Comparitive study on string graph and de-bruijn graph for genome reconstruction

Chaitanya cs22b036[1], Yashwanth cs22b002[2]

1 Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, Tamil Nadu, India
2 Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, Tamil Nadu, India

Abstract

Genome assembly is a fundamental problem in bioinformatics where the goal is to reconstruct a genome from a large number of short sequencing reads. In this project, we explored two main approaches used for de novo genome assembly: String Graphs and De Bruijn Graphs. We started by understanding how each of these graph models works and what makes them suitable for assembly. For String Graphs, we focused on detecting overlaps between reads using methods like k-mer indexing and simplified the graph by removing transitive edges and collapsing chains simplifying the graph and improving efficiency. We also used a flow-based technique to classify edges and select the most validated ones for genome reconstruction. On the other hand, for De Bruijn Graphs, we built the graph using fixed-length k-mers in parallel and applied iterative path cover algorithm. We also used SPAdes Genome Assmebler [1] to compare our implementation with it. We implemented both approaches and tested them using real datasets(EColi genome). Finally, we used tools like QUAST [2] to compare our assembled sequences to a reference genome and evaluate the accuracy. This project gave us a good experience with the principles of genome assembly and helped us explore both methods to a certain depth.

Introduction

De novo genome assembly reconstructs an organism's genome from sequencing reads without a reference genome, playing a crucial role in studying novel species and genetic variations [3]. But this is computationally hard because of sequencing errors, repetitive sequences, and huge data.

There are two main approaches for genome assembly: string graph-based and De Bruijn graph-based. Overlap-Layout-Consensus (OLC) builds a string graph by finding overlaps between sequencing reads and simplifying the graph [4]. De Bruijn graphs break sequences into k-mers and build edges based on sequence continuity [5]. De Bruijn graphs are efficient for short reads but struggle with sequencing errors and complex repeats, whereas string graphs are good for long-read sequencing technologies like PacBio and Oxford Nanopore [6].

Recent progress, such as diBELLA 2D, has improved the parallel construction of string graphs, but genome reconstruction is still a hard problem [7]. De Bruijn graphs also have limitations depending on sequencing data [8]. This project will compare genome reconstruction from string graphs and De Bruijn graphs, with metrics of assembly contiguity, error rates, memory usage, and execution time.

Materials and methods

Dataset

The datasets used are all based on *Escherichia coli* K-12 substrain MG1655. This includes both real short-read (Illumina) and long-read (PacBio) sequencing data, along with the official reference genome. The short-read dataset is used in its full form as well as in smaller subsampled versions to test the performance of assembly algorithms under varying input sizes. The long-read dataset is used to evaluate how well string graph-based methods perform on large, realistic datasets. All assembled outputs are compared against the same reference genome to assess reconstruction accuracy.

Dataset Name	Description	Size	Data Type
ecoli_k12_reference.fasta	Official reference genome of *E. coli* K-12 MG1655	4.6 MB	Genomic FASTA
ecoli_subset.fasta	Full Illumina short-read dataset (high coverage, 100 bp reads)	267 MB	Short reads
ecoli_small_25000.fasta	Subsample of 25000 reads from `ecoli_subset.fasta`	3.6 MB	Short reads
ecoli_small_5000.fasta	Subsample of 5000 reads from `ecoli_subset.fasta`	600 KB	Short reads
ecoli_full.fasta	Long-read PacBio sequencing dataset of the same *E. coli* strain	1.2 GB	Long reads

Table 1. Summary of datasets used for genome assembly and evaluation.

De-Bruijn Graph

De Bruijn grpahs are the fundamental data structures in modern genome assembly due to their efficiency in handling large datasets. De Bruijn graphs represent k-mers (substrings of length k) as edges between (k-1)-mers. This representation allows for more efficient memory usage when processing high-coverage sequencing data as the graph scales with genome length rather than the total read length.

Our Implementation

The pipeline follows a multi stage approach from read processing to genome reconstruction.

