Genome Assembly Using Overlap Graphs

Amir Mann, 207732132

March 8, 2025

Abstract

We present an efficient genome assembly algorithm for the PhiX genome based on overlap graphs and trie-based indexing. The method generates both error-free and error-prone reads, allowing controlled evaluation of assembly performance under varying error conditions. In our experiments using 2693 reads, the assembly achieved a precision of 1.0 for clean reads. With a low error probability of 0.001, the precision remains high at approximately 0.999, while a higher error probability of 0.01 reduces precision to around 0.92. Additionally, the implementation demonstrates impressive computational efficiency, achieving a sequencing throughput of between 3.5 and 11 genome assemblies per second. These results underscore the method's robustness and its potential for rapid genome assembly, even in the presence of sequencing errors.

1 Introduction

Genome assembly is a cornerstone task in computational biology, aimed at reconstructing a complete genome from a collection of short, overlapping sequencing reads. This process is inherently challenging due to sequencing errors, repetitive regions, and the enormous volume of data generated by modern high-throughput sequencing platforms [7, 8].

Traditional assembly methods are typically divided into two major paradigms. The Overlap-Layout-Consensus (OLC) approach represents reads as nodes and the overlaps between them as edges, forming an overlap graph that models the continuity of the genome [1]. This method was a key element in early assemblers such as the Celera Assembler [2] and remains particularly effective when the reads are long enough to span repetitive regions. In contrast, de Bruijn graph methods fragment reads into smaller k-mers and are more suited for high-coverage datasets of short reads.

In our project, we focus on an OLC-based assembly strategy to reconstruct the PhiX genome. PhiX is a well-characterized bacteriophage with a small, balanced genome, making it an ideal model system for evaluating assembly algorithms [3]. To assess the robustness of our method, we simulate both error-free and error-prone reads (with base mismatch probabilities of 0.001 and 0.01, respectively) and quantify assembly performance using metrics such as recall, precision, and intersection-over-union (IoU). Recall measures the proportion of the true genome bases that are recovered in the assembly, precision quantifies the fraction of bases in the assembled sequence that correctly map to the genome, and IoU (also known as the Jaccard index) captures the overall agreement between the assembly and the reference genome [9, 6].

A notable feature of our implementation is the use of a trie-based data structure for rapid overlap detection. The real strength of the trie lies in its ability to simultaneously check for matching between any given suffix and the entire collection of reads, thereby expediting the construction of the overlap graph [5, 4]. Preliminary analysis suggests that our approach exhibits strong worst-case performance and excellent average-case runtime; further details on the runtime complexity will be incorporated after a complete analysis of the code.

Overall, this report demonstrates a complete implementation of genome assembly using overlap graphs, fulfilling the objectives of the assignment.

2 Methods

The full code for this project is available at https://github.com/Amir-Mann/GenomeAssemblyUsingOverlapGraphs .

2.1 Notations and Preliminaries

Let S denote the true genome (or sequence) to be assembled, with $|S|$ representing its length. In our experiments, we will later compare the assembled candidate sequence, denoted by $\hat{S}$, to the true sequence S.

Coverage. Coverage, denoted by C, is defined as the average number of times each base in S is read during sequencing. In other words, it measures the redundancy of the sequencing data. A higher coverage increases the likelihood that every base in S is represented in the set of reads, which is critical for accurate assembly and error correction. Assuming that the sequencing process uniformly samples from S, the total number of reads required is computed by

$$N = \frac{C \times |S|}{l},$$

where l is the read length (i.e., the number of bases in each read). For instance, given the PhiX genome with $|S| \approx 5386$ bases and using reads of length $l = 100$, a $50\times$ coverage requires approximately

$$N \approx \frac{50 \times 5386}{100} \approx 2693 \text{ reads.}$$

Read Simulation and Error Modeling. Reads are generated by randomly selecting a starting position in S and extracting a substring of length l. To mimic real sequencing conditions, each base in a read is allowed to undergo a substitution with a given error probability p. The parameter `allow_mis_matches`, denoted by Δ, specifies the maximum number of mismatches permitted when comparing the suffix of one read to the prefix of another. This parameter strongly influences the result of the trie-based search function and the search complexity.

