
    - Collapsing repeats: The assembler may merge all repeats into one, losing the real count.
    - Overestimating repeats: Short reads may misalign, falsely increasing the repeat count.

Long-Read Sequencing

Long-read sequencing produces much longer DNA fragments, typically more than 10,000 base pairs in length. Common techniques for long-read sequencing include Nanopore sequencing and PacBio sequencing.

Long reads are useful for:

- **De novo assembly:** Constructing entire genomes from scratch without a reference genome as it provides better continuity.
- **Structural variant detection:** Identifying large genetic variations that are difficult to detect with short reads.

Challenges with Long-Read Sequencing:

- Long-read sequencing often has higher error rates than short-read methods.

- The process relies on detecting electrical or optical signals, which can introduce noise and misinterpretation of bases, which can generate errors.
- If a single character is misread, it can cause the entire read to be misaligned or contain sequencing errors.

GENOME ASSEMBLY WITH REFERENCE GENOME

Read Mapping:

Read mapping is the process of matching sequencing reads to a genome. This can be done by aligning them to a known reference genome or by piecing together the genome directly from the reads without using a reference.

Read Mapping - Naive Approach

The naive approach for read mapping is a brute-force method that searches for a given read in a reference genome by checking every possible position for a match. It does this by sliding(traversing) the read along the genome, one position at a time, and comparing characters one by one. If a mismatch occurs, the algorithm moves to the next position and starts the comparison again. If a full match is found, the position is recorded. We continue until the read has been compared with every possible position in the genome.

Example:

Reference Genome: GCAGCTCA

Read: GCTC

Step 1:GCAGCTCA → GC matched, but the third character did not match.
GCTC

Step 2:GCAGCTCA → The first character did not match. Move forward.
GCTC

Step 3: GCAGCTCA → The first character did not match. Move forward.
GCTC

Step 4: GCAGCTCA → Match found.
GCTC

Step 5: GCAGCTCA → The first character did not match.
GCTC

Whenever a mismatch occurs, the read shifts one position to the right, as seen in **Steps 2** and **3**. In **Step 1**, when **"GC"** matched but the third character did not, the next comparison did not restart from **"A"** but instead continued from **"C"** which was the next character in the sequence. This ensures that every possible match is checked without skipping any potential alignments. Even when a full match is found, the read still shifts one position forward to check for additional occurrences. The key point here is that while checking character-by-character, the pointer moves forward sequentially, but once a mismatch occurs, the read starts from the next character in the reference genome rather than continuing from the position where the mismatch happened. This brute-force approach guarantees that all positions are examined, making it exhaustive but computationally expensive.

Time Complexity: **$O(m^2kn)$**, where m = read length, k = number of reads, n = reference genome length.

Calculation of Time Complexity:

- Comparing one read at one position takes O(m) time as it is required to check up to m characters in the worst case.
- Since all positions in the reference genome are checked, this happens n times, giving a total complexity of O(m × n).
- If there are k reads, the process repeats k times, making it O(m × n × k).

Worst-case scenario: Imagine a read and reference where almost every character matches except for the last one. In this case, the algorithm will compare m characters before detecting the mismatch. After a mismatch, it shifts the read by one position and starts comparing it all over again. This means for each position, we may perform up to m comparisons, leading to an extra O(m) factor.

This results in a final worst-case complexity of **$O(m^2kn)$**.

Thus, in cases where mismatches occur late, the naive approach becomes extremely slow, especially for long reads(because m is more and the time complexity is quadratic of m), making it inefficient for large-scale genome sequencing.

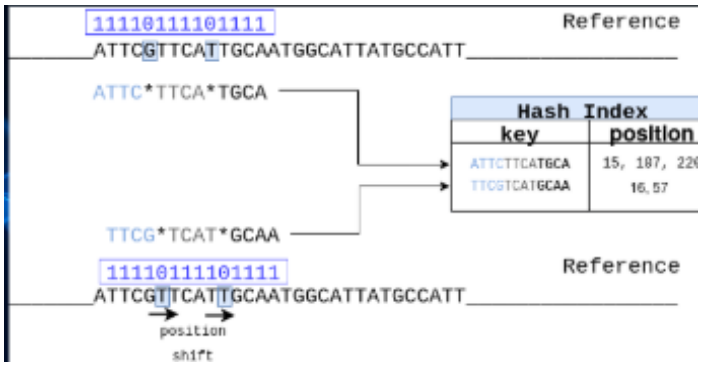
Seeds:

Seeds are short, fixed-length sections of DNA or RNA that help us to identify matching regions within a genome or dataset. They narrow down the search space, allowing seeds to act as a starting point for alignment instead of aligning an entire read directly, making genome analysis more efficient.

Steps of Read Mapping:

1. Indexing

Indexing is a preprocessing step that divides the reference genome into k-mers (substrings of length k) to enable fast and efficient searching during read mapping. Instead of scanning the entire genome, indexing allows sequencing reads to be quickly matched to potential locations. These k-mers are stored in data structures such as hash tables, suffix arrays, or Burrows-Wheeler Transform (BWT) indexes, making genome searches much faster.



Indexing Methods in Read Mapping:

1) **Hashing:**

   A hash table is a data structure used to store and retrieve data in constant time O(1). In DNA sequencing, hash tables store DNA sequences (seeds) along with their positions in the reference genome.

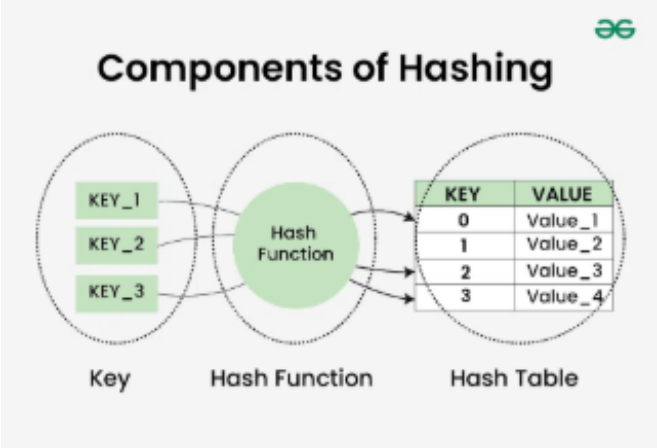
   How Hashing Works in Read Mapping:

- The reference genome is scanned to extract all possible seeds (small DNA sequences).
- These seeds are stored in a hash table along with their positions in the genome.

- When aligning a sequencing read, the algorithm extracts seeds from the read and checks the hash table to find potential matching locations.

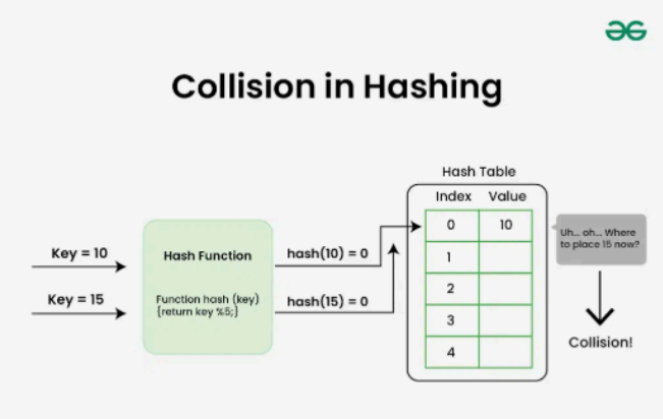
Components of Hashing:

- **Search Key:** The input (e.g., a DNA sequence) used to determine storage location.
- **Hash Function:** A function that maps a key to an index in the hash table. Common functions include modular hashing (K mod n), mid-square method, and folding method.
- **Hash Table:** An array-like structure that stores values based on the hash function. It also ensures that each value has a unique location.



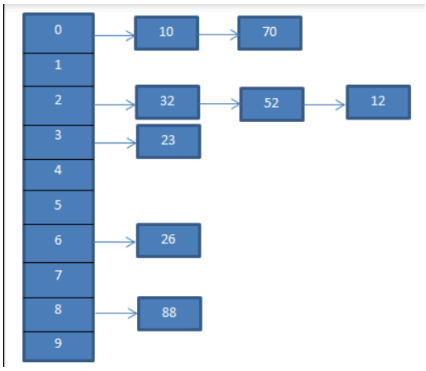
Collision Handling in Hash Tables

Collisions occur when different keys map to the same index. Handling collisions is important for efficient data retrieval. Several methods exist to handle collisions, as follows:



1. **Separate Chaining:**
    - Each index in the hash table points to a linked list storing multiple keys.
    - It is simple to implement and supports dynamic resizing of lists.

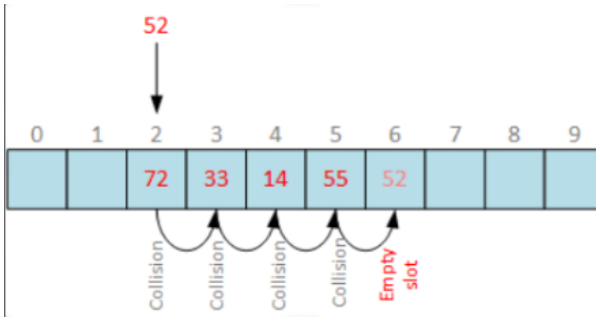
- It helps prevent collisions but slows down searches if the list grows too long.



