Building and visualizing colored, compressed de Bruijn graphs for pangenomic representation

Anika Misra, Brian Rui, Cindy Zhang, and Lambert Kober

Abstract

In this project, we attempt to build a compressed colored De Bruijn graph for pan-genomic sequences. Due to the multi-genomic nature of the input data and the size of the sequences, it is important that these data structures are efficiently represented. For this reason, we will look at and implement succinct representations of the colored De Bruijn graphs, specifically for pan-genomic applications.

Keywords: De Bruijn graphs, visualization, compression

Introduction

The colored De Bruijn graph and its succinct and compressed versions are applicable for representing and storing pan-genomic sequences. A pan-genome is a practical model that captures the full set of genomic elements that can be applied to a dataset such as a given species and represent inherent variability of the genome through direct non deterministic relationships. Traditional, reference-based genomes that use a consensus model-based approach can introduce biases and are not necessarily an accurate reference for comparison [Einzenga]. Beyond interest in the capabilities of a pan-genome, we recognized that versions of the colored De Bruijn graphs are still a relatively new field of research with a number of recent developments. We hope to explore these topics further and build on the existing contemporary literature.

Our approach is based around two papers by Bowe *et al.* (2012) and Muggli *et al.* (2017). Bowe et al. introduces a succinct classical De Bruijn graph that uses an edge-BWT-based representation to efficiently encode edges and nodes [Bowe]. Muggli *et al.* would further extend this succinct model to colored De Bruijn graphs via the addition of an efficient representation of a color matrix [Muggli]. The algorithm proposed by Muggli *et al.* will be the primary focus for our implementation of the succinct colored De Bruijn graph since it is currently one of the most efficient encodings of colored De Bruijn graphs. Other approaches that have been proposed seem to be largely variations or extensions to the framework and algorithms laid out by Muglie and Bowe *et al* [Almodaresi].

Prior Research

A De Bruijn graph is a directed graph that encodes overlaps between different sequences. Classical De Bruijn graphs, introduced in 1995, promised faster sequence assembly via the use of "oligomers"—length k substrings of the input sequences that are represented as vertices in a directed graph with edges marking occurrences of specific k-mers [Idury]. Due to algorithms for compression of these De Bruijn graphs and efficient storage mechanisms for encodings of sequencing reads, De Bruijn graphs became widely utilized in the field of bioinformatics for de novo genome assembly and variance representation across individuals and populations.

Colored De Bruijn graphs, which associates each edge to a color conditional on the sample it originated from, served as an extension to this model to represent multi-strain inputs. In a colored De Bruijn graph, the edges are colored according to the DNA strain from which they originated, allowing for the identification of shared genetic sequences between different strains [Iqbal]. However, the introduction of distinct, colored edges requires immense memory overhead as the number of strains represented increases, and conventional succinct structures and compression algorithms that existed for classical De Bruijn graphs were not easily applicable to the additional color structures [Muggli].

A number of approaches have been proposed to efficiently represent colored De Bruijn graphs with the main focus of these representations being succinct or compressed ways to encode the edge colors in this active area of research. Cortex, the "first de novo assembler capable of assembling multiple eukaryotic genomes simultaneously", utilizes a hash-table-based approach to efficiently represent colored De Bruijn graphs [Iqbal]. The approach offered a largely linear algorithmic complexity but required significant resources to build and store the graph [Iqbal]. The space requirement was addressed by VARI, a novel generalization of the succinct colored data structure for classical De Bruijn graphs by including a succinct color matrix [Muggli]. VARI relies on the previously defined succinct BOSS representation of a graph $G = (V, E)$ by storing node and edge information in three succinct structures–a Burrows-Wheeler Transform for edges of the graph, a bit vector for the number of incoming edges for nodes, and a bit vector for the number of outgoing edges for nodes–and one additional two-dimensional binary array to indicate the presence of an edge in a subgraph for color. Beyond the utilization of Burrow-Wheeler transform having independent theoretical importance, the succinct structure enables many operations for construction to be performed in O(1) time. VARI was later refined into VariMerge to support building large updatable colored De Bruijn graphs via merging. The innovation with VariMerges lies in how merging of a new graph can directly be merged into existing multigraphs represented by the BOSS structures and the color matrix simply converting the new graph into the accepted structure without having to adapt the existing multigraph. This is significant for creation and scalable updating of large scale datasets [Muggli].