2.2 Generating the Graph

To construct the overlap graph, we employ a trie data structure for fast suffix–prefix matching. Every read is first inserted into a single trie, and then, for each read, we search its suffixes to find overlaps with other reads. The search starts with the longest suffix and proceeds downward until a match is found or until the suffix length drops below a threshold of

$$L_{\min} = 2\log_4(N),$$

where N is the total number of reads. This threshold is chosen because shorter suffixes are more likely to yield random matches (with probability approximately $(1/4)^L$ for a suffix of length L), so for $L > \log_4(N)$ the expected number of random matches becomes negligible.

Furthermore, our algorithm employs an early stopping strategy: once the first (and thus the largest) overlap is detected for a given read, no further (shorter) suffixes are examined. Although this approximation may sometimes miss additional overlaps, it greatly simplifies the trie—by storing only one vertex per node—and significantly improves runtime for both graph construction and later traversal.
Complexity Analysis.

- *Insertion Phase:* Each read of length l is inserted into the trie in $O(l)$ worst-case time. With N reads, the overall insertion cost is $O(Nl)$.
- *Search Phase:* For each read, up to $O(l)$ suffixes may be examined. In the worst-case scenario (when no exact match is found), and when mismatches are allowed, the search may explore up to $O(4^k)$ branches at each character position, where k is the allowed number of mismatches. Therefore, each suffix search incurs a worst-case cost of $O(4^k \cdot l)$, leading to a worst-case total cost of $O(4^k \cdot l^2)$ per read and $O(N \cdot 4^k \cdot l^2)$ overall. However, under high coverage the probability of finding a long matching suffix is very high, so on average only a constant number of suffixes is examined per read. Consequently, the average-case cost per read is $O(4^k \cdot l)$ (with 4^k being a small constant for typical values of k), yielding an overall average-case complexity of $O(Nl)$.

We note that an alternative to this approach would be to use a suffix tree and query it with prefixes. However, this approach is more complex to implement and results in the same average-case complexity as analyzed in Auxiliary Material 1.

2.3 Assembly

The assembly process begins by identifying origins in the overlap graph. An origin is defined as a read with 0 incoming connections, indicating it is likely at the start of a contig. The traversal proceeds by following the single outgoing connection from each vertex, which corresponds to the read with the longest suffix-prefix overlap.

Repetitive and ambiguous regions pose challenges, as certain reads may have multiple valid overlaps. Our approach resolves such ambiguities by consistently selecting the longest available overlap, allowing for efficient and reasonably accurate genome reconstruction. However, this heuristic approach may struggle in highly repetitive genomic regions where multiple reads have nearly identical overlaps.

From a computational perspective, our method ensures efficiency. Since each read has at most one outgoing connection, the number of edges in the graph is equal to the number of reads ($E = N$). As the traversal visits each vertex once, the overall time complexity of the assembly process is $O(N)$. While a globally optimal genome reconstruction would require exploring all possible overlaps and sequencing paths—an NP-complete problem—our greedy approach provides a practical and computationally feasible solution for genome assembly using overlap graphs.

2.4 Evaluation

The quality of the assembled genome $\hat{S}$ is assessed by comparing it to the reference genome S using three key metrics:

- **Recall**: The fraction of S that appears correctly in $\hat{S}$.
- **Precision**: The fraction of $\hat{S}$ that correctly matches S.
- **Intersection over Union (IoU)**: A combined measure of correctness that balances recall and precision.

To compute these metrics, we perform an alignment between S and $\hat{S}$ using a simplified global alignment approach inspired by the Needleman–Wunsch algorithm. Given that our sequencing process does not introduce insertions or deletions, but only substitutions or strand misalignment, our alignment method is adapted to reflect these constraints.