2. **Linear Probing:**
   - When a collision occurs, the algorithm checks the next available index.

Clustering in Linear Probing:

Clustering happens in open addressing methods like linear probing, where multiple keys get stored close to each other in the hash table due to collisions. This creates long sequences of occupied slots, making future insertions and searches slower.

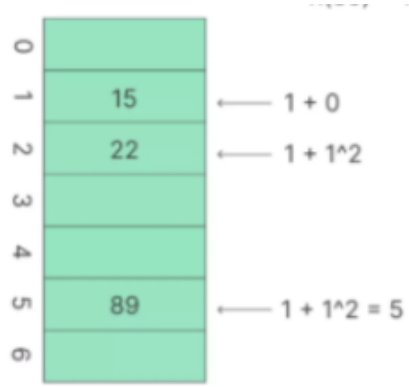


In the given example, the key **52** was initially mapped to **index 2**, but since that position was already occupied, linear probing checked the next available slots sequentially (3, 4, 5) until it found an empty slot at index 6. As a result, any future insertions in the same clustered area will also have to check multiple positions, further reducing efficiency. To minimize clustering, alternative collision resolution techniques like quadratic probing or double hashing can be used, as they distribute values more evenly across the table, preventing long chains of occupied slots.

3. **Quadratic Probing:**
    - Instead of moving to the next slot, it jumps by quadratic steps **$h(x) + i^2$**, where i is the probe number, starting from 1.
    - Reduces clustering but may leave some slots unused.

h(89) = 1

In the given image, the hash function for **89** produces an index of **1**, but since index **1** is occupied by **15**, a collision occurs. Using quadratic probing, the next index is calculated as **$h(89) + 1^2 = 2$**, but index **2** is also occupied by **22**. The next probe uses **$h(89)+ 2^2 = 5$**, which is available, so **89** is placed in index **5**. Quadratic probing helps in reducing clustering compared to linear probing and avoids placing colliding elements in consecutive slots, but it requires a prime-sized table for efficient probing.



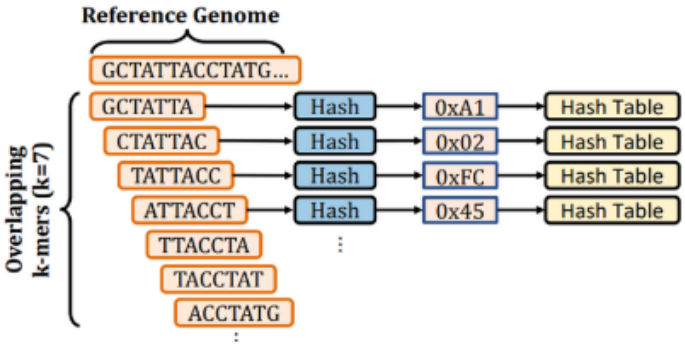
4. **Double Hashing:**
    - Uses two hash functions to determine the next probe location (index = **$h1(x) + i * h2(x) \bmod table\_size$**, where table_size is the total number of indexes present in the hash table.
    - Reduces clustering but is rarely used due to complexity.

Seeding Techniques:

Seeding techniques help efficiently find matching regions between a sequencing read and a reference genome. Instead of comparing entire sequences, seeds act as starting points for alignment, making the process much faster.

Types of Seeding Techniques are:

1) **<u>All K-Mers:</u>**
   A K-mer is a substring of length K derived from a longer sequence (such as a DNA, RNA, or protein sequence). This technique extracts all overlapping k-mers and stores them in a hash table.

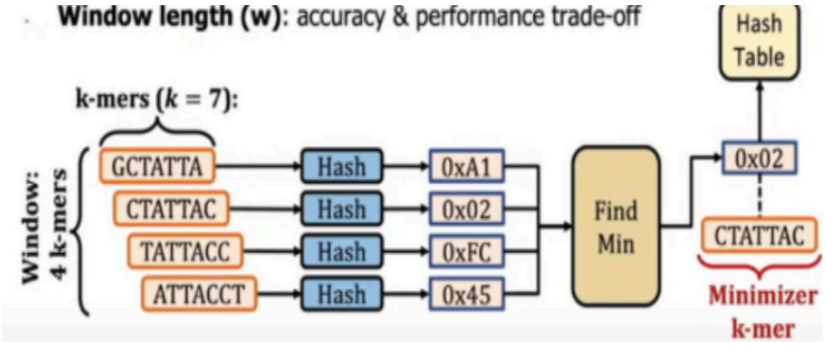


One of the main advantages of using this method is that it finds all possible matching regions, leading to higher alignment accuracy. It is particularly effective when the reference genome has low mutation rates, as it ensures that every potential match is considered. However, a major drawback is that it requires a lot of memory since it stores all possible k-mers. Additionally, the search process becomes slower because too many k-mers need to be compared, increasing computational time and complexity.

2) Minimisers:

Minimizers are a smart way to reduce the number of k-mers stored while maintaining accurate sequence alignment. Instead of extracting and storing all possible k-mers, minimizers select only a few representative k-mers from a sequence, making the process faster and more memory-efficient. This method is particularly useful in large-scale sequencing, where storing and comparing all k-mers would be computationally expensive.

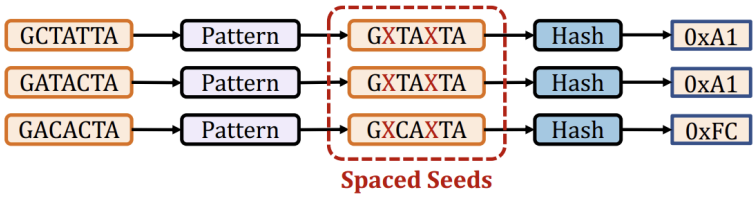
The minimizer selection process involves dividing the DNA sequence into overlapping windows and extracting all k-mers from each window. Among these k-mers, the smallest (lexicographically or numerically smallest) k-mer is chosen as the minimizer. The window then slides forward by one base, and the process repeats. Instead of storing every k-mer, only a subset of them is retained, reducing storage requirements.



In the above example, k = 7 (k-mer size) and w = 4 (window size), so 4 overlapping k-mers are taken from a sequence. Each k-mer is hashed to get a hash value. The smallest hash value among the k-mers in each window is selected as the minimizer.

Minimizers help make sequence alignment faster and use less memory, making them efficient for handling large datasets. They also reduce computational costs since only a few k-mers are stored instead of all possible ones. However, there are some drawbacks. Because only a subset of k-mers is selected, some important k-mers might be missed, especially if a mutation occurs in a minimizer. Additionally, in highly repetitive regions, minimizers may fail to capture enough variation, leading to less accurate alignment. Despite these limitations, minimizers are widely used in modern genome sequencing because they speed up alignment and reduce computational costs, making them ideal for large-scale sequencing projects

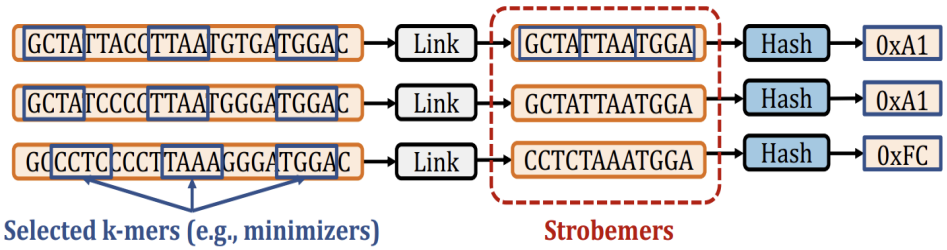
3) Spaced seeds



This technique adds some don't care characters at specific positions in the seed. This allows mismatches between the seeds of the reference genome and sequence seed. This gives the same hash value even if there are mismatches, increasing accuracy.

The disadvantage here is that the positions of don't care characters are random, which reduces flexibility as we don't know where precisely the mismatch is.

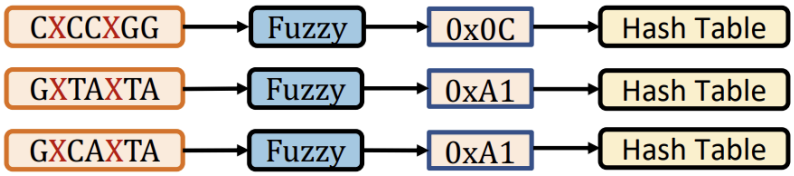
4) Strobemers



This technique selects specific k-mers, like minimizers, and then calculates their hash values. This allows for handling insertions, deletions, and mismatches in the positions of the seeds that are not selected.

The disadvantage is that it again reduces flexibility, as this will only give the same hash values when the insertions, deletions, and mismatches are not in the selected k-mers. The k-mers should match even in order for the same hash values.

