EN 601.447 Final Project: Wheeler Graphs – Investigating Pan-genome Query Optimization

Group 4d: Andrea Cheng, Nikhil Choudhary, Avery Kuo, Annie Wang

Abstract:

Reference bias, or the tendency for genetic analyses to favor sequences similar to a single linear reference genome, often leads to the misalignment or exclusion of reads. In genomic studies this results in incomplete variant detection and biased genomic studies. In recent years, pangenomes have emerged as a way to address these limitations by capturing the full spectrum of genetic diversity within a species. While great in theory, existing approaches (including the current state of the art) for pangenome analysis exhibit distinct trade-offs in scalability, storage efficiency, and query performance. Therefore, to facilitate greater adoption of pangenomes over traditional linear reference genomes, there is a critical need for novel data structures that can efficiently store and query the vast diversity of genetic information while maintaining computational feasibility at scale. This paper explores the application of Wheeler Graphs to pangenome storage and querying by leveraging their path coherence and compression properties. Wheeler Graphs allow for efficient pattern matching and storage by encoding repetitive sequences and their variations in a succinct, ordered structure. Using a dataset that includes 14 variants of the DMPK gene, a gene with clinically significant repeat expansions, we benchmark the Wheeler Graph implementation against custom implementations of k-mer indexes and suffix trees. Results demonstrate that while our implementation of the Wheeler Graph query algorithm is naive, we believe that a SOTA Wheeler Graph query algorithm offers competitive query performance while significantly reducing memory usage, especially at scale.

Introduction:

In genomics, accurate analysis relies heavily on the quality of reference genomes, with the reference genome serving as a benchmark for variant detection, alignment and other downstream bioinformatic applications. A major limitation of reference genomes is reference bias, where legitimate reads diverging from the reference genome - especially those from underrepresented populations - are prone to misalignment or outright exclusion[1]. GRCh38, the current gold-standard human reference genome which is assembled primarily from individuals of European descent, is particularly prone to reference bias, with some studies demonstrating that up to 10% of the total human genome may be missing from the human reference genome[2]. As a consequence, rare variants present in non-European populations are often not well characterized leading to incomplete disease models, biased population studies, reduced efficacy of precision medicine and even potential overlooking of valuable therapeutic targets[3]. For example, nonsense variants in the PCKS9 gene, significantly more prevalent in individuals of African ancestry (prevalence of 2.6%) are associated with substantial (88%) reductions in coronary heart disease risk. These findings, which were initially overlooked due to their rarity in European populations (prevalence of 0.006%) led to the development of evolocumab, which is a highly effective monoclonal antibody PCKS9 inhibitor[1]. In general, single linear reference genomes are fundamentally limited in their ability to represent the full array of genetic variation (i.e., single nucleotide polymorphisms, insertions, deletions, structural variants, and population-specific sequences) that naturally occur in the human species[4]. In recent years, pangenomes have emerged as promising solution to the challenges posed by reference bias. Pangenomes capture the full spectrum of genetic diversity present in a species via a core genome, highly conserved sequences present in all individuals, accessory genomes, sequences present in subsets of a population, and genetic variants[5]. By incorporating genetic variants into their structure, pangenomes mitigate the exclusion of reads that diverge from a single reference, enabling more comprehensive variant detection. Despite their advantages, pangenomes are still not widely implemented, as scaling existing pangenome data structures (discussed below) to store hundreds of genomes is an intensive task that poses a significant computational burden[6]. We aim to address this issue by investigating the utility of Wheeler Graphs to efficiently store and query pangenomes for the purpose of variant detection.

Wheeler graphs are a special designation given to graphs that allow for a special ordering of their nodes. For a directed, edge-labeled graph to be Wheeler, the following criteria must be met[7]:

1. 0-indegree nodes come before all other nodes in the ordering
2. For all pairs of edges (u,v) and (u',v') labeled a and a' respectively:
    a. $a \prec a' \rightarrow v < v'$
    b. $a = a' \wedge u < u' \rightarrow v \leq v'$

These criteria create a consistent ordering of nodes and edges, which allows for efficient query and pattern matching using rank and select queries, similar to the search algorithm of the BWT-based FM index[7]. This specific type of alphabetical ordering also leads to path coherence. A graph is path coherent if nodes can be ordered such that: For any consecutive range of nodes [i,j] and character c, the nodes reached by following edges matching c also form a consecutive range[7]. By leveraging path coherence, Wheeler Graphs can significantly improve upon the search and pattern matching performance of other graph structures, as all relevant nodes to the search will be clustered together. In addition to performing efficient querying and pattern matching, Wheeler Graphs are also good candidates for addressing the computational and memory resource burdens of storing long, repetitive sequences. This is particularly useful for mammalian organisms where approximately 25-50% of the genome consists of repetitive sequences[8]. In pangenomes, this repetition scales linearly with the number of genomes included in the pangenome[9]. The wheeler graph's path coherence property also allows for run-length encoding and compression[10,11]. Using run length compression reduces the storage requirement of a wheeler graph to a constant factor which scales with the size of the alphabet (i.e., ACTG) instead of the number of genomes. Furthermore, bitvector representations of nodes and edges can further minimize the memory requirement of pangenomic representations. In this study, we build and benchmark a custom Wheeler Graph implementation against a custom Suffix Tree and K-mer index implementation by comparing the time it takes to perform the task of querying a pangenome.