Our method searches for the best possible alignment of $\hat{S}$ within S while allowing for sequence offsets. However, unlike the full Needleman–Wunsch algorithm, our approach applies two key optimizations:

- **Indels are only considered at the beginning of the sequence**: Since our read model does not introduce indels within reads, only initial offset shifts are examined.
- **Early stopping**: If the mismatch count exceeds $\log |\hat{S}|$, where $|\hat{S}|$ is the length of the assembled sequence, the alignment process terminates early to reduce computational overhead.

A full Needleman–Wunsch alignment has a complexity of $O(|S| \cdot |\hat{S}|)$. Our optimizations significantly reduce this cost by allowing only initial offsets and stopping early when mismatches grow too large, leading to a practical complexity closer to $O(|S| \cdot \log |\hat{S}|)$ in most cases, as a misaligned string would have nearly 3 mismatches for every match, terminating after $\frac{4}{3} \log |\hat{S}|$ steps. This enables efficient evaluation while maintaining a meaningful assessment of assembly accuracy. We note that could have runned a full Needleman-Wunsch algorithm to evaluate our results, with higer and more accurate measurement, but then we would not be able to replicate them over many iterations. We choose to provide a statistically based result with our relaxed alignment algorithm.

Finally, in order to evaluate different methods accurately, we conduct I repetitions of the error-prone reads assembling process and report the average recall, precision, and IoU over these iterations.

3 Results

We conducted a series of experiments to evaluate the effect of sequencing error probability (p), allowed mismatches (k), read length (l), and number of reads (N) on the accuracy and efficiency of our genome assembly approach. All experiments were replicated in 100 iterations of running the same test. The primary performance metrics considered are recall, precision, and Intersection over Union (IoU).

We note that our results on clean reads always had run-time of around 10-30 seconds for 100 replications with recall, precision, and IoU of above 99% in all cases. For that reason we include only 1 analysis of error probability 0. Both dirty and clean read results can be replicated using the run_tests.sh and run_clean_read_tests.sh files.

3.1 Effect of Error Probability (p)

To analyze how error probability affects assembly accuracy, we compare results for different values of p. The reference experiment uses $l = 100$, $N = 3000$, and $k = 2$ with $p = 0.01$. The results are presented in Table 1.

p	Recall	Precision	IoU	Runtime (s)
0.0 (clean reads)*	0.9995	1	0.9995	15.61
0.001*	0.9956	0.9989	0.9946	22.34
0.01 (ref)	0.9119	0.9206	0.9029	63.84
0.1	0.1421	0.5585	0.1318	232.38

Table 1: Effect of sequencing error probability (p) on assembly quality. (*) for $p = 0, 0.001$ we used $k = 0, 1$.

As expected, higher error probabilities significantly degrade assembly performance. For clean reads $p = 0$ and at $p = 0.001$, accuracy remains near perfect. At $p = 0.01$ we achive good results with 91% mean recall and 92% mean precision. However, at $p = 0.1$, recall drops to 0.14, and IoU to 0.13, indicating poor reconstruction, exactly covering only 14.2% of the genome.

3.2 Effect of Allowed Mismatches (k)

We examine how varying k affects performance at $p = 0.01$, as shown in Table 2.

k	Recall	Precision	IoU	Runtime (s)
0	0.1428	0.8332	0.1414	51.88
1	0.5887	0.9458	0.5819	73.3
2 (ref)	0.9119	0.9206	0.9029	63.84
3	0.4872	0.4798	0.4729	162.96

Table 2: Effect of allowed mismatches (k) on assembly accuracy at $p = 0.01$. The reference experiment uses $k = 2$.

Increasing k from 0 to 2 improves recall significantly but adding more mismatches beyond this reduces performance due to incorrect overlaps.