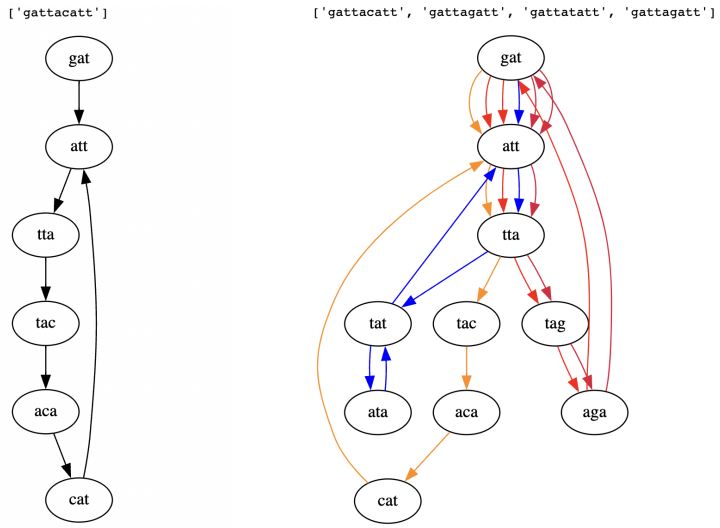
Building on the VARI approach, Almodaresi *et al.* proposed Rainbowfish. This would use equivalence relations to further reduce the space requirement for color information in the existing VARI model. They claim to achieve up to a 20x improvement in space performance with space requirements now being close to the theoretical optimum [Almodaresi]. We also looked into graphical representations of pan-genomes outside of De Bruijn graphs. One such approach is the bloom filter trie which uses a bloom filter to reduce space requirements and a trie to support efficient string matching [Holley]. There also exists a whole other class of graphical representations called 'Variation Graphs' which aim to represent variance across genomic sequences though these seem to be a far more specialized approach [Eizenga].

Methods and Software

Python Visualization Library

As part of our preliminary research and to help build a framework for both the classical and colored De Bruijn graphs, we built a simple visualization library in Python. This would serve to familiarize our team to the graph structures, assist in debugging our succinct algorithm, and support our full-stack visualization. The library was built on top of **NetworkX** and **Graphviz**. NetworkX was used to construct the individual graphs and PyGraphviz was used for visualization purposes due to the improved flexibility it provides on top of NetworkX.

We had several issues with running this in Google Colab due to a bug in a past version of PyGraphviz. Thus, this needs to be run locally on the most up-to-date version of PyGraphviz in order to fully visualize the multigraphs. Additionally, we originally intended to explore Wheeler graphs as well thus our library has some initial work in generating and displaying Wheeler graphs from a sequence of inputs.



Outputs from visualization library: Classical (left) and colored De Bruijn graph (right)

The library can handle sequences of 1 million nucleotides in less than 10 seconds on a CPU-bound system. However, visualizing full genome sequences created graphs were difficult to interpret due to the large number of edges and was addressed in our full-stack visualization.

Python Graph Generation

To build a colored De Bruijn graph (henceforth referred to as a cDBG) that the web application (described in the next section) could properly visualize, we had to perform more rigorous processing of our genomic data for a more structured output. To do this, we began by creating some basic classes, including Edge, Node, and Graph, for an object-oriented approach to cDBG creation. At the core of the graph, we created a hash structure to maintain all of the nodes, and a list of edges in association. We began by building, in the most naive method possible, a single DBG of a single input genome. We initially chose a kmer size of 31 as is often standard, and iterated over the whole genome, adding unique nodes and links. This served as our initial "testing the waters" so to speak, and after ensuring that the runtime of a naive build was not egregious (e.g., over one minute), we pursued this structure to build upon.

After building a single-genome DBG, we added genomes of additional related strains. In constructing the overall pangenome graph, it was essentially a merging of each individual DBG. Suppose we have a graph representing strains {1, 2, 3, 4, 5}. Regarding coloring, there are then 31 possible colors in the graph, each representing some subset of strains. While we chose not to color edges, we colored nodes according to which strain(s) the kmer (and later sequence) represented by a given node were present in. Another step we took to reduce the number of edges

was to ensure that given any node, every edge label exiting should be unique. As is discussed further in later sections, the number of edges between nodes rapidly became a bottleneck in terms of our ability to adequately visualize the graphs in the web application, which is why this step was taken.

After building the cDBG of our pangenome, we next sought out to compress it. While testing with five strains of the bacteria *Mycoplasma genitalium*, we used this to monitor our progress. Prior to edge reduction, there were ~600,000 nodes present, and ~2.3 million edges. After edge reduction, the number of nodes was unchanged, but the number of edges reduced to ~600,000. Then, after our compression step, there were ~12,000 nodes present, and ~16,000 edges, a significant decrease in memory consumption. In order to compress, our method first builds the uncompressed graph. Although not the most sophisticated method, a future consideration could be to develop a method to build the compressed graph from onset. After building the uncompressed version, we then explore paths within the graph for which between a pair of any two branching nodes, there exists a path of unbranching nodes. These paths were collapsed into a single node, introducing variable-length nodes rather than k-length nodes. This compression step enabled us to have a more space-efficient visualization that is interpretable as well.

We also implemented a feature to enable the user to "query" the cDBG by generating a fasta file corresponding to all the nodes associated with a given strain, or all the nodes corresponding to the "core" genome, e.g., the sequences present in all strains. This is similar to functionalities that other packages employ, and we added this feature in hopes that it can make our results more applicable beyond the specific JSON format required by the web application, and serves as a better platform for analysis and comparison of raw genomic data.