Prior Work:

Various existing data structures have been developed to represent pangenomes, each offering trade-offs in scalability, computational efficiency and query performance. Through an extensive literature review, we categorized existing approaches into five distinct archetypes (summarized in Supplementary Figure 1):

Multiple Sequence Alignment (MSA): The most trivial representation of a pangenome leverages multiple sequence alignment to align multiple linear reference genomes. This approach introduces gap characters to create a $G \times L$ matrix where G is the number of reference genomes and L is the length of the reference genomes. Each column in the matrix corresponds to homologous positions across the aligned genomes[31]. While there are some advantages to using such approaches (i.e., support for a variety of comparison tasks), these are largely outweighed by high computational costs and inability to capture more complex genetic variants (i.e., inversions and translocations). As such, even the most efficient methods (i.e., MAFFT, MUSCLE, or CLUSTALW) are limited to short genomic regions or highly similar genomes[32].

K-mer Approaches: These approaches break input genomes into fixed-length k-mers. The size of these k-mers is flexible; however a size of 31 is often chosen to limit the rate of non-unique k-mers[33]. Additionally, size 31 kmers have been shown to effectively capture variation in prior studies[34]. In general, these approaches use data structures that are able to store and query the presence, frequency, and distribution of k-mers across genomes. The choice of data structure is variable in the approaches we looked at: PanKmer uses compact hash tables, KMC uses disk-based data structures, BFCounter uses Bloom filters and Jellyfish uses lock-free hash

tables. These data structures are typically constructed by iterating over input sequences to extract all k-mers, which are then inserted into the choice of data structure with associated counts or presence/absence flags[19]. Like MSA based approaches, k-mer based methods struggle to represent larger structural variants due to their local, context-independent nature and require significant memory and other computational resources for constructing and querying k-mer indices. As such, even with efficient algorithms, like sliding windows for k-mer extraction and optimized hash functions, these approaches are not suitable for large-scale pangenomes[35].

Suffix Trees: Suffix tree representations generally allow for efficient indexing and exact pattern matching by organizing all suffixes of a string into a hierarchical tree structure. Each edge represents a substring in the input genomes and each path from the root to a leaf corresponds to a suffix in the input genomes. No explicit suffix-tree representations have been developed for large-scale pangenomic analysis, but generalized suffix trees (GSTs) have been used to integrate suffixes from multiple genomes into a single tree. Some existing tools to create GSTs include MUMmer[23] and the GenomeTools suite[24]. The primary advantage of using such representations is query efficiency, where presence/absence of a subsequence can be performed in O(m) time where m is the query length. However, this is often not enough to overcome memory limitations, as the storage of nodes, edges and suffix links grows exponentially with the size and number of input genomes. Given that constructing a suffix tree for a single human genome is known to require gigabytes of memory, any pangenome representation is likely computationally infeasible[36].

De Bruijn Graphs: De Bruijn graphs are an improvement on the simpler k-mer based approaches by encoding sequences as nodes (representing unique k-mers) connected by edges that reflect shared overlaps of length k-1[31]. In a pangenome context, each unique k-mer across the entire set of genomes is encoded as a node, and edges still denote overlapping k-mers (of length k-1)[31]. In practice, "colored" or "labeled" De Bruijn graph can be used to represent pangenomes, where each node is annotated with information about which genomes contributed the associated k-mer[37]. Implementations of DeBruijn Graphs tend to be scalable and efficient to construct, making them good candidates to represent pangenomes. However, they can be difficult to interpret and have limited utility in downstream applications except for presence/absence queries[6].