1. Read processing and k-mer extraction
2. De Bruijn graph construction
3. Path finding and genome reconstruction
4. Validation and assessment

Tools/Libraries used

We used Python libraries `os`, `re`, and `collections` for file handling and text processing, `Bio.SeqIO` from Biopython for sequence parsing, and `subprocess` to run external tools

De Bruijn graph preparation

- **Parallel k-mer extraction:** The first step in graph preparation is k-mer extraction from the input reads. This begins by reading the raw sequencing reads in FASTA format. In the next step, the reads are split across multiple threads to

accelerate processing (*parallel processing*). Now each thread extracts all-mers from its assigned reads. The frequency of each k-mer is also kept track to identify potential errors.

- **De Bruijn graph construction:** Following the k-mer extraction, the graph is constructed through a distributed process. Each process/thread constructs a local graph from the chunk of reads assigned to it. These local graphs are then merged to form a complete De Bruijn graph. The count of each edge (representing k-mer abundance) is preserved. Finally this graph is written to an output file in the format $< prefix > -> suffix > [count = n]$

Reconstruction using Iterative Path Cover

The iterative path cover algorithm is designed to construct contigs from non-Eulerian De Bruijn graphs as these are commonly encountered in real sequencing data which in this case is *E. coli*. Hence there will be many nodes with $in - degree \neq out - degree$. So, rather than aiming for a single, global traversal, this method repeatedly extracts multiple paths from the graph until no valid paths remain. Here, each path contributes to the final set of contigs that collectively represent the final genome assembly .

- **Graph traversal strategy:** The algorithm starts by selecting an appropriate starting node, which is usually a semi-balanced node where the out-degree exceeds the in-degree by one. This is done by *find_path_start()* function. From here, we employ a modified version of the Hierholzer's algorithm, implemented in the *extract_single_path()* function. As we traverse the edges, they are removed from the graph and the node degree information is updated accordingly.
- **Contig construction and graph reduction:** Once the path has been extracted, it is converted into a contig by concatenating the sequence of nodes along the path. the *clean_graph()* function then reduces the graph by removing the used edges and updating the node degrees. Thus the graph is prepared for the next iteration. this process continues until no valid paths can be identified.
- **Comparison with standard assembly approaches:** Unlike traditional De Bruijn graph assemblers that attempt to identify a single Eulerian path, this method acknowledges that real world graphs are rarely Eulerian due to sequencing errors and uneven coverage. A single Eulerian path may not exist.

Strengths and limitations

This method is well-suited for handling the complexities of real sequencing data. Its strengths include efficiency, parallel k-mer extraction, construction of contigs.

The limitations include producing chimeric contigs when traversing through ambiguous regions or branch points. It does not include coverage based filtering to nullify sequencing errors and the results can be sensitive to the initial choice of the starting node.

SPAdes Genome Assembler [1]

SPAdes is a widely used tool designed for both single-cell and standard short read data. It produces high-quality assemblies by integrating advanced error correction methods and graph based strategies.

- **Read error correction:** To improve accuracy, read error correction is performed which helps remove sequence noise and reduces complexities of the resulting graph.