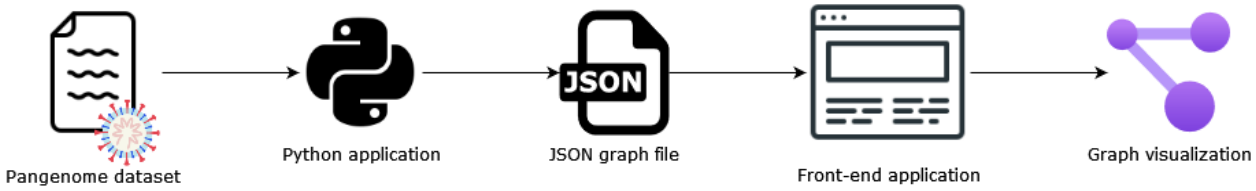
Full-stack Application

Tech Stack and Design

As this is by no means a web-application development course, we will not go into too much detail regarding how the frontend components were built. With that being said, we believe there is still some value in understanding some of the design decisions we made and why we made them. Let's briefly go over each of the tools and libraries used to build the application:

- React: a JavaScript framework that makes building web applications easy and includes ways to manage the application states through their Hooks API
- TypeScript: a statically typed extension of JavaScript (so we can enforce the types of the graphs and reduce the number of bugs at compile time)
- Mantine: a UI library with predefined components that we can use in our application
- Zod: a schema validation library that we use to validate the input JSON file to ensure that it matches our expected format

The following design diagram describes how the front-end application is meant to interact with the rest of the project.



In its current state, the application implements the following features (on manageable input data):

- Parse a JSON graph file to generate an interactive graph visualization
- Display node information (corresponding *k*-mer and strains) when on selecting a node
- Filter the graph to highlight all nodes that correspond to a selected strain

Input Data

In order to visualize the generated graphs, we first had to agree on an output format that would encode the graph information. While there are several more efficient choices, JSON was by far the easiest to work with since many web-based graph visualization libraries use the JSON format to encode nodes and links. Therefore, we decided on the following format (expressed as a TypeScript interface):

```
interface GraphFile {
  nodes: Node[],
  links: Link[],
  strains: Strain[],
}
```

The types for `Node`, `Link`, and `Strain` are as follows (also expressed as TypeScript interfaces):

```
interface Node {
  id: string,
  color: string,
  strains: number[],
  val: number,
}

interface Link {
  source: string,
  target: string,
}
```

```
interface Strain {
  id: number,
  name: string,
  colors: string[],
}
```

Further down the line, we realized that JSON was perhaps not the best way to encode the graph data due to its sheer size. CSV files are able to encode the same data while taking up half the space. However, this approach would require the user to upload two files (one encoding the graph edges and another encoding the node information) and we simply ran out of time to implement these changes.

Finding a Suitable Library and Visualizing the Uncompressed cDBG

With the graph file format settled, we then had to find a suitable library that could suit our needs. The ideal library should support the following features:

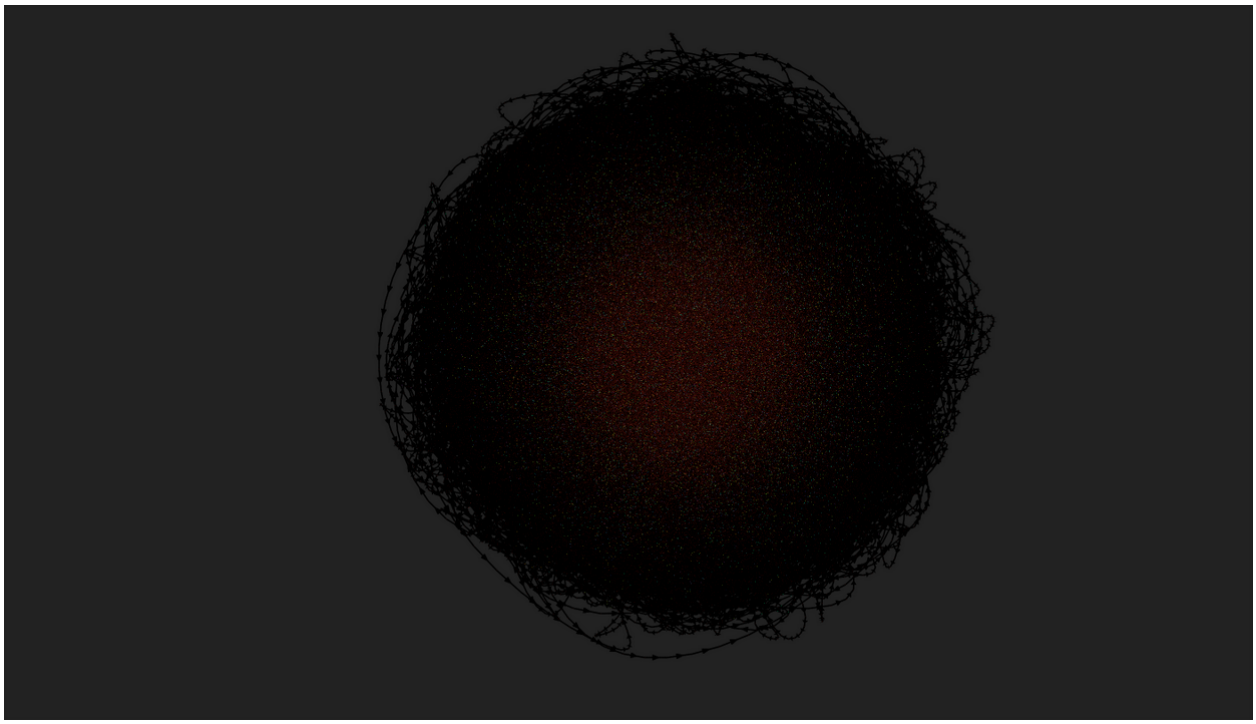
- Colored nodes and links
- Directional links
- Interactable nodes and links (e.g., click and select)
- Zoom and drag
- Self-edges

A great deal of time was spent on experimenting with different libraries. Before we were able to generate our compressed colored De Bruijn graphs, we made an attempt to visualize the uncompressed graphs. Below is a table comparing each library and our findings when passing in the uncompressed graph file.