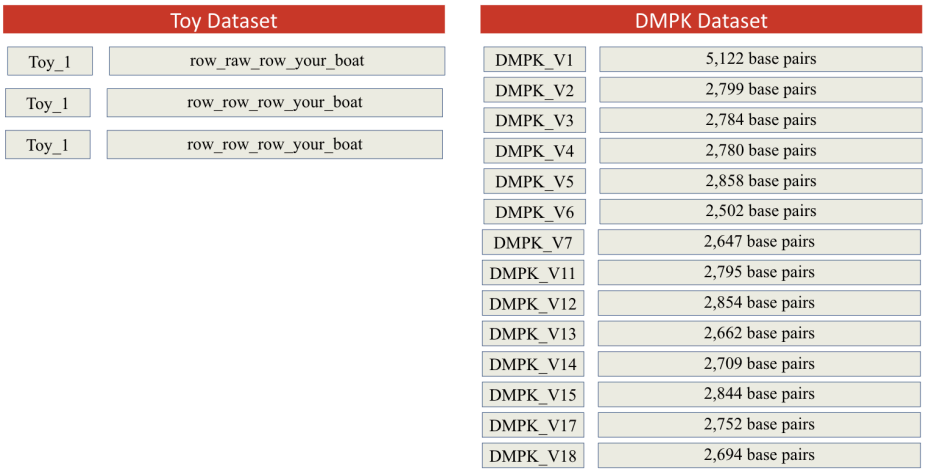
Variation Graphs: The current state of the art for pangenome representations are variation graphs. They represent pangenomes by directly encoding genomic variation into a unified graph structure built from whole-genome alignments rather than isolated k-mers[31]. In this representation, nodes correspond to genomic segments and edges link these segments to reflect the variation in sequence order and content across multiple haplotypes[29,30]. Implementations of variation graphs tend to be excellent for explicitly modeling all of the complex genetic variants present in human pangenomes, are efficient for variant detection applications and are highly interpretable[6,38]. However, constructing and maintaining variation graphs can be computationally intensive, leaving room for improvement[6].

<u>Methods and Software:</u>

Datasets

For our analysis, we tested each implementation on two datasets. The first dataset, a toy dataset, served as a baseline to verify the correctness of each implementation. While it allowed us to confirm that the implementations behaved as expected, its limited size was insufficient for any meaningful assessment of computational performance or scalability. To better examine query and computational efficiency, we selected a more biologically relevant target: the human DMPK gene. This gene is characterized by expansions of CTG repeats and is clinically relevant because these expansions cause myotonic dystrophy, a devastating genetic condition. The presence of variable-length repeats and structural complexity make it an ideal test case for

benchmarking the capabilities of our different implementations, and we hypothesize that the Wheeler graph's storage of variants will prove to be demonstrably more efficient in a genome of this structure. We obtained 14 DMPK variants[39-52] from the NCBI database ranging from 2,502 - 5,122 base pairs. We also considered expanding our analysis to entire microbial or yeast genomes, but found that the results derived from the DMPK variants were sufficient to highlight meaningful differences and characterize our benchmarking efforts.



Toy Dataset	
Toy_1	row_raw_row_your_boat
Toy_1	row_row_row_your_boat
Toy_1	row_row_row_your_boat

DMPK Dataset	
DMPK_V1	5,122 base pairs
DMPK_V2	2,799 base pairs
DMPK_V3	2,784 base pairs
DMPK_V4	2,780 base pairs
DMPK_V5	2,858 base pairs
DMPK_V6	2,502 base pairs
DMPK_V7	2,647 base pairs
DMPK_V11	2,795 base pairs
DMPK_V12	2,854 base pairs
DMPK_V13	2,662 base pairs
DMPK_V14	2,709 base pairs
DMPK_V15	2,844 base pairs
DMPK_V17	2,752 base pairs
DMPK_V18	2,694 base pairs

Figure 2: Dataset Structures. Individual sequences are concatenated into a dictionary structure where keys are the file name and values are the FASTA sequences.

Kmer Index

Implementation: We built a k-mer index to allow rapid lookups of genomic variants without relying on expensive full-sequence scans for each query. Each unique 31-mer (choice of k-mer size explained in the Prior Work section) serves as a key to a hash-based mapping data structure (i.e., a python dictionary). The associated value is a list of tuples, with each tuple containing the k-mer's starting position and an identifier for one of the genomes in which it occurs. This allows us to quickly determine the presence or absence of any given k-mer in the dataset, as well as pinpoint its exact location.

Variant Detection Algorithm: Break read into 31-mers. Check to see if all overlapping 31-mers exist in at least one genome. If the read exists in all genomes, it is likely to be part of a highly conserved sequence (i.e., core genome). If the read exists in a subset of the genomes, it is likely to be part of a variant sequence. If it exists in no genomes, it is likely a sequencing error.

Suffix Tree

Custom Implementation: We build the suffix tree by iteratively inserting every suffix from each genome in the dataset to the tree based on their prefixes. For each suffix, we traverse the tree from the root, comparing prefixes of the suffix to existing edges. If a partial match is found along an edge, we introduce an internal node to split it and insert the remainder of the suffix. If no match is found, we create a new leaf node. Internal nodes represent common prefixes of multiple suffixes, and leaf nodes mark endpoints of individual suffixes. Each node stores a list of sequence identifiers and the corresponding start indices of the suffixes that pass through it, enabling us to map every substring in the tree back to its original genomic context.