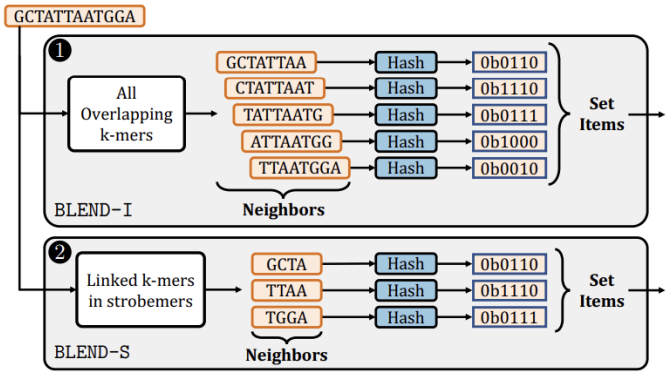
5) Fuzzy seeds



This technique uses don't care characters at certain positions but still gives the same hash values even if there are some mismatches in the rest of the seed. So, it assigns identical hash values to highly similar seeds. This reduces false negatives (failing to find matches due to slight variations). This improves space efficiency because we don't need multiple hash functions for similar seeds. This is especially helpful in handling sequencing errors and genetic variation.

This is implemented using a method called Blend, which uses the SimHash technique to get the same hash value for highly similar seeds.

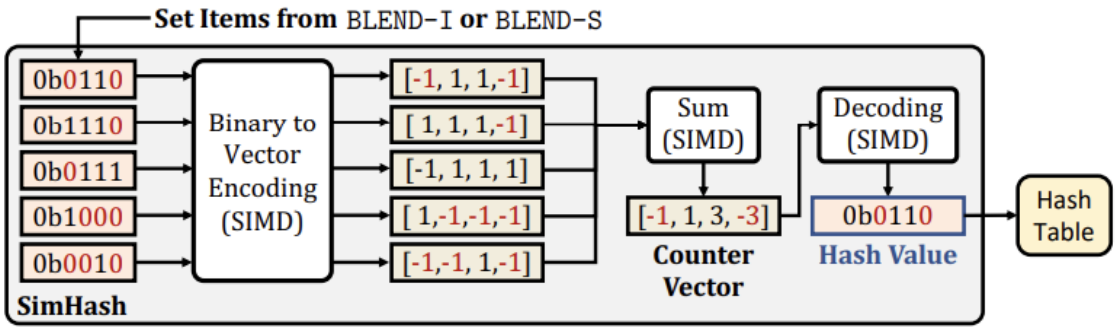
Blend using the SimHash technique



There are two mechanisms used for determining the set items of the input sequence:

1) BLEND-I uses the hash values of all the overlapping k-mers of an input sequence as the set items.

2) BLEND-S uses the hash values of only the k-mers selected by the strobemer seeding mechanism.



BLEND employs the SimHash technique in three steps: 1) encoding the set items as vectors, 2) performing vector additions, and 3) decoding the vector to generate the hash value for the set that BLEND-I or BLEND-S determines, as shown in Figure.

1) The set items are the hash values represented in their binary form. Binary to Vector Encoding converts these set items to their corresponding vector representations. For each hash value in set items, the resulting vector includes 1 for the positions where the corresponding bit of a hash value is 1 and –1 for the positions where the bit is 0.
2) Sum performs the vector additions and stores the result in a separate vector called the counter vector.
3) The decoding operation is a simple conditional operation where each bit of the final hash value is determined based on its corresponding value at the same position in the counter vector. BLEND assigns the bit either 1 if the value at the corresponding position of the counter vector is greater than 0 or 0 if otherwise.

The vector addition operation aims to determine the bit positions where the number of 1 bits exceeds the number of 0 bits among the set items, which we call determining the majority of bits. The key insight in determining these majority bits is that highly similar sets are likely to result in similar majority results because a few differences between two similar sets are unlikely to change the majority bits at each position, given that there is a sufficiently large number of items involved in this majority calculation.

2)Suffix Trees in Genome Analysis for Reference Genomes

1. Introduction

A suffix tree is a tree-like data structure that represents all suffixes of a given string. It is widely used in genome analysis for efficient sequence matching, indexing, and searching. Unlike basic hashing techniques, suffix trees provide a structured way to handle variable-length sequences and allow for efficient substring queries.

2. Advantages of Using Suffix Trees

Traditional hashing methods require predefined hash functions and struggle with variable-length sequence searches. In contrast, suffix trees offer:

- **Efficient search operations:** Enables fast pattern matching in **O(m)** time, where **m** is the query length.
- **Compact storage:** Avoids redundant storage of common prefixes, reducing space complexity.
- **Flexible matching:** Allows for approximate matching and pattern recognition in genomic sequences.

3. Suffix Tree Structure and Functionality

A suffix tree is structured such that:

- Each edge represents a substring of the reference genome.
- Each leaf node stores the starting index of a suffix in the genome.
- Internal nodes represent common prefixes among different suffixes.

4. Suffix Tree Construction

To construct a suffix tree for a given reference genome T, follow these steps:

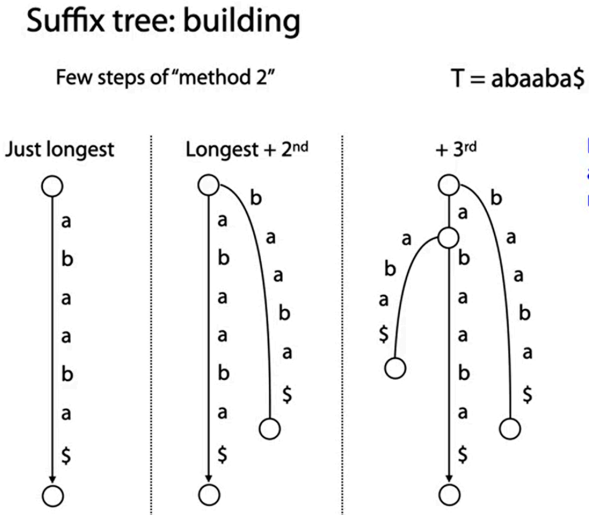
4.1 Step-by-Step Construction Process

Using the reference sequence T = "abaaba$", where "$" is a termination marker:

1. Insert the longest suffix: "abaaba$"
2. Insert the second longest suffix: "baaba$"
3. Insert the third longest suffix: "aaba$"
4. Insert the fourth longest suffix: "aba$"

5. Insert the fifth longest suffix: "ba$"
6. Insert the sixth longest suffix: "a$"
7. Insert the shortest suffix: "$"

Each insertion extends the tree while sharing common prefixes, thereby optimizing space usage. The “$” sign is used to represent unique suffixes, such that no suffix is a prefix of another suffix.



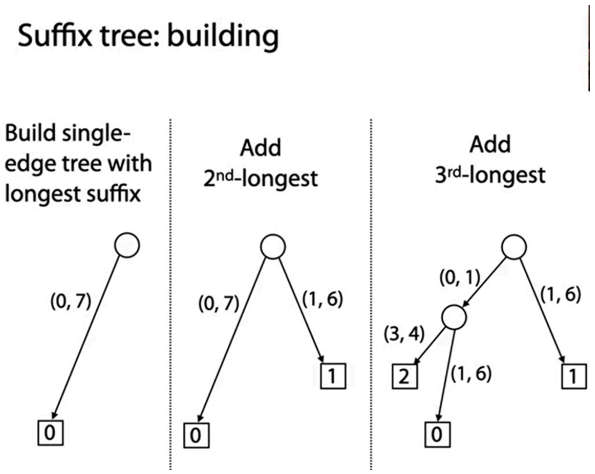
4.2 Optimized Node Storage

Instead of storing substrings explicitly, we store:

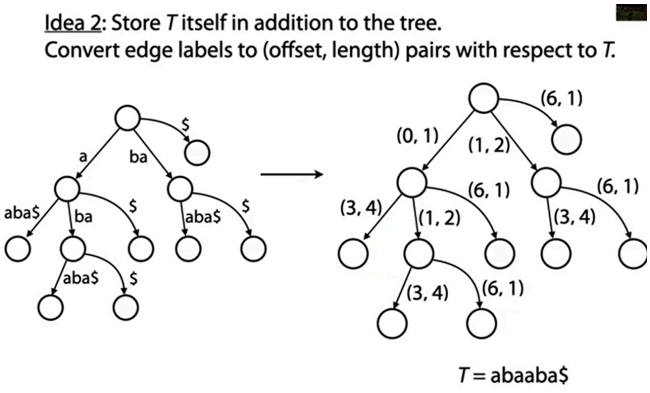
- The starting index of each substring
- Length of the substring

  Example with T = "abaaba$"

- Node a(0,1)
  - Represents the substring "a" starting at index 0.
  - Instead of storing "a," we store:
    - Starting index = 0
    - Length = 1
- Node aba(3,4)
  - Represents the substring "aba$" starting at index 3.
  - Instead of storing "aba$," we store:
    - Starting index = 3
    - Length = 4



This reduces space complexity from $O(n^2)$ to $O(n)$ while keeping construction time at $O(n^2)$.



The location of a suffix stored at the leaf node can be calculated using:

Location=Total length−(sum of segment lengths)

For example, if the total length of the sequence is 7, and the traversal path has segment lengths 1 and 3, then the final location is:

$$7-(1+3)=3$$

This helps efficiently store and retrieve suffix locations in the suffix tree.