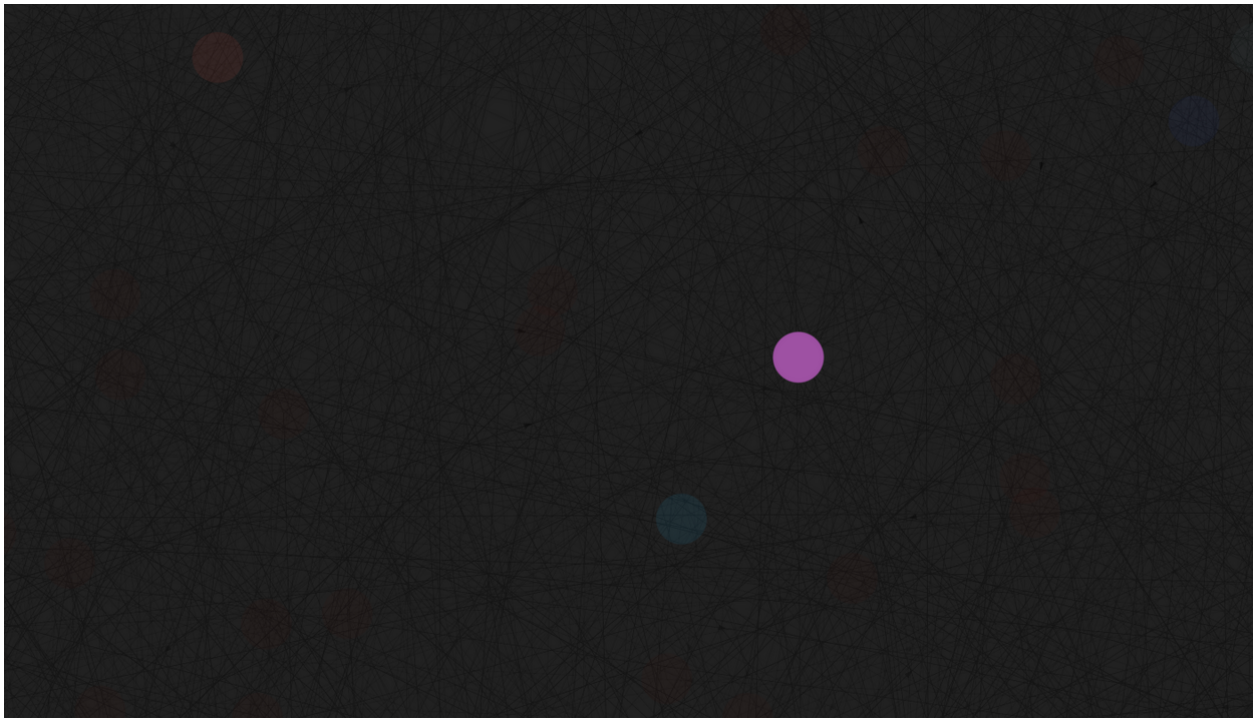
	Colored nodes/links, directional links, interactable nodes/links, zoom/drag	Self-edges	Performance	Additional notes
react-d3-graph	Yes	No	Crashes	None
d3-graph-controller	Yes	Yes, but only up to one self-edge per node	Crashes	None
react-force-graph-2d	Yes	Yes, but only up to one self-edge per node	Crashes	None
react-force-graph-3d	Yes	Yes, but only up to one self-edge per node	Crashes	Generates 3D graphs, but possibly at the cost of performance
cosmograph	Yes	No	Loads, but is extremely slow	GPU-accelerated

While most libraries support the simple requirements such as colored nodes/edges and directed graphs, very few support self-edges. We suspect that this is due to how self edges have to be organized and displayed (since they can possibly overlap if there are multiple). Also, most of these graph libraries use straight edges rather than curved as implementing curves is more computationally intensive. This proves to be a problem since DBGs are allowed to contain several self-edges, and without them, our visualization would be inaccurate. We found only one library that supported multiple self-edges, but it required additional code for each individual edge to orient the edges in a 3D environment so they do not overlap with each other.

Next, we had to address the even larger problem—the sheer amount of data that we needed to process. Even just uploading the JSON file to the application takes ~10 seconds. It is no secret that JavaScript and web browsers were not built for this type of data visualization, and trying to visualize hundreds of thousands of nodes and potentially millions of links is no easy task for any consumer-grade hardware. In fact, cosmograph is likely only able to load the graph successfully due to the fact that it is the only GPU-accelerated library on the list of libraries we experimented with (ran on an Nvidia GeForce RTX 3070). The following page includes some images of the generated visualizations from cosmograph.