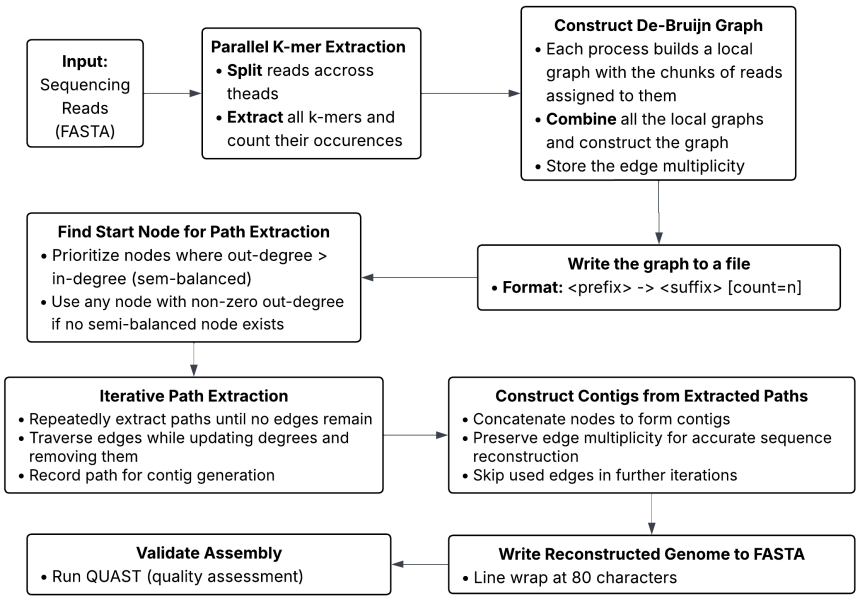

Fig 1. Pipeline of our implementation.

It then constructs **multi-k de Bruijn graphs** by combining information from different k-mer sizes.

- **Graph simplification techniques** such as removing tips, resolving bubbles and eliminating low-coverage edges are employed. Contigs are then extracted by identifying the Eulerian paths through the simplified graph.
- **Parallelization:** To enhance performance and scalability, core components of the SPAdes pipeline are parallelized.

In this work, we used SPAdes as a benchmark to evaluate the quality and effectiveness of our own iterative path cover implementation.

String Graphs

String graphs are one of the most used data structures for genome reconstruction using de novo genome assembly. Unlike De-Bruijn Graphs where we have fixed length k-mers(precisely (k-1)-mers if we focus on nodes) the nodes of a string graph correspond to the actual reads obtained. The edges denote the overlap between the reads.

String graph implements certain graph complexity reduction techniques like transitive reduction which makes it suitable for this task.

We followed the below methodology for genome reconstruction:

1. Data processing.
2. String graph construction
3. Performing reconstruction techniques on the obtained graph
4. Validation

Tools/Libraries Used

1. Python Libraries like **collections, subprocess** and **multiprocessing** were used.
2. **QUAST** was used for validation

Construction of String graph

Parsing the fasta file is the first step from where we got the read sequences.

We tried various approaches to construct the string graphs(overlap detection etc.). They are mainly:

- **Naive Method:** This method takes all the pairs of reads and find the overlap between them and add those edges. As this method has a large time complexity of $O(n^2l)$, it is not efficient for large datasets.
- **Burrows-Wheeler Transform (BWT):** The Burrows-Wheeler Transform is a technique that rearranges a string to make it more compressible. In genome assembly, BWT helps in detecting overlaps between reads more efficiently. Instead of comparing every pair of reads directly, BWT allows us to quickly search for matching substrings using something called the FM-index. This reduces both the time complexity and space usage compared to brute-force overlap detection. So, it helps speed up the assembly process while also using less memory.
- **k-mer index:** A k-mer index is a data structure that stores each unique k-mer and keeps track of which reads it appears in. This is helpful because if two reads share a common k-mer, there is a chance they might overlap. Instead of comparing every read with every other read (which would take a lot of time), we can just look at the reads that have a k-mer in common. This helps us narrow down the number of comparisons and makes the overlap detection process much faster.

The obtained graph still contains a lot of nodes making it complex to perform reconstruction on, so we do the following:

- **Transitive Reduction:** If we have edges $A \to B$, $B \to C$ and $A \to C$, we can see that the edge $A \to C$ is redundant for reconstruction. Therefore we remove all such edges in the graph. We perform this by running a dfs and removing all edges which do not lie on the dfs forest. This removes all the transitive edges.
- **Collapsing Chains:** We have a not of nodes with in-degree = out-degree = 1, such nodes have no structural significance and can be collapsed down to an edge between two junctions. This helps in simplifying the graph.