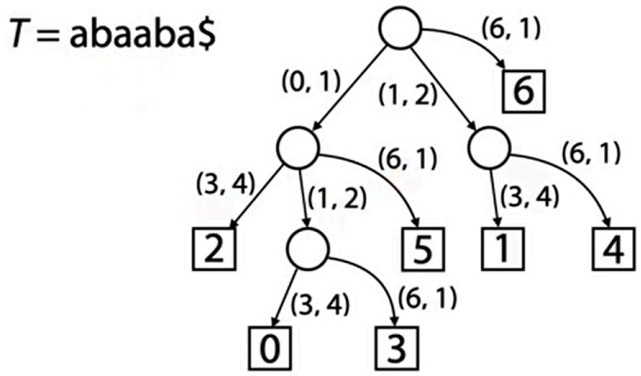
Example with T = "abaaba$"

To find the location of the suffix "aaba$":

- The first node represents "a" (0,1).
- The next node represents "aba$" (3,4).
- The final location is calculated as:

  $7-(1+4)=2$

This method ensures efficient suffix storage and lookup within the tree.



5. Traversal and Searching in Suffix Trees

5.1 Step-by-Step Traversal Process

To search for a query sequence in the suffix tree:

1) Start at the root.
2) Follow edges based on query characters.
3) If a full match is found, The search may end on an edge that has no child nodes, meaning the query sequence corresponds exactly to a suffix of the reference. In this case, the stored suffix location is returned directly.
4) If a partial match is found, The node has multiple outgoing edges. The number of child nodes represents the number of occurrences of the query in the reference.
5) Traverse downward to collect locations

.

5.2 Example Search Query

- For the query "aba" in the suffix tree of "abaaba$":
- Start at the root and follow "a" → "b" → "a".
- The node reached has multiple children, indicating multiple occurrences.
- Traversing downward gives the exact positions of "aba" in the reference genome.
- In the above example, we follow the path from "a" to "ba," achieving a full match. However, since "ba" has two child nodes, this indicates that the query appears in

two different locations. Traversing down these child nodes will provide the exact positions of each occurrence.

6. Time and Space Complexity

- Suffix Tree Construction: Time Complexity → O(n^2), Space Complexity→O(n)
- Search Operation: Time Complexity → O(m+k), where k is the number of occurrences of the query in the reference sequence.
- Getting the Locations: Time Complexity → O(k)
- Overall Complexity: Time Complexity → O(m+k), Space Complexity → O(n)

2. Global Positioning

Purpose:

- Identify probable regions in the reference genome where sequencing reads might align.
- Reduce computational burden by limiting alignment to specific areas instead of scanning the entire genome.

Key Steps:

1. Index Lookup:
    - Use pre-built genome indexes (e.g., hash tables or suffix trees like the Burrows-Wheeler Transform).
    - Retrieve positions in the genome where the extracted seeds might occur.
2. Location Filtering:
    - Focus on regions near identified seed locations to prioritize alignment.
    - Eliminate irrelevant genome regions to save computation.

Factors Affecting Accuracy:

- Seed Length: Balances the number of potential locations and the reliability of matches.
    - Short seeds:
        - Provide many potential matches but introduce noise (false positives).
    - Long seeds:
        - Reduce noise but risk missing true matches.

3. Pairwise Alignment

Pairwise alignment determines the exact similarity or the score between sequencing reads and their candidate locations in the reference genome.

Steps involved:

- Compare each read with its potential genome segments identified in the global positioning step.

- This is done using alignment algorithms which determine the best match based on edit distance or scoring functions. There are two types of algorithms :
    - Dynamic Programming (DP) Algorithms
        - Dynamic programming is a technique used to solve optimization problems by breaking them down into smaller subproblems and storing their solutions to avoid redundant calculations.
        - Examples: Needleman-Wunsch for global alignment and Smith-Waterman for local alignment.
        - It provides optimal accuracy but has a high computational cost.
    - Non-DP Algorithms:
        - Non-dynamic programming refers to methods that do not rely on breaking problems into overlapping subproblems and storing intermediate results.
        - Faster but less accurate.

- Outputs:
    - Alignment score: Measures match quality using user-defined scoring functions.
    - Edit Distance: Counts the minimal number of edits (insertions, deletions, or substitutions) needed to align sequences.

BWT (Burrows-Wheeler Transform)

Genomes typically don't have long runs of identical characters but have many repetitive patterns scattered throughout.

Challenges with sorting for Compression

- Sorting the text lexicographically can group repeats into runs, but there's no way to invert the sorting to recover the original sequence.
- The compression must be reversible to preserve genomic data integrity.
- The Burrows-Wheeler Transform (BWT) solves this by creating a reversible transformation that clusters similar characters into runs, enabling efficient compression and subsequent querying.

BWT with an example -

Let the input be **"BANANA"**.

1) Add '$' at the end of the read and create cyclic rotations.

    BANANA$
    $BANANA
    A$BANAN
    NA$BANA
    ANA$BAN
    NANA$BA
    ANANA$B

2) Sort the rotations lexicographically.

   $BANANA
   A$BANAN
   ANA$BAN
   ANANA$B
   BANANA$
   NA$BANA
   NANA$BA

3) Take the last character of each sorted rotation.
   BWT Output: ANNB$AA

Why do we use BWT?

- Reduces storage requirements using clustering and run-length encoding (RLE).
- Fast pattern matching for large-scale sequencing data.

Why do we add $ at the end of the read in the first step?

Here, "$" allows for efficient reconstruction of the original string during the inverse BWT. Since "$" is unique and marks the end of the original string, it helps us determine the starting point for reconstructing the original sequence.

BWT is implemented on reads of the reference genome. Now, to find a sequence read in the read of the reference genome, we first calculate the FM Index and suffix array of the reference genome and then use a backward search algorithm to find the sequence read.

FM Index

Example -
Let the read be T=**”abracadabra$”**.

```
$abracadabra
a$abracadabr
abra$abracad
abracadabra$
acadabra$abr
adabra$abrac
bra$abracada
bracadabra$a
cadabra$abra
dabra$abraca
ra$abracadab
racadabra$ab
```

The figure shows the sorted rotations. The BWT output is **“ard$rcaaaabb.”**

We compute two tables using this BWT output.

C[c] of "ard$rcaaaabb"

c	$	a	b	c	d	r
C[c]	0	1	6	8	9	10

C Table: The C-table stores the cumulative frequency of lexicographically smaller characters that appear before a given character.
Here, the characters smaller than ‘$’ are zero, characters smaller than a is one(‘$’), characters smaller than b are six(one($) + five(a)), and so on.

Occ(c, k) of "ard$rcaaaabb"

	a	r	d	$	r	c	a	a	a	a	b	b
	1	2	3	4	5	6	7	8	9	10	11	12
$	0	0	0	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	2	3	4	5	5	5
b	0	0	0	0	0	0	0	0	0	0	1	2
c	0	0	0	0	0	1	1	1	1	1	1	1
d	0	0	1	1	1	1	1	1	1	1	1	1
r	0	1	1	1	2	2	2	2	2	2	2	2

Occurrence Table(Occ): The OCC table stores the cumulative count of each character up to a given position in the BWT string.
Here, at the first position, ‘a’ is present, so the count of ‘a’ goes to one, then at the second position, the count of ‘r’ goes to one, and so on.

FM Index Backward Search

Let our search pattern (sequence read) be **'bra.'**
For every character in the search pattern starting from the end, we will find a range where the occurrence of that string lies.
The formula for calculating the range where s is the starting index and e is the end is:

$$s' = C[P[i]] + rank(s-1, P[i]) + 1$$
$$e' = C[P[i]] + rank(e, P[i])$$

where,
P[i]: the character being evaluated
rank: Occ table
s’,e’: new start and end index

- We start with P[i]=’a’ , s=1 , e=12

s'= 1 + 0 + 1 =2
e' = 1 + 5 = 6
Thus, all occurrences of 'a' lies in [2,6]

- Now, P[i]='r' , s=2 , e=6
  s'= 10 + 0 + 1 =11
  e' = 10 + 2 = 12
  Thus, all occurrences of 'ra' lie in [11,12]
- Finally, P[i]='b' , s=11 , e=12
  s'= 6 + 0 + 1 =7
  e' = 6 + 2 = 8
  Thus, all occurrences of 'bra' lie in [7,8]

The range [7,8] in the FM-index corresponds to the 7th and 8th positions in the suffix array. This indicates that the pattern 'bra' occurs at these positions in the suffix array of the reference genome.

Mapping to the reference genome

Suffix Index	Suffix	Sorted Suffix	Original index
1	$abracadabra	$abracadabra	0
2	abracadabra	a	11
3	bracadabra	abra	8
4	racadabra	abracadabra	1
5	acadabra	acadabra	4
6	cadabra	adabra	6
7	adabra	bra	9
8	dabra	bracadabra	2
9	abra	cadabra	5
10	bra	dabra	7
11	ra	ra	10
12	a	racadabra	3

The suffix array of the above example is: [0, 11, 8, 1, 4, 6, 9, 2, 5, 7, 10, 3]

As we can see, the 7th and 8th positions of the suffix array, which show the 9th and 2nd positions of the original index, have the occurrence of ‘bra.’

Thus, the final range [s,e] represents the indices in the suffix array corresponding to where the read matches in the BWT.

- Once the range [s,e] is determined:
    - If the range size is nonzero (e−s>0), the indices in this range are looked up in the suffix array.
    - The suffix array maps these indices to starting positions in the reference genome.
- If no valid range is found (e−s=0), the sequencing read is absent in the reference genome.