3.3 Effect of Read Length (l)

We evaluate how read length impacts assembly quality. The results for different values of l at $p = 0.01$ are shown in Table 3.

l	Recall	Precision	IoU	Runtime (s)
50	0.8733	0.9090	0.8641	40.6
100 (ref)	0.9119	0.9206	0.9029	63.84
150	0.8881	0.9001	0.8786	103.46

Table 3: Effect of read length (l) on assembly accuracy at $p = 0.01$. The reference experiment uses $l = 100$.

Shorter reads ($l = 50$) lead to lower recall and IoU, as shorter overlaps increase fragmentation. Increasing l to 150 results in slightly lower recall and precision, likely due to reduced read redundancy.

read length	num reads	error prob	allowed misses	recall	precision	IoU	runtime	graph gen runtime	assembly runtime	eval runtime
100	3000	0	0	0.99954	1	0.99954	15.61	0.149	0.001	0.001
100	3000	0.01	2	0.91186	0.92064	0.90286	63.84	0.29	0.196	0.135
100	3000	0.001	1	0.99563	0.99895	0.99459	22.34	0.162	0.036	0.01
100	3000	0.1	2	0.14209	0.5585	0.13179	232.38	1.845	0.003	0.446
100	2000	0.01	2	0.90441	0.92624	0.89194	44.15	0.197	0.075	0.157
100	4000	0.01	2	0.86432	0.88089	0.85569	102.4	0.378	0.412	0.21
50	3000	0.01	2	0.87328	0.90902	0.86414	40.6	0.11	0.123	0.164
150	3000	0.01	2	0.88808	0.90015	0.87864	103.46	0.614	0.218	0.177
100	3000	0.01	0	0.14281	0.83315	0.14139	51.88	0.274	0.005	0.222
100	3000	0.01	1	0.58868	0.94578	0.58187	73.3	0.268	0.089	0.358
100	3000	0.01	3	0.48716	0.47984	0.47295	162.96	0.398	0.322	0.891

Figure 1: Full results from all experiments, including all evaluation metrics and per method-subsection timing results.

3.4 Effect of number of reads (N)

To examine the role of sequencing depth, we vary N while keeping other parameters fixed. The results are presented in Table 4.

C	Recall	Precision	IoU	Runtime (s)
2000	0.9044	0.9262	0.8919	44.15
3000 (ref)	0.9119	0.9206	0.9029	63.84
4000	0.8643	0.8809	0.8557	102.4

Table 4: Effect of number of reads (N) on assembly accuracy at $p = 0.01$. The reference experiment uses $N = 3000$.

Increasing N does not always improve accuracy. At very high coverage, repeated reads may introduce ambiguity, slightly lowering recall and IoU, while also significantly increasing runtime.

3.5 Runtimes

As can be seen in our results, probability mainly affects the runtime of graph generation. When p is high, we observe higher graph generation time. This is because the average-case analysis does not hold when the longest overlaps become 20 on expectation with $p = 0.1$, making the runtime closer to $O(N \cdot l^2)$.

We can also see that the only cause for longer assembly runtime is an increase in the number of reads, as they represent the size of the graph, increasing the traversal time. The rest of the assembly runtimes remain fairly short.

The evaluation time, which is $O(|S| \log |S|)$, usually does not require checking all possible offsets but only those necessary to find the correct alignment, leading to an expected runtime of $O(|S| \cdot (1 - \text{precision}) \cdot \log |S|)$. The worst cases align with situations where performance drops.

Lastly, the effect of the number of allowed mismatches on graph generation time is much lower than a fourfold increase per k value. This is likely because searching for a suffix that replaces a sequence with a letter that does not appear in the genome terminates in very few steps.

4 Discussion

We demonstrated that by using early stopping optimizations, we can produce high-quality genome assemblies efficiently with fairly simple logic, based on the algorithms discussed in the course.

Our results show that for high-quality reads, we can achieve near-perfect performance, processing over five genome assemblies per second. Even for moderately error-prone reads, we maintain a rate of approximately two genome assemblies per second, allowing us to replicate our results over many evaluation iterations.