Reconstruction using the string graph

We assigned weight to the edges based on the support they get from the reads i.e. the number of reads which confirm its presence. To estimate edge weights, we count how many times each overlap appears in the graph. If many reads have the same overlap, it means that overlap is probably important, so we give it a higher weight. Then, in the classification step, we try to see if the number of reads going into and coming out of each node is balanced. If an edge has no support, we mark it as "not_required". If only one read supports it, we call it "unreliable". Otherwise, we say the edge is "required". This helps us keep only the useful overlaps and ignore the ones that might be due to sequencing errors.

After that we perform the greedy traversal based on whether the edge is required or not by keeping an adjacency list based on it and also track the indegree and outdegree for identifying starting nodes.

Strengths and Limitations

Since we use actual read overlaps (not broken k-mers like in De Bruijn graphs), the assembly has good accurate.The k-mer index helps us avoid checking all pairs of reads, which makes the overlap detection step faster. Steps like removing transitive edges and collapsing chains help clean up the graph and make the genome easier to reconstruct.

Even with the k-mer index, building the string graph can be slow and memory-heavy for big genomes. This is the reason complete reconstruction is not possible without higher level equipments like GPUs, Supercomputers etc. Results depend heavily on the minimum overlap cap and the k-mer size.

Results

To assess the accuracy and quality of the assembled genome, we used QUAST (Quality Assessment Tool for Genome Assemblies) [2]. Since we had a reference genome for *E. coli*, QUAST was run in reference-based mode, calculating accurate assessment of genome fraction, number of misassemblies and duplication ratio. The analysis was performed using the following command.

```
quast.py assembled_genome.fasta -r reference_genome.fasta -o quast_output
```

- **GC content distribution** reveals base composition trends and anamolies.
- **Cumulative length distribution** illustrates assembly continuity and size distribution of contigs.

De-Bruijn graph

Our Implementation

Alignment-based statistics	reconstructed_contigs
Genome fraction (%)	0.004
Duplication ratio	1
Largest alignment	87
Total aligned length	171
NGA50	-
LGA50	-
Misassemblies	
# misassemblies	0
Misassembled contigs length	0
Per base quality	
# mismatches per 100 kbp	0
# indels per 100 kbp	0
# N's per 100 kbp	0.27
Statistics without reference	
# contigs	1
Largest contig	146 042 200
Total length	146 042 200
Total length (>= 1000 bp)	146 042 200
Total length (>= 10000 bp)	146 042 200
Total length (>= 50000 bp)	146 042 200

Fig 2. QUAST report.

As we can see the genome coverage is 0.4% which is extremely low. The number of contigs is just one and the size of that is $146 Mbp$ which is massively larger than E. coli ($4.6 Mbp$).
From *fig 3* we can observe that the reconstructed genome's GC content distribution doesn't overlap with the reference genome's GC content distribution.
From *fig 4* we can conclude that the reconstructed genome's largest contig doesn't have the highest coverage. This graph needs to steep at the start and then achieve a low slope.

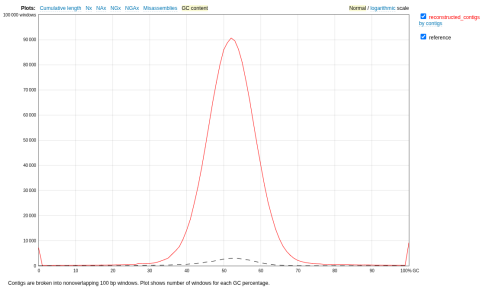

Fig 3. GC content distribution.

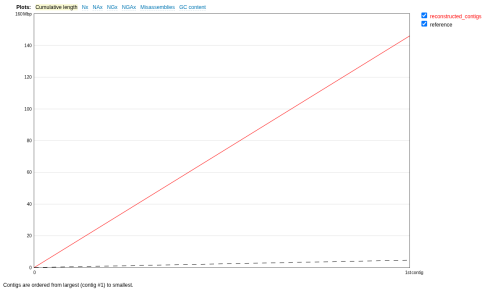

Fig 4. Cumulative length distribution.