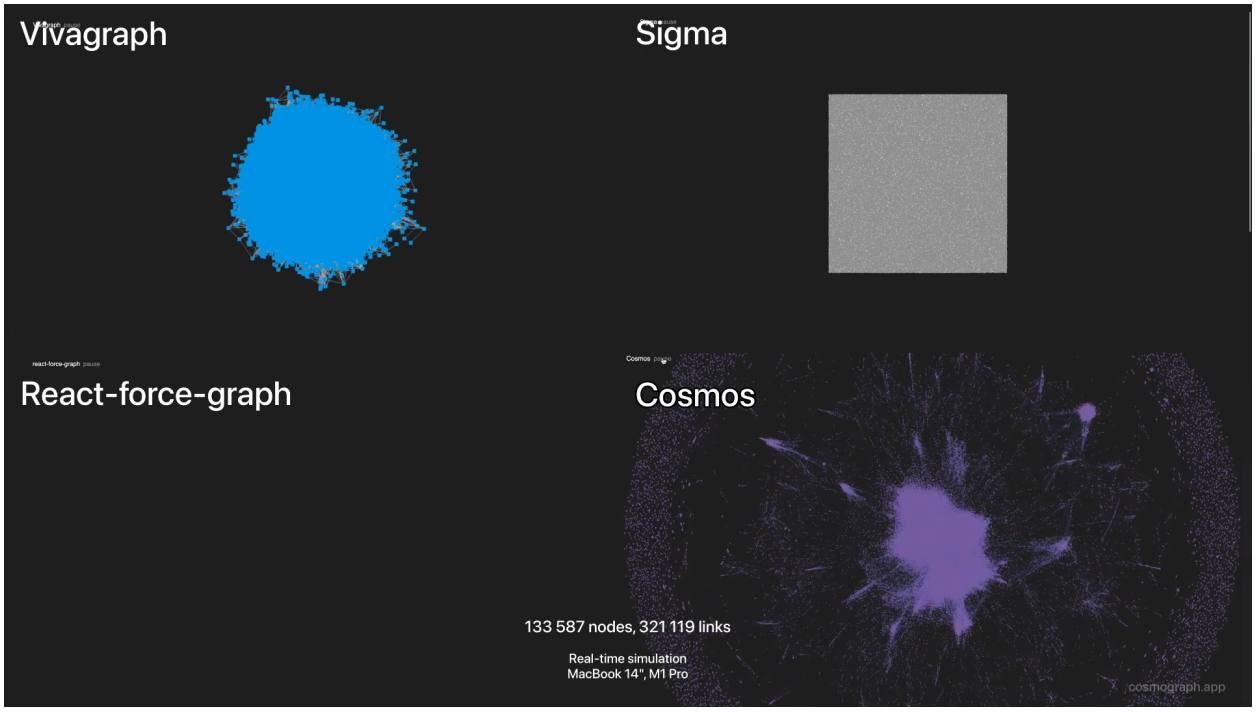


k = 10; 282,831 nodes; 2,898,861 links



Focused onto a single node

Below is a comparison of different graph visualization libraries (taken from this video), the bottom two being ones we experimented with. Note that react-force-graph is unable to render the graph and causes the web browser to become unresponsive.