Specific Use and Comparison with STree[53]: There are several ways to construct suffix trees in order to align with user goals (query efficiency, counting efficiency, special purposes, ease of understanding etc). To compare

ours with a well-documented implementation, we used STree from the suffix-tree library. Our implementation later proved to be more efficient in query time for longer substrings, which we prioritized in this project. However, STree is much more efficient memory-wise, and it's better at counting variants whereas our tree is ill-equipped to count variants in large datasets without profusely straining memory capacity.

Variant Detection Algorithm: To identify variant nodes we compare the node's sequence IDs against both the parent node's sequence IDs and the full set of input sequence IDs. If the node's set of IDs differs from both the parent's and the universal set of all sequences, the node is designated as a variant node. We then apply this logic recursively to the node's children, continuing down the tree for all hierarchical levels.

DeBruijn + Wheeler Graphs

Implementation: To build a Wheeler Graph, we first build a De Bruijn Graph. To do this, we use a modified version of a prewritten script to build a De Bruijn Graph[54]. First, all k-mers from our input sequences are extracted for a chosen k. Each unique (k-1)-mer becomes a node, and edges are drawn from one node to another whenever a k-mer connects them. Essentially, if a k-mer is formed by appending a character c to a (k-1)-mer X, it points to the (k-1)-mer Y formed by shifting one character forward. The edge label of the edge which connects the (k-1)-mer nodes X and Y is the last character of (k-1)-mer Y, the character one character after the (k-1)-mer X. After constructing this graph, we export it as a .dot file and run it through the Wheelie package[55]. If the resulting graph meets the requirements for a Wheeler graph, Wheelie produces the Gagie structure (I, O, L, and Nodes arrays), creating a more compact and efficiently searchable representation. We also attempted to modify the Succinct de Bruijn Graphs implementation[56], which alphabetically orders nodes and creates a FM-index inspired structure. However, we struggled to confirm that the graph was indeed wheeler as we moved from the toy pangenome to larger datasets, and we had a better understanding of the query algorithm described for the Gagie structure[57], which is the pattern matching algorithm described below and implemented.

Pattern Matching Algorithm: Because a custom implementation of the Wheeler Graph was not built, we were unable to implement a way to keep track of counts of occurrences in Wheeler Graph. The algorithm searches for patterns in the pangenome via rank and select functions on 4 data structures. The L string stores edge labels in graph order. The I and O strings are bit vectors for incoming and outgoing edges, respectively. Finally, a dictionary called C is created to map the starting and ending indices of a string of alphabetically ordered edge labels. Before performing rank and select operations, we also extract positions of 0's and 1's in I and O, where 0's represent the edges and 1's represent nodes. The rank and select functions are used to navigate between nodes and their corresponding outgoing and incoming edges. The rank functions count the number of 1's or 0's in a bit vector given a specific range. The select functions retrieve the positions of 1's or 0's to specify a range of nodes/edges. The backward_search method uses rank and select operations to search for a pattern P in the graph. It iterates over P in reverse and determines whether the pattern exists by narrowing down ranges of edge labels and corresponding nodes. The select operation is performed on I to find the range of incoming edges which correspond to the last character in P. Then the rank operation is used to refine the search and find the rank of the nodes represented by 1's in I that are within and just bordering the range of 0's, essentially finding which nodes the edges lead to. Next, the select operation is performed on O to find the range where the outgoing edges corresponding to these nodes will be located. Then, the rank operation is used to find the rank of these edges, which is used to find corresponding outgoing edge labels in L. Within these labels, if the next character in P is found, the range of nodes and edges to search for the next character is narrowed. This series of operations is repeated until the entire pattern is found or until no valid range of characters in L is found.

Benchmarking

To assess the relative performance of the Wheeler graph, we compared it against more conventional methods like the k-mer index and suffix tree. We used Python's time and tracemalloc libraries to benchmark both time efficiency and memory usage for each method on two datasets (toy and DMPK). For time efficiency, we tested increasing query lengths (with a minimum length of 31). The query sequences used are known to be present in our pangenome, as this represents the worst-case scenario. In the worst case, if a sequence is valid, each implementation must examine every character, which ensures that the time taken reflects the full processing requirement. For memory usage, we tracked peak memory consumption to build each implementation. While the smaller dataset ran smoothly, a challenge we ran into was the scalability of our custom implementations – especially that of the suffix tree for counting variants in larger genomes. To address this case, we borrowed from the PyPI suffix-tree library[53] to benchmark the larger DMPK genome. In the process of benchmarking, we discovered a bug in the implementation of the pattern matching algorithm used for the Wheeler Graph, which we were unable to resolve, resulting in an inability for the algorithm to find certain strings that are present in the pangenome. For the purpose of benchmarking, we identified strings that the algorithm is able to identify, and the results shown below are obtained from benchmarking each implementation with these strings.