Advantages of BWT+FM-Index:

- Computational Efficiency: Querying is extremely fast, with a time complexity of O(m), where m is the read length, which is because of the back search.
- Pre-processing reference genome indexing using BWT, suffix array, C, and occ table is a one-time operation.
- Memory Efficiency: BWT compresses repetitive genomes into long runs, making it suitable for large genomes.

Types of Dynamic Programming Algorithms

Global Alignment	Local Alignment
Global alignment attempts to align two sequences from start to end, considering the entire length of both sequences. It is best suited for sequences of similar lengths with a high degree of similarity. The **Needleman-Wunsch algorithm** is commonly used for global alignment. Example: Sequence 1: GATTACA Sequence 2: GATACA Alignment: GATTACA GAT_ACA	Local alignment identifies the most similar subsequence between two sequences, focusing on highly conserved regions rather than aligning the full length. It is useful when sequences differ significantly in length or contain similarities in certain regions. The **Smith-Waterman algorithm** is commonly used for local alignment. Example: Sequence 1: ATGCT Sequence 2: AGCT Alignment: ATGCT AGCT

Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm is a global sequence alignment algorithm used to find the best possible alignment between two biological sequences (such as DNA). It ensures an optimal alignment by considering the entire length of both sequences, even if they have mismatches or gaps.

Step 1: Setting a Scoring Scheme

Before filling the matrix, we must define a scoring scheme to determine how matches, mismatches, and gaps influence the alignment. The scoring scheme consists of:

1. **Match Score (+M)**→ A reward for aligning identical characters.
2. **Mismatch Score (−S)** → A penalty for aligning different characters.
3. **Gap Penalty (−G)** → A penalty for inserting a gap in either sequence.

Step 2: Initialization

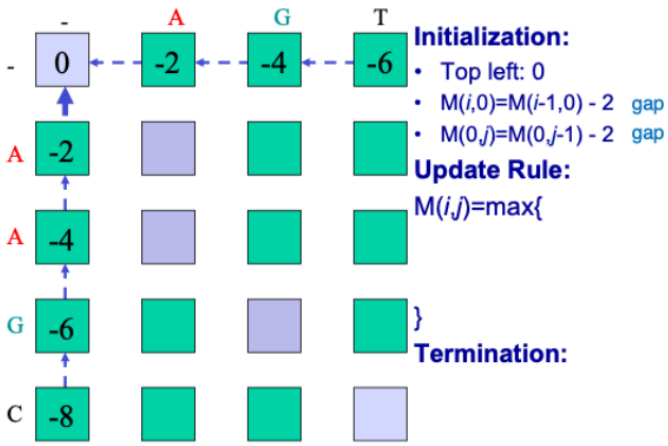
1. Construct a matrix of size (m+1)×(n+1), where m and n are the lengths of the sequences.
2. Place one sequence along the top margin and the other along the left margin.
3. Initialize the first row and column by incrementally adding gap penalties to represent gaps at the start of the alignment.
4. The top-left cell is set to zero if using a standard approach or a negative gap penalty if required.

Assume we want to align two sequences, S1 and S2, where

S1 = AGT

S2 = AAGC

Considering the gap penalty -2, the match score +1, and the mismatch score -1.

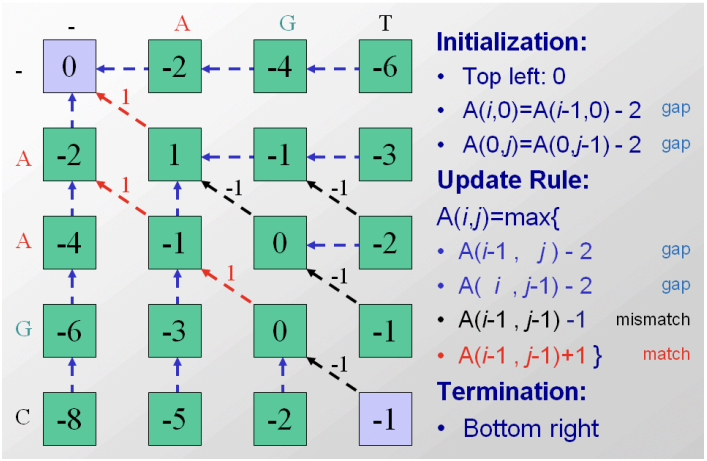


Step 3: Matrix Filling with Multiple Pointers

For each cell **F(i, j)**, compute three possible scores depending on the scores of three adjacent matrix cells (specifically the entry above, the one diagonally up and to the left, and the one to the left).:

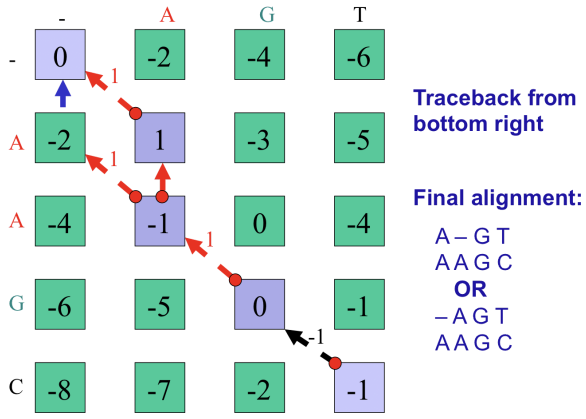
- **F(i-1, j-1) + match/mismatch score**
- **F(i, j-1) + gap penalty**
- **F(i-1, j) + gap penalty**

The maximum score of these three possible tracebacks is assigned to the cell **F(i, j)**, and the corresponding pointer is also stored. If two adjacent matrix cells give the same maximum value, then multiple pointers to both of these adjacent cells will be stored.



Step 4: Traceback and constructing the alignment

1. Begin traceback from the bottom-right corner of the matrix.
2. Follow the stored pointers backward to the top-left corner.
3. If a cell has multiple pointers, it creates a branching point, meaning that the algorithm will explore multiple possible alignments.
4. The traceback process continues until the starting cell (0,0) is reached
5. Since the traceback is performed from end to start, the recorded alignments need to be reversed to obtain the correct sequence order.
6. If, during traceback, we go diagonally from a larger value to a lower value, then it's a match, and if it goes from a smaller value to a larger value, then it's a mismatch.
7. If moving upwards (from F(i,j)to F(i−1,j)), then there is a gap in seq 1.
8. If moving leftwards (from F(i,j)to F(i,j−1)), then there is a gap in seq 2.
9. If multiple paths were followed during traceback, multiple valid alignments are generated.



Time Complexity:

Initialization: Assigning scores to the first row and column takes O(M + N).

Matrix Filling: Computing each cell score using the three neighbors requires O(MN).

Traceback: Backtracks from the bottom-right, moving at most O(M + N) steps.

Total Time Complexity: Summing up all steps gives **O(MN)**.

Smith-Waterman Algorithm

The Smith-Waterman algorithm is a local sequence alignment algorithm designed to find the most similar subregions between two sequences. Unlike Needleman-Wunsch, it does not force alignment across the entire sequence but rather finds the best subsequence matches.

Steps:

1. **Scoring System:**
   - **Match (+M)** → Reward for identical characters.
   - **Mismatch (-S)** → Penalty for different characters.
   - **Gap (-G)** → Penalty for insertions/deletions.

2. **Initialization:**
   - Create a matrix (m+1)×(n+1).
   - Set the first row and column to 0 (unlike Needleman-Wunsch).

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
A	0											
C	0											
T	0											
T	0											
A	0											
C	0											

3. **Matrix Filling:**
   - Compute each cell from diagonal, left, and above values similar to Needleman Wunsch

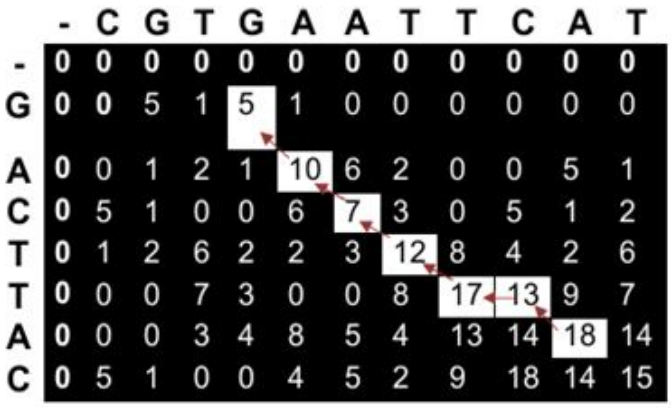
     $$F(i,j)=\max(0,\ F(i-1,j-1)+\text{match(+M)/mismatch(-S)},\ F(i,j-1)-G,\ F(i-1,j)-G)$$

   - Set negative values to 0 to ensure local alignment.

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	14	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15

4. **Traceback:**

- Start from the highest-scoring cell and move along the best path.
- Stop when reaching a cell with a **score of 0**.