Alignment-based statistics	contigs
Genome fraction (%)	0.591
Duplication ratio	1
Largest alignment	13 973
Total aligned length	27 442
NGA50	-
LGA50	-
Misassemblies	
# misassemblies	0
Misassembled contigs length	0
Per base quality	
# mismatches per 100 kbp	2412.36
# indels per 100 kbp	225.93
# N's per 100 kbp	0
Statistics without reference	
# contigs	133
Largest contig	303 257
Total length	4 881 752
Total length (>= 1000 bp)	4 867 989
Total length (>= 10000 bp)	4 720 850
Total length (>= 50000 bp)	3 554 819

Fig 5. QUAST report.

SPAdes achieved a coverage of 59.1% which is pretty impressive compared to our implementation. Also no missamblies indicate that it is a structurally clean assembly.
From *fig 6* we can see that the reconstructed genome almost overlaps with reference genome which is a good sign.
From *fig 7* we can observe that the curve starts with a high slope at first and then it reaches a saturation. This is the expected result.

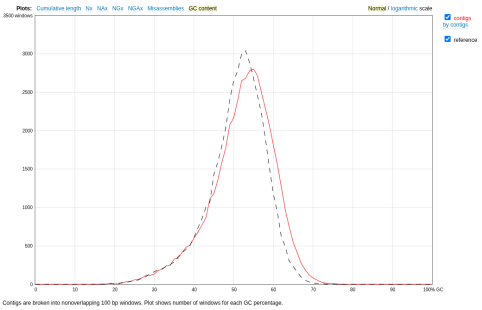

Fig 6. GC content distribution.

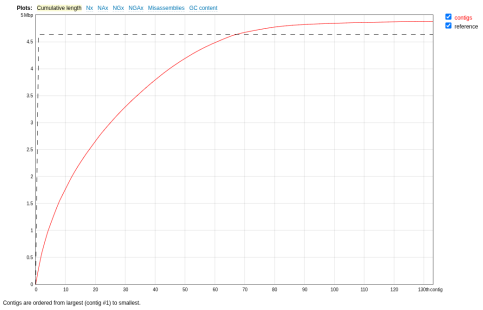

Fig 7. Cumulative length distribution.

String Graphs

We have not got fruitful results while working with string graph. Our implementation is too slow to work with all the reads and cannot be used for complete reconstruction at this stage but we got some results worth mentioning and help us move forward.

We took a subset of 25000 reads of the graph and able to generate a contig of $\tilde{3}$0,000 base pairs but it is unaligned as we have not taken all reads.

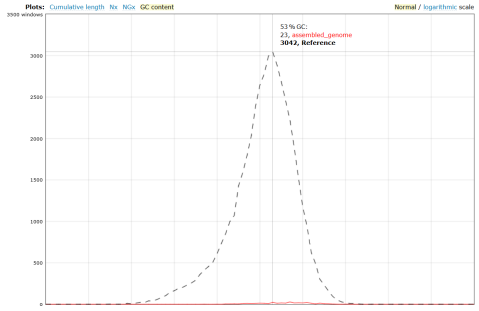

Fig 8. GC content distribution.

Statistics without reference	assembled_genome
# contigs	1
# contigs (>= 0 bp)	1
# contigs (>= 1000 bp)	1
# contigs (>= 5000 bp)	1
# contigs (>= 10000 bp)	1
# contigs (>= 25000 bp)	1
# contigs (>= 50000 bp)	0
Largest contig	30 228
Total length	30 228
Total length (>= 0 bp)	30 228
Total length (>= 1000 bp)	30 228
Total length (>= 5000 bp)	30 228
Total length (>= 10000 bp)	30 228
Total length (>= 25000 bp)	30 228
Total length (>= 50000 bp)	0
N50	30 228
N90	30 228
auN	30 228
L50	1
L90	1
GC (%)	55.49
Unaligned	
# fully unaligned contigs	1
Fully unaligned length	30 228
Mismatches	
# N's per 100 kbp	0
# N's	0
Genome statistics	
NG50	-
NG90	-
auNG	196.9
LG50	-
LG90	-

Fig 9. Statistics of Contig

This is an improvement as it struggled for giving reconstructed genome for 5000 reads using Naive method but using k-mer index and weight based filtering, we were able to obtain a genome of 30,000 base pairs.