Results:

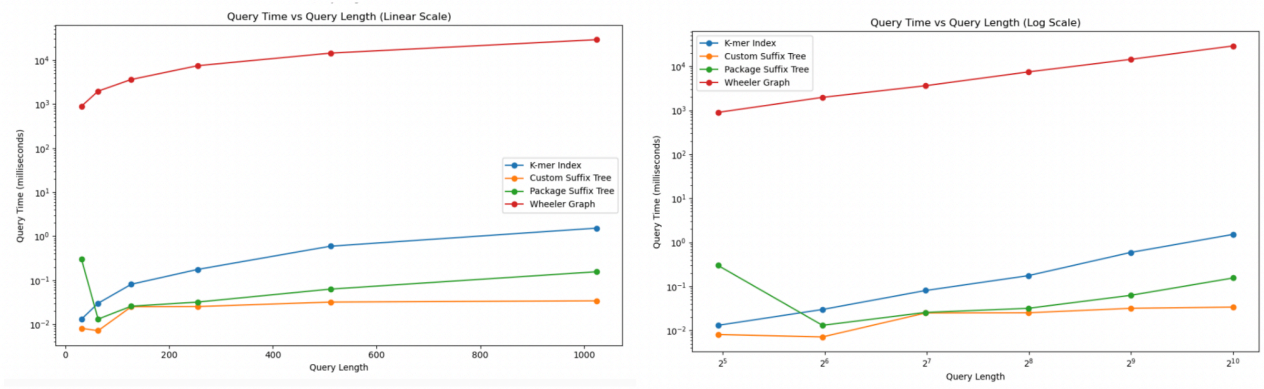


Figure 3: Time Benchmarking Results for DMPK Dataset. Each implementation was run on the same device to ensure valid comparisons. The left plot uses a linear scale, while the right plot uses a logarithmic scale for both axes. Each curve represents a different implementation: K-mer Index (blue), Custom Suffix Tree (orange), Package Suffix Tree (green), and Wheeler Graph (red). These comparisons highlight the varying time efficiencies of the indexing methods with increasing query lengths.

Method	Time to Build (in seconds)	Peak Memory Usage (in MB)
K-Mer Index	0.152s	8.604214 MB
Suffix Tree (Custom)	140.080s	243.488984 MB
Suffix Tree (STree)	1.124s	40.855962 MB
Wheeler Graph	19.899s	27.217067 MB

Figure 4: Memory Benchmarking Results for DMPK Dataset (using 31-mers). Each implementation was run on the same computer to ensure valid comparisons. The table shows the time taken to build each index and the peak memory usage for four different indexing methods: K-mer Index, Suffix Tree (Custom Implementation), Suffix Tree (STree Implementation), and Wheeler Graph.

Conclusions:

No free lunch: Between the traditional methods, the data illustrate the tradeoffs of each structure. The k-mer index's build and query times are the fastest and use the least memory initially, but we observed that it exhibits poor scalability when dealing with queries of length greater than k. The time to query grows linearly with query length because the k-mer index does not have any methods to compact substrings greater in length than k and instead must split longer sequences into k=31-mers, match them against the index individually. This leads to redundant operations when processing overlapping segments and becomes computationally more and more expensive as the length in sequencing reads increases, which is evident in Figure 3 as the query time grows linearly as the length increases. On the other hand, suffix trees offer a more computationally expensive (initially) and comprehensive approach as evidenced by longer build/query times for 31-mers, but they process higher query substring lengths at a more reasonable rate. The suffix tree itself takes a significant amount of time to build initially – far greater than that of the kmer index – but because of this structure, it has greater scaling capacity as query length increases as demonstrated in Figure 3. The upfront cost of constructing the tree can be a barrier for use in time-sensitive applications, and it uses much more memory to store the tree itself and process it, but it is best suited for use cases involving variable or long query lengths, or for large datasets where we expect to make many queries.

Wheeler graph reflection: In absolute terms, the Wheeler graph performed poorly, needing several times the query time to process queries, which was consistently inefficient for increasing length queries. The memory efficiency surpassed that of the suffix trees but was behind that of the k-mer index (although this may be due to the memory overhead required to first construct the Debruijn graph). This is somewhat suboptimal as to what we hoped to see; previous research has presented Wheeler graphs as demonstrating outstanding memory performance while offering competitive query efficiency, especially in repeat-heavy pangenomes. However, given that the Wheeler graph searching algorithm is not as well-documented, we expect our own implementation to have flaws that lead to poorer performance. For example, our implementation of rank and select was naive instead of using the optimal data structures such as the Wavelet Tree Structure for performing rank queries on the L string. Even so, benchmarked against traditional k-mer index and suffix tree, our implementation of the Wheeler graph exhibits a higher starting point for query time for shorter substring but is comparatively faster with longer substrings; that is, although the Wheeler graph takes much longer to query in general, the time to query increases very slowly as length of query increases. This makes it very suitable for large datasets and long substrings, especially those that can be compactly searched for in the graph, and provides support for our hypothesis that its underlying design principles suggest a significant improvement in both time and memory efficiency compared to traditional suffix trees.