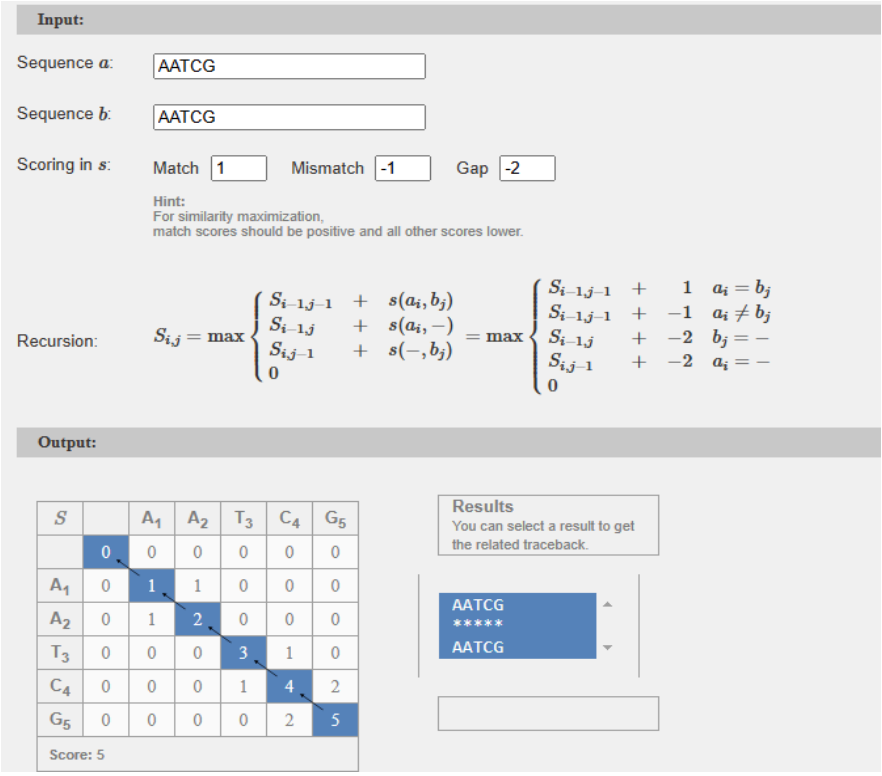


	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	14	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15

Final Alignment: C G T G A A T T C A T

G A C T T _ A

Case of perfectly aligned sequences



S		A_1	A_2	T_3	C_4	G_5
	0	0	0	0	0	0
A_1	0	1	1	0	0	0
A_2	0	1	2	0	0	0
T_3	0	0	0	3	1	0
C_4	0	0	0	1	4	2
G_5	0	0	0	0	2	5

GENOME ASSEMBLY: CONSTRUCTING THE REFERENCE GENOME

1. Introduction

Genome assembly is the process of reconstructing an organism's complete genome sequence from short DNA fragments (reads) obtained through sequencing technologies. Since no current technology can read an entire genome in a single pass, genome assembly is essential for combining these fragments into a contiguous sequence.

2. Sequencing and Read Generation

Sequencing technologies produce short genome fragments known as **reads**, which must be assembled into a complete reference genome. Two major approaches exist for genome reconstruction. We will mainly focus on **WGS** Sequencing.

2.1 Hierarchical Sequencing

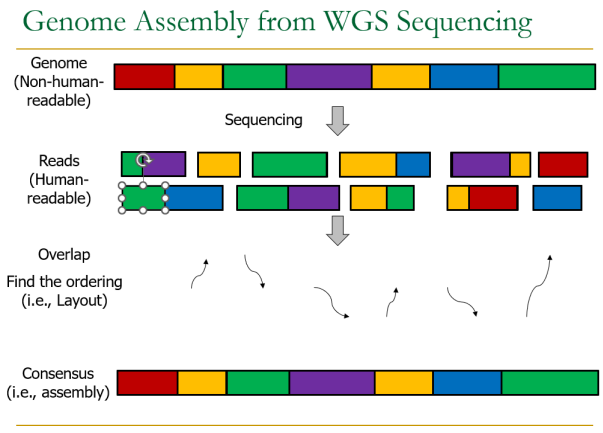
- Used in the Human Genome Project in the 1990s.
- Advantages: Highly accurate and contiguous.
- **Disadvantages:** Slow and expensive.

2.2 Whole Genome Shotgun (WGS) Sequencing

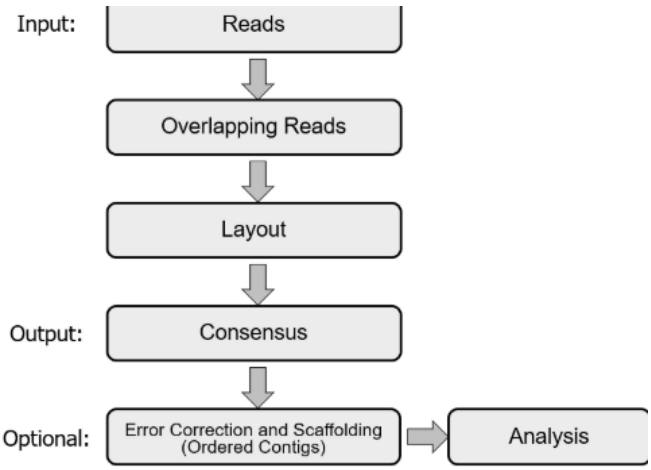
- Faster and more cost-effective.
- Less accurate and less contiguous compared to hierarchical sequencing.

3. Whole Genome Shotgun (WGS) Sequencing Process

The WGS approach follows these key steps:



1. **Sequencing**: The genome is fragmented into short reads.
2. **Overlapping Reads**: Identifying overlapping regions between reads.
3. **Consensus Generation**: Combining overlapping reads to construct a draft genome.
4. **Error Correction (Assembly Polishing)**: Refining the consensus sequence to reduce sequencing errors.
5. **Scaffolding**: Ordering contigs (assembled fragments) to finalize the genome assembly.



4. Identifying Overlapping Reads Using K-mers

To assemble the genome, overlapping reads must be identified and combined. The overlap condition states that the suffix of one read must overlap with the prefix of another.

4.1 Step 1: Extract k-mers from Reads

Each read is broken down into smaller overlapping sequences of length k (k-mers).

Example:

Reads:

- Read 1: AGCTGATCG
- Read 2: TGATCGACC

With k = 4, we extract:

- Read 1 k-mers: AGCT, GCTG, CTGA, TGAT, GATC, ATCG
- Read 2 k-mers: TGAT, GATC, ATCG, TCGA, CGAC, GACC

4.2 Step 2: Storing k-mers in a Hash Table

Each k-mer is stored as a **key** in a hash table, mapping to the **read ID and position**.

k-mer	Read ID	Position
AGCT	1	0
GCTG	1	1
CTGA	1	2
TGAT	1, 2	3 (R1), 0 (R2)
GATC	1, 2	4 (R1), 1 (R2)
ATCG	1, 2	5 (R1), 2 (R2)

- Each k-mer is stored in a hash table, mapping to the read ID and position.
- This structure enables fast overlap detection between reads.
- **Advantages:**
    - Speeds up overlap search.
    - Reduces memory usage (only stores k-mers, not entire reads).
    - Handles sequencing errors (matching multiple k-mers reduces error impact).

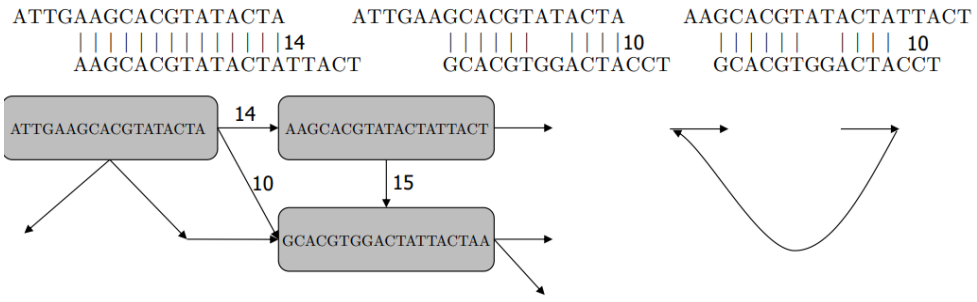
Graph-Based Representation of Sequence Overlaps

Efficiently storing and analyzing overlapping sequences is important in genome assembly. A graph-based representation provides a compact structure that helps avoid redundant reads while accurately identifying the ordering of overlaps between sequences. However, due to the high volume of nucleotide sequences and repetitive regions in genomes, graph complexity increases significantly, and therefore, there is a need to simplify the graph by removing redundant information.

Graph Representation

In an overlap graph:

- Nodes represent individual reads or segments of reads.
- Directed Edges indicate overlaps, where the suffix of one read aligns with the prefix of another.
- Edge Labels quantify the number of matches/mismatches in the overlapping region.



Several optimization techniques, such as transitive reduction, bubble collapsing, and tip removal, are used to simplify overlap graphs and improve genome assembly quality.