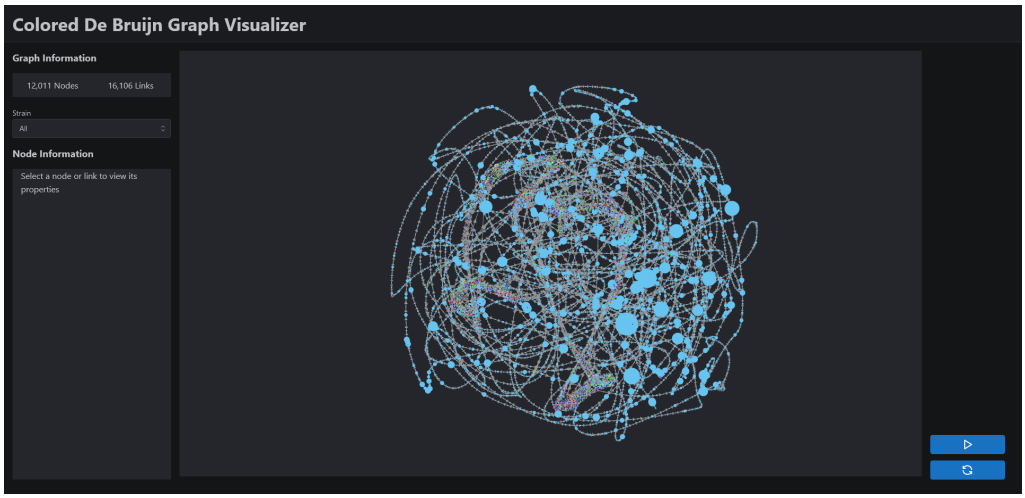


Comparison of graph visualization libraries

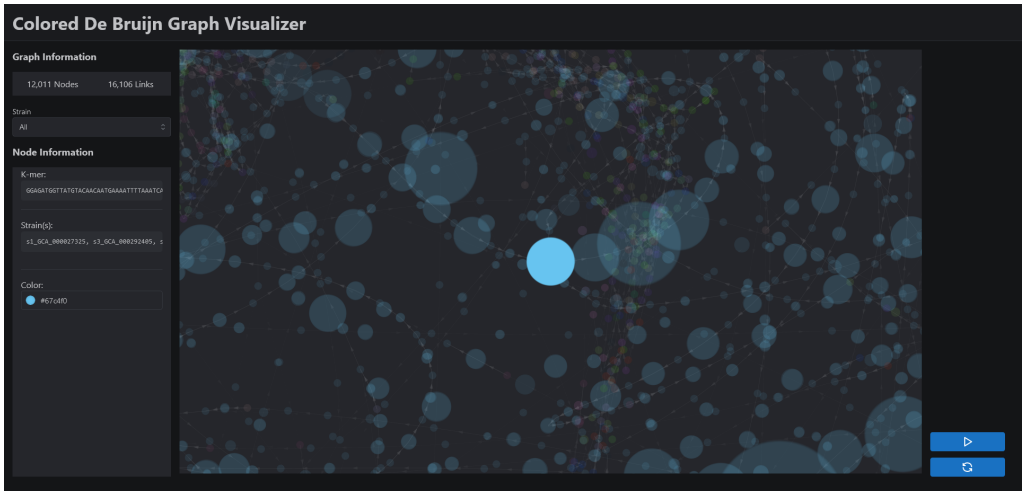
With larger k values, the number of nodes in the graph grows exponentially. For testing purposes, we also tried reducing k and visualizing the resulting graph. At $k = 3$, the graph only had 64 nodes, but the number of edges remained the same, thus resulting in similar performance issues as before. At this point, it became clear that attempting to visualize the uncompressed cDBG in a web-browser is not worthwhile. The hardware limitations, stuttering, lag, and the sheer number of nodes and edges on the graph canvas makes the entire interactive experience clunky and cumbersome.

Visualizing the Compressed cDBG

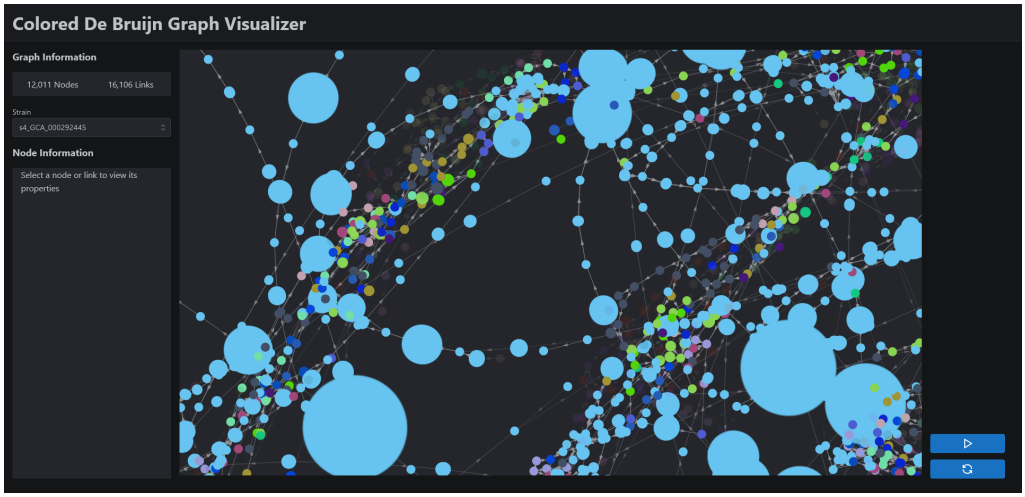
Fortunately, once we were able to compress our De Bruijn graphs, we found much more success in passing those into the frontend application. Since compression reduced the number of nodes and links significantly, the application was able to handle the input data much better. It is important to note that we were unable to get self-edges to render properly. The following page contains some screenshots of the generated visualizations.