Using a SOTA searching algorithm and data structure implementation, we would expect better results. Memory efficiency would also likely achieve superiority over traditional methods as our process must construct the Debruijn graph, run Wheelie, and construct the Wheeler graph while there are other theoretical documented methods to construct the graph in a more direct way. This would allow us to fully leverage the excellent storage properties of the underlying automata. Most importantly, we expect the expert query algorithm to take advantage of much more suitable data structures and search in a much more efficient manner, implying an improvement of magnitude upon our searching algorithm. Since scalability was exhibited in the benchmarking through substrings of length 1023, we hope that the Wheeler graph's compact representation and ability to encode pangenomes in a scalable manner mark it as a promising tool for large-scale genomic queries.

Appendix

Contributions

Andrea referenced various sources to modify the implementation of the De Bruijn graph, used the Wheeler Graph Toolkit Package to convert the De Bruijn Graph into Wheeler graph structure, coded a pattern matching algorithm based on the Wheeler Graph Structure, then wrote various scripts to allow all parts of the implementation to be run seamlessly from one command line script. Avery built the k-mer index structure to represent pangenomes, investigated ways to detect variation, and created a variation detection algorithm for the structure based on past literature review. She developed benchmarking standards for all structures. Using these standards, she created scripts to generate benchmarking data and results from all three structures so they can all be run consistently with the same command line. Annie built the custom suffix tree structure to represent pangenomes and constructed the proprietary variation detection algorithm optimized for longer queries with variation nodes defined based on consensus definition. She also wrote substring processing scripts to determine and consolidate the data used for the benchmarking setup and iteratively processed the benchmarking results. Nikhil conducted prior literature review, investigated prior implementations and other methods to represent pangenomes, and assembled deliverables (i.e., writeup, references, slides). He also created the final visualizations and figures in the paper. We all assisted each other when we ran into bugs and discussed potential issues/solutions and the writeup.

Supplementary Figures

Data Structure	Storage Efficiency	Query Efficiency	Existing Implementations
Multiple Sequence Alignment	Low	Low	MAFFT[12]; Muscle[13]; CLUSTALW[14]; DIALIGN[15]; PROBCONS[16]; ProDA[17], T-COFFEE[18]
K-mer Indexes	Low	Medium	PanKmer[19];KMC[20]; BFCounter[21]; Jellyfish[22];
Suffix Trees	Low	High	Mummer[23]; GenomeTools[24]
De Bruijn Graphs	High	Low	mdBG[25]; Bifrost[26]
Variation Graphs (SOTA)	Medium	High	vg[27]; PGGB[28]; Minigraph[29]; Minigraph-Cactus[30]

Supplementary Figure 1: Summary of Existing Data Structures to Represent Pangenomes