However, our results also highlight some limitations of our method. Most notably, for $p = 0.1$, our assembly quality is poor. Additionally, even in cases with lower error rates, we do not employ any read correction algorithms, meaning we do not fully utilize the benefits of high coverage for each nucleotide in the genome.

Given more time, the next improvement we would implement is read error correction. Using a suffix tree of all reads, we could identify subreads that appear an unusually low number of times in the genome. For example, if a read's first nine characters appear around 30 times in the genome, the 10th character is likely to match the majority of those occurrences. Such an algorithm could improve recall, particularly for higher error probabilities like $p = 0.1$.

Another possible enhancement to improve precision, after obtaining a high base recall, would be to locally align reads to the assembled sequence. If a large number of reads disagree with a specific nucleotide in the sequence, we could modify it based on a majority vote. This would help correct sequencing errors and further refine the final assembly.

References

[1] Overlap-layout-consensus. `https://omics.crumblearn.org/genomics/assembly/de-novo/overlap-layout-consensus/`. Accessed: 2025-03-06.

[2] Celera assembler. `https://de.wikipedia.org/wiki/Celera_Assembler`, 2018. Accessed: 2025-03-06.

[3] Phi x 174. `https://en.wikipedia.org/wiki/Phi_X_174`, 2025. Accessed: 2025-03-06.

[4] Suffix tree. `https://en.wikipedia.org/wiki/Suffix_tree`, 2025. Accessed: 2025-03-06.

[5] Trie. `https://en.wikipedia.org/wiki/Trie`, 2025. Accessed: 2025-03-06.

[6] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[7] V. Jayakumar and Y. Sakakibara. Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de bruijn-graph. *Briefings in Functional Genomics*, 11(1):25–37, 2012.

[8] Manuela Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, and Paola Bonizzoni. Overlap graphs and de bruijn graphs: data structures for de novo genome assembly in the big data era. *Quantitative Biology*, 7(4):278–292, 2019.

[9] Takaya Saito and Michael Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PloS One*, 10(3):e0118432, 2015.

Auxiliary Material 1 - Suffix Tree Optimization

An alternative approach to the full-read trie is to construct a suffix tree for the entire collection of reads. A suffix tree stores all suffixes of all reads in a compressed structure, enabling efficient substring searches. Each query can be answered in time proportional to the length of the query. However, implementing a suffix tree presents several challenges:

Construction Complexity: Building a suffix tree for a large dataset requires sophisticated algorithms such as Ukkonen's algorithm to achieve linear-time construction. This involves intricate handling of edge labels, node pointers, and careful memory management.

Implementation Overhead: Unlike a full-read trie, where each node corresponds directly to a character in a read, a suffix tree compresses repeated substrings and maintains suffix links. This adds considerable complexity to both the code and the underlying data structures.

Memory Overhead: While suffix trees are theoretically efficient, their memory usage includes additional bookkeeping, such as suffix links, which can be substantial for large numbers of reads.

Despite these complexities, the average-case search performance of a suffix tree remains comparable to that of the full-read trie, both providing near-linear time searches (i.e., $O(l)$ per query for a read of length l). However, the suffix tree introduces additional implementation overhead, making it less attractive unless its compressed representation of all suffixes is necessary.

In our case, we only require the longest matching suffix rather than any arbitrary suffix. A suffix tree implementation for all strands would require either multiple end-string symbols (which would incorrectly extend matches) or significantly more complex topologies in memory to maintain correct boundaries between different reads.

Efficient implementation of such a suffix tree would yield a worst-case matching complexity of $O(N \cdot l)$, matching our average complexity. However, incorrect implementations can degrade performance significantly, resulting in $\Omega(N^2)$ complexity due to mistakes such as incorrect edge compression or using an alphabet of N termination symbols (\$), leading to excessive node expansions and inefficiencies.

For our purposes, the full-read trie, with its early stopping mechanism and straightforward design, meets performance requirements while keeping the codebase manageable.