Compressed cDBG; 12,011 nodes; 16,106 links

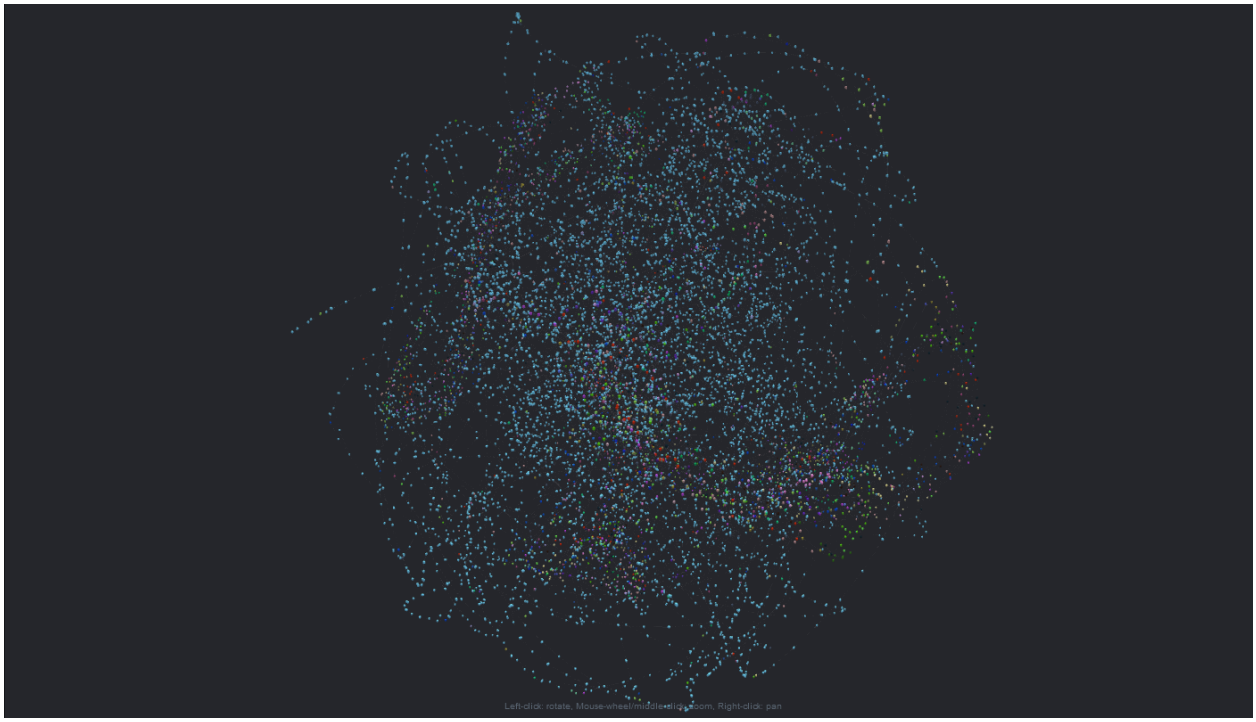


Compressed cDBG; node selected



Compressed cDBG; filtered by a specific strain (note the highlighted vs. unhighlighted nodes)

We found that the cosmograph graph would overlap nodes, meaning larger nodes would cover smaller nodes entirely. This was not a huge issue since selecting a node causes it to become transparent, thus revealing the nodes behind it. However, we thought it would be an interesting idea to give the user the option to visualize the graph in a three-dimensional space; this would allow the user to rotate the graph and see nodes without any overlap. Thus, we went back and tried to use the react-force-graph-3d library to visualize the data. While the graph was successfully generated, we were unable to change the node sizes, which is an important property we wished to keep. The graph was also extremely slow and unresponsive to user input.



react-force-graph-3d output on compressed cDBG input

Future Improvements

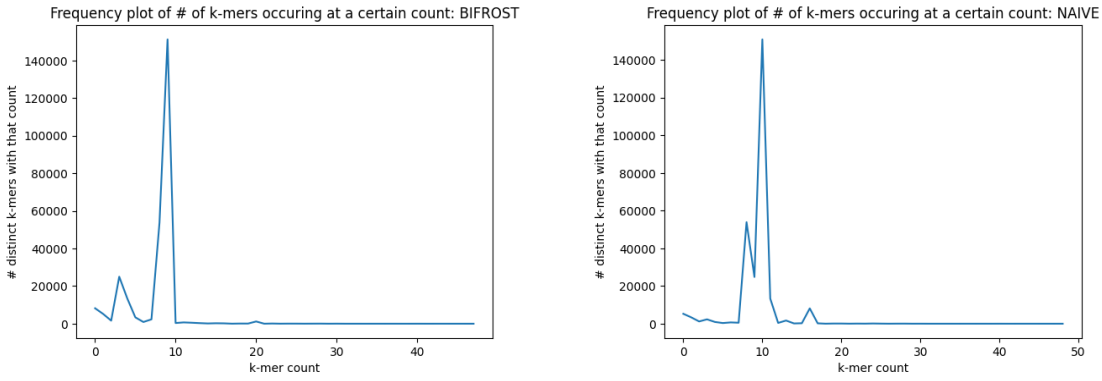
Due to time constraints and the aforementioned limitations, we were not able to accomplish as much as we initially sought out to do. The following list includes some features and improvements that we originally wanted to support and implement:

- Allow the user to download a FASTA file containing the information of the selected nodes
- Download the graph file that corresponds to a specific strain of the pangenome
- Encode self-edge information within each node so that when a node was clicked, it would show how many self-edges correspond to that node
- Currently, the frontend is handling a lot of the data loading and processing when the user interacts with the graph. For example, filtering the graph to only display nodes that correspond with a selected strain was implemented naively. If we moved a lot of the computational logic to a backend server with a corresponding API or to a serverless runtime environment (e.g., AWS Lambda or GCP Cloud Functions), it could significantly reduce the time taken to run these computations and perhaps make the user experience smoother.

Despite the application's shortcomings in being unable to handle very large graphs as well as its inability to display self-edges, we believe that it still serves as an interesting proof-of-concept to visualize De Bruijn graphs.