References

1. Bentley, A. R., Callier, S., & Rotimi, C. N. (2017). Diversity and inclusion in genomic research: Why The uneven progress? *Journal of Community Genetics*, *8*(4), 255–266. https://doi.org/10.1007/s12687-017-0316-6
2. Sherman, R.M., Salzberg, S.L. Pan-genomics in the human genome era. Nat Rev Genet 21, 243–254 (2020). https://doi.org/10.1038/s41576-020-0210-7
3. Chen, NC., Solomon, B., Mun, T. et al. Reference flow: reducing reference bias using multiple population genomes. Genome Biol 22, 8 (2021). https://doi.org/10.1186/s13059-020-02229-3
4. Chelsea A Matthews, Nathan S Watson-Haigh, Rachel A Burton, Anna E Sheppard, A gentle introduction to pangenomics, Briefings in Bioinformatics, Volume 25, Issue 6, November 2024, bbae588, https://doi.org/10.1093/bib/bbae588
5. Caudal, É., Loegler, V., Dutreux, F. et al. Pan-transcriptome reveals a large accessory genome contribution to gene expression variation in yeast. Nat Genet 56, 1278–1287 (2024). https://doi.org/10.1038/s41588-024-01769-9
6. Andreace, F., Lechat, P., Dufresne, Y. et al. Comparing methods for constructing and representing human pangenome graphs. Genome Biol 24, 274 (2023). https://doi.org/10.1186/s13059-023-03098-2
7. Gagie, T., Manzini, G., & Sirén, J. (2017). Wheeler graphs: A framework for BWT-based data structures. Theoretical computer science, 698, 67–78. https://doi.org/10.1016/j.tcs.2017.06.016
8. Liao, X., Zhu, W., Zhou, J. et al. Repetitive DNA sequence detection and its role in the human genome. Commun Biol 6, 954 (2023). https://doi.org/10.1038/s42003-023-05322-y
9. Teaching materials. Langmead Lab @ JHU - Teaching Materials. (n.d.). https://www.langmead-lab.org/teaching.html
10. Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, Giovanna Rosone, Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform, Bioinformatics, Volume 28, Issue 11, June 2012, Pages 1415–1419, https://doi.org/10.1093/bioinformatics/bts173
11. Kuhnle, A., Mun, T., Boucher, C., Gagie, T., Langmead, B., & Manzini, G. (2020). Efficient Construction of a Complete Index for Pan-Genomics Read Alignment. Journal of computational biology : a journal of computational molecular cell biology, 27(4), 500–513. https://doi.org/10.1089/cmb.2019.0309
12. Multiple alignment program for amino acid or nucleotide sequences. MAFFT. (n.d.). https://mafft.cbrc.jp/alignment/software/
13. Edgar, R.C. MUSCLE: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics 5, 113 (2004). https://doi.org/10.1186/1471-2105-5-113
14. Sievers, F., Wilm, A., Dineen, D., Gibson, T. J., Karplus, K., Li, W., Lopez, R., McWilliam, H., Remmert, M., Söding, J., Thompson, J. D., & Higgins, D. G. (2011). Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. Molecular systems biology, 7, 539. https://doi.org/10.1038/msb.2011.75
15. Layal Al Ait, Zaher Yamak, Burkhard Morgenstern, DIALIGN at GOBICS—multiple sequence alignment using various sources of external information, Nucleic Acids Research, Volume 41, Issue W1, 1 July 2013, Pages W3–W7, https://doi.org/10.1093/nar/gkt283
16. Do, C. B., Mahabhashyam, M. S. P., Brudno, M., & Batzoglou, S. (2005). Probcons: Probabilistic consistency-based multiple sequence alignment. Genome Research, 15(2), 330–340. https://doi.org/10.1101/gr.2821705

17. Tu Minh Phuong, Chuong B. Do, Robert C. Edgar, Serafim Batzoglou, Multiple alignment of protein sequences with repeats and rearrangements, Nucleic Acids Research, Volume 34, Issue 20, 1 November 2006, Pages 5932–5942, https://doi.org/10.1093/nar/gkl511
18. Notredame, C., Higgins, D. G., & Heringa, J. (2000). T-Coffee: A novel method for fast and accurate multiple sequence alignment 1 1edited by J. Thornton. Journal of Molecular Biology, 302(1), 205–217. https://doi.org/10.1006/jmbi.2000.4042
19. Anthony J Aylward, Semar Petrus, Allen Mamerto, Nolan T Hartwick, Todd P Michael, PanKmer: k-mer-based and reference-free pangenome analysis, Bioinformatics, Volume 39, Issue 10, October 2023, btad621, https://doi.org/10.1093/bioinformatics/btad621
20. Marek Kokot, Maciej Długosz, Sebastian Deorowicz, KMC 3: counting and manipulating k-mer statistics, Bioinformatics, Volume 33, Issue 17, September 2017, Pages 2759–2761, https://doi.org/10.1093/bioinformatics/btx304
21. Melsted, P., Pritchard, J.K. Efficient counting of k-mers in DNA sequences using a bloom filter. BMC Bioinformatics 12, 333 (2011). https://doi.org/10.1186/1471-2105-12-333
22. Guillaume Marçais, Carl Kingsford, A fast, lock-free approach for efficient parallel counting of occurrences of k-mers, Bioinformatics, Volume 27, Issue 6, March 2011, Pages 764–770, https://doi.org/10.1093/bioinformatics/btr011
23. Marçais, G., Delcher, A. L., Phillippy, A. M., Coston, R., Salzberg, S. L., & Zimin, A. (2018). MUMmer4: A fast and versatile genome alignment system. PLoS computational biology, 14(1), e1005944. https://doi.org/10.1371/journal.pcbi.1005944
24. Gremme, G., Steinbiss, S., & Kurtz, S. (2013). GenomeTools: a comprehensive software library for efficient processing of structured genome annotations. IEEE/ACM transactions on computational biology and bioinformatics, 10(3), 645–656. https://doi.org/10.1109/TCBB.2013.68
25. Ekim, B., Berger, B., & Chikhi, R. (2021). Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. Cell Systems, 12(10). https://doi.org/10.1016/j.cels.2021.08.009
26. Holley, G., Melsted, P. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. Genome Biol 21, 249 (2020). https://doi.org/10.1186/s13059-020-02135-8
27. Garrison, E., Sirén, J., Novak, A. et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat Biotechnol 36, 875–879 (2018). https://doi.org/10.1038/nbt.4227
28. Garrison, E., Guarracino, A., Heumos, S. et al. Building pangenome graphs. Nat Methods 21, 2008–2012 (2024). https://doi.org/10.1038/s41592-024-02430-3
29. Li, H., Feng, X. & Chu, C. The design and construction of reference pangenome graphs with minigraph. Genome Biol 21, 265 (2020). https://doi.org/10.1186/s13059-020-02168-z
30. Hickey, G., Monlong, J., Ebler, J. et al. Pangenome graph construction from genome alignments with Minigraph-Cactus. Nat Biotechnol 42, 663–673 (2024). https://doi.org/10.1038/s41587-023-01793-w
31. The Computational Pan-Genomics Consortium , Computational pan-genomics: status, promises and challenges, Briefings in Bioinformatics, Volume 19, Issue 1, January 2018, Pages 118–135, https://doi.org/10.1093/bib/bbw089
32. Edgar, R. C., & Batzoglou, S. (2006). Multiple sequence alignment. Current Opinion in Structural Biology, 16(3), 368–373. https://doi.org/10.1016/j.sbi.2006.04.004
33. Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, Sandra Smit, PanTools: representation, storage and exploration of pan-genomic data, Bioinformatics, Volume 32, Issue 17, September 2016, Pages i487–i493, https://doi.org/10.1093/bioinformatics/btw455