Conclusion

- Our Implementation for de-Bruijn graphs was not as we expected. We first employed a single Eulerian cycle method using Hierholzer's algorithm and then later changed it to iterative path cover algorithm but that didn't yield expected results. The accuracy of the former was 0.4% but the latter as also produced the same accuracy. Still, it was able to run on full E. coli dataset under 1 minute. SPAdes was able to run on the full dataset under 10 seconds and produced an accuracy of 59.1%
- Future extensions can be employing methods to remove sequencing noise by performing error correction on reads, applying graph simplification techniques like removing tips, bubbles and low-coverage edges into our implementation.
- For String graphs, using BWT reduced the memory requirement significantly and using the k-mer indexing reduced the time complexity which helped us to do reconstruction for over 25000 reads.
- Future extensions can be further simplification of graph and acquiring adequate equipment to do complete genome reconstruction. As we constructed only 1 contig, we can split it to construct multiple contigs further simplifying the problem.
- After it is able to construct genome completely, we can play with the parameters and do comparative study with the De-Bruijn graphs.

Acknowledgments

We used publicly available Illumina sequencing data for *E. coli*, assembled using SPAdes [1], and evaluated the results with QUAST [2]. We thank the developers of these tools and datasets for making them publicly available.

Author Contribution

Yashwanth: Finding Dataset and data preprocessing, construction of De-Bruijn graphs and reconstruction from it, understanding SPAdes Assembler to understand sequencing errors and their rectification, validating using reference genome.

Chaitanya: Understanding string graph principles, construction of string graphs and reconstruction of genome using it, validating using reference genome.

9. GitHub Link

`https://github.com/YASHwanth-1411/CS6024-Comp-Bio`

10. Progress After Presentation

- Modified the Hierholzer's algorithm which tried to find a single Eulerian path in De Bruijn graph to iterative path cover algorithm which finds maximal eulerian paths at each iteration.
- Implemented the naive greedy approach of string graph reconstruction. Tried the Burrow-Wheeler Transform and Kmer Indexing for reducing the complexity/increasing the efficiency.
- Used support based weights for further simplifying the graph. Reconstructed a genome using 25,000 reads.

References

1. Prjibelski A, Antipov D, Meleshko D, Lapidus A, Korobeynikov A. Using SPAdes De Novo Assembler. Current Protocols in Bioinformatics. 2020;70(1):e102. doi:10.1002/cpbi.102.
2. Alekseyev Lab. QUAST: Quality Assessment Tool for Genome Assemblies; 2023. `https://github.com/ablab/quast`.
3. Zhang W, Chen J, Yang Y, Tang Y, Shang J, Shen B. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. PLOS One. 2011;6(3):e17915.
4. Myers EW. The fragment assembly string graph. Bioinformatics. 2005;21(suppl 2):ii79–ii85.
5. Pevzner PA, Tang H, Waterman MS. A new approach to fragment assembly in DNA sequencing. Genome Research. 2001;11(5):789–803. doi:10.1101/gr.171001.
6. Goodwin S, Gurtowski J, Ethe-Sayers S, Deshpande P, Schatz MC, McCombie WR. Oxford nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome. Genome Research. 2015;25(11):1750–1756.

7. Guidi G, Selvitopi O, Ellis M, Oliker L, Yelick K, Buluç A. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2021. p. 517–526.

8. Simpson JT, Durbin R. Efficient de novo assembly of large genomes using compressed data structures. Genome Research. 2012;22(3):549–556.