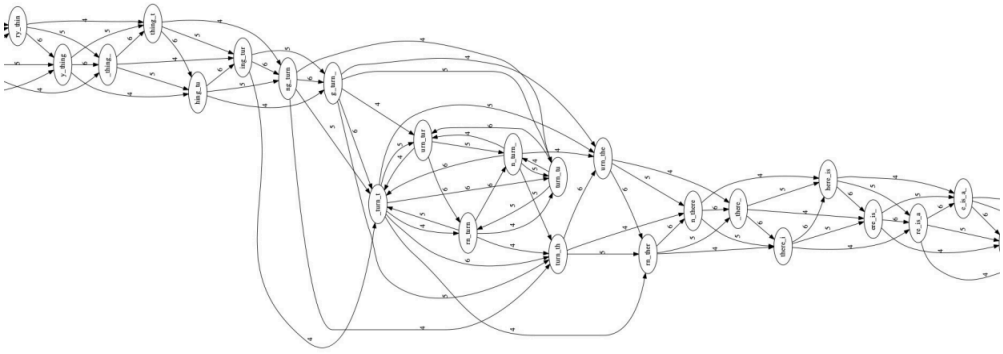
Transitive Reduction

Transitive edges introduce redundancy in an overlap graph. An edge from node v to node w (v → w) is considered transitive if:

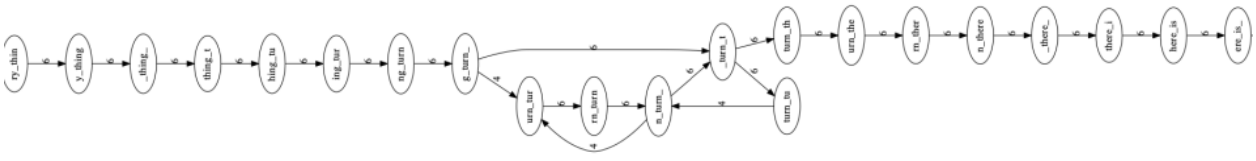
- There exist edges v → u and u → w.
- The edge v → w can be removed without affecting the connectivity between v and w.

By eliminating transitive edges, the graph becomes more compact, making it easier to determine the correct ordering of overlaps. Since the remaining edges maintain the original connectivity, no essential information is lost.

Consider the graph

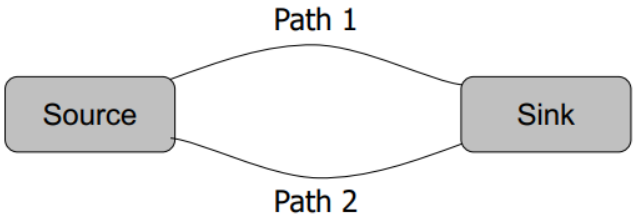


After removing transitive edges, the graph is simplified to



Bubble Collapsing

A bubble is a directed acyclic subgraph with a source node v and a sink node w, where at least two distinct paths exist between them. Bubbles often arise from sequencing errors or repeat regions.

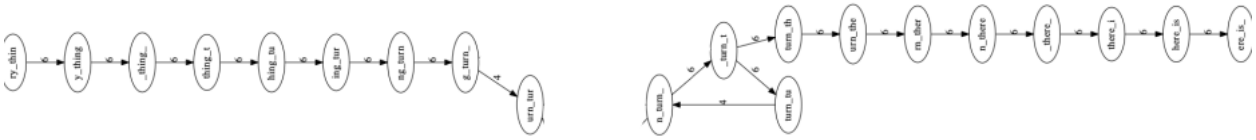


Collapsing bubbles help in:

- Reducing graph complexity by eliminating redundant paths.
- Enhancing genome assembly contiguity, ensuring a more coherent sequence representation.

Typically, the shorter path within a bubble is collapsed, as it often results from sequencing artifacts or partial overlaps.

The earlier graph can be simplified more by bubble collapsing.



However in this, some connectivity information of the graph is lost.

Tip Removal

Some branches in an overlap graph terminate prematurely, providing no useful information for genome assembly. These are referred to as tips and can arise due to sequencing errors, low coverage regions, or short contigs that do not contribute meaningfully to the final assembly. Tip removal improves the clarity of the graph by eliminating such uninformative branches.

Optimizing overlap graphs through transitive reduction, bubble collapsing, and tip removal significantly enhances genome assembly. By applying these methods, the ordering of overlaps becomes clearer, reducing redundancy and computational complexity in genome sequencing projects.

5. Constructing the Consensus Genome

Once overlapping reads are identified, a consensus sequence is generated.

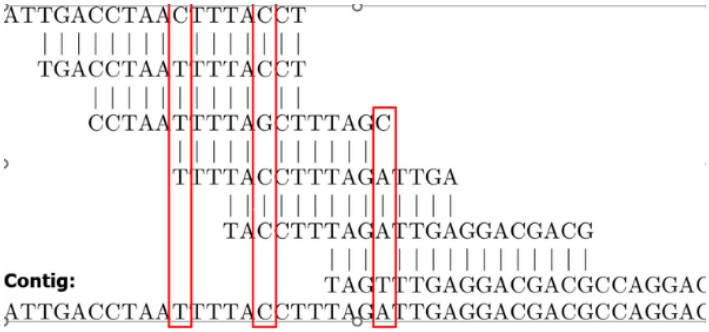
5.1 Layout Overlaps

The overlapping reads are organized based on their positions.

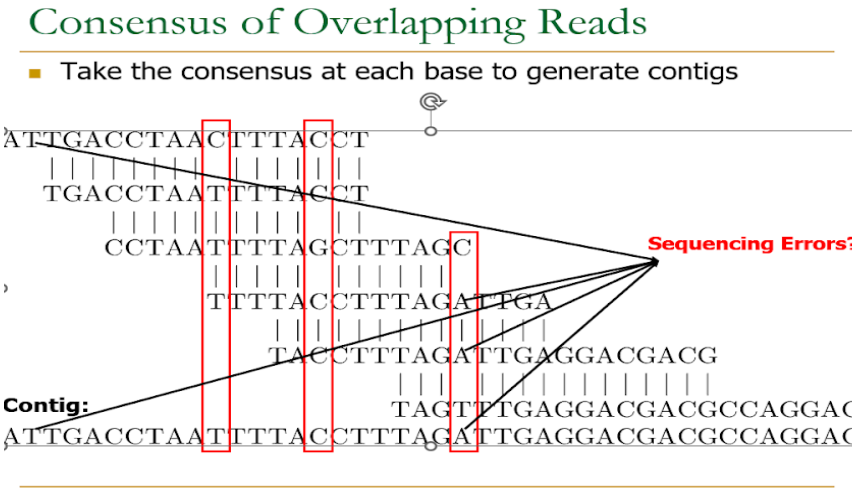
5.2 Generating the Consensus Sequence

Organize the overlapping reads as identified from the overlap graph. At some base positions, multiple reads may have slight differences due to sequencing errors.

A consensus sequence is computed by selecting the most frequent base at that position.



6. Error Correction: Assembly Polishing



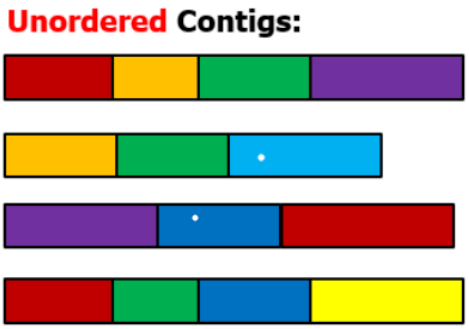
- **Challenge**: Due to sequencing errors, the most frequent base at a position may not always reflect the true sequence. For example, in the column [C T T T], we select T as the consensus base. However, if the original base was actually C, the column would have been [C C C T], leading to C being chosen instead. This means sequencing errors can propagate into the final contigs, introducing inaccuracies in the assembled genome.
- **Solution: Read Realignment to Contigs**

To minimize the propagation of sequencing errors and improve the accuracy of the consensus sequence, reads are realigned to contigs in an iterative process. Here's how it works:

1. **Initial Contig Assembly:**
    - Raw sequencing reads are first assembled into contigs by aligning overlapping reads and determining a consensus sequence at each position.
    - However, due to sequencing errors, the consensus may not always reflect the true sequence.
2. **Realignment of Reads to Contigs:**
    - Once an initial draft of the contigs is formed, the original sequencing reads are realigned to these contigs.
    - This helps correct potential errors in the consensus sequence by providing additional data to reconsider base frequencies at each position.
    - If a sequencing error results in a frequent but incorrect base being chosen (e.g., T instead of the correct C), realigning the reads can reveal this discrepancy and help correct it.

7. Scaffolding: Ordering the Contigs

Once contigs are polished, they must be **ordered** correctly to form the final genome assembly.



- **Contig Ordering Strategy**
  Once contigs are assembled from sequencing reads, they need to be arranged in the correct order to reconstruct the full genome. This step is crucial because assembling contigs independently does not provide information about their relative positions. The contig ordering strategy relies on overlapping reads and pairwise comparisons to determine the correct arrangement.

Step 1: Using Read Overlaps to Determine Contig Order

- Reads that span multiple contigs (i.e., reads that overlap the edges of two contigs) provide clues about their relative positioning.
- If a read starts in Contig A and extends into Contig B, it suggests that Contig A precedes Contig B in the genome.
- By analyzing many such overlapping reads, we can establish a rough order of contigs.