34. Rahman, A., Hallgrímsdóttir, I., Eisen, M., & Pachter, L. (2018). Association mapping from sequencing reads using K-MERS. eLife, 7. https://doi.org/10.7554/elife.32920
35. Fan, J., Khan, J., Singh, N.P. et al. Fulgor: a fast and compact k-mer index for large-scale matching and color queries. Algorithms Mol Biol 19, 3 (2024). https://doi.org/10.1186/s13015-024-00251-9
36. Kurtz, S., Phillippy, A., Delcher, A.L. et al. Versatile and open software for comparing large genomes. Genome Biol 5, R12 (2004). https://doi.org/10.1186/gb-2004-5-2-r12
37. Iqbal, Z., Caccamo, M., Turner, I. et al. De novo assembly and genotyping of variants using colored de Bruijn graphs. Nat Genet 44, 226–232 (2012). https://doi.org/10.1038/ng.1028
38. Eizenga, J. M., Novak, A. M., Sibbesen, J. A., Heumos, S., Ghaffaari, A., Hickey, G., Chang, X., Seaman, J. D., Rounthwaite, R., Ebler, J., Rautiainen, M., Garg, S., Paten, B., Marschall, T., Sirén, J., & Garrison, E. (2020). Pangenome Graphs. Annual review of genomics and human genetics, 21, 139–162. https://doi.org/10.1146/annurev-genom-120219-080406
39. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 1, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
40. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 2, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
41. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 3, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
42. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 4, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
43. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 5, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
44. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 6, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_001288764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_001424165,NM_001424166,NM_001424168,NM_001424169,NM_004409
45. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 7, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128

8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409

46. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 11, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
47. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 12, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
48. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 13, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
49. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 14, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
50. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 15, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
51. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 17, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
52. U.S. National Library of Medicine. (n.d.). Homo sapiens DM1 protein kinase (DMPK), transcript variant 18, mrna. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/nuccore/NM_001081560,NM_001081562,NM_001081563,NM_00128
8764,NM_001288765,NM_001288766,NM_001424162,NM_001424163,NM_001424164,NM_0014
24165,NM_001424166,NM_001424168,NM_001424169,NM_004409
53. Suffix-tree documentation. suffix-tree Documentation - suffix-tree 0.1.2 documentation. (n.d.). https://cceh.github.io/suffix-tree/
54. Langmead, B. (n.d.). Benlangmead/Comp-Genomics-class: Code and examples for JHU Computational Genomics class. GitHub. https://github.com/BenLangmead/comp-genomics-class/tree/master
55. Chao, K. H., Chen, P. W., Seshia, S. A., & Langmead, B. (2023). WGT: Tools and algorithms for recognizing, visualizing, and generating Wheeler graphs. iScience, 26(8), 107402. https://doi.org/10.1016/j.isci.2023.107402

56. Bowe, A. (2023, July 31). *Succinct de bruijn graphs*. alexbowe.com. https://www.alexbowe.com/succinct-debruijn-graphs/
57. Langmead, B. (n.d.-b). *Wheeler graphs, part 5: Data structures*. *Langmead Lab*. Retrieved 2024, from https://www.cs.jhu.edu/~langmea/resources/lecture_notes/bwt/278_wheeler_graph_5.pdf.