Results

Regarding the results of our graph generation, it is difficult to compare to existing methods, simply due to the fact that our approach is much more naive. Furthermore, our output does not follow a standard format such as a .gfa file or .csv that can be adequately compared. That being said, there is a comparison to be made as far as graph "correctness", specifically in regards to tool Bifrost. Bifrost also computes colored, compacted De Bruijn graphs, albeit through more sophisticated methods. When constructing graphs with Mycoplasma Genitalium via Bifrost and saving all the node sequences into fasta files, the number of nodes between the Bifrost output and ours are regularly < 10% of the total nodes. While we aren't sure the exact source of this discrepancy, upon examining the sequences of the nodes themselves, it seems that Bifrost does some more work examining sequence similarities. For example, when running with the build -s option, Bifrost omits all kmers that only appear once, a filtering step which we do not take. They also seem to do some more pruning of nodes, searching for "ghost" *k*-mers and removing them. These are all steps that we do not take, hence the slight discrepancy between node length and edge construction. That being said, overall, for each node that is present in our graph, there is a corresponding node present in the Bifrost graph with > 97% sequence identity, indicating a robust construction in comparison on our part. A final comparison of the frequency plot of the number of *k*-mers occurring at a certain count was performed for the two with multiple *k*-mers occurring with a frequency of "1" indicating high error. Notice how NAIVE has fewer *k*-mers that occur fewer times.



Error correction *k* = 10, *k*-mers with errors usually occur fewer times than error-free k-mers

Regarding the web application, it is difficult to compare to other existing methods. However, comparison between libraries used and techniques attempted has been performed above.

Conclusion

From this project, we learned about the De Bruijn graph, explored application of the graph, and utilized technical tools for visualization of what we have learned. We gained understanding for the necessity of efficient data storage methods and compression methods for working with large scale data. On a smaller scale, we learned about succinct structures and how theoretically simple operations can be adapted to be even more spatially and computationally efficient. While exploring the multitude of approaches and taking the time to thoroughly understand what has been done was initially overwhelming, we have an understanding and appreciation of all the innovation made in the sphere of De Bruijn graphs.

For those reviewing our paper, we hope that our research project has served as a comprehensive walk through of colored deBruijn graphs and its various implementations. We also hope that those reviewing our visualization appreciate its capability for handling the genome of the compressed bacterial pangenomes

References

1. Almodaresi, Fatemeh, et al. 'Rainbowfish: A Succinct Colored de Bruijn Graph Representation'. BioRxiv, Cold Spring Harbor Laboratory, 2017, https://doi.org10.1101/138016.

2. Bowe, Alexander, et al. 'Succinct de Bruijn Graphs'. Algorithms in Bioinformatics, edited by Ben Raphael and Jijun Tang, Springer Berlin Heidelberg, 2012, pp. 225–235.

3. Eizenga JM, Novak AM, Sibbesen JA, Heumos S, Ghaffaari A, Hickey G, Chang X, Seaman JD, Rounthwaite R, Ebler J, Rautiainen M, Garg S, Paten B, Marschall T, Sirén J, Garrison E. pan-genome Graphs. Annu Rev Genomics Hum Genet. 2020 Aug 31;21:139-162. doi: 10.1146/annurev-genom-120219-080406. Epub 2020 May 26. PMID: 32453966; PMCID: PMC8006571.

4. Gagie, Travis & Manzini, Giovanni & Sirén, Jouni. (2017). Wheeler Graphs: A Framework for BWT-Based Data Structurese. Theoretical Computer Science. 698. 10.1016/j.tcs.2017.06.016.

5. Holley, G., Wittler, R. & Stoye, J. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms Mol Biol 11, 3 (2016). https://doi.org/10.1186/s13015-016-0066-8

6. 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

7. Inglin, R.C., Meile, L. & Stevens, M.J.A. Clustering of Pan- and Core-genome of Lactobacillus provides Novel Evolutionary Insights for Differentiation. BMC Genomics 19, 284 (2018). https://doi.org/10.1186/s12864-018-4601-5

8. Iqbal, Zamin et al. “De novo assembly and genotyping of variants using colored de Bruijn graphs.” Nature genetics vol. 44,2 226-32. 8 Jan. 2012, doi:10.1038/ng.1028

9. 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

10. Marcus, S., Lee, H., & Schatz, M. C. (2014). SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics (Oxford, England), 30(24), 3476–3483. https://doi.org/10.1093/bioinformatics/btu756

11. Muggli, Martin & Alipanahi, Bahar & Boucher, Christina. (2019). Building large updatable colored de Bruijn graphs via merging. Bioinformatics (Oxford, England). 35. i51-i60. 10.1093/bioinformatics/btz350.

12. Muggli, Martin D et al. “Succinct colored de Bruijn graphs.” Bioinformatics (Oxford, England) vol. 33,20 (2017): 3181-3187. doi:10.1093/bioinformatics/btx067

13. Uwe Baier, Timo Beller, Enno Ohlebusch, Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform, Bioinformatics, Volume 32, Issue 4, 15 February 2016, Pages 497–504, https://doi.org/10.1093/bioinformatics/btv603