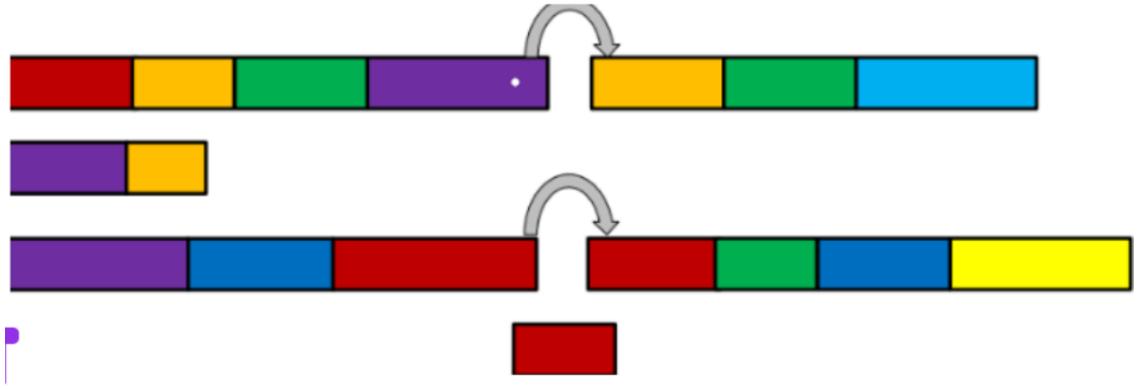
Example:

Consider two contigs:

- Contig A: `...ATGCGTACGATCGA...`
- Contig B: `...GATCGAGGTCACTG...`

If a read spans both contigs:

- Read X: `...ATGCGTACGATCGAGGTCACTG...`
- The read starts in Contig A and extends into Contig B, indicating that Contig A comes before Contig B in the genome.



Genome assembly is a crucial process in bioinformatics, enabling the reconstruction of entire genomes from short sequencing reads. The WGS approach provides a fast and cost-effective solution, while k-mer-based overlap detection and assembly polishing enhance accuracy. Despite sequencing challenges, these techniques ensure a reliable genome reconstruction for biological research and medical applications.

CONCLUSION

The report provides a thorough review of genome analysis with an emphasis on the value of intelligent genome analysis in overcoming the drawbacks of conventional techniques, including their high storage requirements and delayed processing. In existing genomic analysis it highlights the necessity of flexibility, faster findings, scalability, and accuracy. The report explores important ideas such as sequencing technologies (short-read and long-read sequencing), the use of suffix trees and hashing for effective sequence matching and indexing in reference genomes and seeding methods, read mapping (naive approach), and advanced algorithms like the Burrows-Wheeler Transform (BWT) and FM-index for effective genome indexing and searching. It also covers pairwise alignment methods (Needleman-Wunsch and Smith-Waterman) and genome assembly techniques, including whole genome shotgun sequencing and error correction strategies.

In conclusion, genome analysis is an important discipline that helps researchers understand evolutionary features, disease causes, and genetic variants. Genome analysis has become much more accurate, scalable, and efficient with the use of advanced computational techniques like BWT, FM-index, and suffix trees. Large-scale genetic research, customized treatments, and medical advances all depend on these developments. The creation of intelligent genome analysis tools will be essential in addressing issues of computational effectiveness and accurate interpretation of complicated genomic data.

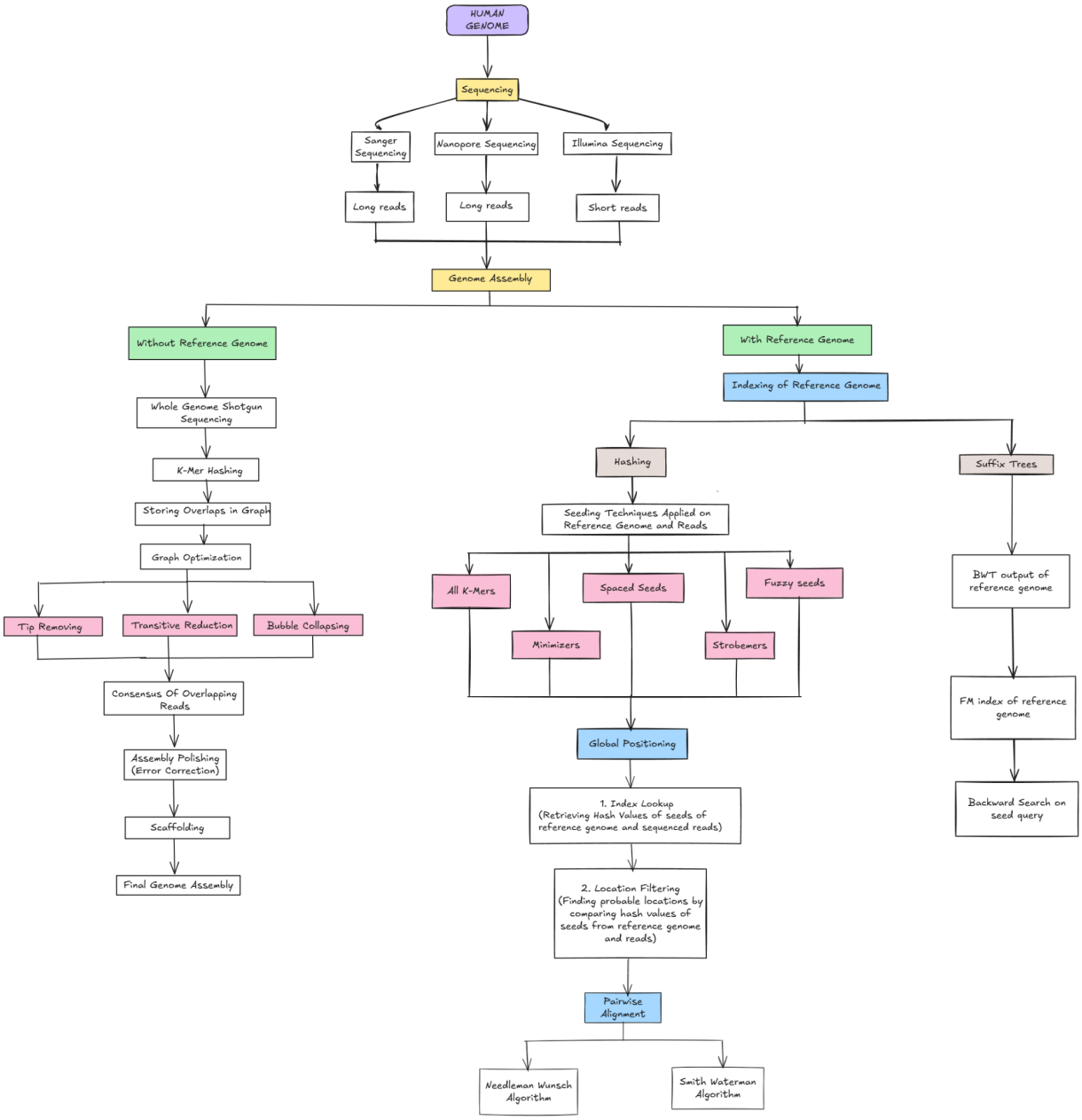


Fig. Flowchart

REFERENCES

1. O. Mutlu, "Bioinformatics," *ETH Zurich*, 2024. [Online]. Available: https://safari.ethz.ch/projects_and_seminars/fall2024/doku.php?id=bioinformatics. [Accessed: 21-Feb-2025].
2. Google Images, "Suffix tree illustration," [Online]. Available: https://images.app.goo.gl/yzUM9xiFrdRp4MR37. [Accessed: 21-Feb-2025].
3. Google Images, "FM-index visualization," [Online]. Available: https://images.app.goo.gl/7K9byPwsD16yErfw7. [Accessed: 21-Feb-2025].
4. H. Lin, W. Xu, and M. Li, "Efficient BWT and FM-index construction for long sequences via a bicameral approach," *arXiv preprint arXiv:2112.08687*, 2021. [Online]. Available: https://arxiv.org/pdf/2112.08687. [Accessed: 21-Feb-2025].
5. Wikipedia, "FM-index," *Wikipedia, The Free Encyclopedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/FM-index. [Accessed: 21-Feb-2025].
6. A. Bowe, "FM-index explained," 2025. [Online]. Available: https://www.alexbowe.com/fm-index/. [Accessed: 21-Feb-2025].
7. M. Kellis et al., *Computational Biology: Genomes, Networks, and Evolution*, LibreTexts. [Online]. Available: https://bio.libretexts.org/Bookshelves/Computational_Biology/Book%3A_Computational_Biology_-*Genomes_Networks_and_Evolution*(Kellis_et_al.)/02%3A_Sequence_Alignment_and_Dynamic_Programming/2.05%3A_The_Needleman-Wunsch_Algorithm. [Accessed: 21-Feb-2025].
8. Figshare, "Suffix-based indexing data structures learning materials," [Online]. Available: https://figshare.com/collections/Suffix-based_indexing_data_structures_learning_materials/7205547. [Accessed: 21-Feb-2025].
9. Rahmann Lab, "Suffix trees and their applications," [Online]. Available: https://www.rahmannlab.de/lehre/alsa21/02-1-suffixtrees.pdf. [Accessed: 21-Feb-2025].
10. GeeksforGeeks, "Hashing Data Structure," *GeeksforGeeks*, 2025. [Online]. Available: https://www.geeksforgeeks.org/hashing-data-structure/. [Accessed: 21-Feb-2025].
11. hre/alsa21/02-1-suffixtrees.pdf. [Accessed: 21-Feb-2025].